Visual Basic.NET. Flujo de programas

Indice

Nº 4 - Visual Basic.NET. Flujo de programas

1. Operaciones con variables

Ya conocemos las variables, sus tipos, las matrices, constantes, enumeraciones... vamos a trabajar ahora con ellas. Ya se que esto te será muy sencillo pero verás algunos detalles interesantes que no conocías. Por ejemplo, ante esta situación:

```
Dim numero as integer

Numero= 4 + 2 * 3

'Número será 10

Numero=(4 + 2) * 3

'Número será 18
```

Fíjate que sutil e importante diferencia tenemos ahí, ya que dependiendo de si ponemos paréntesis o no obtendremos un resultado u otro. Esto es por el orden en que se realizan las operaciones, que son: exponentes, multiplicación, división, suma y resta. Pero esto lo solucionaremos siempre poniendo paréntesis que además de evitarnos estos problemas hará mucho mas legible el código. Sigamos un poco mas, ahora con las cadenas de caracteres o "string":

```
Mi_nombre= Nombre & Apellidos
```

El resultado es la concatenación de las dos cadenas, así que utilizaremos el operador "&" para concatenar cadenas. Podríamos utilizar "+" pero nos quedaría confuso ya que parece que estamos sumando y no añadiendo cadenas. Sigamos con una forma peculiar de realizar operaciones sencillas con variables:

```
' Sumar 10 a la variable Valor: Valor = Valor + 10

Valor +=10

' Multiplicar la variable Valor por 3: Valor = Valor * 3

 Valor *=3

' Dividir Valor por 12: Valor = Valor / 12

Valor /=12
```

Es una forma de abreviar y que los programadores del lenguaje C conocen muy bien. Lo dejamos al gusto del consumidor, yo utilizaré la notación de siempre porque es mas clara a la hora de aprender el código.

Nota En muchas ocasiones las líneas de instrucciones será largas y nos desaparecerán de la pantalla. Podemos añadir un carácter que indica que la línea de instrucción continua debajo:

```
' Una línea con muchas operaciones:

Valor = Valor1 + Valor2 + Valor3
```

```
' El mismo código con división de la línea:

Valor = Valor1 + Valor2 + _

        Valor3
```

1.1 Matemáticas avanzadas

Antes cada lenguaje tenía sus propias instrucciones matemáticas y al cambiar de uno a otro teníamos que ver cual era su equivalente. Ahora podemos centralizar todas las operaciones matemáticas avanzadas utilizando la clase "System.Math" que es parte de .NET. Esto nos asegurará por ejemplo que si pasamos a otro lenguaje de .NET las operaciones son las mismas, además de contar con un gran número de operaciones.

Para utilizar estas operaciones simplemente llamaremos a los métodos de esta clase Math. Estos métodos están compartidos (Shared) que significa que están listos para utilizarse. Este concepto de compartido lo veremos mas adelante. Veamos algunos ejemplos

```
 Dim valor as double

Valor=Math.Sqrt (81)            'Valor = 9.0

Valor=Math.Round (42.889,2)     'Valor = 42.89

Valor=Math.Abs (-10)            'Valor = 10

Valor=Math.Log (24.22)          'Valor = 3.18...

Valor=Math.Pi                   'Valor = 3.14...

' Dividir Valor por 12: Valor = Valor / 12

Valor /=12
```

Y una lista de algunos de los métodos disponibles:

Nombre	Descripción
Abs	Devuelve el valor absoluto de un número
Acos	Arco coseno
Asin	Arco seno
Atan	Ángulo cuya tangente es el valor especificado.
Atan2	Ángulo cuya tangente es el valor especificado del cociente de dos números.
BigMul	Producto de dos números de 32 bits
Ceiling	Devuelve el menor entero mayor o igual que el número especificado.
Cos	Coseno
Cosh	Coseno hiperbólico
DivRem	Calcula el cociente entre dos números y devuelve el resto en un parámetro de salida
Exp	Potencia
Floor	Devuelve el mayor entero menor o igual que el número especificado.
Log	Logaritmo.

Log10	Logaritmo en base 10
Max	El mayor de dos números
Min	El menor de dos números
Round	Redondeo
Sign	Signo de un número
Sin	Seno
Sinh	Seno hiperbólico
Sqrt	Raíz cuadrada
Tan	Tangente
Tanh	Tangente hiperbólica
Truncate	Parte entera de un número.

1.2 Conversión de tipos

Las conversiones de tipos de datos es muy importante ya que nos puede modificar el valor devuelto si no hemos tenido en cuenta que tipos de datos estamos operando. Utilizaremos mucho la conversión de datos, por ejemplo, el usuario meterá un valor numérico en un cuadro de texto que son siempre de tipos string, y deberemos, por tanto, convertirlo a numérico.

Por ejemplo:

```
Dim valorgrande as Integer = 100

Dim valorpequeño as Short

Dim mitexto as String = "100"


' Convertimos el número grande de 32 bits a uno de 16 bits:

Valopequeño= Valorgrande

'Convertimos el texto de "mitexto" a numérico

Valorgrande= mitexto
```

En Visual Basic esto nos puede funcionar pero no es prudente realizar estas operaciones sin realizar la conversión de datos. Además podremos convertir un numérico grande en un numérico pequeño pero no al revés, así que tendremos que asegurarnos de poder hacer esas operaciones. Podemos decirle a Visual Basic que no nos deje hacer estas asignaciones diciéndole con una instrucción que sea "estricto" y que no permita asignaciones de tipos de datos que no sean iguales. Es una práctica muy recomendable pero que nos dará mas trabajo ya que tendremos que utilizar instrucciones de conversión de tipo. Pero ganaremos en que el código sea mas seguro de ejecutar. Así que si añadimos la instrucción:

```
Option Strict On
```

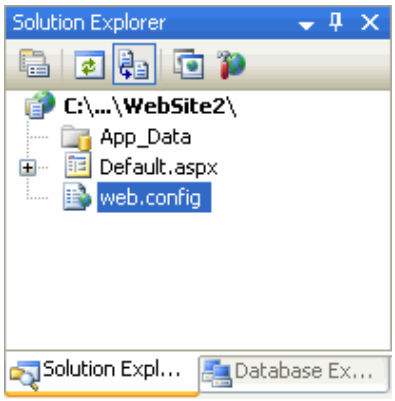
Activaremos ese control y nos dejará hacer las asignaciones anteriores que son realmente peligrosas, por ejemplo decir a una variable numérica que coja el valor de un string. Así que VB nos dará error si no hemos convertido previamente los tipos de datos.

Para realizar esta conversión utilizaremos la función Ctype() que precisa dos argumentos. El primero es la variable a convertir y el segundo el tipo de datos al que queremos convertir, así que nuestro ejemplo anterior quedaría:

```
Dim valorgrande as Integer = 100

Dim valorpequeño as Short

Dim mitexto as String = "100"


' Convertimos el número grande de 32 bits a uno de 16 bits:

Valopequeño= CType (Valorgrande, Short)

'Convertimos el texto de "mitexto" a numérico

Valorgrande= CType (mitexto, Integer)
```

Si hubiésemos hecho la asignación del primer ejemplo con el "Option Strict" a "on" habríamos obtenido un error en tiempo de ejecución. Podemos decirle a nuestra página que haga esa comprobación de tipos con esa instrucción o podemos decirle que lo haga para todo el proyecto entero. Esto se hace en nuestro entorno de desarrollo, dentro del fichero de configuración de nuestro sitio web "web.config"

Este fichero todavía no lo hemos estudiado pero lo vamos a editar de un momento para añadir esta opción. En nuestro explorador de archivos del proyecto en la parte derecha de la pantalla, hacemos doble clic sobre el fichero "web.config":



Que nos mostrara una serie de instrucciones que configuran nuestro web. Para poner esta instrucción iremos a la sección "<compilation>" y añadimos:

```
            Set strict="true" to disallow all data type conversions
            where data loss can occur.
            Set explicit="true" to force declaration of all variables.
        -->
    <compilation debug="true" strict="true" explicit="true">
      <assemblies>
        <add assembly="System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
        <add assembly="System.Web.Extensions, Version=3.5.0.0, Culture=neutral, Publi
```

Pulsa para ver como activar las directivas Explicit y Strinct

1.3 Manipulación de objetos

Sabemos que todo lo que utilicemos en .NET son objetos ya que .NET está orientado a objetos. Es un concepto que todavía no entendemos pero que a lo largo del curso iremos encajando. Por ejemplo, las cadenas de caracteres "string" son objetos en los que utilizaremos sus métodos y propiedades para acceder y manipular sus contenidos.

Por ejemplo, todos los tipos de las clases .NET tienen el método "ToString()" que convierte a cadena de caracteres la variable a la que estemos aplicando ese método. Por ejemplo:

```
Dim cadena as String

Dim valor as Integer = 100

' Convertimos un número a string

cadena= valor.ToString()
```

El tipo de datos String

Uno de los mejores ejemplo de cómo los miembros de las clases pueden remplazar funciones. Veamos ejemplos de cómo podemos manipular las cadenas son los métodos que incorpora:

```
Dim cadena as String = "Esto es una cadena de prueba     "

cadena= cadena.Trim ()          ' = "Esto es una cadena de prueba"

cadena= cadena.Substring (0,4)  ' = "Esto"

cadena= cadena.ToUpper          ' = "ESTO"

cadena= cadena.Replace ("TO","TA")      ' = "TA"

Dim longitud as Integer = cadena.Length ' = 4
```

Cada uno de estos métodos genera otra cadena de caracteres que remplaza la cadena original. El último método devuelve la longitud de la cadena: 4

El método "Substring" , que devuelve una subcadena necesita dos parámetros: el punto de inicio y la longitud. Para esto debes tener en cuenta que las cadenas de caracteres empiezan desde el valor 0. Es decir, como en las matrices, el primer valor es la posición 0, el segundo carácter es la posición 1. Además estos métodos pueden utilizarse de forma anidada, es confuso pero totalmente válido:

```
cadena= cadena.Trim ().SubString(0,4).ToUpper().Replace ("TO","TA")
```

Y otro ejemplo mas interesante todavía, aunque también un poco extraño de utilizar y es proporcionar el propio valor de la cadena de caracteres:

```
cadena= "Hola".ToUpper() ' Devuelve HOLA
```

Dada la importancia y la cantidad de veces que utilizaremos estos métodos veamos una lista de las mas importantes:

Miembro	Descripción
Compare	Compara dos objetos **String** especificados.
Concat	Concatena una o más instancias de **String**
Copy	Crea una nueva instancia de **String** con el mismo valor que una **String** especificada.
CopyTO	Copia un número especificado de caracteres situados en una posición especificada de la instancia en una posición determinada de una matriz de caracteres
Equals	Determina si dos objetos **String** tienen el mismo valor.
IndexOf	Devuelve el índice de la primera aparición de un objeto **String**, o de uno o más caracteres de la instancia en cuestión.
IndexOfAny	Devuelve el índice de la primera aparición en la instancia de un carácter de una matriz de caracteres especificada.
Insert	Inserta una instancia especificada de **String** en una posición de índice especificada de la instancia.
Join	Concatena un objeto **String** separador especificado entre cada uno de los elementos de una matriz **String** especificada, generando una sola cadena concatenada.
LastIndexOf	Devuelve la posición de índice de la última aparición de un carácter Unicode especificado o de un objeto **String** en la instancia.
LastIndexOfAny	Devuelve la posición de índice de la última aparición en la instancia de uno o varios caracteres especificados de una matriz de caracteres Unicode.
PadLeft	Alinea a la derecha los caracteres de la instancia e inserta a la izquierda espacios en blanco o un carácter Unicode especificado hasta alcanzar la longitud total especificada.
PadRight	Alinea a la izquierda los caracteres de la cadena e inserta a la derecha espacios en blanco o un carácter Unicode especificado hasta alcanzar la longitud total especificada.
Remove	Elimina un número de caracteres especificado de la instancia a partir de una posición especificada.
Replace	Reemplaza todas las apariciones de un carácter Unicode o un objeto **String** en la instancia por otro carácter Unicode u otro objeto **String**.
Split .	Identifica las subcadenas de la instancia que están delimitadas por uno o varios caracteres especificados en una matriz, y las coloca después en una matriz de elementos **String**.

Substring	Recupera una subcadena de la instancia.
TToLower	Devuelve una copia de **String** en minúsculas.
ToUpper	Devuelve una copia de **String** en mayúsculas.
Trim	Quita todas las apariciones de un conjunto de caracteres especificados desde el principio y el final de la instancia.
TrimEnd	Quita todas las apariciones de un conjunto de caracteres especificado en una matriz del final de esta instancia.
TrimStart	Quita todas las apariciones de un conjunto de caracteres especificado en una matriz del principio de esta instancia.

Estas son las mas importantes y nos van a permitir realizar todas las operaciones posibles, como ves la versatilidad de los métodos del objeto String es tremenda. Ahora ya ves que String es un objeto con una gran cantidad de métodos para trabajar con él.

Los tipos de datos DateTime y TimeSpan

Sin duda este es el tipo de datos mas complejo de manipular que hay en el mundo de la informática. Es distinto según países (mes/año/día), según programas ya que unos utilizan 10.1.2007 en lugar de 10/1/2007... Pero como no podía ser de otra forma, tenemos multitud de métodos y funciones para manipular este tipo de datos. Las clases que tendremos para este tipo de datos nos van a permitir:

- Extraer una parte de la fecha, por ejemplo, el año. O convertirlo en un valor como el número de días transcurridos.
- Realizar cálculos con fechas y horas de forma sencilla
- Determinar información de las fechas, por ejemplo, el día de la semana que es o será en una determinada fecha.

Por ejemplo, el siguiente código crea un objeto de tipo fecha, establece la fecha y hora y añade determinados días. Y crea una cadena de caracteres que indica el año actual:

```
Dim Mifecha as DateTime = DateTime.Now

MiFecha= Mifecha.AddDays (100)

Dim cadena_fecha as string= Mifecha.Year.ToString
```

Versátil ¿verdad? Son operaciones que antes eran complicadas pero gracias a los método del objeto Fecha serán operaciones triviales. El objeto TimeSpan simplemente representa un intervalo de tiempo, así que será apropiado para almacenar datos de este tipo. El ejemplo que haremos a continuación utilizaremos el TimeSpan para calcular el número de minutos entre dos objetos DateTime

```
Dim mifecha1 as Date = DateTime.Now

Dim mifecha2 as Date = DateTime.Now.Addhours (3000)


Dim Diferencia as TimeSpan

Diferencia= mifecha2.Substract (mifecha1)
```

```
Dim numero_de_minutos as double

numero_de_minutos = Diferencia.TotalMinutes
```

Ya ves que es muy interesante el ejemplo porque hemos utilizado el método en el objeto "Diferencia" que antes habíamos declarado de tipo "TimeSpan", por tanto tiene todos los métodos disponibles para operar con ellos.

Las clases DateTime y TimeSpan también soportan los operadores + y -, con lo que podemos rescribir el ejemplo anterior de esta forma:

```
Dim mifecha1 as DateTime = DateTime.Now

Dim Intervalo as TimeSpan = TimeSpan.FromHours (3000)

Dim mifecha2 as DateTime = mifecha1 + Intervalo


'Restar un objeto de tipo DateTime de otro de tipo TimeSpan


Dim Diferencia as TimeSpan

Diferencia= Mifecha2-Mifecha1
```

Igual que con las cadenas String veamos ahora los métodos mas interesantes de estos dos objetos:

Miembros de DateTime	Descripción
Now	Obtiene la fecha y hora actual del sistema
Today	Obtiene la fecha del sistema, la hora sería: 00:00:00
Year, Date, Month, Day, Hour, Minute	Devuelve una parte del objeto DateTime
DayOfWeek	Devuelve un valor numérico indicando el día de la semana. Por ejemplo, si es domingo, devuelve DayofWeek.SunDay
Add()	Añade un TimeSpan o un DateTime a otro. El resultado es otro DateTime
Subtract	Resta un TimeSpan o un DateTime de otro. El resultado es otro DateTime
AddYears(), AddMonths(), AddDays, AddHours, AddMinutes, AddSeconds, AddMiliseconds	Acepta un entero que es representa un número de años,

	meses... y devuelve un DateTime
DaysInMonth()	Devuelve el número de días del mes especificado
IsLeaYearp()	Devuelve True o False si es un año bisiesto
ToString()	Devuelve una cadena de caracteres que representa el objeto DateTime

Nota Los métodos como Add() o Substract() no cambian el objeto DateTime, en su lugar devuelven un nuevo objeto de tipo DateTime o TimeSpan

Miembros de TimeSpan	Descripción
Days, Hours, Minutes, Seconds, Miliseconds	Devuelve una parte del TimeSpan.
TotalDays, TotalHours, TotalMinutes, TotalSeconds, TotalMiliseconds	Devuelve el valor total del TimeSpan con los días, horas, ... El valor devuelto es de tipo Double, por tanto es fraccionario y el total de días puede ser 23,34
Add() y Substract()	Combina objetos TimeSpan juntos
FromDays, FromHours, FromMinutes, FromSeconds, FromMiliseconds	Permite crear TimeSpan rápidamente, por ejemplo, TimeSpan.FromHOurs (24) crea un TimeSpan de 24 horas
ToString()	Devuelve una cadena de caracteres que representa el objeto TimeSpan

Puede que haya quedado un poco confuso las diferencia entre un tipo y otro, para aclararte te comento que los tipos de valor de DateTime y TimeSpan se diferencian en que DateTime representa un instante de tiempo, mientras que TimeSpan representa un intervalo de tiempo. Esto significa, por ejemplo, que se puede restar una valor de DateTime de otro para obtener el intervalo de tiempo entre ambas. O bien, se podría agregar un TimeSpan positivo al DateTime actual para calcular una fecha futura.

Se puede agregar o restar un intervalo de tiempo a un objeto DateTime. Los intervalos de tiempo pueden ser negativos o positivos y se pueden expresar en unidades como pasos o segundos, o como objetos TimeSpan.
El tipo Array

El tipo Array

El tipo de datos Array también puede manipularse como objeto, ya lo vimos anteriormente al hacer pequeñas operaciones con matrices pero vamos a repetirlo aquí para repasarlo, dada su importancia. Veamos por ejemplo, cómo obtener el total de elementos de un array:

```
Dim Matriz() as Integer= {1, 2, 3, 4, 5}

Dim elementos as Integer

elementos= Matriz.Length ' devuelve 5
```

Podemos utilizar GetUpperBound para averiguar el índice mayor de la matriz:

```
Dim Matriz() as Integer= {1, 2, 3, 4, 5}

Dim mayor as Integer

' 0 representa el primer elemento de la matriz

mayor = Matriz.GetUpperBound (0) ' devuelve 4
```

En una matriz de una dimensión GetUpperBound devuelve un número menos que la longitud. Es decir nos devuelve 4 si tiene 5 elementos, ya que el primero se almacena en la posición 0. En una matriz de dos dimensiones podemos averiguar ese valor en la dimensión que queramos. Veamos un ejemplo en una matriz de dos dimensiones:

```
' Creamos una matriz de dos dimensiones 4 x 2 (4 filas x dos columnas)

Dim matriz (,) as Integer {{1,2},{3,4},{5,6},{7,8}} as Integer

Dim Filas, Columnas as Integer

Filas= Matriz.GetUpperBound (0)         'Filas = 4

Columnas= Matriz.GetUpperBound (1)      'Columnas = 2
```

Con estos valores, dimensiones y su tamaño es fácil hacer un bucle para recorrer esa matriz, que ya veremos mas adelante en los bucles. Como en los anteriores casos, los Array disponen de métodos muy útiles para la manipulación de sus datos, veamos algunos de ellos:

Miembros	Descripción
Length	Devuelve un entero con el número total de elementos de todas las dimensiones de la matriz
GetLowerBound y GetUpperBound	Devuelve la posición del índice del último elemento de la matriz.
Clear()	Elimina los valores contenidos en la matriz.
IndexOf() y LasIndexOf()	Busca un elemento en una matriz unidimensional y devuelve su índice o devuelve el índice de la última aparición de un valor en una matriz
Sort	Ordena una matriz unidimensional
Reverse()	

Invierte el orden de los elementos
de la matriz

2. Mas sobre variables y estructuras

Vamos a seguir viendo algunos detalles mas de declaración de variables, son pocas cosas pero muy útiles...

2.1 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 mas sencilla la primera porque sólo indicamos el tipo de datos una sola vez.

2.2 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**.

2.3 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
```

Estaría disponible para trabajar con esta variable, pero... ¿qué tipo de datos es? Históricamente esto estaba permitido también en anteriores versiones de VB y el tipo utilizado para estos casos era el Variant. Ya lo comentaremos mas adelante, ahora con VB.NET cuando no se define el tipo de datos de una variable se dice que es de tipo **Object**.

El tipo de datos Variant presumía de ser muy inteligente, no hacía falta decirle el tipo de datos que iba a contener porque se adaptaba a todo, es decir le podíamos asignar en el mismo programa esto:

```
Dim mivariable

mivariable=45
mivariable="Jose Rodriguez"
mivariable=True
```

Esta supuesta ventaja se nos volvía enseguida en nuestra contra porque no podíamos hacer un correcto seguimiento de los errores ya que provocaba incoherencias, pero esto se ha quedado en VB...

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

```
Dim z
```

Seria lo mismo que hacer esto otro:

```
Dim z As Object
```

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

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

Pero lo que no podemos hacer es declarar, por ejemplo, dos variables de tipo Integer y 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 si que podemos hacerlo:

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

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

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

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

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

Pero... la recomendación es que no compliques las declaraciones de las variables de esa forma... Procura usar **Dim**s diferentes para diferentes declaraciones, si no quieres declarar cada variable con un Dim, al menos usa un Dim para cada tipo de variable. Por ejemplo el ultimo 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...

3. Flujo de programa. Condiciones

El siguiente tema del que vamos a tratar son las expresiones lógicas. Aquí empezaremos a crear nuestra lógica de programa, es decir podremos decidir que instrucciones ejecutar dependiendo del resultado de unas comparaciones. Evaluaremos expresiones cuyo resultado pueda ser un valor verdadero o falso.

Antes, en algún ejemplo vimos por encima las instrucciones IF /THEN. Ahora vamos a verlas en profundidad, son muy sencillas de utilizar y atentos a la novedad en VB.NET

3.1 If... Then ... Else

Hay ocasiones en las que necesitaremos decidir que hacer dependiendo de algún condicionante, para tomar decisiones usaremos:

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

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

Eso mismo también se suele usar de esta otra forma:

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

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

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

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

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

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

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

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

Como ves, en medio de cada If / Then he usado lo que se llama un **comentario**. Los comentarios empiezan por una comilla simple (apóstrofe), en los comentarios podemos poner lo que queramos, con la seguridad de que no será tenido en cuenta por el Visual Basic.

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

Cuando VB.NET encuentra con algo como esto:

```
If i > 25 Then
```

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

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

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

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

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

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

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

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

Aunque esto mismo podríamos haberlo hecho de esta otra forma:

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

Detengámonos un poco para ver estos operadores lógicos. Repito su definición: los operadores lógicos devuelven un valor de tipo Boolean (true o false). Veamos con mas detalle los operadores que hay:

- **And**. Para que se cumple 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 mas de 25 años OR es Varón entonces...
- **Not**. Esta operación realiza una negación entre dos expresiones.
- **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 "false" el resto no se evaluará devolviendo falso como resultado final.
- **OrAlso**. 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 (+,-). Luego los ampliamos un poco...

Sigamos con los comparadores... 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, vamos a poner lo mismo pero en una expresión mas larga:

```
resultado=30 + 5 *5 /12 * 56
```

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

Por tanto quedaría más claro de esta otra forma:

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

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

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

En cuyo caso sólo se cumplirá cuando A sea 100 o B mayor de 50, pero SIEMPRE x debe ser igual a n * 2. Es decir, debemos utilizar los paréntesis dependiendo de lo que realmente quieras comprobar cuando utilicemos mas 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. Antes se evaluaban todas las expresiones, (con las posibles consecuencias que podían tener si alguna de ellas contenía una función que hacía algún tipo de cálculo largo y al final resultaba que no era necesario haberlo calculado...) y después se empezaba a descartar posibilidades...

3.2 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
```

Mejor ¿no?. Queda mucho mas elegante y mas claro, además su uso se justifica con sólo tener 3 posibilidades de evaluación: dejamos de utilizar el IF..Then para usar este tipo de evaluación. Luego en casos de decisión múltiple utilizaremos este sistema.

Tiene varias opciones mas 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 que hubiese sido 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
End if
```

Menuda diferencia. Pero podemos poner mas cosas, por ejemplo un "Case Else" al final le dice y en los demás valores haz esto. 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 mas 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 mas 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.

Detalles del If...then...else

Vistos estos detalle 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 voy a resumir 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:

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 mas a la derecha de tabulación, de este forma el código es mucho mas legible. A continuación de la evaluación escribiremos las líneas de código necesarias. Veamos un ejemplo:

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

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"

3. Decisión doble

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

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

```
If edad<18 then
     Response.write ("Eres menor de edad")
else
     Response.write ("Eres mayor de edad")
End if
```

4. Doble decisión en una línea

```
If Expresión Then InstrucciónsiCierto Else InstrucciónSiFalso
```

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

5. Decisión múltiple

```
If expresión then
     'código cuando se cumpla
     '...
elseif expresion 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 mas de dos opciones por ejemplo:

```
If valor>0 then
     Response.write ("Valor es mayor que cero")
elseif valor<0 then
     Response.write ("Valor es menor que cero")
else
     Response.write ("Valor igual a cero")
End if
```

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 ElseIf, podríamos seguir haciendo mas evaluaciones.

6. Utilizar Mas de un comparador: AndAlso y OrAlso

Recordamos también que podemos utilizar mas 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. Vale, de acuerdo esto es normal.... mira si se cumplen las dos de acuerdo... o no?

Pero podría mejorarse, por ejemplo si la primera no se cumple la expresión debería devolver ya un "False" y no evaluar la siguiente condición. Pues si esto es una buena mejora y para cuando tengamos practica una gran mejora. Vamos a ver si entiendes este ejemplo (los programadores de VB seguro)

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 lo que va a hacer es 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. Bueno... y aquí ¿donde se puede mejorar? Pues 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: nuestro programa provocaría un error.

Esto nos hace desear una variante del AND para que evalué 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) ANDALSO (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 de lo que pasaba con AND, aunque N sea igual a 3, también se comprobará si X vale más de 10, aunque no sea estrictamente necesario.

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

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

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

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

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

4. Flujo de programa. Bucles

Seguimos viendo las instrucciones básica de VB.NET para el control de nuestro programa, ahora le toca el turno a los bucles. Cada vez que queramos repetir una instrucción o un grupo de ellas un número determinado de veces, e incluso un número indeterminado de veces utilizaremos los bucles..

En Visual Basic .NET existen diferente instrucciones para hacer bucles, cada tipo tiene sus peculiaridades que al conocerlas nos va a permitir elegir el que mas se adecue para la resolución de nuestro problema.

4.1 Bucles For / Next.

Con este tipo de bucle podemos repetir un bloque de instrucciones un número determinado de veces. Su sintaxis es:

```
For <variable numérica> = <valor inicial>  To <valor final> [Step
<incremento>]
   ' contenido del bucle, lo que se va a repetir
 Next
```

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

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

4.2 Bucles For Each

Una instrucción **For Each...Next** repite bucles basados en los elementos de una expresión. Una instrucción **For Each** especifica una variable de control de bucle y una expresión enumeradora. El tipo del valor devuelto por la expresión en una instrucción **For Each** debe ser un *tipo de colección*.

El bucle **For Each...Next** es parecido al bucle **For...Next**, pero ejecuta el bloque de instrucciones una vez por cada elemento de una colección, en vez de un número de veces especificado. Se trata de una variante del bucle For-Next: recorrerá los elementos de una colección (aunque veremos las colecciones mas adelante), digamos que es como moverse por los elementos de un array o matriz. Su sintaxis es:

```
For Each <variable>  In <colección del tipo de la variable>
   ' lo que se hará mientras se repita el bucle
Next
```

Los elementos de *la colección* pueden ser de cualquier tipo de datos.

Este tipo de bucle lo veremos con más detalle en otros temas, pero aquí veremos un par de ejemplos. Sólo decirte que podemos usar este tipo de bucle para recorrer cada uno de los caracteres de una cadena, este será el ejemplo que veremos, ya que, como tendrás la oportunidad de comprobar, será un tipo de bucle que usaremos con bastante frecuencia.

```
Dim s As String

'

For Each s In "Hola Mundo"

    Escribe (s)

Next
```

Que viene a decir que "por cada carácter en 'Hola Mundo' escríbelo."

4.3 Bucles While / End While

Este bucle se repetirá mientras se cumpla la expresión lógica que se indicará después del While. Su sintaxis es:

```
While <expresión>
   ' lo que haya que hacer mientras se cumpla la expresión
End While
```

Tiene una gran diferencia con los bucles For-Next. Así como éstos debíamos decirle cuantas veces se debe repetir, en este caso no hace falta ya que se va a cumplir mientras se cumpla una condición. Por ejemplo, tenemos un fichero con nuestros clientes, deseo recorrerlo entero y para voy a utilizar un bucle. Me planteo inicialmente el For-Next porque es mas sencillo de implementar (escribir en nuestro código) pero... ¿cuantas veces debo realizar el bucle? No lo se, hoy puede que tenga 200 clientes pero luego o mañana ese valor cambiará así que debo utilizar otro tipo de bucle.

Puedo utilizar el While porque le puedo añadir una evaluación y es que mientras existan registros, así que este es el adecuado:

```
While no_finfichero
    Escribe_datos
End While
```

Así que no se va a ejecutar 200 ni 300 veces sino cuando se cumpla que ha llegado al final del fichero.

Con este tipo de bucles, se evalúa la expresión y si el resultado es un valor verdadero, se ejecutará el código que esté dentro del bucle, es decir, entre While y End While. La expresión será una expresión lógica que se evaluará para conseguir un valor verdadero o falso. Veamos algunos ejemplos:

```
Dim i As Integer

'
```

```
While i < 10
    Response.write (i)
    i = i + 1
End While
'
'
Dim n As Integer = 3
i = 1
While i = 10 * n
    ' no se repetirá ninguna vez
End While
```

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

Nota Un bucle For-Next se repite un número finito de veces indicado por el valor "To". Los bucles condicionales se ejecutan mientras se cumplen una expresión así que tendré que tener mucho cuidado para que se llegue a cumplir alguna vez. Si no se cumple esa terminación se quedará en un bucle infinito bloqueando la aplicación. En el ejemplo anterior incrementa la variable i=i+1 para seguir con las iteraciones o repeticiones del bucle.

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

4.4 Bucle Do / Loop

Es muy parecido al anterior pero mas versátil, junto con el For-Next será el que mas utilicemos. Si se utiliza sólo con esas dos instrucciones, este tipo de bucle no acabará nunca y repetirá todo lo que haya entre Do y Loop de forma infinita, así que le pondremos una expresión para que la evalué y pueda terminar con el bucle.

```
'Está incompleto porque debemos poner una condición para que finalice el
bucle:
Do
    'instrucciones
Loop
```

Podemos utilizar dos instrucciones que nos permitirán evaluar expresiones lógicas: **While** y **Until**

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

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

```
Caso 1:
Do While <expresión>
    'instrucciones
Loop

Caso 2:
Do
    'instrucciones
Loop While <expresión>

Caso 3:
Do Until <expresión>
    'instrucciones
Loop

Caso 4:
Do
    'instrucciones
Loop Until <expresión>
```

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

Until, a diferencia de While, la expresión se evalúa cuando no se cumple la condición, es como si negáramos la expresión con While (Not <expresión>) o mejor dicho y traducido uno hace "Mientras se cumpla esta condición haz esto: (Do While)" y el otro dice: "Haz esto hasta que se cumpla esta condición. (Do Until)"

Veamos un ejemplo con DO Until: Do Until X > 10 (repite hasta que X sea mayor que 10):

```
i = 0

Do Until i > 9

    Response.write (i)

    i = i + 1

Loop
```

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

```
i = 0

Do While Not (i > 9)

    Response.write (i)

    i = i + 1

Loop
```

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

4.5 Finalización anticipada de bucles

Normalmente los bucles terminarán cuando cumplan su condición de salida, que para eso la ponemos entre otras cosas. Pero a veces nos interesa romper el bucle y terminarlo sin que llegue a evaluarse mas condiciones. O en el caso del For-Next sin llegar a cumplirse el valor de "To"

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

- **Exit For**
- **Exit While**
- **Exit Do**

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

```
Dim i As Integer
'
For i = 1 To 10
    ' contará de 1 hasta 10
    ' la variable i tomará los valores 1, 2, 3, etc.
Next
'
For i = 1 To 100 Step 2
    ' contará desde 1 hasta 100 (realmente 99) de 2 en 2
    ' la variable i tomará los valores 1, 3, 5, etc.
Next
'
For i = 10 To 1 Step -1
    ' contará desde 10 hasta 1
    ' la variable i tomará los valores 10, 9, 8, etc.
Next
'
For i = 100 To 1 Step -10
    ' contará desde 100 hasta 1, (realmente hasta 10)
    ' la variable i tomará los valores 100, 90, 80, etc.
Next
'
For i = 10 To 1
```

```
    ' este bucle no se repetirá ninguna vez. Le falta el STEP
Next
'
For i = 1 To 20 Step 50
    ' esto sólo se repetirá una vez
Next
```

Como ves la sintaxis es muy sencilla pero en algunos casos hay que tener en cuenta que el valor final del bucle puede que no sea el indicado, todo dependerá del incremento que hayamos especificado. Por ejemplo, en el tercer bucle, le indicamos que cuente desde 1 hasta 100 de dos en dos, el valor final será 99.

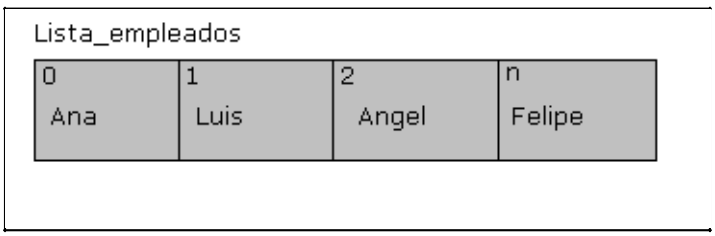
En otros casos puede incluso que no se repita ninguna vez... este es el caso del penúltimo bucle, ya que le decimos que cuente de 10 a 1, pero al no indicar Step con un valor diferente, Visual Basic "supone" que será 1 y en cuanto le suma uno a 10, se da cuenta de que 11 es mayor que 1 y como le decimos que queremos contar desde 10 hasta 1, pues... sabe que no debe continuar. Pero en líneas generales lo utilizaremos sin mayores problemas, no suele ser la fuente de errores...

4.6 Ejemplo

Vamos a ver dos algoritmos ampliamente utilizados para ver con mas detalles esto de los bucles y su ruptura. El problema es el siguiente: tengo una matriz de 200 elementos y quiero buscar la cadena "Felipe" dentro de esa matriz. Todas las instrucciones y declaraciones que vamos a utilizar en este ejemplo ya las hemos visto, así que haz dos cosas: un esfuerzo en repasar lo que veamos aquí y no te acuerdes y por otro lado haz una lectura tranquila y concentrada de este ejemplo, es muy importante porque vamos a resolver un problema, buscaremos la mejor solución y la implementaremos.

Un algoritmo es un conjunto de instrucciones que resuelven un problema, así que a partir de ahora cuando digamos que vamos diseñar un algoritmo o te lo pida en las prácticas significa que hay que diseñar una rutina o grupo de instrucciones para resolver el problema.

Sigamos con nuestro caso y vamos gráficamente la matriz o array:



Tenemos la ventaja de que sabemos que como máximo hay 200 elementos así que parece que el bucle a utilizar puede ser el For-Next. Pero que pasa si "Felipe" está en la posición 20, que lo hemos encontrado pero el bucle seguirá hasta el final lo cual es una evidente pérdida de tiempo y un fallo de diseño (aunque funcione)

Podríamos solucionar la ruptura del bucle con el truco que hemos visto antes de "Exit For". Pero vayamos por partes, primero vamos a hacerlo mal y vemos el porqué de introducirle mejoras:

1ª Solución: bucle For-Next

Vemos su resolución, partimos de una matriz que se definió así:

```
dim lista_empleados (199) as string
```

Como existe el elemento 0, para que tenga 200 el tamaño debe ser en la declaración de 199. Luego otro procedimiento o rutina insertó los nombres en las 200 posiciones. Así que partimos de un problema habitual que es la búsqueda de un elemento en una matriz. Una resolución sencilla puede ser

```
Sub buscar()
      dim i as long
      dim posicion as long

      For i=0 to 199
          if lista_empleados (i)="Felipe" then
              posicion=i
          end if
      Next

      Response.Write ("Encontrado en la posición: " & posicion)

End Sub
```

Lo he puesto sin comentarios para ver el código mas claro, ahora lo analizaremos. Primero creamos un procedimiento que va a buscar el elemento y le pondremos de nombre buscar, luego su sintaxis es: (mas adelante veremos en profundidad esta sintaxis).

```
Sub buscar()
     ...
     ...
End Sub
```

De acuerdo, ahora necesito recorrer el bucle entero para buscar el elemento así que declaro una variable "i" para el bucle For. Habitualmente se utilizan las variables i, j , k par recorrer bucles For. También voy a declarar otra variable que va a almacenar la posición de "Felipe" en la matriz que es el objetivo del algoritmo. Y no necesito mas preparativos:

```
dim i as long
dim posición as long
```

Ya podemos realizar un bucle desde el primer elemento hasta el último:

```
For i=0 to 199
    ...
Next
```

Con esto recorreré todos los elementos de la matriz desde el 0 al 199. Ahora debo comparar si el elemento en el que esté es el que estoy buscado:

```
if lista_empleados(i)="Felipe" then
    ...
end if
```

Es decir, si el elemento de la matriz posición "i" es igual a la palabra "Felipe" haz lo siguiente. Recuerda que tanto para acceder como para introducir un valor en una matriz basta con indicarle su posición o índice, por ejemplo para acceder al elemento 13 será valor=matriz(12) y para meter un valor en la matriz será matriz(34)="Antonio". Sigamos, como ya he encontrado a "Felipe" asignaré en una variable el valor de i, es decir la posición del bucle en la que estoy:

```
if lista_empleados (i)="Felipe" then
    posicion=i
end if
```

Luego tengo el resultado almacenado en la variable posición y el problema resuelto, finalmente le escribo al usuario la posición donde se encontraba "Felipe"

```
Response.Write ("Encontrado en la posición: " & posicion)
```

Aquí he utilizado la instrucción conocida para escribir en una aplicación de consola y he escrito dos cosas que aparecen concatenadas con el símbolo de concatenar "&". Por un lado he concatenado la frase "Encontrado en la posición: " y por otro el valor de una variable 'posición' por eso al escribir uno es constante y va entre comillas y el otro no porque es una variable. Si "Felipe" estuviese en la posición 30 escribiría:

```
Encontrado en la posición: 30
```

Y ya está terminado el algoritmo. Repasa ahora el ejemplo completo de arriba y verás que se lee bastante mejor... Ahora veamos los fallos... está claro el primero y mas importante y ya lo sabes porque lo he comentado antes y es que si "Felipe" aparece en la posición 30 es absurdo que el bucle se realice 200 veces: OBVIO. Así que vamos con la primera mejora...

2º Solución, Bucle For con ruptura

Sabemos ya que debemos finalizar el bucle For cuando ya hayamos logrado nuestro objetivo así que introduciremos la instrucción "Exit For" para decirle al programa que rompa el bucle para finalizarlo. ¿Dónde tendré que colocarlo? Pues lógicamente cuando encuentre lo que quiera, es decir cuando se cumpla la comparación del elemento de la matriz con "Felipe":

```
Sub buscar()
    dim i as long
    dim posicion as long

    For i=0 to 199
        if lista_empleados (i)="Felipe" then
            posicion=i
                Exit For
        end if
    Next

    Response.Write ("Encontrado en la posición: " & posicion)
```

```
End Sub
```

Y funcionará bien, en el momento que se cumpla esto se interrumpirá el bucle For y así aunque lo haya encontrado en la posición número 5 no continuará con las repeticiones: problema solucionado.

Pero... esto no es lo mas elegante, si un bucle "For" está diseñado para realizar una repetición un número determinado de veces, provocar una interrupción parece introducirle algo poco natural, como si fuera un "parche" para que funcione. Rompemos la filosofía del bucle For.

Así que vamos a pensar en otra solución mas elegante: ¿Por qué no hacemos un bucle condicional que simplemente termine cuando encuentre a "Felipe"? Pues si, esa es la mejor solución. Vamos con ella...

3ª Solución: Bucle condicional

Utilizaremos un bucle "Do While" para recorrer nuestra matriz, así que de momento vamos a ver cómo recorremos la matriz con este tipo de bucles, puede parecer un poco mas costoso pero es tremendamente versátil trabajar de esta forma.(ya me lo diréis cuando esté terminado)

Para hacer un bucle por la matriz con un "Do-While" tengo un problema y es que así como en el bucle "For-Next" éste incrementa automáticamente el índice en un "Do-While" lo tendré que crear a mano así que sería algo parecido a esto, compáralo con el "For-Next" de la derecha.

```
Sub buscar()
    dim i as long

    i=0
    Do While (i<199)
        i=i+1
    Loop

End Sub
```

```
Sub buscar()
        dim i as long

        For i=0 to 199

        Next

    End Sub
```

Tampoco hay tanta diferencia... en el de la parte derecha (For) al incrementarse de forma automática el índice "i" no tenemos que preocuparnos de nada en cambio en el otro tendremos que hacer dos modificaciones. Por un lado le tenemos que decir el valor inicial: i=0 y luego dentro del "Do While" incrementaremos a mano el índice para que pase al siguiente...

Recuerda que en estos bucles la condición de salida se debe cumplir alguna vez sino se quedaría en un bucle infinito. Si nosotros nos equivocamos (cosa absolutamente normal y necesaria en cualquier proceso de aprendizaje) y nos olvidamos de ese "i=i+1" resulta que nunca se podrá cumplir la condición de "i<199" porque "i" tendrá siempre el mismo valor "i=0" lo que provocará un bucle infinito y nos bloqueará la aplicación.

Pero esto sólo alguna vez al principio, luego es muy sencillo de escribir y a medida que veamos las posibilidades que nos ofrece lo utilizaremos muy a menudo. Luego la misma resolución sin romper el bucle sería:

```
Sub buscar()
    dim i as long
```

```
    i=0
    Do While (i<199)
        if lista_empleados (i)="Felipe" then
            posicion=i
        end if
        i=i+1
    Loop

    Response.write ("Encontrado en la posición: " & posicion)

End Sub
```

Luego tenemos lo mismo que el caso 1. Lo encontramos pero seguimos recorriendo el bucle así que vamos a mejorarlo.

4ª Solución: bucle condicional con ruptura

Así que una solución "a medias" es la inclusión de un "Exit Do" en el bucle cuando hayamos encontrado lo que queremos:

```
Sub buscar()
    dim i as long

    i=0
    Do While (i<199)
        if lista_empleados (i)="Felipe" then
            posicion=i
            Exit Do
        end if
        i=i+1
    Loop

    Response.write ("Encontrado en la posición: " & posicion)

End Sub
```

Funcionará pero no es lo mejor porque ya que este bucle se cumple si se cumplen determinadas condiciones ¿por qué no son estas condiciones las que interrumpen el bucle? Me explico: podemos hacer que la condición de salida de ese bucle sea o cuando llegue hasta el último elemento o cuando lo encuentre, veamos el último caso.

5ª Solución: bucle condicional con "centinela"

La idea final es hacer que el bucle se repitan x veces mientras se cumpla que :

- No hemos llegado hasta el final de la matriz
- No hemos encontrado a "Felipe"

Así que cuando sea "True" alguna de esas dos comparaciones debemos finalizar el bucle. Para esto nos vamos a apoyar en una variable de tipo Boolean que te recuerdo sólo puede contener dos valores: True ó False. La vamos a llamar "encontrado" y funcionará de esta forma: le pondremos como valor inicial "False". Cuando se vaya recorriendo el bucle haremos la comparación para buscar a "Felipe"... en cuanto lo encontremos le cambiaremos su valor a "True" para que provoque que se interrumpa el bucle al cumplirse esta comparación, mira este código:

```
encontrado=False

Do while Not encontrado 'o encontrado=false que es lo mismo

    if lista_empleados(i)="Felipe" then

        encontrado=True

    end if

Loop
```

He omitido la parte del contador "i" para que se vea mejor. Así que simplemente lo que hace es que repite el bucle mientras "Not encontrado" o lo que es lo mismo: "encontrado=false". En el momento que se encuentre a "Felipe" le asignaremos el valor "True" a la variable "Encontrado". Esto provocará que ya no se cumpla la condición del "Do" porque "encontrado" vale "True" por lo que finalizará el bucle. Veamos todo junto:

```
Sub buscar()
      dim i as long
      dim posicion as long
      dim encontrado as boolean

      encontrado=False
      i=0
      Do while (i<199) and (Not encontrado)
          if lista_empleados(i)="Felipe" then
              encontrado=True
              posicion=i
          end if
          i=i+1
      Loop

      If encontrado=true then
      Response.Write ("Encontrado en la posición: " & posicion)
   else
      Response.Write ("No he encontrado a Felipe")
   end if

  End sub
```

Fíjate bien en los pasos:

- Declaramos las variables a utilizar
- Iniciamos las variables
- Comenzamos el recorrido

- Termina el recorrido bien porque llegamos al final o porque hemos encontrado a "Felipe"
- Escribimos el resultado

La parte de condición del bucle queda como:

```
Do while (i<199) and (Not encontrado)
    ...
Loop
```

Es decir cuando una de las dos condiciones se cumpla o su valor sea True terminará el bucle, con lo que queda perfecto. Finalmente escribimos un resultado dependiendo de si lo ha encontrado o no, que en este caso será cuando la variable booleana "encontrado" tenga el valor "true".

Este algoritmo es muy especial porque pertenece a la lista de los estándar que se enseñan en todos los libros de programación e incluso tiene nombre: "algoritmo del centinela". El nombre es evidente, hay un centinela (variable encontrado) que indica cuando se debe detener el bucle.

Espero que haya estado claro, si es así enhorabuena a los dos, a mi por haber plasmarlo la idea y a tú porque con esto hemos dado un paso enorme en la programación aunque te haya parecido sencillo. :-)

Sólo una mejora mas para este ejemplo. En lugar de poner fijo el valor 199 de la matriz porqué no le ponemos una función que nos diga cuantos valores tienen la matriz. Me explico, ya sabes que con "Redim" se puede cambiar el tamaño de la matriz... puede que a lo largo del programa se haya modificado y ahora tengamos 250 elementos... entonces nuestro bucle no funcionará bien. Para solucionarlo, en lugar de ponerle 199 le pondremos: Ubound(lista_empleados). Si recuerdas, cuando vimos las matrices "Ubound" nos decía el valor máximo de la matriz, así que:

```
Sub buscar()
    dim i as long
    dim posicion as long
    dim encontrado as boolean

    encontrado=False
    i=0
    Do while (i<Ubound(lista_empleados)) and (Not encontrado)
        if lista_empleados(i)="Felipe" then
            encontrado=True
            posicion=1
        end if
        i=i+1
    Loop

    If encontrado=true then
    Response.Write ("Encontrado en la posición: " & posicion)
 else
    Response.Write ("No he encontrado a Felipe")
 end if

End sub
```

Es decir mientras sea menos que el número de elementos (Ubound), haya los que haya. Con esto nos ha quedado "niquelada", debes guardarla muy cerca porque la utilizaremos mucho, cuando lleguemos a los ficheros será fundamental para recorrerlos.

Ya hemos visto todos los bucles posibles, sigamos con mas cosas importantes del código de VB .Net

5. Métodos

Los métodos son las forma mas eficaz de agrupar código. Estos métodos no los debes confundir con los métodos que utilizábamos en los objetos. Los métodos son lo que hasta hace poco llamábamos procedimientos. En esta nueva filosofía de orientación a objetos llamaremos a estos procedimientos métodos ya que de alguna forma son parte del gran objeto que forma nuestro programa o sitio web.

Como hemos visto en algunos de los ejemplo hemos escrito en varias ocasiones las mismas líneas de código. Visual Basic.NET permite agrupar estas líneas de código que se repiten para poderse llamar tantas veces como nos haga falta. Estos mini-programas se llaman procedimientos. Será muy habitual que ASP.NET ejecute una serie de instrucciones, se llame a un procedimiento para que ejecute unas instrucciones según nuestra petición y luego sigamos en el punto donde estábamos en nuestro programa principal.

Por ejemplo, queremos escribir código para poner unas líneas de cómo contactar con el departamento de ventas. En lugar de escribir estas líneas cada vez que queramos esto podemos escribir un procedimiento que haga esta operación, y luego invocar a este procedimiento tantas veces como necesitemos.

Existen dos tipos de métodos:

- Procedimientos. Realizan distintas operaciones, por ejemplo ejecutar unas instrucciones al cargarse la página (Sub Form_Load)
- Funciones. Realizan procesos que devuelven un resultado. Por ejemplo para calcular el valor máximo de unos datos le proporcionamos éstos y la función nos devuelve un valor con el máximo de ellos.

¿Cuales son los beneficios de esta modularización?

5.1 Modularización

Es habitual ver en los libros se programación que la filosofía de la correcta escritura de código es utilizar el "divide y vencerás". Esto es, lo mejor para resolver problemas en dividirlos en problemas mas pequeños, así en vez de resolver un problema grande resolveremos otros mas pequeños con lo el código es mas reducido y el error más fácil de localizar. A esta forma de escribir el código se le llama también "diseño descendente". Por lo tanto nuestro programa lo dividiremos en pequeños fragmentos más fáciles de escribir, depurar y reutilizar utilizando procedimientos y funciones.

Como ves todo son ventajas al modularizar nuestro código, es evidente que si tenemos que llamar 10 veces a un código que tiene 100 líneas si éstas las reunimos en un procedimiento llamaremos a éste 10 veces en lugar de ejecutar 1000 líneas :-)

5.2 Procedimientos

Un procedimiento **Sub** consiste en una serie de instrucciones de Visual Basic delimitadas por las instrucciones **Sub** y **End**. Cada vez que se llama a un procedimiento, se ejecutan las instrucciones de éste, desde la primera instrucción ejecutable tras la instrucción **Sub** hasta la primera instrucción **End Sub**, **Exit Sub** o **Return** que se encuentre.

Un procedimiento **Sub** ejecuta acciones, pero no devuelve ningún valor. Puede tomar argumentos, como constantes, variables o expresiones, que le pasa el código de llamada.

La sintaxis para declarar un procedimiento **Sub** es la siguiente:

```
Sub nombreproc[(listaargumentos)]

   ' Instruciones para realizar el proceso.

End Sub
```

Nota Su recuerdas, que cuando vimos las variables podían ser declaradas como "Private" que significan que solo estarán disponibles en ese módulo. Lo mismo sucede con los procedimientos, podemos declararlos con "Private Sub calcula ()" Donde definimos un procedimiento que solo estará disponible en ese módulo, que corresponderá a nuestra página.

Declaración de argumentos

Los argumentos son los valores que necesita para realizar el proceso. Por ejemplo si creamos un procedimiento para buscar un dato en un fichero los argumentos serán el fichero y el dato a buscar. Pero no son obligatorios, en muchas ocasiones podemos llamar a un procedimiento sin indicarle ningún parámetro, por ejemplo para que ponga en negrita un dato en pantalla o para que suene un sonido en el ordenador...

Los argumentos de un procedimiento se declaran igual que las variables, especificando el nombre y el tipo de datos del argumento. También puede especificarse el mecanismo que se va a utilizar para pasar el argumento, así como si se trata de un argumento opcional.

La sintaxis de los argumentos en una lista de argumentos es la siguiente:

```
[Optional] [ByVal|ByRef] [ParamArray] argumento As tipodatos
```

Si el argumento es opcional, la declaración de éste debe contener también un valor predeterminado, como se muestra a continuación:

```
Optional [ByVal      argumento As tipodatos = valorpredeterminado
```

Tranquilo, esto que estamos viendo es la sintaxis oficial, luego utilizaremos menos cosas y verás como es mas sencillo.

Sintaxis de llamada

Los procedimientos **Sub** se invocan de forma explícita, con una instrucción de llamada independiente. No se les puede llamar utilizando su nombre en una expresión. La instrucción de llamada debe suministrar el valor

de todos los argumentos que no sean opcionales, y debe incluir la lista de argumentos entre paréntesis. Si no se proporcionan argumentos, se puede omitir el paréntesis.

La sintaxis para llamar a un procedimiento **Sub** es la siguiente:

nombresub[(*listaargumentos*)]

El procedimiento **Sub** que aparece a continuación notifica al usuario del equipo la tarea que está a punto de realizar la aplicación, y también muestra una marca de tiempo. En lugar de duplicar este fragmento de código al principio de cada tarea, la aplicación simplemente llama a `"Avisaoperador"` desde varios lugares. Cada llamada pasa una cadena al argumento `Tarea` que identifica la tarea que se va a iniciar.

```
Sub Avisaoperador(ByVal Tarea As String)

   Dim hora_inicio As Date      ' Variable local de tipo fecha.

   hora_inicio = TimeOfDay()    ' Obtiene la hora actual.


   ' Utilizamos el response.write para escribir un texto.

   Response.Write("Iniciando " & Tarea & " a las " & CStr(hora_inicio))

End Sub
```

Una llamada a `"avisaoperador"` sería:

```
 AvisaOperator("Actualizar fichero")
```

Delante de las llamada se puede poner la palabra "Call" pero es opcional y no hace nada, mas que otra cosa en por compatibilidad con otras versiones.

A lo largo del proyecto iremos creando muchos procedimientos para ir realizando pequeños procesos. Dividiremos tareas complicadas en pequeños Sub que nos facilitarán su depuración y mantenimiento.

Es decir, en lugar de escribir 100 líneas de código seguidas para resolver un programa, lo estructuraremos en pequeños fragmentos que irán realizando pequeños procesos para en conjunto, resolver el problema principal.

Nos va a simplificar las tareas de programación al dividir los programas en componentes lógicos más pequeños. Los procedimientos resultan muy útiles para condensar las tareas repetitivas o compartidas, como cálculos utilizados frecuentemente, manipulación de texto y controles, y operaciones con bases de datos.

Hay dos ventajas principales cuando se programa con procedimientos:

- Los procedimientos permiten dividir los programas en unidades lógicas, cada una de las cuales se puede depurar más fácilmente que un programa entero sin procedimientos.
- Los procedimientos que se utilizan en un programa pueden actuar como bloques de construcción de otros programas, normalmente con pocas o ninguna modificación

5.3 Funciones

Las funciones son similares a los procedimientos pero devuelven siempre un valor. Por ejemplo una función para calcular una raíz cuadrada: le proporciono unos datos y me devuelve un resultado.

Una **función (Function)** consiste en una serie de instrucciones de Visual Basic delimitadas por las instrucciones **Function** y **End Function**. Cada vez que se llama a un procedimiento de este tipo, se ejecutan las instrucciones de éste, desde la primera instrucción ejecutable tras la instrucción **Function** hasta la primera instrucción **End Function**, **Exit Function** o **Return** que se encuentre.

Un procedimiento **Function** es similar a un procedimiento **Sub**, pero además devuelve un valor al programa que realiza la llamada al procedimiento. Como en los procedimientos puede tomar argumentos, como constantes, variables o expresiones, que le pasa el código de llamada.

La sintaxis para declarar un procedimiento **Function** es la siguiente:

```
Function  nombrefuncion[(listaargumentos)] As tipodatos

   ' Instrucciones de la función...

End Function
```

Los argumentos se declaran del mismo modo que en un procedimiento **Sub**.

Valores devueltos

El valor que un procedimiento **Function** devuelve al programa que realiza la llamada se denomina *valor devuelto*. La función puede devolver dicho valor de dos maneras:

- La función asigna un valor a su propio nombre de función en una o más instrucciones del procedimiento. No se devuelve el control al programa que realiza la llamada hasta que se ejecuta una instrucción **Exit Function** o **End Function**, como en el siguiente ejemplo:

```
Function nombrefuncion[(listaargumentos)] As tipodatos

   ' ...

   nombrefuncion = expresión  .

   ' ...

End Function
```

- La función utiliza la instrucción **Return** para especificar el valor devuelto, e inmediatamente devuelve el control al programa de llamada, como en el siguiente ejemplo:

```
Function nombrefunción[(listaargumentos)] As tipodatos

   ' ...

   Return expresión

   ' ...

End Function
```

La ventaja de asignar el valor devuelto al nombre de la función es que el control permanece en la función hasta que el programa encuentra una instrucción **Exit Function** o **End Función**, lo que permite asignar un valor previo y, si es necesario, se puede ajustar después.

Todos los procedimientos **Function** tienen un tipo de datos, al igual que las variables. La cláusula **As** de la instrucción **Function** especifica el tipo de datos, y determina el tipo del valor devuelto. En las siguientes declaraciones de ejemplo se ilustra esto último:

```
Function ayer As Date

   ' ...

End Function

Function BuscaSqrt(ByVal Numero As Single) As Single

   ' ...

End Function
```

Sintaxis de llamada

Para llamar a un procedimiento **Function**, hay que incluir el nombre y los argumentos de éste en la parte derecha de una asignación o en una expresión. Esto último se ilustra en los siguientes ejemplos de llamada:

```
valor= nombrefuncion[(listaargumentos)]

If ((nombrefuncion[(listaargumentos)] / 3) <= expression) Then ...
```

El siguiente procedimiento **Function** calcula la hipotenusa de un triángulo rectángulo a partir de los valores de los catetos:

```
Function Hypotenusa (ByVal Cara1 As Single, ByVal Cara2 As Single) As Single

   Return Math.Sqrt((Cara1 ^ 2) + (Cara2  ^ 2))

End Function
```

Atento a estas dos formas de llamar a esta función `Hypotenusa`:

```
Dim TestLength, PruebaHypotenusa, X, Y, Area As Single

Pruebahipotenusa = Hypotenusa(20, 10.7)

If Hypotenuse(X, Y)=30 then

        ...

end if
```

Como ves hay dos formas de llamar a las funciones (no te preocupes nos cansaremos de hacer funciones), una es asignando su valor a una variable como es en el primer caso, así ya tengo ese valor en la variable "Pruebahipotenusa" y puedo seguir con el programa. La otra forma es llamarla directamente como si fuera una expresión: la parte del If... en ese caso realizará la llamada para calcular el valor y luego hará la comparación

para ver si su valor es 30.

Common Language Runtime (recuerda que es el lenguaje de .NET) proporciona multitud de funciones, que se encuentran en el espacio de nombres **Microsoft.VisualBasic**. Entre las funciones más comunes se encuentran, por ejemplo, **Beep**, **MsgBox** y **StrComp**. Podemos llamar a estas funciones de la misma manera que llama a los procedimientos **Function** que hemos creado nosotros.

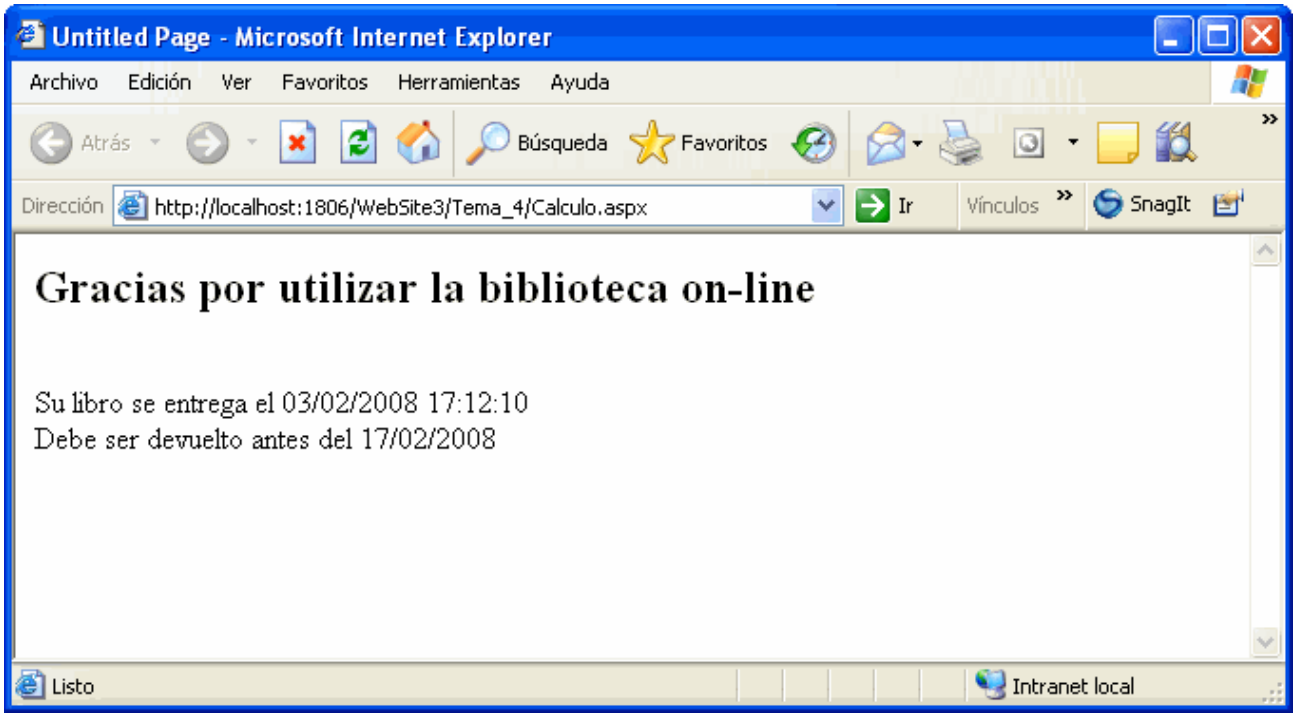
Nos quedan dos partes de ver es los procedimientos y funciones: los argumentos y el alcance o visibilidad. Pero antes hagamos un ejemplo. Crea una página web y escribes el siguiente código. Ten en cuenta que no utilizaremos el método de "response.write" para escribir sino que volvemos a poner dos controles de tipo "Label" en los que hemos puesto nombre a sus "ID": "mensaje1" y "mensaje2" así para escribir pondremos: mensaje1.text="hola". Atento al ejemplo:

Es un título con dos controles label debajo. El código será el siguiente:

```
<script runat="server">
sub page_load()
    dim fecha_devolucion as date
    dim fecha_comprobacion as date

    'Asignamos la fecha de hoy
    fecha_comprobacion=Now()

    'Comprobamos cuando debe haberlo devuelto
    fecha_devolucion=busca_fecha (fecha_comprobacion)
    mensaje1.text="<br>Su libro se entrega el " & fecha_comprobacion
    mensaje2.text="<br>Debe ser devuelto antes del " & fecha_devolucion
end sub

function busca_fecha (fecha as date) as date
    return datevalue(dateadd("d",14,fecha))
end function

</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <h2>
        Gracias por utilizar la biblioteca on-line</h2>
    <form id="form1" runat="server">
    <p>
        <asp:Label ID="mensaje1" runat="server" Text="mensaje1"></asp:Label>
        <asp:Label ID="mensaje2" runat="server" Text="mensaje2"></asp:Label>
    </p>
    <div>
```

A los dos label de la pantalla les he puesto el nombre de "mensaje1" y "mensaje2". Esto lo puedes hacer escribiendo en el código en la vista "Source" del editor, justo cuando estamos escribiendo la otra parte.

2. Ejecuta la página en el navegador:



En este caso el código es muy diferente a lo visto anteriormente, por un lado la secuencia ya no es igual, ya no se ejecutan todas las líneas en un sólo procedimiento "Form_Load" sino que se ejecutan en otros módulo, en nuestro caso en una llamada a una función. Por otro lado hemos utilizado alguna función de Visual Basic .NET para hacer un cálculo de fechas y por otro un procedimiento Sub que se ejecuta en la cargar de la página.

La función creada es:

```
function busca_fecha (fecha as date) as date

   return datevalue(dateadd("d",14,fecha))

end function
```

Hemos seguido la sintaxis ya vista antes de "function" + nombre de función + (parámetros de entrada) + parámetros de salida. El dato que le proporcionamos a la función es "fecha" y el parámetro de salida es de tipo fecha "as date".

La función sólo añade 14 días a la fecha proporcionada como parámetro. Las operaciones con fechas normalmente son complicadas así que .NET nos ofrece una buena cantidad de funciones para realizar cálculos. Estas funciones las veremos al final en un anexo. De momento la función:

```
dateadd("d",14,fecha)
```

Necesita tres argumentos para calcular, el primero es el intervalo, es decir que parte de la fecha va a ser el operador: días, meses, años, horas, ...el segundo argumento es lo que queremos incrementar y el tercer

argumento a que dato. Así que incrementa 14 días al valor contenido en "fecha". Puesto que cuando trabajamos con el tipo date se almacena también la hora utilizamos una segunda función para extraer sólo la parte de la fecha:

```
datevalue(dateadd("d",14,fecha))
```

También hemos utilizado una de las funciones de .NET para que nos devuelva la fecha del sistema:

```
fecha_comprobacion=Now()
```

Para llamar a nuestra función como siempre devuelven valores debemos hacerlo siempre estableciendo su dato devuelto a una variable:

```
fecha_devolucion=busca_fecha (fecha_comprobacion)
```

5.4 Ámbito de las variables

Las variables tienen una vida determinada, es decir, dependen de como se creen se crean y destruyen en nuestras páginas. Así las funciones declaradas dentro de un procedimiento sólo son válidas para ese procedimiento, si intentamos utilizar su valor en otro se producirá un error.

El ámbito o visibilidad de las variables definirá su capacidad de acceso que tenemos a esta variable. Si sintaxis es:

ámbito Dim variable as tipovariable

La palabra Dim la podremos omitir en algunas partes de nuestro código, ahora veremos los tipos de ámbito y ejemplos.

Nivel de procedimiento

Una variable declarada dentro de un procedimiento sólo es visible para él: es **local** o a **nivel de procedimiento**.

Nivel de módulo

Una variable declarada en la zona de declaraciones del módulo es visible dentro de ese módulo. Pueden acceder a estas variables todos los procedimientos:

Atención, deben estar declaradas en el módulo, esto es en la parte del código de nuestra página

Duración de las variables

Este el tiempo en el que las variables están activas y dependen de su ámbito:

- Procedimiento: Se crean al entrar al procedimiento y se liberan o finalizan al terminar el procedimiento.

- Módulo. Si son privadas para ese módulo se inician con él y finalizan al descargarse el módulo.

Observa este código:

```
Sub proceso1()
     ' Variable declarada a nivel de procedimiento
    Dim cadena As string
     '
    cadena="Hola, soy la variable 1"
End Sub

Sub proceso2()
     ' Variable declarada a nivel de procedimiento
    Dim cadena As string
     '
    cadena="Hola, soy la variable 2"
End Sub
```

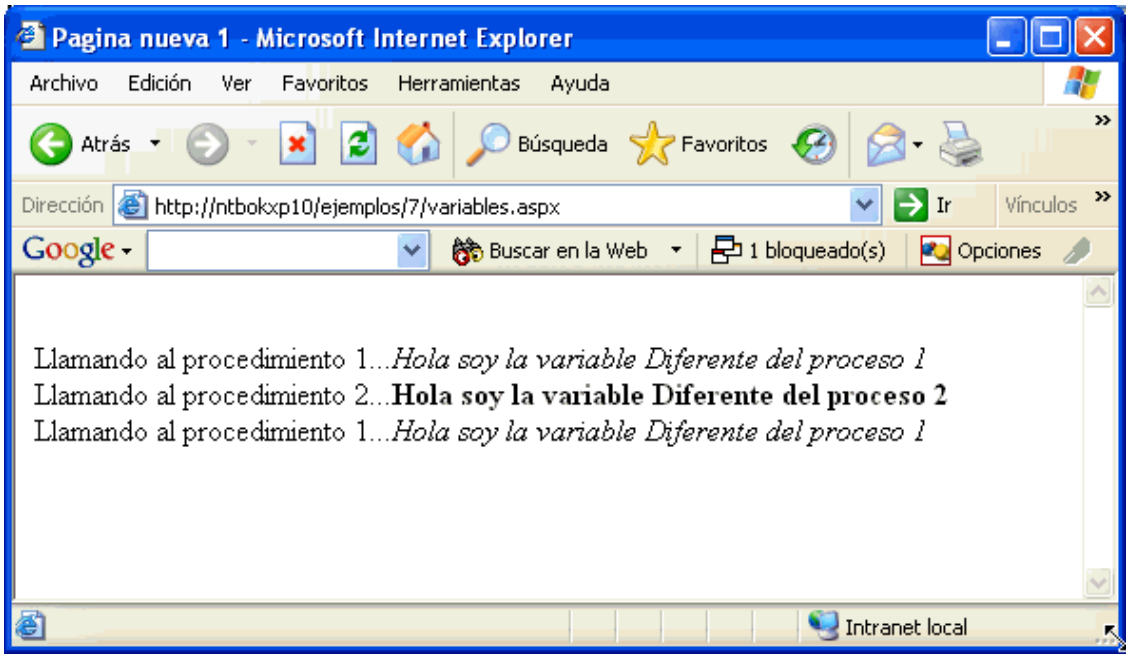
Son procedimientos diferentes así que no hay problema en utilizar el mismo nombre de variable en los dos ya que no son visibles entre ellas, son variables locales. Se crean al entrar al procedimiento y se destruyen al salir, liberando la memoria y recursos utilizados.

Veamos un sencillo ejemplo de procedimientos y variables:

1. Crea una página con el nombre "variables.aspx" e incluye este código:

```
4
5  <script runat="server">
6  sub page_load()
7      Response.write ("<BR />Llamando al procedimiento 1...")
8      proceso_1
9      Response.write ("<BR />Llamando al procedimiento 2...")
10     proceso_2
11     Response.write ("<BR />Llamando al procedimiento 1...")
12     proceso_1
13 end sub
14
15 sub proceso_1
16     dim diferente as string
17     diferente="<i>Hola soy la variable Diferente del proceso 1</i>"
18     response.write (Diferente)
19 end sub
20
21 sub proceso_2
22     dim diferente as string
23     diferente="<b>Hola soy la variable Diferente del proceso 2</b>"
24     response.write (Diferente)
25 end sub
26 </script>
```

2. Ejecuta la página:



Veamos ahora que ha pasado.... este programa tiene dos procedimientos idénticos:

```
sub proceso_1
   dim diferente as string
   diferente="<i>Hola soy la variable Diferente del proceso 1</i>"
   response.write (Diferente)
end sub

sub proceso_2
   dim diferente as string
   diferente="<b>Hola soy la variable Diferente del proceso 2</b>"
   response.write (Diferente)
end sub
```

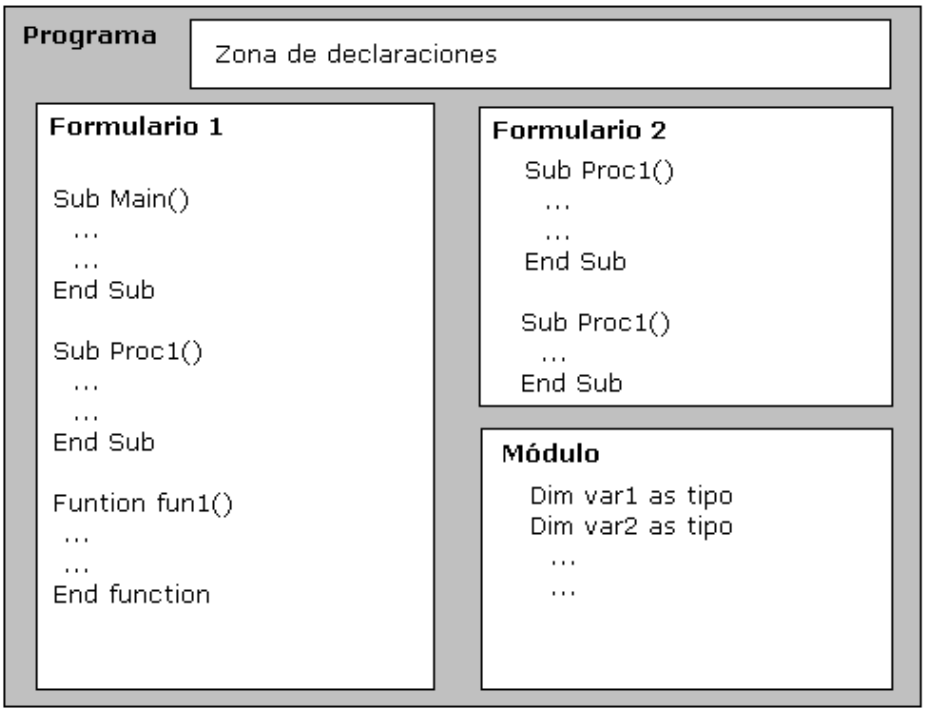
La variable "diferente" se ha declarado a nivel del procedimiento así que no será visible por el otro, por eso se permite que se llame de igual forma. Luego hemos realizado tres llamadas a los dos procedimientos para escribir el contenido de las variables que permanece igual:

```
Response.write ("<BR />Llamando al procedimiento 1...")
proceso_1
Response.write ("<BR />Llamando al procedimiento 2...")
proceso_2
Response.write ("<BR />Llamando al procedimiento 1...")
```

```
proceso_1
```

Variables globales

Sabemos crear y manejar variables en los procedimientos pero... ¿cómo hacemos para que puedan utilizar la misma variable en los dos procedimientos?. La solución está en las variables globales. Estas se declaran fuera de los procedimientos y son visibles para todos. Observa este gráfico de un programa de VB:

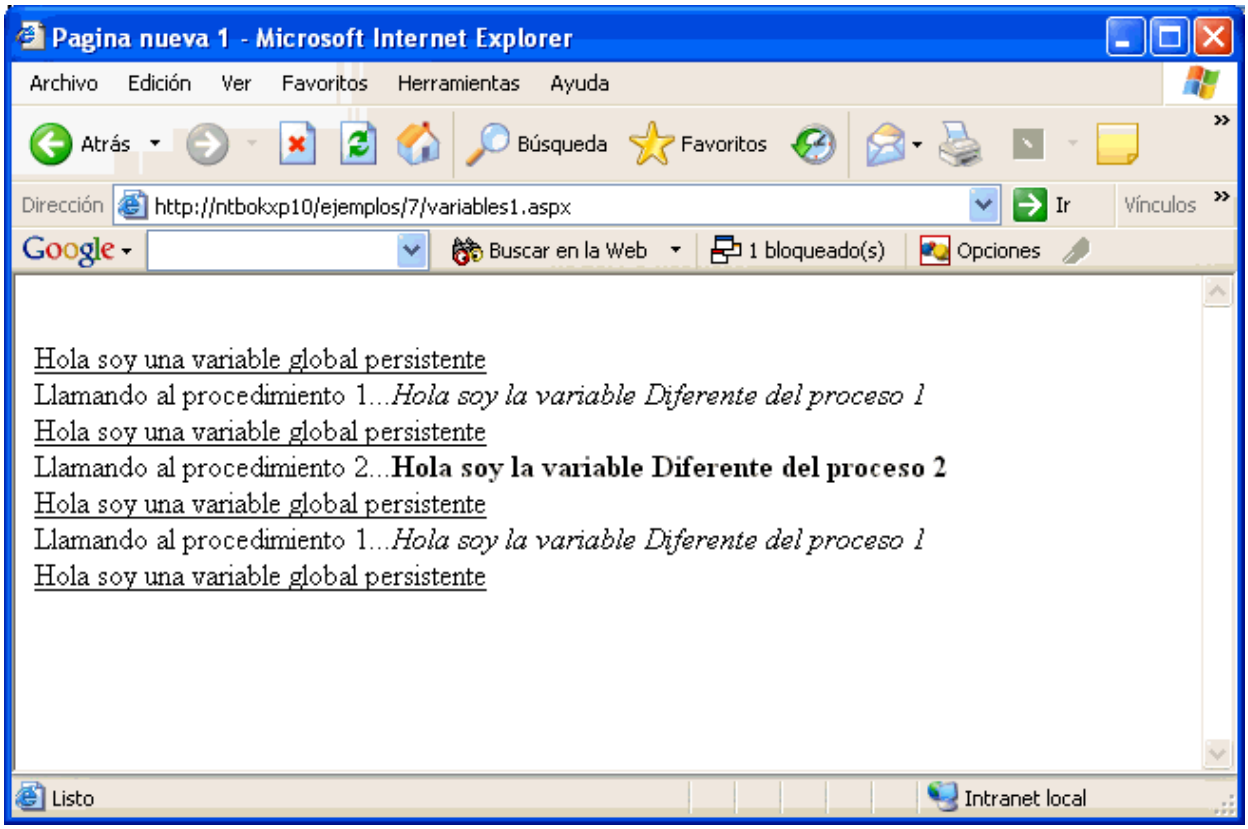


Aunque el gráfico corresponde a un programa de Windows de Visual Basic.NET no difiere mucho de nuestra filosofía. Por un lado en la parte del "Programa" tenemos una zona de declaraciones, todas las variables que declaremos serán visible por todos los formularios. Esa es nuestra idea, las variable que pongamos en la parte superior de la página se comportarán como globales. Hagamos un ejemplo:

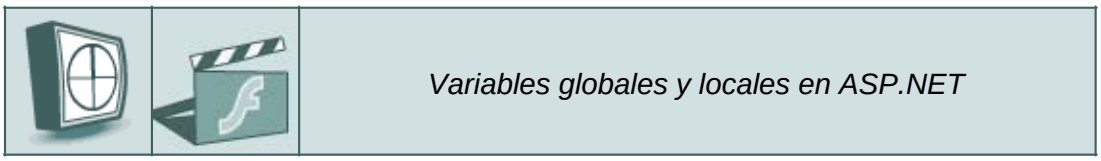
1. Copia la página anterior, llámala variables1.aspx e introduce estos cambios:

```
<script runat="server">
Dim mivariableglobal as string="<br><u>Hola soy una variable global persistente</u>"

sub page_load()
    Response.write (mivariableglobal)
    Response.write ("<BR />Llamando al procedimiento 1...")
    proceso_1
    Response.write ("<BR />Llamando al procedimiento 2...")
    proceso_2
    Response.write ("<BR />Llamando al procedimiento 1...")
    proceso_1
end sub

sub proceso_1
    dim diferente as string
    diferente="<i>Hola soy la variable Diferente del proceso 1</i>"
    response.write (Diferente)
    Response.write (mivariableglobal)
end sub

sub proceso_2
    dim diferente as string
    diferente="<b>Hola soy la variable Diferente del proceso 2</b>"
    response.write (Diferente)
    Response.write (mivariableglobal)
end sub
</script>
```

2. Ejecuta la página:



Puedes ver las diferencias de las variables ya que "mivariableglobal" estará visible para toda la página y las otras serán locales en sus procedimientos.



Con esta parte ya hemos dado otro paso importante en nuestro aprendizaje de Visual Basic. Ahora nos queda el último paso que es aprender la programación orientada a eventos. Una vez vista ya podemos pasar al manejo de las bases de datos y hacer páginas mas complejas. Paciencia, ASP.NET es un mundo impresionante y necesita su tiempo de aprendizaje.

5.5 Métodos sobrecargados

La sobrecarga de métodos es una de las particularidades de la programación orientada a objetos, en inglés se dice "OverLoading" y es un concepto muy sencillo y potente que vamos a ver ahora. Esta técnica permite realizar varias versiones distintas de un mismo método. Por ejemplo, puedes hacer una consulta a una base de datos que devuelve una matriz de objetos de productos que representan registros o filas de la base de datos. Puesto que queremos crear tres funciones distintas para obtener distintas consultas (para obtener todos los productos, categorías de los productos y Productos en stock) podíamos crear solo una que trabaje con las tres operaciones. Cada método tendrá el mismo nombre pero distinta declaración, dependiendo del número de parámetros necesarios. Además pondremos la palabra clave "overload" para que se sepa que esos métodos están "conectados" y se utilizarán según el número o tipo de parámetros.

No es tan complicado como parece, veamos un ejemplo con dos versiones sobrecargadas del método ObtenerPrecio_del_producto()

```
Private Overload Function ObtenerPrecio_del_producto (id as Integer) as Decimal

    'Código...

End function

Private Overload Function ObtenerPrecio_del_producto (nombre as String) as Decimal

    'Código...

End function
```

Nota Fíjate que se están declarando las funciones como de ámbito local: Private Overload Function: función privada sobrecargada.

Ahora dependiendo de si hacemos la llamada a esta función con un valor de tipo string (nombre) o integer (id) se ejecutará una función u otra. Así que no hay que crear distintas funciones con distintos nombres, solo con llamarla con el tipo de datos que queremos trabajar ejecutará una función u otra.

```
Dim precio as Decimal

'Obtiene el precio de un producto por su numero de identificación ID

Precio=ObtenerPrecio_del_Producto (1001)


'Obtiene el precio de un producto por su nombre

Precio=ObtenerPrecio_del_Producto ("Reproductor DVD")
```

Obviamente no podremos sobrecargar funciones que tengan el mismo número y tipo de parámetros porque el CLR (common language runtime) no sabría cual debería ejecutar.

Los métodos sobrecargados se utilizan mucho, prácticamente en todas las clases de .NET existen métodos así. Los veremos cuando realicemos nuestros programas ya que la ayuda del entorno de desarrollo nos indicará cuando utilicemos una función que ésta está sobrecargada y así la podremos llamar con los parámetros que necesitemos en ese momento.

5.6 Delegación

La delegación permite crear variables que apunten a métodos y así podemos utilizar esta variables en cualquier momento para invocar o llamar a un método. Estos conceptos de sobrecarga y delegación son íntimos de la programación orientada a objetos y .NET por lo que si conocías los anteriores mundos del ASP (no ASP.NET) y Visual Basic 6.0 todo esto será bastante novedoso.

El primer paso para crear un "delegate" es definir una forma o "signature", esta forma es una combinación de varias partes: el tipo devuelto, número de parámetros y el tipo de datos de cada parámetro. Una variable puede apuntar solo a un método que coincida con una "signature" específica, es decir, el método debe devolver el mismo tipo de datos, el mismo número de parámetros y si tipo de datos para cada parámetro.

Por ejemplo, si tenemos un método que acepta un único parámetro de tipo string y otro que acepta dos necesitaremos crear tipos "delegate" distintos para cada método. Veamos un ejemplo:

```
Private Function TraduceIngles_a_Español (ingles as string) as string

    'código....

End Function
```

Esta función acepta un único parámetro de tipo string y devuelve otro. Teniendo en cuenta estos dos detalles podemos definir un tipo "delegate" que coincida con estos tipos:

```
Private Delegate Function Funcion_cadena (texto_entrada as string) as string
```

Fíjate que el nombre que hemos elegido no coincide ya que no es necesario pero si el tipo de datos devuelto y el de los parámetros de entrada. Una vez creado el tipo "delegate" podemos crear y asignar variables de este tipo en cualquier momento. Podemos crear por ejemplo:

```
Dim Mifuncion as Funcion_cadena
```

Y ahora empieza lo bueno, utilizando una variable delegada (delegate) podemos apuntar a cualquier método que coincida con su formato (signature). En el ejemplo, Funcion_cadena necesita un parámetro de entrada y devuelve otro. Por tanto podremos utilizar la variable Mifuncion para almacenar una referencia a la función TraduceIngles_a_Español. Para actualizar y almacenar esta referencia a la función TraduceIngles_a_Español utilizaremos el operador "AddressOf" de esta forma:

```
Mifuncion= AddressOf TraduceIngles_a_Español
```

Ahora que tenemos una variable delgada que hace referencia a una función podremos llamar a la función a través de su delegada, de esta forma:

```
Dim Cadena_Español as String

Cadena_Español=Mifuncion ("Hello")
```

En este ejemplo el método que apunta a la variable delegada Mifuncion se llama com el texto "Hello" y el valor devuelto se almacena en la variable Cadena_Español.

¿Complicado? Pues si, para que nos vamos a engañar. Esto lo utilizaremos realmente poco pero debemos conocer que se puede realizar esta serie de operaciones. De todas formas mas adelante haremos ejemplos mas concretos que nos ayudarán a entender este interesante sistema de asignar una función a una variable.

6 Prioridad de los operadores

Esto ya lo hemos comentado antes pero vamos a ampliarlo un poco mas dada su importancia al realizar operaciones. 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. Como hay varios tipos de operadores: aritméticos, de comparación y lógicos. Veamos cada grupo por separado lo separamos en tres grupos, algunos no los hemos visto todavía pero caerán enseguida. 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.

Pulsa aquí para descargar los ejemplos de este tema

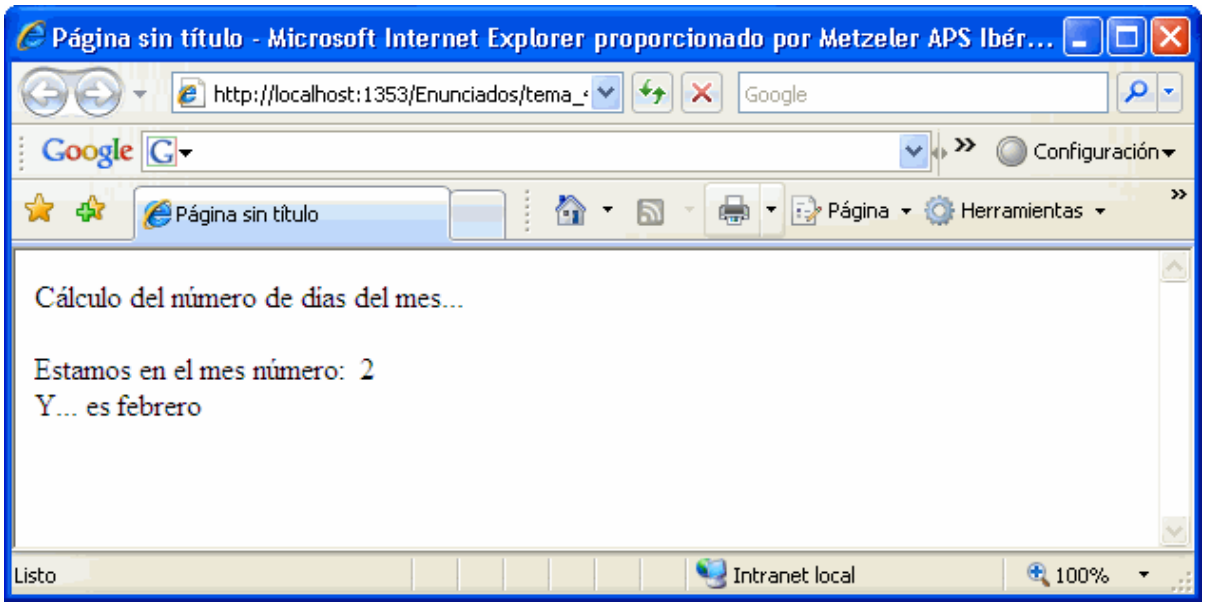
Ejercicios

Ejercicio 1

Crea una página con una secuencia "select case" para que según el mes actual nos diga:

"Este mes tiene 30 días"
"Este mes tiene 28 ó 29 días" Para el caso de febrero, nos da igual que sea bisiesto, no haremos ese cálculo.
"Este mes tiene 31 días"

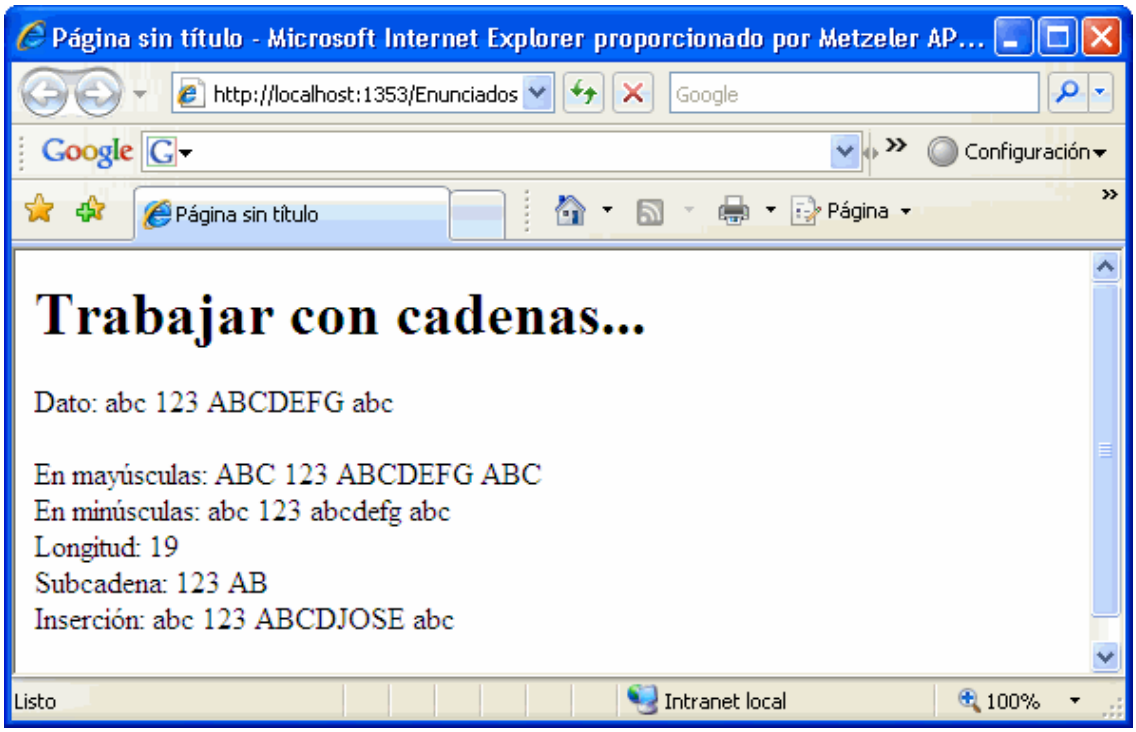
El número del mes lo obtenemos con la función "Now.Month()"



Ejercicio 2

Dada una varaible de tipo string con el valor "abc 123 ABCDEFG abc" realiza las operaciones siguientes:

- Escribir la variable en mayúsculas
- Escribir la variable en minúsculas
- Escribir su longitud
- Escribir la subcadena resultante de extraer los 6 caracteres siguientes a la 4 posición
- Sustituye "EFG" por "JOSE"


Página sin título - Microsoft Internet Explorer proporcionado por Metzeler AP...
http://localhost:1353/Enunciados
Google
Configuración
Página sin título
Página
Trabajar con cadenas...
Dato: abc 123 ABCDEFG abc
En mayúsculas: ABC 123 ABCDEFG ABC
En minúsculas: abc 123 abcdefg abc
Longitud: 19
Subcadena: 123 AB
Inserción: abc 123 ABCDJOSE abc
Listo
Intranet local
100%