Constantes, Variables y Operadores de Visual Basic

1.- Introducción.

En este tema aprenderemos a utilizar variables para almacenar temporalmente datos en su programa y a emplear operadores matemáticos, relacionales y lógicos. También comenzaremos a utilizar funciones matemáticas con las que podremos realizar cálculos con números

2.- Constantes

Al igual que las variables, una constante es un elemento del lenguaje que guarda un valor, pero que en este caso y como su propio nombre indica, dicho valor será permanente a lo largo de la ejecución del programa, no pudiendo ser modificado.

Para declarar una constante, debemos utilizar la palabra clave **Const**, debiendo al mismo tiempo establecer el tipo de dato y asignarle valor.

```
Sub Main()
    Const Color As String = "Azul"
    Const ValorMoneda As Double = 120.48
End Sub
```

Si intentamos asignar un valor a una constante después de su asignación inicial, el IDE nos subrayará la línea con un aviso de error de escritura, y se producirá igualmente un error si intentamos ejecutar el programa.

```
Sub Main()
    Const Color As String = "Azul"
    Const ValorMoneda As Double = 120.48

    Color = "Verde"
            Una constante no puede ser el destino de una asignación.
End Sub
```

La ventaja del uso de constantes reside en que podemos tener un valor asociado a una constante, a lo largo de nuestro código para efectuar diversas operaciones. Si por cualquier circunstancia, dicho valor debe cambiarse, sólo tendremos que hacerlo en el lugar donde declaramos la constante.

Supongamos como ejemplo, que hemos escrito un programa en el que se realiza una venta de productos y se confeccionan facturas. En ambas situaciones debemos aplicar un descuento sobre el total resultante.

```
Sub Main()
    ' venta de productos
    Dim Importe As Double
    Dim TotalVenta As Double
    Console.WriteLine("Introducir importe de la venta")
    Importe = Console.ReadLine()

    ' aplicar descuento sobre la venta
    TotalVenta = Importe - 100
    Console.WriteLine("El importe de la venta es: {0}", TotalVenta)
    Console.WriteLine()
    ' ............
    ' ............
    ' ............

    ' factura de mercancías
    Dim PrecioArt As Double
    Dim TotalFactura As Double
    Console.WriteLine("Introducir precio del artículo")
    PrecioArt = Console.ReadLine()

    ' aplicar descuento a la factura
    TotalFactura = PrecioArt - 100
    Console.WriteLine("El total de la factura es: {0}", TotalFactura)
    Console.WriteLine()
    ' ............
    ' ............
    ' ............
    Console.ReadLine()
End Sub
```

En el anterior ejemplo, realizamos el descuento utilizando directamente el valor a descontar. Si en un momento dado, necesitamos cambiar dicho valor de descuento, tendremos que recorrer todo el código e ir cambiando en aquellos lugares donde se realice esta operación.

Empleando una constante para el descuento, y utilizando dicha constante en todos aquellos puntos del código en donde necesitemos aplicar un descuento, cuando debamos modificar el descuento, sólo necesitaremos hacerlo en la línea en la que declaramos la constante.

```
Sub Main()
    ' crear constante para calcular descuento
    Const DESCUENTO As Integer = 100

    ' venta de productos
    Dim Importe As Double
    Dim TotalVenta As Double
    Console.WriteLine("Introducir importe de la venta")
    Importe = Console.ReadLine()

    ' aplicar descuento sobre la venta, atención al uso de la constante
    TotalVenta = Importe - DESCUENTO
    Console.WriteLine("El importe de la venta es: {0}", TotalVenta)
    Console.WriteLine()
    ' ............
    ' ............
    ' ............

    ' factura de mercancías
    Dim PrecioArt As Double
    Dim TotalFactura As Double
    Console.WriteLine("Introducir precio del artículo")
    PrecioArt = Console.ReadLine()

    ' aplicar descuento a la factura, atención al uso de la constante
    TotalFactura = PrecioArt - DESCUENTO
    Console.WriteLine("El total de la factura es: {0}", TotalFactura)
    Console.WriteLine()
    ' ............
    ' ............
    ' ............
    Console.ReadLine()
End Sub
```

3.- Variables.

Una variable es un identificador del programa que guarda un valor que puede ser modificando durante el transcurso de dicha aplicación.

DECLARACIÓN

La declaración de una variable es el proceso por el cual comunicamos al compilador que vamos a crear una nueva variable en el programa.

Para declarar una variable utilizaremos la palabra clave **Dim**, seguida del identificador o nombre que daremos a dicha variable.

```
Sub Main()
    Dim MiValor
End Sub
```

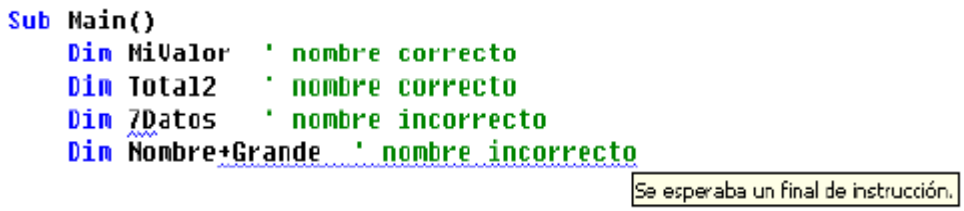
DENOMINACIÓN

Respecto al nombre de la variable, debe empezar por letra, y no puede ser ninguna de las palabras reservadas del lenguaje, ni contener caracteres como operadores u otros símbolos especiales.

```
Sub Main()
    Dim MiValor  ' nombre correcto
    Dim Total2   ' nombre correcto
    Dim Mis_Datos  ' nombre correcto
    Dim 7Datos   ' nombre incorrecto
    Dim Nombre+Grande  ' nombre incorrecto
    Dim End  ' nombre incorrecto
End Sub
```

AVISOS DEL IDE SOBRE ERRORES EN EL CÓDIGO

Al declarar una variable con un nombre incorrecto, o si se produce otro tipo de error en la escritura del código, el propio IDE se encarga de avisarnos que existe un problema subrayando el fragmento de código conflictivo y mostrando una viñeta informativa al situar sobre dicho código el cursor.



Estos avisos constituyen una gran ayuda, ya que permiten al programador observar problemas en la escritura del código, antes incluso de ejecutar el programa.

Existen multitud de avisos de muy diversa naturaleza, teniendo en cuenta que la tónica general consiste en que el código problemático quedará subrayado por el IDE hasta que no modifiquemos la línea en cuestión y la escribamos correctamente.

LUGAR DE LA DECLARACIÓN

Podemos declarar variables en muy diversos lugares del código. El punto en el que declaremos una variable será determinante a la hora del ámbito o accesibilidad a esa variable desde otros puntos del programa. Por ahora, y ciñéndonos a la declaración de variables dentro de procedimientos, recomendamos declarar todas las variables en la cabecera o comienzo del procedimiento, para dar una mayor claridad al mismo. Después de la declaración, escribiríamos el resto de instrucciones del procedimiento.

TIPIFICACIÓN

La tipificación de una variable es la operación por la cual, al declarar una variable, especificamos qué clase de valores o tipo de datos vamos a poder almacenar en dicha variable.

En VB.NET utilizamos la palabra clave **As** seguida del nombre del tipo de datos, para establecer el tipo de una variable.

```
Sub Main()
    Dim Valor As String   ' cadena de caracteres
    Dim Cuenta As Integer ' numérico entero
    Dim FhActual As Date  ' fecha
End Sub
```

La siguiente tabla muestra la relación de tipos de datos disponibles en el lenguaje.

Tipo de dato en VB.NET	Tipo correspondiente en el entorno de .NET Framework	Tamaño	Rango de valores
Boolean	System.Boolean	1 byte	True o False (Verdadero o Falso)
Byte	System.Byte	1 byte	0 a 255 (sin signo)
Char	System.Char	2 bytes	0 a 65535 (sin signo)
Date	System.DateTime	8 bytes	1 de enero de 1 a 31 de diciembre de 9999
Decimal	System.Decimal	12 bytes	+/- 79.228.162.514.264.337.593.543.950.335 sin punto decimal; +/-7,9228162514264337593543950335 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es +/-0,0000000000000000000000000001
Double (punto flotante con precisión doble)	System.Double	8 bytes	-1,79769313486232E308 a -4,94065645841247E-324 para valores negativos; 4,94065645841247E-324 a 1,79769313486232E308 para valores positivos
Integer	System.Int32	4 bytes	-2.147.483.648 a 2.147.483.647

Long (entero largo)	System.Int64	8 bytes	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
Short	System.Int16	2 bytes	-32.768 a 32.767
Single (punto flotante con precisión simple)	System.Single	4 bytes	-3,402823E38 a –1,401298E-45 para valores negativos; 1,401298E-45 a 3,402823E38 para valores positivos
Object	System.Object	4 bytes	Cualquier tipo
String (cadena de longitud variable)	System.String	10 bytes + (2 * longitud de la cadena)	Desde 0 a unos 2.000 millones de caracteres Unicode
Estructura (tipo de dato definido por el usuario)	Hereda de System.ValueType	Suma de los tamaños de los miembros de la estructura	Cada miembro de la estructura tiene un intervalo de valores determinado por su tipo de datos e independiente de los intervalos de valores correspondientes a los demás miembros

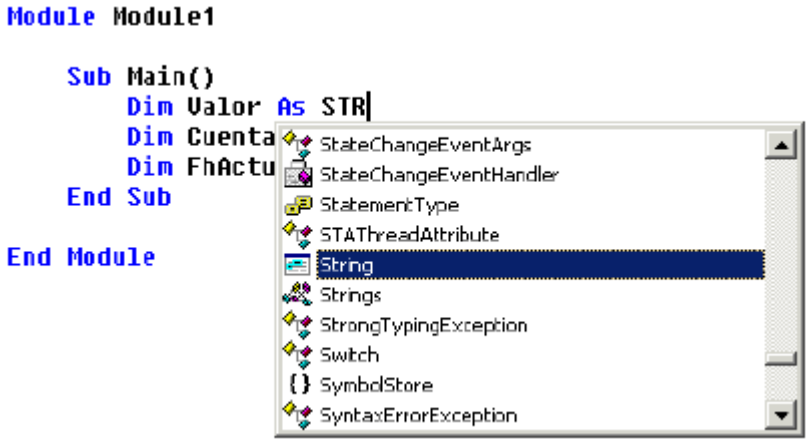
Si al declarar una variable no indicamos el tipo, por defecto tomará **Object**, que corresponde al tipo de datos genérico en el entorno del CLR, y admite cualquier valor.

Según la información que acabamos de ver, si declaramos una variable de tipo **Byte** e intentamos asignarle el valor 5899 se va a producir un error, ya que no se encuentra en el intervalo de valores permitidos para esa variable. Esto puede llevar al lector a preguntar: "¿por qué no utilizar siempre **Object** y poder usar cualquier valor?, o mejor ¿para qué necesitamos asignar tipo a las variables?".

El motivo de tipificar las variables reside en que cuando realizamos una declaración, el CLR debe reservar espacio en la memoria para los valores que pueda tomar la variable, como puede ver el lector en la tabla anterior, no requiere el mismo espacio en memoria una variable **Byte** que una **Date**. Si además, declaramos todas las variables como **Object**, los gastos de recursos del sistema serán mayores que si establecemos el tipo adecuado para cada una, ya que como el CLR no sabe el valor que puede tomar en cada ocasión la variable, debe realizar un trabajo extra de adecuación, consumiendo una mayor cantidad de recursos.

Una correcta tipificación de las variables redundará en un mejor aprovechamiento de las capacidades del sistema y en un código más veloz en ejecución. Cuantos más programas se diseñen optimizando en este sentido, el sistema operativo ganará en rendimiento beneficiándose el conjunto de aplicaciones que estén en ejecución.

VS.NET dispone de una ayuda al asignar el tipo a una variable, que nos muestra la lista de tipos disponibles para poder seleccionar uno sin tener que escribir nosotros el nombre. Al terminar de escribir la palabra **As**, aparecerá dicha lista, en la que pulsando las primeras letras del tipo a buscar, se irá situando en los más parecidos. Una vez encontrado, pulsaremos la tecla Enter o Tab para tomarlo.



DECLARACIÓN MÚLTIPLE EN LÍNEA

En el caso de que tengamos que declarar más de una variable del mismo tipo, podemos declararlas todas en la misma línea, separando cada una con una coma e indicando al final de la lista el tipo de dato que van a tener.

```
Dim Nombre, Apellidos, Ciudad As String
```

Con esta técnica de declaración, todas las variables de la línea tienen el mismo tipo de dato, ya que no es posible declarar múltiples variables en la misma línea que tengan distintos tipos de dato.

ASIGNACIÓN DE VALOR

Para asignar un valor a una variable utilizaremos el operador de asignación: el signo igual (=), situando a su izquierda la variable a asignar, y a su derecha el valor.

```
Dim Cuenta As Integer
Cuenta = 875
```

Según el tipo de dato de la variable, puede ser necesario el uso de delimitadores para encerrar el valor que vamos a asignar:

- **Tipos numéricos**. Las variables de tipos de datos numéricos no necesitan delimitadores, se asigna directamente el número correspondiente. Si necesitamos especificar decimales, utilizaremos el punto (.) como carácter separador para los decimales.

- **Cadenas de caracteres**. En este caso es preciso encerrar la cadena entre comillas dobles (").

- **Fechas**. Al asignar una fecha a una variable de este tipo, podemos encerrar dicho valor entre el signo de almohadilla (#) o comillas dobles ("). El formato de fecha a utilizar depende del delimitador. Cuando usemos almohadilla la fecha tendrá el formato Mes/Día/Año; mientras que cuando usemos comillas dobles el formato será Día/Mes/Año.

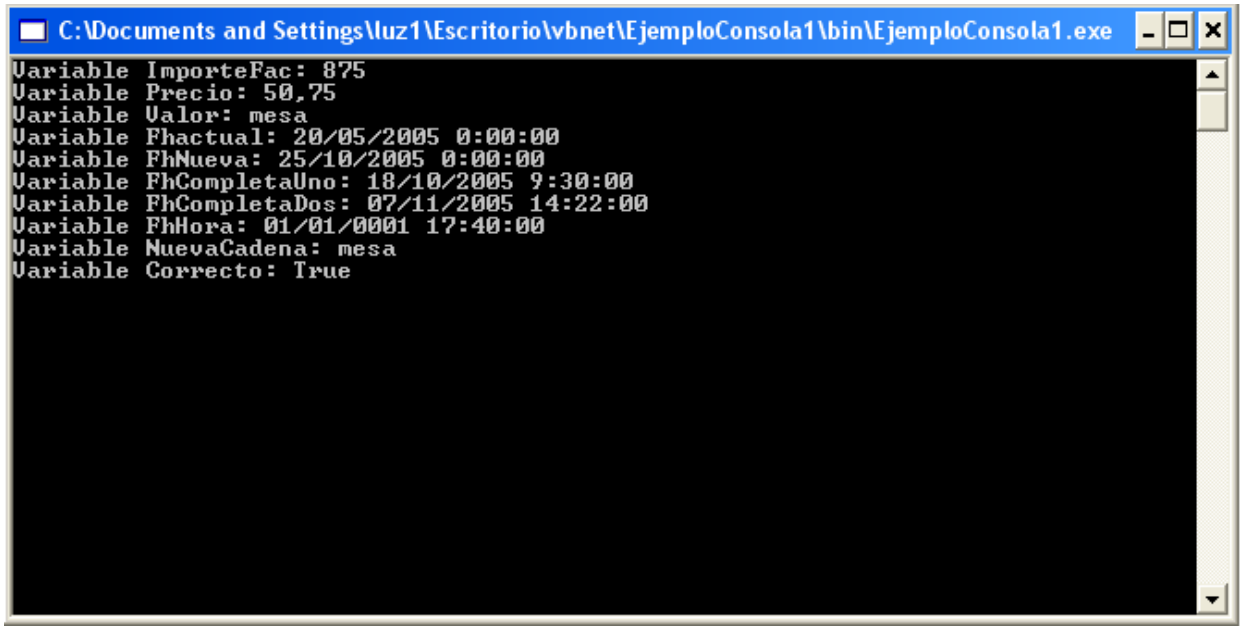
Las fechas pueden contener además información horario que especificaremos en el formato Hora:Minutos:Segundos FranjaHoraria. En el caso de que no indiquemos la franja horaria (AM/PM) y si estamos utilizando el signo almohadilla como separador, el entorno insertará automáticamente los caracteres de franja horaria correspondientes.

- **Tipos lógicos**. Las variables de este tipo sólo pueden tener el valor True (Verdadero) o False (Falso).

Además de asignar valores como acabamos de explicar, podemos asignar el contenido de una variable a otra o el resultado de una expresión, como veremos más adelante en el apartado dedicado a operadores.

```
Module Module1

    Sub Main()
        Dim ImporteFac As Integer
        Dim Precio As Double
        Dim Valor As String
        Dim FhActual, FhNueva, FhCompletaUno, FhCompletaDos, FhHora As Date
        Dim Correcto As Boolean
        Dim NuevaCadena As String

        ImporteFac = 875
        Precio = 50.75
        Valor = "mesa"

        FhActual = #5/20/2005#  'mes/día/año
        FhNueva = "25/10/2005" 'día/mes/año
        FhCompletaUno = #10/18/2005 9:30:00 AM#
        FhCompletaDos = "7/11/2005 14:22:00"
        FhHora = #5:40:00 PM#

        NuevaCadena = Valor ' asignar una variable a otra


        Correcto = True

        'mostrar variables en la consola

        Console.WriteLine("Variable ImporteFac: {0}", ImporteFac)
        Console.WriteLine("Variable Precio: {0}", Precio)
        Console.WriteLine("Variable Valor: {0}", Valor)
        Console.WriteLine("Variable Fhactual: {0}", FhActual)
        Console.WriteLine("Variable FhNueva: {0}", FhNueva)
        Console.WriteLine("Variable FhCompletaUno: {0}", FhCompletaUno)
        Console.WriteLine("Variable FhCompletaDos: {0}", FhCompletaDos)
        Console.WriteLine("Variable FhHora: {0}", FhHora)
        Console.WriteLine("Variable NuevaCadena: {0}", NuevaCadena)
        Console.WriteLine("Variable Correcto: {0}", Correcto)

        Console.ReadLine()
    End Sub

End Module
```

El resultado será:



Otra cualidad destacable en este apartado de asignación de valores, reside en que podemos declarar una variable y asignarle valor en la misma línea de código.

```
Dim Valor As String = "mesa"
Dim ImporteFac As Integer = 875
```

VALOR INICIAL

Toda variable declarada toma un valor inicial por defecto, a no ser que realicemos una asignación de valor en el mismo momento de la declaración. A continuación se muestran algunos valores de inicio en función del tipo de dato que tenga la variable:

- **Numérico**. Cero (0).
- **Cadena de caracteres**. Cadena vacía ("").
- **Fecha**. 01/01/0001 0:00:00.
- **Lógico**. Falso (False).
- **Objeto**. Valor nulo (Nothing).

DECLARACIÓN OBLIGATORIA

Es obligatorio, por defecto, la declaración de todas las variables que vayamos a utilizar en el código. En el caso de que intentemos utilizar una variable no declarada, se producirá un error.

La declaración de variables proporciona una mayor claridad al código, ya que de esta forma, sabremos en todo momento si un determinado identificador corresponde a una variable de nuestro procedimiento, de un parámetro, etc.

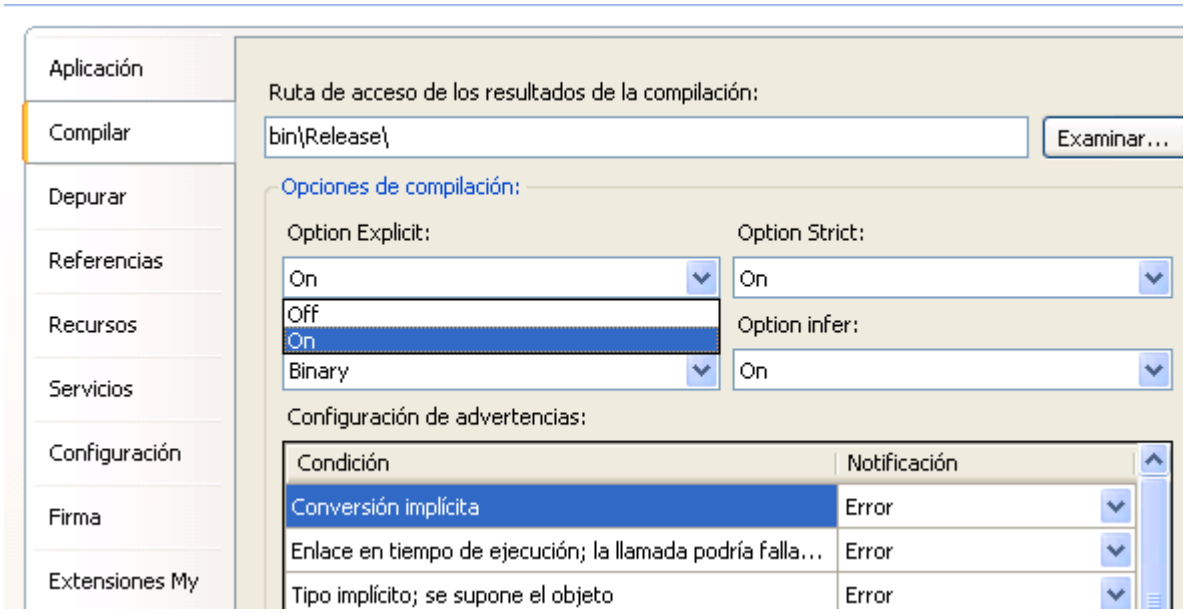
Mediante la instrucción **Option Explicit**, y sus modificadores **On/Off**, podemos requerir o no la declaración de variables dentro del programa.

- **Option Explicit On**. Hace obligatoria la declaración de variables. Opción por defecto.
- **Option Explicit Off**. Hace que no sea obligatoria la declaración de variables.

Podemos aplicar esta instrucción para que tenga efecto a nivel de proyecto y a nivel de fichero de código.

- **Option Explicit a nivel de proyecto**.

Para establecer **Option Explicit a nivel de proyecto**, debemos abrir **My Proyect**. Esto mostrará la ventana de propiedades del proyecto, en cuyo panel izquierdo haremos clic sobre el elemento **Compilar** Finalmente abriremos la lista desplegable del elemento **Option Explicit**, seleccionaremos un valor (On, Off) y pulsaremos Aplicar y Aceptar.



Con la declaración obligatoria desactivada podríamos escribir código como el mostrado:

```
Sub Main()
    Valor = "coche"
    MiDato = 984

    Console.WriteLine("Variable Valor: {0}", Valor)
    Console.WriteLine("Variable MiDato: {0}", MiDato)
    Console.ReadLine()
End Sub
```

En el ejemplo anterior, no hemos declarado las variables en Main( ). Al estar Option Explicit Off esto no produce error, y el CLR al detectar un identificador sin declarar, crea una nueva variable internamente.

Mucho más fácil que tener que declarar las variables ¿verdad?. Pues precisamente esta facilidad es uno de los graves problemas de no declarar variables. En un procedimiento de prueba con poco código, esto no supone una importante contrariedad. Sin embargo pensemos un momento, que en lugar de un pequeño procedimiento, se trata de una gran aplicación con muchas líneas de código, procedimientos, y cientos de variables. Al encontrarnos con una variable de esta forma, no sabremos si esa variable ya la hemos utilizado con anterioridad en el procedimiento, si ha sido pasada como parámetro al mismo, etc. Estas circunstancias provocan que nuestro código se vuelva complejo de interpretar, retrasando la escritura general de la aplicación. Si volvemos a activar **Option Explicit On**, inmediatamente sabremos que algo va mal, ya que toda variable no declarada, quedará subrayada por el IDE como un error de escritura. Las ventajas son evidentes.

OPTION EXPLICIT A NIVEL DE FICHERO.

Para establecer la declaración obligatoria a nivel de fichero, debemos situarnos al comienzo del fichero de código y escribir la instrucción **Option Explicit** con el modificador correspondiente.

```
' desactivar declaración obligatoria de variables
' ahora podemos, dentro de este fichero de código,
' escribir todas las variables sin declarar

Option Explicit Off

Module Module1

    Sub Main()
        Valor = "coche"
        MiDato = 984

        Console.WriteLine("Variable Valor: {0}", Valor)
        Console.WriteLine("Variable MiDato: {0}", MiDato)
        Console.ReadLine()
    End Sub

End Module
```

Option Explicit a nivel de fichero, nos permite establecer el modo de declaración de variables sólo para ese fichero en el que lo utilizamos, independientemente del tipo de obligatoriedad en declaración de variables establecido de forma general para el proyecto. Podemos por ejemplo, tener establecido **Option Explicit On** para todo el proyecto, mientras que para un fichero determinado podemos no obligar a declarar variables escribiendo al comienzo del mismo **Option Explicit Off**.

El hecho de tener **Option Explicit Off** no quiere decir que no podamos declarar variables, podemos, por supuesto declararlas, lo que sucede es que el compilador no generará un error al encontrar una variable sin declarar.

El otro grave problema al no declarar variables proviene por la incidencia en el rendimiento de la aplicación. Cuando tenemos **Option Explicit Off**, el CLR por cada identificador que encuentre sin declarar, crea una nueva variable, y ya que desconoce qué tipo de dato querría utilizar el programador, opta por asignarle el más genérico: Object.

Una excesiva e innecesaria proliferación de variables Object afectan al rendimiento del programa, ya que el CLR debe trabajar doblemente en la gestión de recursos utilizada por dichas variables. En el próximo apartado trataremos sobre la obligatoriedad a la hora de tipificar variables.

Por todo lo anteriormente comentado, a pesar de la engañosa facilidad y flexibilidad de **Option Explicit Off,** nuestra recomendación es tener configurado siempre **Option Explicit On** a nivel de aplicación, nos ahorrará una gran cantidad de problemas.

TIPIFICACIÓN OBLIGATORIA

Cuando declaramos una variable, no es obligatorio por defecto, establecer un tipo de dato para la misma. Igualmente, al asignar por ejemplo, una variable numérica a una de cadena, se realizan automáticamente las oportunas conversiones de tipos, para transformar el número en una cadena de caracteres.

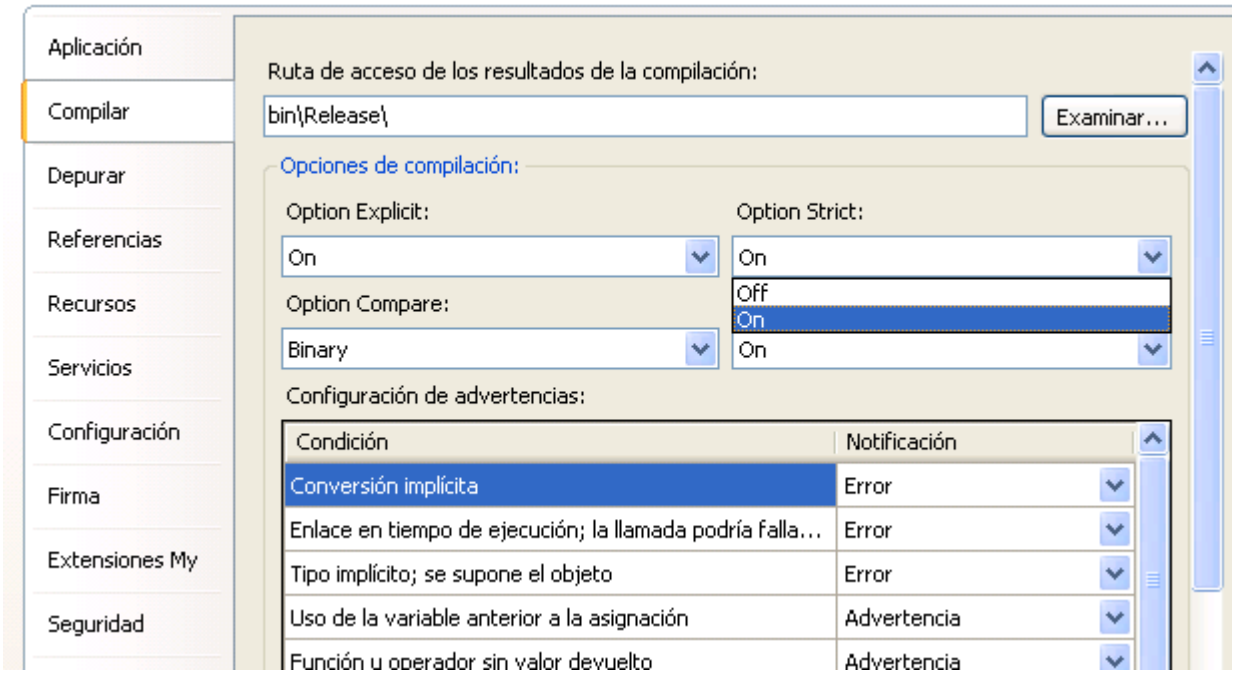
```
Sub Main()
    ' no es necesario tipificar la variable, tipificación implícita,
    ' la variable Valor se crea con el tipo Object
    Dim Valor

    ' tipificación explícita
    Dim Importe As Integer
    Dim UnaCadena As String

    ' al asignar una fecha a la variable Valor,
    ' sigue siendo de tipo Object, pero detecta que
    ' se trata de una fecha y guarda internamente
    ' esta información como un subtipo Date
    Valor = #8/20/2001#

    Importe = 590

    ' no es necesario hacer una conversión de tipos previa
    ' para asignar un número a una variable de cadena,
    ' ya que se realiza una conversión implícita,
    ' la variable UnaCadena contiene la cadena "590"
    UnaCadena = Importe

    Console.WriteLine("Variable Valor: {0}", Valor)
    Console.WriteLine("Variable Importe: {0}", Importe)
    Console.WriteLine("Variable UnaCadena: {0}", UnaCadena)
    Console.ReadLine()
End Sub
```

Como ya comentábamos en el apartado anterior, si no asignamos el tipo de dato adecuado al declarar una variable, el CLR le asigna el tipo Object, lo que afecta negativamente al rendimiento de la aplicación.

La instrucción **Option Strict**, junto a sus modificadores On/Off, nos permite establecer si en el momento de declarar variables, será obligatoria su tipificación. También supervisa la obligatoriedad de realizar una conversión de tipos al efectuar asignaciones entre variables, o de expresiones a variables.

- **Option Strict On**. Hace obligatoria la tipificación de variables y la conversión de tipos explícita.
- **Option Strict Off**. Hace que no sea obligatoria la tipificación de variables. La conversión de entre tipos distinta en asignaciones y expresiones es realizada automáticamente por el entorno. Opción por defecto.

Podemos configurar **Option Strict** a nivel de proyecto y de fichero de código, de igual forma que con **Option Explicit**. En el caso de configurar a nivel de proyecto, deberemos abrir la ventana de propiedades del proyecto, y en su apartado Compilar, establecer el valor correspondiente en la lista desplegable **Option Strict**.



Si configuramos a nivel de fichero de código, escribiremos esta instrucción en la cabecera del fichero con el modificador oportuno. Consulte el lector el anterior apartado para un mayor detalle sobre el acceso a esta ventana de propiedades del proyecto.

En el siguiente código fuente establecemos **Option Strict On** a nivel de fichero de código, y a partir de ese momento, no podremos asignar un tipo de dato **Double** a un **Integer**, o un valor numérico a una variable String, por exponer un par de casos de los más comunes. El código erróneo será marcado por el IDE como un error de sintaxis, e igualmente se producirá un error si intentamos ejecutar el programa.

```
Option Strict On
Module Module1
    Sub Main()
        ' ahora es obligatorio establecer
        ' el tipo de dato a todas las variables
        Dim Valor As Integer
        Dim TotalGeneral As Double
        Dim Dato As String
        TotalGeneral = 500
        Valor = TotalGeneral ' error, no se permite la conversión implícita
        Dato = TotalGeneral  ' error, no se permite la conversión implícita
    End Sub
End Module
```

Establecer **Option Strict On** requiere un mayor trabajo por parte del programador, ya que ha de ser más cuidadoso y escribir un código más correcto y preciso, lo cual es muy conveniente. Sin embargo, ya que la opción por defecto en este sentido es **Option Strict Off**, los ejemplos realizados a lo largo de este texto se ajustarán en este particular a dicha configuración, con ello ganamos en comodidad, ya que evitaremos la obligación de realizar conversiones de tipos en muy diversas situaciones.

4.- Ámbito de variables

El ámbito de variables consiste en la capacidad de acceso que tenemos hacia una variable, de forma que podamos obtener su valor, así como asignarlo. Para determinar su nivel de accesibilidad, aquí intervienen, además de los modificadores de ámbito, el lugar o nivel de emplazamiento de la variable dentro del código.

```
ModificadorÁmbito [Dim] NombreVariable As TipoDato
```

En función del punto de código en el que sea declarada una variable, podremos omitir el uso de la palabra clave **Dim** para realizar dicha declaración.

Ámbito a nivel de procedimiento

Una variable declarada dentro del cuerpo de un procedimiento se dice que tiene un ámbito local o a nivel de procedimiento, no pudiendo ser accedida por otro código que no sea el de dicho procedimiento.

```
Public Sub Main()
    ' declaramos una variable que tiene ámbito
    ' a nivel de este procedimiento
    Dim Nombre As String

    Nombre = "Hola"

    Console.WriteLine("Introducir un valor")
    Nombre &= " " & Console.ReadLine()
End Sub
```

```
Public Sub Manipular()
    ' si intentamos desde este procedimiento
    ' acceder a la variable Nombre del
    ' procedimiento Main(), se produce un error
    Nombre = "nuevo valor"
End Sub
```

En el ejemplo anterior, la variable Nombre puede ser manipulada dentro del procedimiento Main( ), y cualquier intento de acceder a ella desde otro procedimiento provocará un error.

Ámbito a nivel de bloque

Una variable declarada dentro de una estructura de control se dice que tiene ámbito local a nivel de bloque, siendo accesible sólo dentro del código que está contenido en la estructura.

```
Public Sub Main()
    ' variables con ámbito a nivel de procedimiento
    Dim MiNumero As Integer
    Dim Total As Integer

    Console.WriteLine("Introducir un número")
    MiNumero = Console.ReadLine()

    If MiNumero > 0 Then
        ' variable con un ámbito a nivel de bloque
        ' sólo es accesible dentro de esta estructura If
        Dim Calculo As Integer
        Console.WriteLine("Introducir otro número para sumar")
        Calculo = Console.ReadLine()

        MiNumero += Calculo
    End If

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

    ' error, la variable Calculo no es accesible desde aquí
    Total = 150 + Calculo

    Console.ReadLine()
End Sub
```

En este punto debemos aclarar que el ámbito dentro de un bloque se entiende como la parte de la estructura en la que ha sido declarada la variable. Por ejemplo, en una estructura If...End If con Else, si declaramos una variable a continuación de If, dicha variable no será accesible desde el bloque de código que hay a partir de Else.

```
If MiNumero > 0 Then
    ' variable con un ámbito a nivel de bloque
    ' sólo es accesible dentro de esta estructura If
    Dim Calculo As Integer
    ' ......

Else
    ' la variable Calculo no es accesible desde aquí
    ' ......
End If
```

Ámbito a nivel de módulo

Una variable declarada en la zona de declaraciones de un módulo, es decir, fuera de cualquier procedimiento, pero dentro de las palabras clave **Module**...**End Module**, y utilizando como palabra clave **Dim** o **Private**, se dice que tiene ámbito a nivel de módulo.

Aunque tanto Dim como Private son perfectamente válidas para declarar variables a nivel de módulo, se recomienda usar exclusivamente Private; de este modo facilitamos la lectura del código, reservando las declaraciones con Dim para las variables con ámbito de procedimiento, y las declaraciones con Private para el ámbito de módulo.

En el ejemplo declaramos la variable Nombre dentro del módulo, pero fuera de cualquiera de sus procedimientos, esto hace que sea accesible desde cualquiera de dichos procedimientos, pero no desde un procedimiento que se halle en otro módulo.

```
Module General

    'Dim Nombre As String  <--- esta declaración es perfectamente válida...

    Private Nombre As String  ' ...pero se recomienda declarar con Private

    Public Sub Main()

        Console.WriteLine("Procedimiento Main()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Main() es: {0}", Nombre)
        Manipular()
        MostrarValor()

        Console.ReadLine()
    End Sub

    Public Sub Manipular()
        Console.WriteLine("Procedimiento Manipular()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Manipular() es: {0}", Nombre)
    End Sub

End Module

Module Calculos
    Public Sub MostrarValor()
        ' error, no se puede acceder desde este módulo
        ' a la variable Nombre, que está declarada Private
        ' en el módulo General
        Console.WriteLine("Procedimiento MostrarValor()")
        Nombre = "Antonio"
        Console.WriteLine("Valor de la variable Nombre: {0}", Nombre)
    End Sub
End Module
```

Ámbito a nivel de proyecto

Una variable declarada en la zona de declaraciones de un módulo utilizando la palabra clave **Public**, se dice que tiene ámbito a nivel del proyecto, es decir, que es accesible por cualquier procedimiento de cualquier módulo que se encuentre dentro del proyecto.

Si tomamos el fuente anterior y declaramos como Public la variable Nombre, ahora sí podremos manipularla desde cualquier punto de la aplicación.

```
Module General

    ' esta variable será accesible
    ' desde cualquier lugar del proyecto
    Public Nombre As String

    Public Sub Main()
        Console.WriteLine("Procedimiento Main()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Main() es: {0}", Nombre)
        Manipular()
        MostrarValor()

        Console.ReadLine()
    End Sub

    Public Sub Manipular()
        Console.WriteLine("Procedimiento Manipular()")
        Console.WriteLine("Asignar valor a la variable")
        Nombre = Console.ReadLine()

        Console.WriteLine("El valor de la variable en Manipular() es: {0}", Nombre)
    End Sub

End Module

Module Calculos

    Public Sub MostrarValor()
        ' al haber declarado la variable Nombre
        ' como Public en el módulo General, podemos acceder a ella
        ' desde un módulo distinto al que se ha declarado
        Console.WriteLine("Procedimiento MostrarValor()")
        Nombre = "Antonio"
        Console.WriteLine("Valor de la variable Nombre: {0}", Nombre)
    End Sub
```

5.- Variables Static

Este tipo de variables se caracterizan por el hecho de que retienen su valor al finalizar el procedimiento en el que han sido declaradas. Se deben declarar utilizando la palabra clave Static, pudiendo opcionalmente omitir la palabra clave Dim.

```
Static [Dim] Importe As Integer
```

Cuando declaramos una variable normal dentro de un procedimiento, cada vez que llamamos al procedimiento, dicha variable es inicializada. El siguiente ejemplo en cada llamada al procedimiento, se inicializa la variable y le sumamos un número, por lo que la variable siempre muestra el mismo valor por la consola.

```
Public Sub Main()
    Verificar("Primera") ' en esta llamada se muestra 7
    Verificar("Segunda") ' en esta llamada se muestra 9
    Verificar("Tercera") ' en esta llamada se muestra 11
    Console.ReadLine()
End Sub

Public Sub Verificar(ByVal OrdenLlamada As String)
    ' declarar variable con el modificador Static,
    ' en la primera llamada toma el valor inicial de 5,
    ' las sucesivas llamadas no ejecutarán esta línea
    Static Dim Importe As Integer = 5

    Importe += 2
    Console.WriteLine("{0} llamada al procedimiento, la variable contiene {1}", _
        OrdenLlamada, Importe)
End Sub
```

Las variables Static por lo tanto, tienen un periodo de vida que abarca todo el tiempo de ejecución del programa, mientras que su ámbito es a nivel de procedimiento o bloque, ya que también pueden crearse dentro de una estructura de control.

6.- Tipos definidos por el usuario

Además de los tipos de datos que podríamos denominar intrínsecos, predefinidos en el lenguaje, VB .NET nos permite utilizar cualesquiera otros que hayamos podido definir. Siempre ciñéndonos a los tipos que almacenan valores, no referencias, es posible crear básicamente dos tipos de datos: enumeraciones y estructuras.

Tanto las enumeraciones como las estructuras son entidades derivadas de **System.ValueType**. Una enumeración puede definirse prácticamente en cualquier lugar, como las declaraciones de variables, mientras que las estructuras sólo pueden aparecer en módulos y clases, siendo tipos más complejos que las enumeraciones.

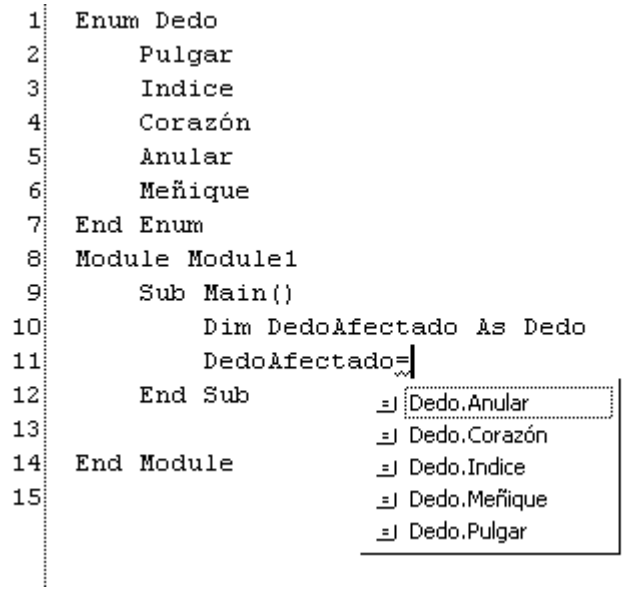
ENUMERACIONES

La mayoría de los tipos intrínsecos permiten que una variable tome un valor de un valor de una lista más o menos limitada. Una variable de tipo Byte, por ejemplo, puede contener un número entre 0 y 255, un Boolean sólo puede tener los valores True o False, y lo mismo ocurre con los tipos enteros y el tipo Char.

Siempre que necesitemos una variable que deba tomar un valor de una lista limitada, no existente en ninguno de los tipos intrínsecos ni enumeraciones predefinidas, podemos crear nuestra propia enumeración. Suponiendo que en una aplicación médica necesitemos variables que contenga n valores representando músculos, huesos y otros elementos, podríamos definir las enumeraciones apropiadas. Una de ellas podría ser la siguiente:

```
Enum Dedo
        Pulgar
        Indice
        Corazón
        Anular
        Meñique
End Enum
```

El identificador **Dedo** es, a partir de esta definición, un nuevo tipo de datos. Podemos utilizarlo como cualquier otro Integer, String, Byte, etc., para declarar variables. Las variables de tipo **Dedo** tan sólo podrán contener uno de los valores que aparecen en la enumeración. El editor de VB .NET, al efectuar una asignación de una variable de este tipo se ocupa de mostrar la lista de posibles valores:



Una enumeración debe tener un tipo de dato. Los tipos que podemos asignar a una enumeración deben ser los numéricos enteros soportados por el lenguaje que estemos utilizando. En el caso de VB.NET, los tipos de datos admisibles son Byte, Integer, Long y Short. En el caso de que no especifiquemos el tipo, tomará Integer por defecto.

El hecho de tipificar una enumeración está relacionado con los valores que podemos asignar a cada una de las constantes que contiene. De ello se deduce, que sólo vamos a poder asignar valores numéricos a estas constantes.

Cuando creamos una enumeración, si no asignamos valores a sus constantes, el entorno asigna automáticamente los valores, comenzando por cero y en incrementos de uno. Podemos en cualquier momento, asignar manualmente valores, no siendo obligatorio tener que asignar a todas las constantes. Cuando dejemos de asignar valores, el entorno seguirá asignando los valores utilizando como valor de continuación, el de la última constante asignada.

```
Public Enum Musicas As Integer
    Rock    ' 0
    Blues   ' 1
```

```
    NewAge   ' 2
    Funky    ' 3
End Enum

Public Enum DiasSemana As Integer
    Lunes             ' 0
    Martes            ' 1
    Miercoles = 278
    Jueves            ' 279
    Viernes           ' 280
    Sabado            ' 281
    Domingo           ' 282
End Enum
```

El valor almacenado en una variable de enumeración corresponderá al número de la constante que hayamos seleccionado. Al declarar la variable, su valor inicial será cero.

No obstante, la manipulación de una enumeración va mucho más allá de la asignación y recuperación simple de las constantes que componen la enumeración. Cuando declaramos una variable de una enumeración, el contenido real de dicha variable es un objeto de la clase Enum; por lo tanto, podemos utilizar los métodos de dicho objeto, para realizar diversas operaciones.

```
Module Module1
    Enum Dedo
        Pulgar = 1
        Indice
        Corazón
        Anular
        Meñique
    End Enum
    Sub Main()

        Dim MisDedos As Dedo
        Dim nombreDedo As String
        'Introduciremos un número. Si él número está entre 1 y 5 se visualizará el
nombre del dedo, sino se visualizará un texto de error
        Console.Write("Introduzca n° de Dedo (1-5): ")
        MisDedos = Console.ReadLine

        If System.Enum.IsDefined(MisDedos.GetType, MisDedos) Then
            Console.WriteLine("El dedo es el " & MisDedos.ToString)
        Else
            Console.WriteLine("El dedo no existe")
        End If

        Console.ReadKey()
        'Introduciremos el nombre del dedo. Si existe mostrará el mensaje de Dedo
existente y sino se visualizará un texto de error
        Console.Clear()
        Console.Write("Introduzca nombre de Dedo: ")
        nombreDedo = Console.ReadLine

        If System.Enum.IsDefined(MisDedos.GetType, nombreDedo) Then
            Console.WriteLine("El dedo existe")
        Else
            Console.WriteLine("El dedo no existe")
        End If
        Console.ReadKey()
    End Sub

End Module
```

ESTRUCTURAS

Una estructura consiste en un conjunto de datos que se unen para formar un tipo de dato compuesto. Este elemento del lenguaje se conocía en versiones anteriores de VB como tipo definido por el usuario (UDT o User Defined Type), y nos permite agrupar bajo un único identificador, una serie de datos relacionados.

Como novedad en VB.NET, los miembros de una estructura pueden ser, además de los propios campos que almacenan los valores, métodos que ejecuten operaciones, por lo cual, su aspecto y modo de manejo es muy parecido al de una clase.

Por ejemplo, si disponemos de la información bancaria de una persona, como pueda ser su código de cuenta, titular, saldo, etc., podemos manejar dichos datos mediante variables aisladas, o podemos crear una estructura que contenga toda esa información, simplificando la forma en que accedemos a tales datos.

Para crear una estructura, tenemos que utilizar la palabra clave **Structure** junto al nombre de la estructura, situando a continuación los miembros de la estructura, y finalizándola con la palabra clave **End Structure**.

```
Public Structure DatosBanco
    Public IDCuenta As Integer
    Public Titular As String
    Public Saldo As Integer
End Structure
```

El modo de utilizar una estructura desde el código cliente, consiste en declarar una variable del tipo correspondiente a la estructura, y manipular sus miembros de forma similar a un objeto. En el siguiente código fuente, manejamos de esta forma, una variable de la estructura DatosBanco.

```
Sub Main()
    Dim lDBanco As DatosBanco
    lDBanco.IDCuenta = 958
    lDBanco.Titular = "Carlos Perea"
    lDBanco.Saldo = 900
End Sub
```

Una estructura admite también métodos y propiedades, de instancia y compartidos, al estilo de una clase. Por lo que podemos añadir este tipo de elementos a nuestra estructura, para dotarla de mayor funcionalidad.

```
Module Module1
    Private Structure DatosBanco
        Dim IdCuenta As Integer
        Dim Titular As String
        Dim Saldo As Integer

        'crear un método a la estructura
        Public Sub Informacion()
            Console.Clear()
            Console.WriteLine("El número de cta es: " & IdCuenta)
            Console.WriteLine("Titular: " & Titular)
```

```
            Console.WriteLine("Saldo: " & Saldo)
            Console.ReadKey() ' pausa
        End Sub

    End Structure
    Sub main()
        Dim lDBanco As DatosBanco
        'introducir datos a la variable de tipo DatosBanco
        Console.Write("Introduzca Num. Cta: ")
        lDBanco.IdCuenta = Console.ReadLine
        Console.Write("Introduzca Titular: ")
        lDBanco.Titular = Console.ReadLine
        Console.Write("Introduzca Saldo: ")
        lDBanco.Saldo = Console.ReadLine

        Console.ReadKey()

        'llamar al método Información de la estructura
        lDBanco.Informacion()
    End Sub
```

La estructura del sistema DateTime

El entorno de .NET Framework proporciona, al igual que ocurre con las clases, una serie de estructuras del sistema, con funcionalidades diseñadas para ayudar al programador en las más variadas situaciones.

Como ejemplo de este tipo de estructura encontramos a DateTime, en la que a través de sus miembros compartidos y de instancia, nos provee de diversas operaciones para el manejo de fechas.

```
Module Module1
    Sub Main()
        ' ejemplos con la estructura DateTime
        ' ====================================
        ' miembros compartidos
        Dim ldtFechaActual As Date
        Dim ldtFechaA, ldtFechaB As Date

        ' la propiedad Today devuelve la fecha actual
        ldtFechaActual = DateTime.Today
        Console.WriteLine("La fecha de hoy es {0}", ldtFechaActual)

        ' el método DaysInMonth() devuelve el número
        ' de días que tiene un mes
        Console.WriteLine("El mes de Febrero de 2002 tiene {0} días", _
            DateTime.DaysInMonth(2002, 2))

        ' el método Compare() compara dos fechas
        Console.WriteLine("Introducir primera fecha")
        ldtFechaA = Console.ReadLine()
        Console.WriteLine("Introducir segunda fecha")
        ldtFechaB = Console.ReadLine()

        Select Case DateTime.Compare(ldtFechaA, ldtFechaB)
            Case -1
                Console.WriteLine("La primera fecha es menor")
            Case 0
                Console.WriteLine("Las fechas son iguales")
            Case 1
                Console.WriteLine("La primera fecha es mayor")
        End Select

        ' miembros de instancia
        Dim loMiFecha As DateTime
        Dim ldtFDias As Date
        Dim ldtFMeses As Date
        Dim lsFHFormateada As String
```

```
        ' usar el constructor de la estructura
        ' para crear una fecha
        loMiFecha = New DateTime(2002, 5, 12)
        ' agregar días a la fecha
        ldtFDias = loMiFecha.AddDays(36)
        ' restar meses a la fecha
        ldtFMeses = loMiFecha.AddMonths(-7)
        ' formatear la fecha
        lsFHFormateada = loMiFecha.ToLongDateString()

        Console.WriteLine("Uso de métodos de instancia de DateTime")
        Console.WriteLine("Fecha creada: {0} - Agregar días: {1}" & _
            " - Restar meses: {2} - Fecha formateada: {3}", _
            loMiFecha, ldtFDias, ldtFMeses, lsFHFormateada)
        Console.ReadLine()
    End Sub
End Module
```

6.- Operadores del Lenguaje

Los operadores son aquellos elementos del lenguaje que nos permiten combinar variables, constantes, valores literales, instrucciones, etc., para obtener un valor numérico, lógico, de cadena, etc., como resultado.

La combinación de operadores con variables, instrucciones, etc., se denomina expresión, mientras que a los elementos integrantes de una expresión y que no son operadores, se les denomina operandos.

En función de la complejidad de la operación a realizar, o del tipo de operador utilizado, una expresión puede ser manipulada a su vez como un operando dentro de otra expresión de mayor nivel. Los operadores se clasifican en las categorías detalladas a continuación, según el tipo de expresión a construir.

Visual Basic dispone de los siguientes operadores:

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
\	División Entera (número entero)
Mod	Resto de la división entera.
^	Exponenciación (elevar a una potencia)
&	Concatenación de cadenas (combinación)

Suma (+)

En función del tipo de dato de los operandos, este operador realiza una suma de números o una concatenación de cadenas de caracteres. Puede producirse un error dependiendo del tipo de dato del operando y la configuración de **Option Strict**. El siguiente ejemplo muestra algunos ejemplos de suma y concatenación, con la instrucción **Option Strict Off.**

```
Sub Main()
    Dim Resultado As Double
    Dim Cadena As String
    Dim Valor As Integer
    Dim Nombre As String
    Dim CadenaResulta As String

    ' suma de números
    Resultado = 12 + 7              ' devuelve: 19
    Resultado = 450 + 130           ' devuelve: 580

    ' concatenación de cadenas
    Cadena = "hola " + "amigos"     ' devuelve: "hola amigos"

    ' suma de variables
    Cadena = "15"
    Valor = 20
    CadenaResulta = Cadena + Valor  ' devuelve: "35"

    ' operaciones incorrectas
    Valor = 25
    Nombre = "Alfredo"

    CadenaResulta = Valor + Nombre ' error
    Resultado = Valor + Nombre  ' error
End Sub
```

Si cambiamos a continuación la configuración a **Option Strict On**, la siguiente operación que antes se ejecutaba, ahora provocará un error.

```
' suma de variables
Cadena = "15"
Valor = 20
CadenaResulta = Cadena + Valor   ' error
```

Para solucionar el problema debemos convertir explícitamente todos los operandos al mismo tipo de datos. Observe que en esta situación, no se realiza una suma, sino una concatenación.

```
' suma de variables
Cadena = "15"
Valor = 20
CadenaResulta = Cadena + CStr(Valor)   ' devuelve: "1520"
```

A pesar de que el operador + permite concatenar tipos String, se recomienda el uso del operador específico de concatenación &.

Resta (–)

Efectúa una resta entre dos números, o cambia el signo de un número (de positivo a negativo, y viceversa).

```
Sub Main()
    Dim Resultado As Integer
    Dim Valor As Integer
    Dim OtroValor As Integer
    ' resta de números
    Resultado = 100 - 75
    ' cambiar a signo negativo un número
    Valor = -50
    ' volver a cambiar el signo de un número,
    ' estaba en negativo, con lo que vuelve
    ' a positivo
    OtroValor = -Valor
End Sub
```

Multiplicación (*):

Multiplica dos números. En el caso de que alguno de los operandos sea un valor nulo, se usará como cero.

```
Dim Resultado As Double
Dim DatoSinValor As Integer
Dim Indefinido As Object

Resultado = 25 * 5   ' devuelve: 125

' la variable DatoSinValor no ha sido
' asignada, por lo que contiene cero
Resultado = 50 * DatoSinValor    ' devuelve: 0

' la variable Indefinido no ha sido
' asignada, por lo que contiene Nothing
Resultado = 25 * Indefinido    ' devuelve: 0

Resultado = 24.8 * 5.98   ' devuelve: 148.304
```

División Real (/):

Divide dos números, devolviendo un resultado con precisión decimal.

```
Dim Resultado As Double

Resultado = 50 / 3   ' devuelve: 16.6666666666667
Resultado = 250 / 4 ' devuelve: 62.5
```

División Entera (\):

Divide dos números, devolviendo como resultado un valor numérico entero.

```
Dim Resultado As Integer

Resultado = 50 \ 3   ' devuelve: 16
Resultado = 250 \ 4 ' devuelve: 62
```

Resto (Mod):

Divide dos números y devuelve el módulo o resto de la división.

```
Dim Resultado As Double

Resultado = 10 Mod 3     ' devuelve: 1
Resultado = 100 Mod 27   ' devuelve: 19
Resultado = 38 Mod 4     ' devuelve: 2
```

Exponenciación (^):

```
Dim Resultado As Double

Resultado = 12 ^ 5   ' devuelve: 248832
Resultado = 2 ^ 3 ^ 7    ' devuelve: 2097152
Resultado = (-4) ^ 2     ' devuelve: 16
```

Concatenación (&)

Estos operadores permiten unir dos o más cadenas de caracteres para formar una única cadena. Se recomienda el uso de & para facilitar la legibilidad del código y evitar ambigüedades. El uso de + puede dar lugar a equívoco, ya que en muchas situaciones no sabremos a primera vista si se está realizando una suma o concatenación.

```
Sub Main()
    Dim CadResulta As String
    Dim Nombre As String

    CadResulta = "esto es " & "una prueba"
    Console.WriteLine("Variable CadResulta: {0}", CadResulta)

    Nombre = "Juan"
    CadResulta = Nombre & " Almendro"
    Console.WriteLine("Variable CadResulta: {0}", CadResulta)
    Console.ReadLine()
End Sub
```

7.- Operadores Avanzados

Estos operadores simplifican la escritura de expresiones, facilitando la creación de nuestro código. El resultado empleado operadores abreviados en una expresión, es el mismo que utilizando la sintaxis normal, pero con un pequeño ahorro en la escritura de código.

Operación	Sintaxis en formato largo	Sintaxis abreviada
Suma (+)	X=X+6	X+=6
Resta (-)	X=X-6	X-=6
Multiplicación(*)	X=X*6	X*=6
División (/)	X=X/6	X/=6
División Entera (\)	X=X\6	X\=6
Exponenciación (^)	X=X^6	X^=6
Concatenación (&)	X=X & "VB"	X &= "VB"

8.- Manejo de los métodos matemáticos en .NET FrameWork.

Ahora y siempre sus programas tendrán que realizar ciertos trabajos extras con los números. Tal vez necesite redondear un número, calcular una expresión matemática compleja o introducir cierta aleatoriedad en sus programas. Los métodos matemáticos mostrados en la siguiente tabla pueden ayudarle a trabajar con números. Estos métodos son proporcionados por .NET FrameWork, una biblioteca de clases que le permitirá incrementar la potencia del sistema operativo Windows y llevar a cabo muchas de las tareas de programación más frecuentes.

.NET FrameWork es una nueva herramienta de Visual Studio . Net que comparten Visual Studio Basic, Visual C++, Visual C# y Otras herramientas contenidas en Visual Studio. Es una interfaz subyacente que forma parte del propio sistema operativo Windows. .NET FrameWork se encuentra organizada por clases que podrá incluir utilizando sus nombres en sus proyectos de programación utilizando la instrucción **Imports**. El proceso resulta bastante sencillo y mostraremos cómo funciona utilizando un método matemático contenido en la clase System.Math de .NET FrameWork.

La siguiente tabla muestra una lista parcial de los métodos matemáticos incluidos en la clase **System.Math**

Método	Propósito
Abs(*n*)	Calcula el valor absoluto de *n*.
Atan(*n*)	Calcula el arcotangente de *n* en radianes
Cos(*n*)	Calcula el coseno del ángulo *n*. El ángulo *n* se expresa en radianes.
Exp(*n*)	Calcula la constante *e* elevada a *n*
Sign(*n*)	Devuelve -1 si *n* es menor que cero, 0 si *n* es cero y +1 si *n* es mayor que cero.
Sin(*n*)	Calcula el seno del ángulo *n*. El ángulo *n* se expresa en radianes.
Sqrt(*n*)	Calcula la raíz cuadrada de *n*.
Tan(*n*)	Calcula la tangente del ángulo *n*. El ángulo **n** se expresa en radianes.

El siguiente ejemplo muestra como se pueden utilizar las funciones de la clase **System.Math**

```
Imports System.Math
Module Module1
    Sub Main()
        Dim n As Double

        n = 100.23
        Console.WriteLine("El valor Absoluto es " & Abs(n))
        Console.WriteLine("La tangente es " & Atan(n))
        Console.WriteLine("El coseno es " & Cos(n))
        Console.WriteLine("El exponente es " & Exp(n))
        Console.WriteLine("El signo es " & Sign(n))
        Console.WriteLine("El seno es " & Sin(n))
        Console.WriteLine("La raíz cuadrada es " & Sqrt(n))
        Console.WriteLine("La tangente es " & Tan(n))
        Console.ReadLine()
    End Sub

End Module
```

9.- Operadores de Comparación.

Estos operadores permiten comprobar el nivel de igualdad o diferencia existente entre los operandos de una expresión. El resultado obtenido será un valor lógico, True (Verdadero) o False (Falso). La siguiente tabla muestra la lista de los operadores disponibles de este tipo.

Operador		El resultado es Verdadero cuando	El resultado es Falso cuando
<	Menor que	ExpresiónA < ExpresiónB	ExpresiónA >= ExpresiónB
<=	Menor o igual que	ExpresiónA <= ExpresiónB	ExpresiónA > ExpresiónB
>	Mayor que	ExpresiónA > ExpresiónB	ExpresiónA <= ExpresiónB
>=	Mayor o igual que	ExpresiónA >= ExpresiónB	ExpresiónA < ExpresiónB
=	Igual a	ExpresiónA = ExpresiónB	ExpresiónA <> ExpresiónB
<>	Distinto de	ExpresiónA <> ExpresiónB	ExpresiónA = ExpresiónB

El siguiente código fuente muestra algunas expresiones de comparación utilizando números.

```
Dim Resultado As Boolean
Resultado = 10 < 45      ' devuelve: True
Resultado = 7 <= 7       ' devuelve: True
Resultado = 25 > 50      ' devuelve: False
Resultado = 80 >= 100    ' devuelve: False
Resultado = 120 = 220    ' devuelve: False
Resultado = 5 <> 58      ' devuelve: True
```

10.- Operadores de lógicos.

Visual Basic permitirá comprobar más de una expresión condicional en las estructuras de decisión y bucles, en el caso de que quiera incluir más de un criterio. Las condiciones adicionales se enlazarán mediante el uso de uno o más de los siguientes operadores lógicos:

Operador lógico	Significado
And	Si ambas expresiones condicionales son verdaderas, el resultado es Verdadero.
Or	Si alguna de las dos expresiones es Verdadera, el resultado es Verdadero.
Not	Si la expresión condicional es Falsa, el resultado es Verdadero es Falso. Si la expresión condicional es Verdadera, el resultado es Falso.
Xor	Si una, y sólo una, de las expresiones condicionales es Verdadera, el resultado es Verdadero. Si ambas son Verdaderas o Falsas, el resultado es Falso (Xor son las siglas de Or eXclusivo).

La siguiente tabla muestra algunos ejemplos de operadores lógicos en funcionamiento. En las expresiones se ha supuesto que la variable de cadena **Vehiculo** contiene el valor "**Moto**"y que la variable entera **Precio** contiene el valor **200**.

Expresión Lógica	Resultado
Vehiculo = "Moto" And Precio <300	Verdadero (ambas expresiones son Verdaderas).
Vehiculo= "Coche" Or Precio < 500	Verdadero (una condición es Verdadera).
Not Precio < 100	Verdadero (la condición es Falsa).
Vehiculo = "Moto" Xor Precio < 300	Falso (ambas condiciones son verdaderas).

Operadores cortocircuitos AndAlso y OrElse

Visual Basic .NET dispone de dos nuevos operadores lógicos que podrá utilizar en las sentencias condicionales. Estos operadores funcionan en la misma forma que And y Or, respectivamente, pero ofrecen una importante diferencia en la forma en que se evalúan.

El operador **AndAlso** es un operador que realiza una conjunción lógica de tipo cortocircuito entre dos expresiones. En este tipo de operación, en cuanto la primera expresión devuelva falso como resultado, el resto no será evaluado devolviendo falso como resultado final.

La siguiente tabla muestra los diferentes resultados obtenidos con el uso de este operador en función de los valores que tengan sus expresiones.

Cuando la ExpresiónA devuelve	**Y la ExpresiónB devuelve**	**El resultado es**
True	True	True
True	False	False
False	No se evalúa	False

Ejemplo:

```
Dim Resultado As Boolean

Resultado = (58 > 20) AndAlso ("H" = "H")    ' devuelve: True
Resultado = ("H" = "H") AndAlso (720 < 150) ' devuelve: False
Resultado = (8 <> 8) AndAlso (62 < 115)      ' devuelve: False
```

El operador **OrElse** realiza una disyunción lógica de tipo cortocircuito entre dos expresiones. En este tipo de operación, en cuanto la primera expresión devuelva verdadero como resultado, el resto no será evaluado devolviendo verdadero como resultado final.

La siguiente tabla muestra los diferentes resultados obtenidos con el uso de este operador en función de los valores que tengan sus expresiones.

Cuando la ExpresiónA devuelve	**Y la ExpresiónB devuelve**	El resultado es
True	No se evalúa	True
False	True	True
False	False	False

Ejemplo:

```
Dim Resultado As Boolean

Resultado = ("H" = "H") OrElse (720 < 150)   ' devuelve: True
Resultado = (8 <> 8) OrElse (62 < 115)       ' devuelve: True
Resultado = ("W" > "b") OrElse ("Q" = "R")   ' devuelve: False
```

11.- Prioridad de operadores.

Dentro de una línea de código que contenga varias operaciones, estas se resolverán en un orden predeterminado conocido como prioridad de operadores. Dicha prioridad se aplica tanto entre los operadores de un mismo grupo como entre los distintos grupos de operadores.

Prioridad entre operadores del mismo grupo.

Los operadores aritméticos se ajustan a la prioridad indicada en la siguiente tabla.

Prioridad de operadores aritméticos
Potenciación (^)
Negación (-)
Multiplicación y división real (* , /)
División entera (\ )
Resto de división (Mod)
Suma y resta (+ , -)

El operador de mayor prioridad es el de potenciación, los de menor son la suma y resta. En el caso de operadores con idéntica prioridad como multiplicación y división, se resolverán en el orden de aparición, es decir, de izquierda a derecha.

Los operadores de comparación tienen todos la misma prioridad, resolviéndose en el orden de aparición dentro de la expresión.

Los operadores lógicos se ajustan a la prioridad indicada en la siguiente tabla.

Prioridad de operadores lógicos
Negación (Not)
Conjunción (And, AndAlso)
Disyunción (Or, OrElse, Xor)

Prioridad entre operadores de distintos grupos.

Cuando una expresión contenga operadores de distintos grupos, estos se resolverán en el orden marcado por la siguiente tabla.

Prioridad entre operadores de distintos grupos
Aritméticos
Concatenación
Comparación
Lógicos

Uso de paréntesis para alterar la prioridad de operadores.

Podemos alterar el orden natural de prioridades entre operadores utilizando los paréntesis, encerrando entre ellos los elementos de una expresión que queramos sean resueltos en primer lugar. De esta forma, se resolverán en primer lugar las operaciones que se encuentren en los paréntesis más interiores, finalizando por las de los paréntesis exteriores. Es importante tener en cuenta, que dentro de los paréntesis se seguirá manteniendo la prioridad explicada anteriormente.

El siguiente código fuente en condiciones normales, devolvería **False** como resultado. Sin embargo, gracias al uso de paréntesis, cambiamos la prioridad predeterminada, obteniendo finalmente **True**.

```
Dim Resultado As Boolean

Resultado = ((30 + 5) * 5 > 100) And (52 > 200 / (2 + 5)) ' devuelve: True
```