

1. Nuevos conceptos en Studio Net

1.1 Introducción.

En esta nueva versión de Studio Net aparecen nuevos conceptos que es necesario comprender para un buen uso del mismo.

Estos conceptos son

- Enumeraciones.
- Delegados.
- Estructuras
- Regiones

En teoría a estas alturas no vamos a hablar de los conceptos de Clase, Método, Propiedad, Eventos.

1.2 Enumeraciones.

Las *enumeraciones* son tipos de valor que heredan de **System.Enum** y representan simbólicamente un conjunto de valores de uno de los tipos enteros primitivos. En un tipo de enumeración E, el valor predeterminado es el valor producido por la expresión CType(0, E).

El tipo subyacente de una enumeración debe ser un tipo entero que puede representar todos los valores del enumerador definidos en la enumeración. Si se especifica un tipo subyacente, debe ser **Byte**, **Short**, **Integer** o **Long**. Si no se especifica explícitamente ningún tipo subyacente, la opción predeterminada es **Integer**.

El siguiente ejemplo declara una enumeración con el tipo subyacente **Long**:

```
Enum Color As Long
    Red
    Green
    Blue
End Enum
```

La instrucción Enum sólo puede aparecer en el nivel de módulo, espacio de nombres o archivo. Esto es, puede declarar enumeraciones en un archivo de código fuente o dentro de un módulo, una clase o una estructura, pero no dentro de un procedimiento. Una vez definido el tipo Enum, puede utilizarse para declarar el tipo de variables, argumentos de procedimientos y el valor devuelto por Function.

Se puede tener acceso a las enumeraciones desde cualquier lugar del módulo, la clase o la estructura en que se declaran.

Una enumeración y todos sus miembros de forma predeterminada son Public.

Para especificar la accesibilidad de manera más detallada se puede incluir Public, Protected, Friend, Protected Friend o Private en la instrucción Enum.

La accesibilidad que especifique se aplicará a todos los miembros, así como a la propia enumeración.

Una enumeración es un conjunto de constantes relacionadas. Los miembros de la enumeración comprendidos entre las instrucciones Enum y End Enum se inicializan en valores constantes.

Los valores definidos no se pueden modificar en tiempo de ejecución.

Los valores pueden incluir números positivos y negativos, como muestra el siguiente ejemplo:

```
Enum Nivel
    Indebido = -1
    Minimo = 0
    Maximo = 1
End Enum
```

Si el valor de un miembro rebasa el intervalo permitido para el tipo de datos subyacente, o si se inicializa un miembro al valor máximo permitido por él, el compilador notifica un error.

Enumeration variables son variables declaradas para ser del tipo Enum. Declarar una variable de este modo le ayudará a controlar los valores que le asigne.

Si Option Strict es On, sólo se pueden asignar miembros de la enumeración a la variable de enumeración. En este caso, se puede utilizar la palabra clave CType para convertir explícitamente un tipo de datos numérico al tipo Enum.

Deben calificarse todas las referencias a un miembro de enumeración, ya sea con el nombre de una variable de enumeración o con el nombre de la propia enumeración. Por ejemplo, en el ejemplo anterior, puede hacer referencia al primer miembro como Nivel.Indebido, pero no como Indebido.

```
Private Enum Ejemplo as Integer
    Minimo = 0
    Maximo = 99
End Enum

Private Sub Enumerador()
    Dim A As [Enum]
End Sub
```

Un programador podría utilizar un tipo subyacente **Long**, como en el ejemplo, para habilitar el uso de valores que estén en el intervalo de **Long**, pero no en el de **Integer**, o para preservar esta opción para el futuro.

En éste ejemplo vemos un uso del enumerador.

Cuando el ratón baja aparece el texto en el label de "prohibido el paso", cuando el botón del ratón sube aparece el texto "Permitido el paso".

```
Private Enum Semaforo
    Rojo = -1
    Verde = 1
    Naranja = 0
End Enum

Private Sub Enumerador(ByVal Estado As Short)
    Dim A As Semaforo

    Select Case Estado
        Case A.Rojo
            Label1.Text = "Prohibido pasar"
        Case A.Naranja
            Label1.Text = "Próximo el cierre"
        Case A.Verde
            Label1.Text = "Permitido el paso"
    End Select
End Sub

Private Sub Form1_MouseDown(ByVal sender As Object, ByVal e As System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseDown
    Enumerador(-1)
End Sub

Private Sub Form1_MouseUp(ByVal sender As Object, ByVal e As System.Windows.Forms.MouseEventArgs) Handles MyBase.MouseUp
    Enumerador(1)
End Sub
```

El uso de los enumerados clarifica y mejora en mucho el código en la programación, pues se deja de tener que tener presente cuales son los valores en los que hay que moverse para la selección de opciones.

En la versión actual de Vb son innumerables el número de Enumerados definidos en el lenguaje.

Además los mismos datos que se definen en una enumeración, se pueden cargar en una lista y ofrecerlos en el programa para su visualización.

El siguiente ejemplo muestra como se ha definido una enumeración y se ha cargado en un ListBox.

```

Enum Unid_Distan
    Terrestres = 1
    Marinas = 2
    Leguas = 3
    Yardas = 4
End Enum

Public Sub CargaListaEnumeracionDistancias(ByRef Lista As ListBox)
    Dim Unidad As Unid_Distan
    For Each Unidad In [Enum].GetValues(GetType(Unid_Distan))
        Lista.Items.Add(Unidad)
    Next
End Sub

```

Su uso no es ese únicamente, sino el que ya se ha visto en ejemplos anteriores, y como el que sigue.

```

Public Function Conversion(ByVal M As Single, _
    ByVal T As Unid_Distan) As Single
    '
    '     Conversión
    '
    Dim C As Single

    Select Case T
        Case Unid_Distan.Terrestres
            C = CSng(M / 1609.34)
        Case Unid_Distan.Marinas
            C = M / 1852
        Case Unid_Distan.Leguas
            C = M / 4190
        Case Unid_Distan.Yardas
            C = CSng(M / 0.9144)
    End Select
    Conversion = C
End Function

Conversion(CSng(Campo01.Text), Unid_Distan.Leguas)

```

Como podemos ver, el uso dentro del código en la definición de las variables y en las llamadas a procedimientos, permite que podamos saber cuales son las posibilidades de esa variable.

1.3 Estructuras.

Tipos de usuario en pocas palabras.

Podríamos decir que es el uso personalizado de un vector, de tal forma que se declara del número y tipo de elementos adecuado para almacenar un tipo de información para un uso posterior.

Ese uso puede ser posteriormente una región.

Se utilizan para almacenar información que luego pueda ser utilizada en una región, el ejemplo puede ser este:

```

' Datos de coordenadas del rectángulo
Dim TamanyoRectangulo As RectangleF

```

Donde RectangleF es una estructura que es capaz de almacenar los datos relativos a las coordenadas de un rectángulo, los cuatro números de punto flotante que representan la posición y tamaño de un rectángulo.

```
' Datos de tamaño del rectángulo
Dim Tamanyo As New SizeF (100,100)
```

Esta es otra forma de declarar el tamaño de un rectángulo, utilizando la estructura SizeF, en este caso se declara el ancho y el alto, no sus coordenadas.

El ejemplo que sigue, observamos el uso de structure para la declaración del registro en un archivo random, o para cualquier otro uso.

```
Public Structure FilConfig
    <VBFixedString(25)> Public FontNombre As String
    Public Tamanyo As Single
    Public Negrita As Boolean
    Public Subrayado As Boolean
    Public Tachado As Boolean
    Public Italica As Boolean
    Public ColorTexto As Integer
    Public ColorFondo As Integer
End Structure
```

1.4 Regiones.

En el manejo de algunos métodos de Graphics, aparece lo que se denomina región.

Este elemento es una estructura que permite almacenar los datos necesarios para un uso posterior de los mismos en un método de dibujo.

Por ejemplo

```
' Vector para coordenadas
Dim CoordinadasArea(2) As [Region]
' Datos de coordenadas del rectángulo
Dim TamanyoRectangulo As RectangleF
```

En éste caso se declara un vector de dos elementos del tipo Region, en el cual después se almacenará los datos de las coordenadas de dos rectángulos.

En el ejemplo que sigue se utiliza el vector para almacenar los datos que devuelve el método MeasureCharacterRanges del objeto Grafico.

```
' Obtiene Mide los rangos de la cadena y los deja en el vector
CoordinadasArea = Grafico.MeasureCharacterRanges(Texto, Fuente,
MedidaRectangulo, Formato)
```

Posteriormente esta información es utilizada de la siguiente forma:

```
' Dibuja el rectángulo
TamanyoRectangulo = CoordinadasArea(0).GetBounds(Grafico)
Grafico.DrawRectangle(New Pen(Color.Red, 1),
Rectangle.Round(TamanyoRectangulo))
```

Una región es escalable porque sus coordenadas se especifican en coordenadas universales. Sin embargo, en la superficie de dibujo, su interior depende del tamaño y la forma de los píxeles que la representan.

Una aplicación puede utilizar regiones para fijar el resultado de las operaciones de dibujo. El administrador de ventanas utiliza las regiones para definir el área de dibujo de las ventanas.

Estas regiones se llaman regiones de recorte. Una aplicación puede utilizar también regiones en las operaciones de comprobación de visitas, para comprobar por ejemplo si un punto o un rectángulo forma una intersección con una región.

Las aplicaciones pueden rellenar una región mediante un objeto Brush.

2. Clases.

2.1 Introducción.

Sobre las características de las clases, hay mucho escrito, y por gente que sabe más que nosotros, y cierto es que no hay mucha coincidencia entre los mismos, por lo tanto no vamos a intentar dar ninguna lección, ni mucho menos sentar cátedra, pero si que vamos a intentar entenderlo y que lo entienda quien pueda estar como nosotros.

Lo más parecido a una clase quizás sea lo que conocemos como librerías.

Una clase es como una plantilla, un molde, con el que luego crearemos una serie de objetos que dispondrán de una serie de características, propiedades, que tendrán un comportamiento, métodos, y es capaz de reaccionar ante una serie de hechos, eventos, que están definidos en la clase de la cual se crea el objeto.

Hasta aquí pocas diferencias con lo que son los objetos de un formulario, y un formulario también, y es que esos dos tipos de objetos derivan de una clase, los formularios de la clase System.Windows.Forms, y los objetos de la que les corresponda, según su tipo, en principio de System.Windows.Forms

Según la teoría, la aparición de las clases viene motivada por la complejidad de las aplicaciones actuales, con el fin de facilitar la creación de software compacto y fiable.

Cada día más, se crean aplicaciones en las que intervienen no solo elementos que están ubicados en el equipo local, sino que también interviene el uso de redes, servidores de distinto tipo etc..., y eso complica la creación del software.

La utilización de clases tiene como fin el de la reutilización de código ya escrito, porque en una clase se puede encerrar, aislar, código que no sea visible desde el exterior, y los parámetros que necesite dicho código para funcionar es lo que convertimos en propiedades de dicha clase.

La reutilización es lo que conocemos como herencia, aunque luego no se usa mucho.

El que el código dentro de la clase esté o no accesible, es lo que conocemos como encapsulamiento.

La posibilidad que tienen las clases de derivarse unas de otras, es lo que les da la característica de polimórficas.

En Visual Basic con la versión 6, ya se podían crear clases, pero estas no cumplían todos los requisitos que debe cumplir una clase.

En ésta versión de VB en Studio Net, si que se siguen dichos requisitos, y es posible implementar clases con todas sus características.

2.2 Declaración.

El primer paso para el uso de las clases es su declaración.

Como en el caso de los procedimientos, los tipos y las variables, éstas han de poseer delante el ámbito en el que podrán ser utilizadas, es decir:

Private	Solo podrán ser utilizadas en el ámbito en el que está definida.
Public.	Puede ser usada en el ámbito en el que está definida y fuera del mismo.
Friend	La clase puede verse en el ensamblado al que pertenece, pero no fuera de él.
Protected.	Solo se puede aplicar a los miembros de una clase, no a la clase en sí.
Protected Friend	Los identificadores pueden utilizarse en el ensamblado en el que se han definido, y en las clases derivadas, aunque estén en otros ensamblados.

Por lo tanto para la definición de una clase escribiremos

```
Public Class Ejemplo
    .. / ..
End Class
```

2.3 Qué tienen.

Las clases pueden albergar

- Variables.
- Propiedades.
- Métodos.
- Eventos.

2.3.1 Variables.

Variables, porque son necesarias para la gestión de los datos a utilizar en la clase.

La información puede provenir de la ejecución propia de la clase, o de las propiedades de la misma, que se pueden inicializar en la misma clase, y/o recibir datos desde el programa que genera el objeto.

2.3.2 Propiedades.

Son la parte de la descripción de la clase que se puede definir desde el programa, podrán ser de distinto tipo, solo lectura, solo escritura, lectura escritura.

Existe la posibilidad de disponer de código de validación en la asignación de datos, Set, para su control.

2.3.3 Métodos.

Es lo que le da a la clase la capacidad de acción.

Pueden heredarse de una a otra, y pueden definirse varios métodos con el mismo nombre y distintos argumentos, es lo que se conoce como sobre carga.

También pueden utilizarse o anularse los métodos de una a otra clase, en función de nuestras necesidades, Overridable, Overrides.

En función de las necesidades, se puede utilizar el método de la clase base en una derivada, para así poderlo ampliar en la misma, Shadows.

2.3.4 Eventos.

Con la creación de eventos en las clases se tiene la posibilidad que sea también la clase la que llama al programa, y la de interactuar con el mismo.

2.4 Con las clases se puede hacer.

Crear objetos, que es su fin primordial.

Obtener clases nuevas a partir de las existentes.

Reutilizar los métodos existentes en nuevas clases derivadas de la origen.

Generar eventos para enviar señales al programa.

Crear componentes para utilizar en los programas.

2.5 Clases derivadas, herencia.

Una de las características de las clases es su reutilización, para ampliarlas en una nueva versión.

La clase nueva hereda todos los elementos de la clase anterior, excepto los definidos como Privados.

Además en esta versión de Vb, la clase que es heredada no tiene porque estar escrita en Vb, puede estar en C, o Java.

Para heredar una clase solo hay que colocar en el texto de la nueva clase la palabra Inherits y el nombre de la clase que queremos heredar.

```
Inherits Calculo
```

Con lo que toda la operatividad de la clase Calculo se hereda en la nueva clase.

```

Namespace Ambito
  Public Class Calculo
    ' La variable se hace pública
    Public Shared Contador As Int16 = 0
    Public Sub Incrementa()
      Contador += 1
    End Sub

    Public Sub Resta()
      Contador -= 1
    End Sub

    Public Sub Visualiza()
      Console.WriteLine("Contador vale {0} ", Contador)
    End Sub
  End Class

  Public Class Clase
    ' herencia de la clase anterior
    Inherits Calculo
  End Class

End Namespace ' Ambito

Module Principal
  Sub Main()
    ' definición de la clase
    Dim MiClase As New Ambito.Clase
    MiClase.Visualiza()
    MiClase.Incrementa()
    MiClase.Visualiza()
    Console.ReadLine()
  End Sub
End Module

```

Como podemos ver en el ejemplo, la clase Clase, en realidad no tiene código, solo hemos puesto

```
Inherits Calculo
```

Y todos los elementos de la clase Calculo han pasado a ser utilizables en Clase.

Pero claro también puede ser que a nosotros nos interese que una clase que hemos escrito, no sea reutilizable, en ese caso la solución está en colocar en la definición de esa clase la palabra NotInheritable, y con ello impedimos que se puedan crear clases a partir de la misma.

Si en la clase anterior colocamos

```
Public NotInheritable Class Calculo
```

en la definición de la clase, en la ventana de depuración enseguida aparece el contenido que vemos a continuación.

Lista de errores					
✖ 10 errores ⚠ 0 advertencias ℹ 0 mensajes					
	Descripción	Archivo	Línea	Columna	Proyecto
✖ 1	No es válido el uso de la palabra clave como identificador.	Module1.vb	2	10	ConsoleApplication1
✖ 2	La instrucción no es válida en un espacio de nombres.	Module1.vb	4	3	ConsoleApplication1
✖ 3	La instrucción no es válida en un espacio de nombres.	Module1.vb	6	3	ConsoleApplication1
✖ 4	La instrucción no es válida en un espacio de nombres.	Module1.vb	10	3	ConsoleApplication1
✖ 5	La instrucción no es válida en un espacio de nombres.	Module1.vb	14	3	ConsoleApplication1
✖ 6	'End Class' debe ir precedida de la instrucción 'Class' correspondiente.	Module1.vb	17	3	ConsoleApplication1
✖ 7	El tipo 'Calculo' no está definido.	Module1.vb	22	14	ConsoleApplication1
✖ 8	'Visualiza' no es un miembro de 'ConsoleApplication1.Ambito.Clase'.	Module1.vb	31	5	ConsoleApplication1
✖ 9	'Incrementa' no es un miembro de 'ConsoleApplication1.Ambito.Clase'.	Module1.vb	32	5	ConsoleApplication1
✖ 10	'Visualiza' no es un miembro de 'ConsoleApplication1.Ambito.Clase'.	Module1.vb	33	5	ConsoleApplication1

Otro caso que se puede plantear es que deseemos agrupar varias clases en una misma, pero solo por cuestiones de organización, es decir no se puede usar para crear objetos, sino solo otras clases, y que de éstas sí se puedan crear objetos, en ese caso a ésta clase puente la podemos definir como MustInherit, algo así como que tiene o que debe ser heredada, y con ella no se podrán crear objetos, pero si podrá ser usada para crear nuevas clases.

2.6 Polimorfismo.

Lo que ya hemos visto de herencia, Inherits, y de sobrecarga, OverLoads, forma parte de lo que se denomina polimorfismo.

En un caso porque se hereda una clase y se puede potenciar, cambiar etc..., y en el otro porque se le dan distintas formas al mismo método.

2.7 MyClass, MyBase, Me.

¿Qué es esto.?

Si creamos clases, es para crear objetos con ellas.

Sí las clases tienen como característica que se puedan heredar en otra clase, ¿qué pasa, si a mi me interesa usar el método de la clase heredada en lugar del método que estoy escribiendo para ésta clase en un momento dado?.

El ejemplo que sigue puede dar una idea del uso de Me, y MyClass, pero como realmente se aprecia el ejemplo es ejecutándolo paso a paso, con <F8>, y no solo una vez, para comprenderlo.

Hay que fijarse en que MiMetodo está definido en las dos clases, y en función del uso de Me o MyClass se ejecuta uno u otro.

MyBase, se utiliza para llamar desde la clase derivada a un método de la clase base que ha sido reemplazado en la clase derivada, pero que sin embargo posee código que nos interesa se ejecute, porque lo que estamos haciendo en realidad es una ampliación del código de la clase base.

```

Namespace Ambito
Class ClaseBase
    Public Sub Sombreado()
        Console.WriteLine("Texto desde sombreado")
    End Sub

    Public Overridable Sub MiMetodo()
        Console.WriteLine("Cadena en clase base")
    End Sub

    Public Sub UsandoMe()
        Me.MiMetodo()
    End Sub

    Public Sub UsandoMyClass()
        MyClass.MiMetodo()
    End Sub
End Class

Class ClaseDerivada
    Inherits ClaseBase

    Public Overrides Sub MiMetodo()
        Console.WriteLine("Cadena en clase derivada")
    End Sub

    Public Shadows Sub Sombreado()
        MyBase.Sombreado() ' llama al de clasebase
        Console.WriteLine("Texto desde sombreado derivada")
    End Sub

End Class

Class Prueba
    Sub Inicio()
        Dim Objeto As ClaseDerivada = New ClaseDerivada
        Objeto.UsandoMe() ' Visualiza "Cadena en clase derivada"
        Objeto.UsandoMyClass() ' Visualiza "Cadena en clase base"
        Objeto.Sombreado() ' Usa solo el de la clase derivada
    End Sub
End Class
End Namespace ' Ambito

Module Principal
    Sub Main()
        ' definición de la clase
        Dim MiClase As New Ambito.Prueba
        MiClase.Inicio()
        Console.ReadLine()
    End Sub
End Module

```

En el ejemplo del uso del método sombreado desde la clase derivada, cambiar MyBase por Myclass y probarlo, para observar los resultados, y comentar la instrucción donde esta MyBase, para comprobar que no se ejecuta la de mi base. Observar el uso de Shadows en la declaración en la clase derivada de Sombreado.

3. Clases, Métodos.

3.1 Introducción.

Los métodos están compuestos por funciones, devuelven datos, o por procedimientos, reciben datos y actúan.

Si hay que crear un objeto con una clase y me interesa que tenga un código que lo identifique, el método que me genere el código será una función.

Si hay que escribir un método que recoja datos para algo que hay que hacer, será un procedimiento.

El ejemplo que sigue contiene la declaración de una clase sencilla en el namespace Generalitat.

```
Namespace Generalitat
  Public Class Instituto
    Private Shared Sub GenInstPrivate()
      Console.WriteLine("Método privado llamado desde el public shared de
la misma clase Instituto.")
    End Sub

    Public Shared Sub GenInstPublicShared()
      Console.WriteLine("Está dentro de un sub public shared en la clase
instituto en el namespace Generalitat")
      GenInstPrivate()
    End Sub
  End Class      ` End Class
End Namespace   ' Generalitat

Module Principal
  Sub Main()
    Generalitat.Instituto.GenInstPublicShared()
  End Sub
End Module
```

En este ejemplo disponemos de dos métodos en la clase, el privado no es visible fuera de la clase.

El público sí que es visible, y puede verse como en el Main, es utilizado.

Después en el público se llama desde el interior de la clase al privado.

Es necesario que dispongan de la palabra Shared para que puedan ser usados.

Cuando escribimos un método éste puede ser un procedimiento o una función, todo dependerá de que tenga que devolver o no información.

Hay que tener en cuenta que la diferencia entre un procedimiento y un método es que el primero está en un módulo y el segundo en una clase, pero ambos son Sub, procedimientos.

```
Namespace Ambito
  Public Class Clase
    Private Shared Sub Procedimiento()
      Console.WriteLine("Método privado llamado desde el public shared de
la misma clase ""Clase""")
    End Sub

    Public Shared Sub Metodo()
      Console.WriteLine("Esta dentro de un sub public shared en la clase
instituto en el namespace Generalitat")
      Procedimiento()
    End Sub

    Public Shared Function Funcion() As String
      Funcion = "Texto en la función ""funcion""
    End Function
  End Class      ` End de la clase
End Namespace   ' End del Ambito
```

```

Module Principal
  Sub Main()
    Ambito.Clase.Metodo()
    Console.WriteLine(Ambito.Clase.Funcion)
  End Sub
End Module

```

Pero como en realidad se debe utilizar la clase es como figura en el siguiente ejemplo.

```

Module Principal
  Sub Main()
    ` Definición de un objeto que referencia a la clase
    Dim MiClase As New Ambito.Clase
    MiClase.Metodo()
    Console.WriteLine(MiClase.Funcion)
    Procedimiento()
  End Sub
End Module

```

La diferencia está en que en este ejemplo lo que se hace es crear un objeto que referencia a la clase, y eso nos permite por ejemplo lo siguiente.

```

Namespace Ambito
  Public Class Clase

    Private Sub Procedimiento(ByVal Texto As String)
      Mensaje = Mensaje & Texto
      Console.WriteLine(Mensaje)
    End Sub

    Public Sub Metodo(ByVal Texto As String)
      Procedimiento(Texto)
    End Sub
  End Class
End Namespace ' Ambito

Module Principal
  Sub Main()
    ` definición de la clase
    Dim MiClase As New Ambito.Clase
    Dim OtraClase As New Ambito.Clase
    ` uso de un método de la clase
    MiClase.Metodo("Texto añadido")
    OtraClase.Metodo("Texto añadido dos")
  End Sub
End Module

```

Como podemos observar en el Main, se han creado dos objetos que referencian a misma clase, pudiéndose usar con contenidos distintos, aunque éste uso no tiene mucho sentido en la realidad, donde se aprecia mejor es en el ejemplo con propiedades.

3.2 Cuando usar métodos o propiedades.

Dado que en el momento de asignar datos a una propiedad, se puede escribir código, podría darse el caso de que pudiera escribirse indistintamente un método o una propiedad.

Creemos que eso no es correcto, pues si hay que empezar a pensar que la validación ha de resolver algo más que simplemente eso, y además añadimos que en esa validación puede darse el caso que intervengan otras propiedades, eso nos obligaría a establecer un orden en la asignación de datos de las propiedades.

Si pensamos que las propiedades no forman parte del ejecutivo de un programa, sino de su descripción, o definición, la propiedad no ha de formar parte de la ejecución, por lo tanto no hay que resolver código con las propiedades, sino dejarlo a los métodos, y además así no hay que establecer de forma obligatoria un orden al asignar valores a las propiedades, siempre que eso pueda ser evitado.

3.3 Métodos sobrecargados, OverLoads.

Bien eso es una característica de las clases, es decir que resolvamos con un único nombre una acción que en función del momento necesita de más o menos parámetros, y por lo tanto de distintos procedimientos, pero hay que reconocer que implica un esfuerzo a la hora de su utilización.

La solución a lo anterior pasa por el uso de una circunstancia, un nombre, pero eso implicaría un sinnúmero de nombres y consecuentemente de métodos, por lo que el tema de la sobrecarga es una idea acertada, se reduce el número de métodos y se facilita su localización, solo queda elegir cual de sus posibilidades, a veces complejo porque hay muchas.

Cuando los métodos están en una misma clase no es necesario indicar lo con la palabra OverLoads, VB lo asume directamente, sin embargo si están en clases distintas, sí.

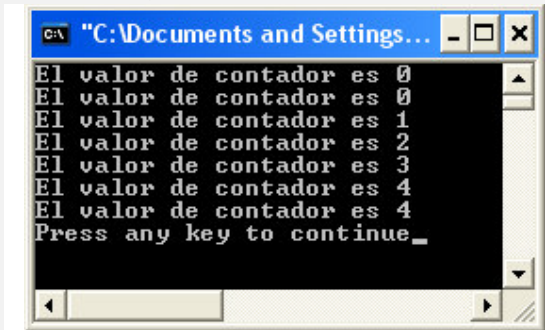
En el ejemplo que hay a continuación hemos creado dos métodos iguales "Visualiza", uno el que ya teníamos, y otro con un argumento texto, los dos se llaman igual, y en el principal, se han utilizado de forma alternativa uno y otro.

El lenguaje discrimina cual ha de usar en cada momento.

Como están en la misma clase no es necesario utilizar la palabra OverLoads.

```
Namespace Ambito
  Public Class Clase
    Private Shared Contador As Int16 = 0
    Public Sub Incrementa()
      Contador += 1
    End Sub
    Public Sub Visualiza()
      Console.WriteLine("El valor de contador es {0} ", Contador)
    End Sub
    Public Sub Visualiza(ByVal Texto As String)
      Console.WriteLine("{0} {1} ", Texto, Contador)
    End Sub
  End Class
End Namespace ' Ambito

Module Principal
  Sub Main()
    ' definición de objetos
    Dim MiClase As New Ambito.Clase
    Dim OtraClase As New Ambito.Clase
    ' valor inicial para cada clase
    MiClase.Visualiza()
    OtraClase.Visualiza("El valor de contador es")
    ' incremento con una clase y
    ' visualizar con la otra
    MiClase.Incrementa()
    OtraClase.Visualiza()
    ' incremento con una clase y
    ' visualizar con la otra
    MiClase.Incrementa()
    OtraClase.Visualiza("El valor de
contador es")
    ' incremento con una clase y
    ' visualizar con la otra
    MiClase.Incrementa()
    OtraClase.Visualiza()
    ' invertir las acciones con una clase y
    ' visualizar con la otra
    OtraClase.Incrementa()
```



```
C:\Documents and Settings...
El valor de contador es 0
El valor de contador es 0
El valor de contador es 1
El valor de contador es 2
El valor de contador es 3
El valor de contador es 4
El valor de contador es 4
Press any key to continue_
```

```

MiClase.Visualiza("El valor de contador es")
' el valor es el mismo en las dos
OtraClase.Visualiza()
Console.ReadLine()
End Sub
End Module

```

Podemos observar que el resultado es el mismo que en la ejecución anterior.

Sin embargo si creamos dos clases distintas con el mismo nombre de método en el mismo ámbito sí que es necesario indicar el OverLoads.

Los valores no se comparten porque cada objeto está definido con una de las clases.

```

Namespace Ambito
    Public Class Suma
        Private Shared Contador As Int16 = 0
        Public Sub Incrementa()
            Contador += 1
        End Sub

        Public Sub Visualiza()
            Console.WriteLine("El valor de contador es {0} ", Contador)
        End Sub
    End Class

    Public Class Resta
        Private Shared Contador As Int16 = 0
        Public Sub Resta()
            Contador -= 1
        End Sub
        Public Overloads Sub Visualiza()
            Console.WriteLine("El valor de contador es {0} ", Contador)
        End Sub
    End Class
End Namespace ' Ambito

Module Principal
    Sub Main()
        ' definición de objetos
        Dim MiClase As New Ambito.Suma
        Dim OtraClase As New Ambito.Resta
        ' valor inicial para cada clase
        MiClase.Visualiza()
        OtraClase.Visualiza()

        MiClase.Incrementa()
        OtraClase.Resta()

        MiClase.Visualiza()
        OtraClase.Visualiza()

        MiClase.Incrementa()
        OtraClase.Resta()

        MiClase.Visualiza()
        OtraClase.Visualiza()
    End Sub
End Module

```

El uso de la palabra clave OverLoads facilita la lectura del programa.

3.4 Ocultar un método, Overridable, Overrides.

Puede pasar que al heredar una clase, alguno de los métodos de ésta, ese método no nos sirva porque en la nueva clase no nos sirva su antigua implementación, en ese caso ese método no es válido y declararlo como OverLoads no nos solucionaría la papeleta, y máxime si además la línea de parámetros es igual a la versión antigua.

En ese caso no queda más remedio que ocultarlo y redefinirlo, de forma que queda anulado el viejo y reescribimos uno nuevo.

En el ejemplo existen dos métodos visualiza, uno en cada clase.

La clase "Clase" hereda a la clase Calculo mediante la instrucción Inherits, y suponemos que no nos sirve el método visualiza de Cálculo.

```
Namespace Ambito
  Public Class Calculo
    ' La variable se hace pública
    Public Shared Contador As Int16 = 0

    Public Sub Incrementa()
      Contador += 1
    End Sub

    Public Sub Resta()
      Contador -= 1
    End Sub
    ' método a ocultar y reescribir
    Public Overridable Sub Visualiza()
      Console.WriteLine("El valor de contador es {0} ", Contador)
    End Sub
  End Class

  Public Class Clase
    ' herencia de la clase anterior
    Inherits Calculo
    ' método nuevo
    Public Overrides Sub Visualiza()
      Console.WriteLine("Contador vale {0} ", Contador)
    End Sub
  End Class
End Namespace ' Ambito

Module Principal
  Sub Main()
    ' definición de la clase
    Dim MiClase As New Ambito.Clase
    ' valor inicial para cada clase
    MiClase.Visualiza()
    MiClase.Incrementa()
    MiClase.Visualiza()
  End Sub
End Module
```

En el Main, solo usamos la clase nueva, ya que ésta ha heredado los métodos de la anterior, y usamos sus métodos heredados.

El usar la palabra clave Overridable y Overrides ayuda a facilitar la lectura del programa.

3.5 Constructores, New, Destructor, Finalize.

Las clases disponen de unos métodos denominados constructores y destructores, que se ejecutan de forma automática cuando se crea un objeto de la clase.

Puede existir uno o varios métodos New, constructor, con distintas líneas de parámetros, OverLoads.

Pero solo puede haber un método destructor, Finalize.

El método Finalize, destructor, es protegido porque no es accesible desde el exterior de la clase.

El método New, se ejecuta siempre, es potestad, o necesidad, el utilizarlo escribiendo código en el mismo, cuando haya que hacer algo en el momento en el que se crea el objeto.

Su ejecución es automática.

```
Namespace Ambito
  Public Class Calculo
    ' La variable se hace pública
    Public Shared Contador As Int16 = 0

    Public Sub New()
      Console.WriteLine("Se ejecutó el constructor New")
    End Sub

    Protected Overrides Sub Finalize()
      Console.WriteLine("Se ejecutó el destructor Finalize")
    End Sub

    Public Sub Incrementa()
      Contador += 1
    End Sub

    Public Sub Resta()
      Contador -= 1
    End Sub

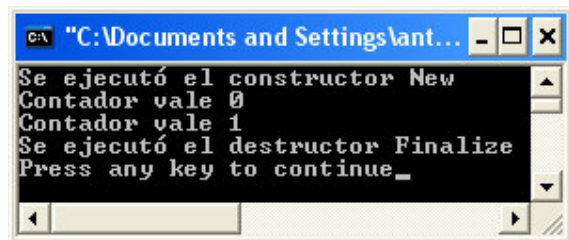
    Public Sub Visualiza()
      Console.WriteLine("Contador vale {0} ", Contador)
    End Sub
  End Class

  Public Class Clase
    ' herencia de la clase anterior
    Inherits Calculo
  End Class
End Namespace ' Ambito

Module Principal
  Sub Main()

    ' definición de la clase
    Dim MiClase As New Ambito.Calculo

    MiClase.Visualiza()
    MiClase.Incrementa()
    MiClase.Visualiza()
  End Sub
End Module
```



```
C:\> "C:\Documents and Settings\ant... - _ X
Se ejecutó el constructor New
Contador vale 0
Contador vale 1
Se ejecutó el destructor Finalize
Press any key to continue_
```

Como podemos ver el constructor New, por defecto no tiene parámetros, y si no lo escribimos no por eso deja de existir, y de ejecutarse, o lo que es lo mismo, se puede crear una instancia de la clase sin tener ningún control sobre la misma, por lo tanto lo que interesa es anular ese constructor, método, New por defecto, y una forma de hacerlo es declarar como Private un Sub New, vacío.

De esa forma cuando se declare una instancia de la clase, el constructor no se podrá ejecutar, pero como tiene que existir un constructor, declaramos uno nuestro con los parámetros que sean necesarios y declarado como Public, con lo que ya no se puede crear una instancia de la clase sin que se pueda controlar.

En el ejemplo anterior probar a declarar como private el método New, y observar lo que sucede.

La solución pasa por modificar la clase Cálculo, como ya hemos dicho con un nuevo New, y como hemos eliminado el estándar hay que crear un New en la clase derivada para que exista, y éste se hace usando el de la clase origen.

En la clase derivada hay varias líneas comentadas, probar a quitar el comentario y observar lo que sucede.

```
Namespace Ambito
    Public Class Calculo
        ' La variable se hace pública
        Public Shared Contador As Int16 = 0
        ' Anulamos el estándar
        Private Sub New()
            Console.WriteLine("Se ejecutó el constructor New")
        End Sub
        ' Creamos el nuestro
        Public Sub New(ByVal Texto)
            Console.WriteLine(Texto)
        End Sub
        Protected Overrides Sub Finalize()
            Console.WriteLine("Se ejecutó el destructor Finalize")
        End Sub
        Public Sub Incrementa()
            Contador += 1
        End Sub

        Public Sub Resta()
            Contador -= 1
        End Sub

        Public Sub Visualiza()
            Console.WriteLine("Contador vale {0} ", Contador)
        End Sub
    End Class

    Public Class Clase
        ' herencia de la clase anterior
        Inherits Calculo
        ' Como no hay New estándar hay que declarar uno.
        ' y hay que usar el de la clase Cálculo, por eso hay que usar MyBase
        Public Sub New()
            ' MyClass.New() ` no vale.
            MyBase.New("Se ejecutó el constructor New llamado desde la clase
derivada")
            ' Console.WriteLine("No, se ejecutó el constructor New en la clase
derivada")
        End Sub
    End Class
End Namespace ' Ambito

Module Principal
    Sub Main()
        ' definición de la clase
        Dim MiClase As New Ambito.Clase

        MiClase.Visualiza()
        MiClase.Incrementa()
        MiClase.Visualiza()
    End Sub
End Module
```

Conclusión, conviene anular el constructor New, Private Sub New, sin parámetros para crear el nuestro y controlar las instancias que se crean de la base.