1. Clases.

1.1 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 implementrar clases con todas sus características.

1.2 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
```

1.3 Qué tienen.

Las clases pueden albergar

Variables.
Propiedades.
Métodos.
Eventos.

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.4 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.

1.5 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 | 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

Lista de errores | Ventana Inmediato | Resultados

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.

1.6 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.

1.7 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.

2. Clases, Métodos.

2.1 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 Clase
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.

2.2 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.

2.3 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 sinfín 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.

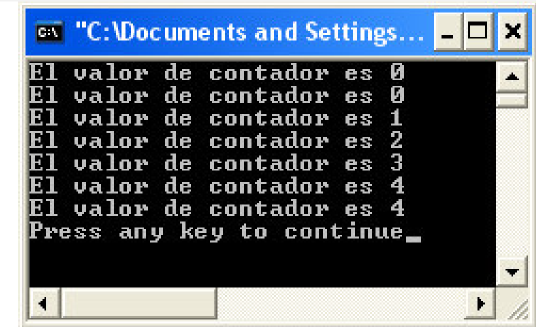
Cuando los métodos están en una misma clase no es necesario indicarlo con la palabra OverLoads, VB lo asume directamente, sin embargo si están en clases distintas, sí.

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()
```



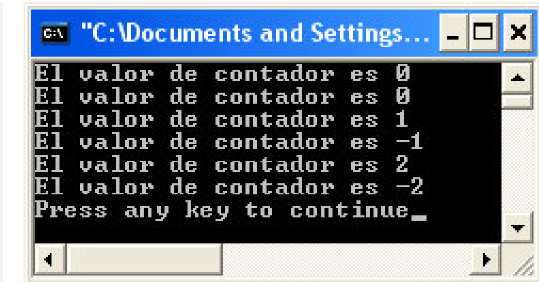
```
    MiClase.Visualiza("El valor de contador es")
    ' el valor es el mismo en las dos
    OtraClase.Visualiza()
    Console.Readline()
  End Sub
End Module
```

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.

2.4 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.

2.5 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.

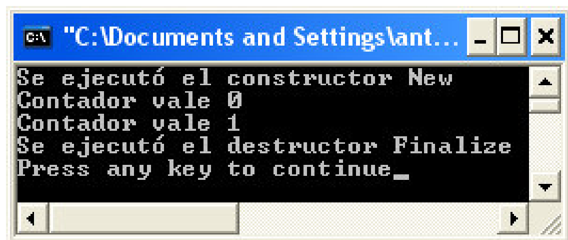
Pero solo puede haber un método destructor, Finalize.

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.Clase

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



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.

3. Clases, Propiedades.

3.1 Propiedades.

Podríamos decir que son la fachada de un edificio, la parte visible de la clase.

Si disponemos de métodos que son capaces de actuar, también nos hará falta características que definan como han de ser cada uno de los ejemplares que creemos con esa clase, sus propiedades, y los datos que se han de guardar en la misma para su proceso normal.

Las propiedades dentro de una clase son eso, únicamente un dato, y como dato que son, se almacenan en una variable, que para que podamos o puedan usar desde el exterior, -el objeto que con la misma se cree-, se denominan propiedades.

Explicado de otra forma.

Las propiedades dentro de una clase son únicamente una variable, que se encargará de almacenar aquella información que sea necesaria, pero como las propiedades se han de ver desde el exterior de la clase, y las variables de la clase no, tiene que haber algo que las diferencie, ese algo es la palabra Property.

Pero claro no es esa la diferencia más importante, sino lo que hay detrás.

Al declarar propiedades impedimos que se pueda manipular la variable de forma directa, sin pasar por la validación del SET, que podemos o no escribir, pero que existe, y de esa forma nuestra clase queda protegida del uso inadecuado de las propiedades.

Evidentemente no se acaba todo con usar la palabra Property, sino que también hay una filosofía detrás.

Es necesario que las propiedades puedan ser inicializadas, eso evidentemente no es ningún problema, puesto que en la declaración de la variable que contiene el dato se puede dar ya un valor por defecto de inicialización.

```
Namespace Ambito
  Public Class Clase
    Private Mensaje As String

    Public Property Texto() As String
      Get     ` devolución de su valor actual
        Return Mensaje
      End Get
      Set(ByVal Value As String)   ` captura del valor asignado
        Mensaje = Value
      End Set
    End Property
  End Class
End Namespace   ' Ambito
Module Principal
  Sub Main()
    ` Definición de un objeto que referencia a la clase
    Dim MiClase As New Ambito.Clase
    Dim OtraClase As New Ambito.Clase
    ` Asignación de valores
    MiClase.Texto = "Propiedad en MiClase  "
    OtraClase.Texto = "Propiedad en Otra Clase "
    ` Visualización de los valores
    Console.WriteLine(MiClase.Texto)
    Console.WriteLine(OtraClase.Texto)
    Console.ReadLine()
  End Sub
End Module
```

En el siguiente ejemplo podemos ver que hemos hecho cambios en el uso de la propiedad de la clase, la variable está inicializada, y además en la captura de su valor ha sido validada, con lo que la propiedad puede pasar por varios valores, o tener varios estados.

```
Namespace Ambito
  Public Class Clase
    Private Mensaje As String = "Valor por defecto"

    Public Property Texto() As String
      Get
        Return Mensaje
      End Get
      Set(ByVal Value As String)
        Select Case Value
          Case Is = ""
            Mensaje = "Inicialización por error"
          Case Else
            Mensaje = Value
        End Select
      End Set
    End Property

    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 objetos
    Dim MiClase As New Ambito.Clase
    ' visualización del contenido de la propiedad
    Console.WriteLine(MiClase.Texto)
    ' asignación de la propiedad
    MiClase.Texto = ""
    ' visualización del contenido de la propiedad
    Console.WriteLine(MiClase.Texto)
    ' asignación de la propiedad
    MiClase.Texto = "Propiedad de MiClase"
    ' visualización del contenido de la propiedad
    Console.WriteLine(MiClase.Texto)
    Console.ReadLine()
  End Sub
End Module
```

3.2 Tipos de propiedades, encapsulamiento.

La propiedad del ejemplo anterior es del tipo lectura y escritura, puede ser usada y asignarle valores, pero podemos declarar la propiedad de solo lectura, readonly, o de solo escritura, writeonly.

Esto es necesario para que de esa forma se pueda respetar una de las características de las clases, y es su encapsulamiento, es decir, protegidas a que el objeto que las usa pueda hacer un uso indiscriminado de la misma, ya que de esa forma se pueden ocultar o proteger en el sentido que nos interese, además de la validación que se pueda hacer en el bloque Set.

Cuando la propiedad es de ReadOnly, el bloque Set desaparece, y al revés, cuando es de WriteOnly, desaparece el bloque Get.

En el ejemplo siguiente hemos creado una propiedad nueva, Visible, de solo lectura.

```
Namespace Ambito
  Public Class Clase
    Private Mensaje As String = "Valor por defecto"
    Private Verse As Boolean = True

    Public Property Texto() As String
      Get
        Return Mensaje
      End Get
      Set(ByVal Value As String)
        Select Case Value
          Case Is = ""
            Mensaje = "Inicialización por error"
          Case Else
            Mensaje = Value
        End Select
      End Set
    End Property
    ' Esta propiedad es de solo lectura,
    ' carece del apartado Set, donde se recibe el
    ' valor asignado en el programa
    ' Pero si puede ser leída
    Public ReadOnly Property Visible() As Boolean
      Get
        Return Verse
      End Get
    End Property
  End Class
End Namespace   ' Ambito

Module Principal
  Sub Main()
    ' definición de la clase
    Dim MiClase As New Ambito.Clase
    ' visualización del contenido de la propiedad
    Console.WriteLine(MiClase.Visible)
    Console.ReadLine()
  End Sub
End Module
```

Probar en el ejemplo a eliminar, o comentar el bloque Set o Get.

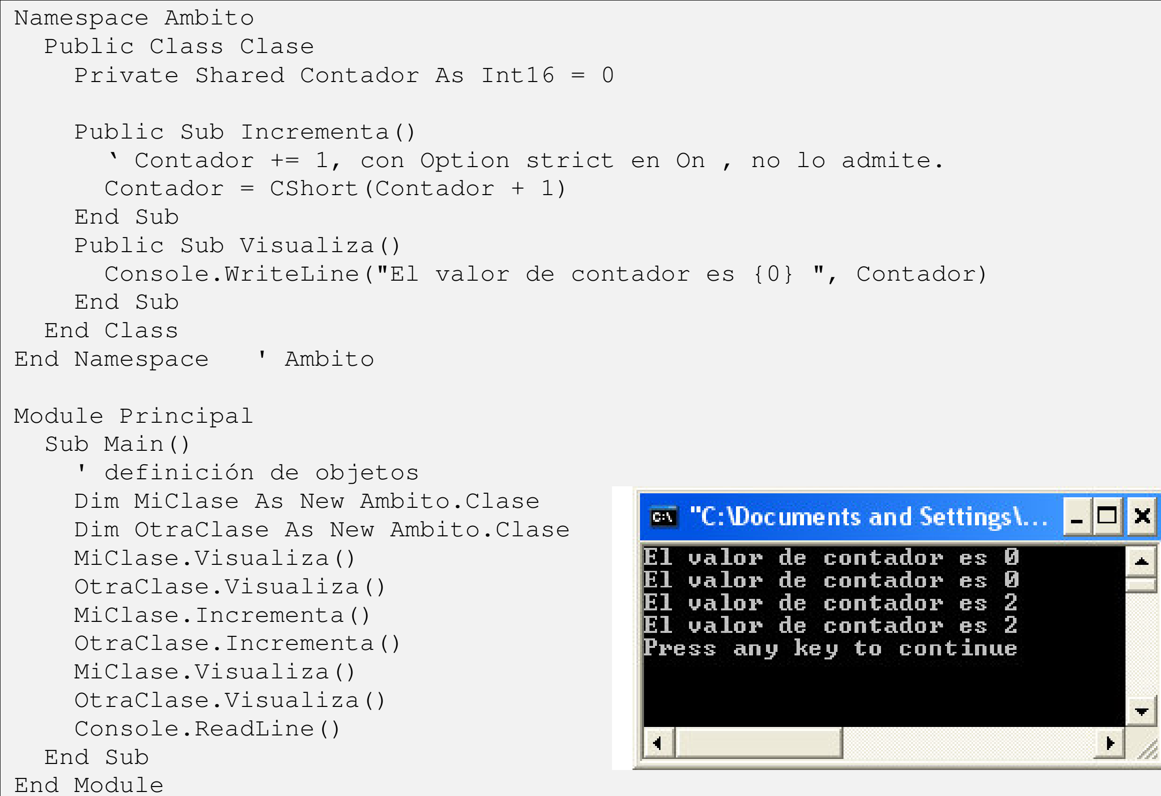
3.3 Variables en las clases, compartidas, no compartidas.

Las variables en las clases pueden ser compartidas, Shared, o no compartidas, sin Shared.

La diferencia es que son de la clase, cuando son Shared, es decir toman valor la primera vez que se hace una instancia de la clase y mantienen su valor en cada instancia de la misma.

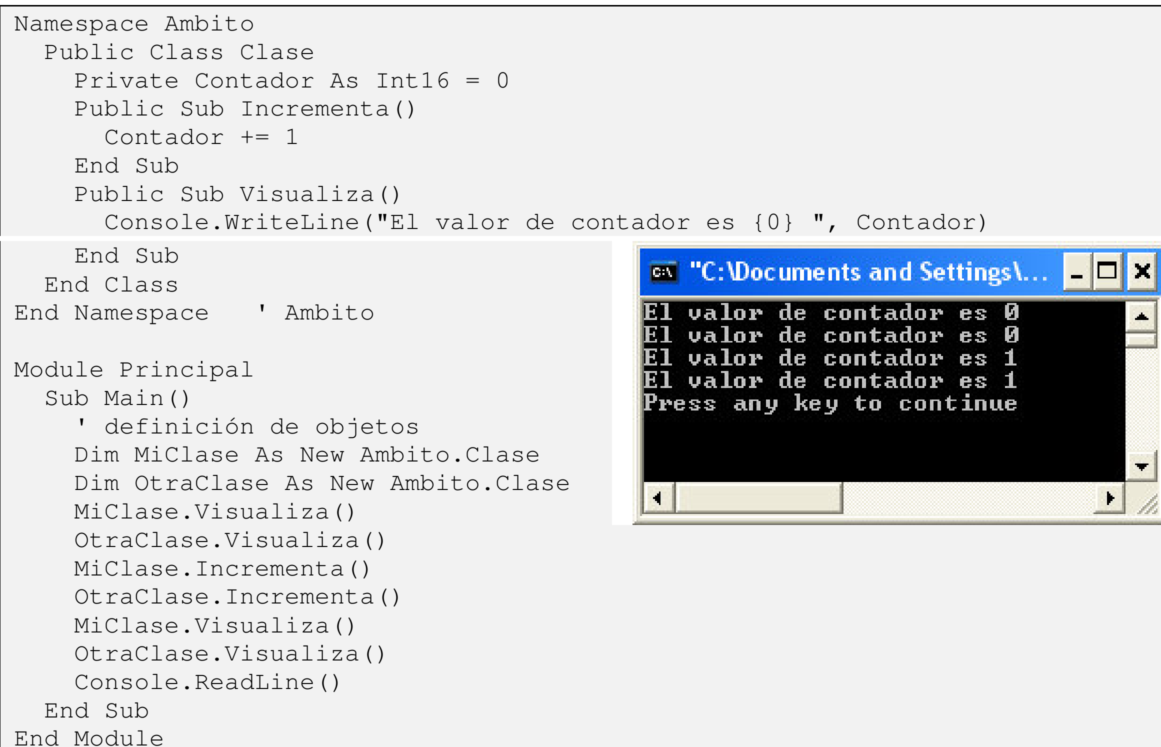
O no son Shared, y tienen un valor propio para cada una de las instancias de la clase.

El primer ejemplo es con Shared y su resultado es el que vemos en la ventana.



```
Namespace Ambito
  Public Class Clase
    Private Shared Contador As Int16 = 0

    Public Sub Incrementa()
      ` Contador += 1, con Option strict en On , no lo admite.
      Contador = CShort(Contador + 1)
    End Sub
    Public 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.Clase
    Dim OtraClase As New Ambito.Clase
    MiClase.Visualiza()
    OtraClase.Visualiza()
    MiClase.Incrementa()
    OtraClase.Incrementa()
    MiClase.Visualiza()
    OtraClase.Visualiza()
    Console.ReadLine()
  End Sub
End Module
```

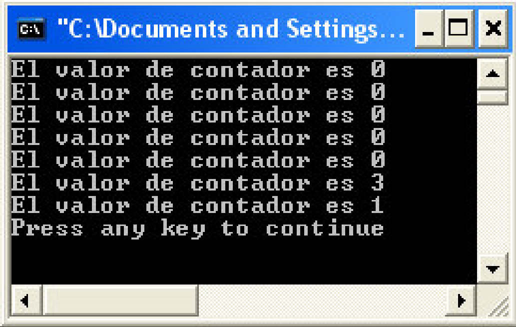

Sin embargo sin tocar una sola línea de código del Main, ni de la clase, solo con cambiar el modo de definición de la variable en la clase, el resultado es el siguiente.



```
Namespace Ambito
  Public Class Clase
    Private 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
End Namespace   ' Ambito

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

Como puede comprobarse la variable contador se ha convertido en propia de cada una de las instancias creadas de la clase, o no se comparte entre las distintas instancias de la clase, a pesar de haber ejecutado el método incrementa en dos ocasiones, una para cada instancia, el valor de contador es uno, si lo ejecutamos más veces cambiará en función de su uso.

```
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()
    ' 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()
    ' 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()
    ` solo ha incrementado una vez
    ` vale uno.
    MiClase.Visualiza()
    OtraClase.Visualiza()
    Console.ReadLine()
  End Sub
End Module
```



3.4 Propiedad por defecto.

En todas las clases se puede asignar una propiedad, una solo, por defecto.

```
Default Public Property Pordefecto(ByVal Indice as Integer)
```

La diferencia entre esa propiedad y las demás es que a la hora de su uso, no es necesario nombrarla sino que recibe el valor de forma automática, vamos es una cuestión de notación.

De todos modos parece mucho más lógico saber con que propiedad se está trabajando que no que se asigne el valor sobre una propiedad por defecto, en la que siempre puede quedar dudas, y más cuando hace mucho tiempo que se ha desarrollado el código.

El ejemplo que sigue posee una propiedad declarada por defecto y el uso de la misma, creemos que solo tiene como positivo que funciona.

En el ejemplo probar a quitar el argumento Index de la propiedad PorDefecto, y observará una peculiaridad de las propiedades por defecto.

Esta característica de propiedades con índice se comenta a continuación.

```
Namespace Ambito
  Class Clase
    ' Variable local para guardar el dato de la propiedad
    Private Variable As String
    ' Declarar propiedad por defecto
    Default Public Property PorDefecto(ByVal Index As Integer) As String
      Get
        Return Variable
      End Get
      Set(ByVal Value As String)
        Variable = Value
      End Set
    End Property
  End Class
End Namespace

Module Ejemplo
  Sub main()
    Dim C As New Ambito.Clase
    ' Acceso estándar a la propiedad
    C.PorDefecto = "Un valor"          ' Asignación
    Console.WriteLine(C.PorDefecto(0))    ' Visualización

    ' Uso del acceso a la propiedad por defecto
    C(1) = "Otro valor"       ' Asignación
    Console.WriteLine(C)   ' Visualización
    Console.ReadLine()
  End Sub
End Module
```

3.5 Propiedades con índices.

Esta característica por lo menos al principio parece muy interesante, pues de hecho permite trabajar con varios datos en una propiedad permitiendo direccionarles a través de un índice.

El ejemplo que sigue se ve como es el código de una clase con una propiedad con índice.

Si repasamos el código y lo comparamos con el del ejemplo anterior, veremos que la única diferencia, a parte del nombre de la propiedad es el de la falta de la palabra Default delante de la propiedad.

```
Namespace Ambito
  Class Clase
    ' Variable local para almacenar los datos
    Private ValorPropiedad As String()
    ' Propiedad por dfecto
    Public Property PropiedadConIndice(ByVal Index As Integer) As String
      Get
        Return ValorPropiedad(Index)
      End Get
      Set(ByVal Value As String)
        If ValorPropiedad Is Nothing Then
          ' The array contains Nothing when first accessed.
          ReDim ValorPropiedad(0)
        Else
          ' Re-dimension the array to hold the new element.
          ReDim Preserve ValorPropiedad(UBound(ValorPropiedad) + 1)
        End If
        ValorPropiedad(Index) = Value
      End Set
    End Property
  End Class
End Namespace
```

```
Module Ejemplo
  Sub main()
    Dim C As New Ambito.Clase
    ' Este ejemplo es de acceso estándar a la propiedad
    C.PropiedadConIndice(0) = "Elemento cero cero"
    C.PropiedadConIndice(1) = "Elemento uno"
    C.PropiedadConIndice(2) = "Elemento dos dos "
    Console.WriteLine(C.PropiedadConIndice(0))
    Console.WriteLine(C.PropiedadConIndice(1))
    Console.WriteLine(C.PropiedadConIndice(2))
    C.PropiedadConIndice(0) = "Elemento cero"
    C.PropiedadConIndice(1) = "Elemento uno uno"
    C.PropiedadConIndice(2) = "Elemento dos "
    Console.WriteLine(C.PropiedadConIndice(0))
    Console.WriteLine(C.PropiedadConIndice(1))
    Console.WriteLine(C.PropiedadConIndice(2))
    Console.ReadLine()
  End Sub
End Module
```

En cuanto a su utilidad, el tiempo lo dirá, pero está claro que si se desconoce, no podemos utilizarla.