

# 1. Clases, Eventos.

## 1.1 Introducción.

La utilización normal de un objeto es la que se da cuando desde el programa se crea una instancia del objeto y se usan los métodos del mismo.

Eso es lo habitual, pero se puede hacer al contrario, que sea el objeto el que active un evento del programa, o dicho de otra forma, al revés.

Entonces lo que hay que ver es de que forma se puede definir un evento dentro de nuestro programa, o de nuestra clase para que ésta sea la que llame al programa principal.

La definición de un evento se realiza utilizando la palabra reservada Delegate en la declaración del procedimiento que se encarga de la gestión del evento.

```
Public Delegate Sub MiEvento (ByVal Argumento as Tipo)
```

## 1.2 Utilización del evento.

Vista la definición del evento dentro del código, su utilización ya depende de su declaración dentro de la clase en la que deseemos utilizarla.

La utilización se divide en varios pasos.

Definición del evento.

Declaración del evento en la clase.

Escritura del procedimiento que alberga del código del evento.

Asociación del evento con el código que se va a ejecutar.

Lanzar el evento desde el sitio que proceda.

Definición

```
Public Delegate Sub MiEvento (ByVal MiArgu As Int16)
```

Declaración del evento en la clase.

```
' Declaración de uso del evento  
Event Evento As MiEvento
```

Escritura del código del evento.

```
Sub CodigoDelEvento (ByVal Valor As Int16)  
    ' Código a ejecutar en el evento,  
    ' valor es el valor que se envía desde su generación  
    ' con la instrucción RaiseEvent  
    Console.WriteLine("Texto desde eventillo {0} ", Valor)  
End Sub
```

Asociación del evento al código del evento.

```
' Asociación del Evento de MiClase al código del evento.  
AddHandler Evento, New MiEvento (AddressOf CodigoDelEvento)
```

En el ejemplo está en el constructor, New, pero también lo hemos probado en el Main y funciona.

Lanzar el evento.

```
RaiseEvent Evento (24)
```

Se han dado distintos valores en cada llamada, para poder observar desde donde se produce el evento.

En el ejemplo podemos ver el código completo.

Hemos probado a poner el código del evento en el Main, y también funciona, al final lo hemos dejado todo en la clase porque parece que es como más organizado, cuando se aplique en la realidad, veremos que es lo que debe ocurrir, si eso influye o no, pero lo anotamos.

```
' Definición del evento
Public Delegate Sub MiEvento(ByVal MiArgu As Int16)

Namespace Ambito
  Public Class Ejemplo
    ' Declaración de uso del evento
    Event Evento As MiEvento

    Public Sub New()
      ' Asociación del Evento de MiClase al código del evento.
      AddHandler Evento, New MiEvento(AddressOf CodigoDelEvento)
      ' Se lanza el evento, se provoca, en la generación
      RaiseEvent Evento(22)
    End Sub

    Public Sub Inicio()
      Console.WriteLine("Hola, lanzando el evento")
      ' Se lanza el evento, se provoca
      RaiseEvent Evento(24)
    End Sub

    Public Sub Metodo()
      Console.WriteLine("Hola desde método, lanzando otra vez el evento")
      ' Se lanza el evento, se provoca
      RaiseEvent Evento(26)
    End Sub

    Sub CodigoDelEvento(ByVal Valor As Int16)
      ' Código a ejecutar en el evento,
      ' valor es el valor que se envía desde su generación
      ' con la instrucción RaiseEvent
      Console.WriteLine("Texto desde el código del evento {0} ", Valor)
    End Sub
  End Class
End Namespace ' Ambito

Module Principal
  Sub Main()
    ' Definición de la clase
    Dim MiClase As Ambito.Ejemplo = New Ambito.Ejemplo
    MiClase.Inicio()
    MiClase.Metodo()
    Console.ReadLine()
  End Sub
End Module
```

Hay otra posibilidad, con el mismo resultado que antes, pero es más breve y quizás más clara. Se quita la declaración de evento del principio, Public Delegate, y en la declaración del evento se realiza directamente la declaración de los argumentos del mismo.

```
' Declaración de uso del evento
Event Evento(ByVal Valor As Int16)
```

En cuanto a la asociación, también hay un cambio, pero queda más claro, antes era

```
' Asociación del Evento de MiClase al código del evento.  
AddHandler Evento, New MiEvento(AddressOfCodigoDelEvento)
```

Y ahora es

```
' Asociación del Evento de MiClase al código del evento.  
AddHandler Evento, AddressOfCodigoDelEvento
```

Que es más sencillo y fácil de recordar.

El ejemplo final queda así.

```
' Definición del evento, ... desaparece ...  
' Public Delegate Sub MiEvento(ByVal MiArgu As Int16)  
  
Namespace Ambito  
    Public Class Ejemplo  
        ' Declaración de uso del evento  
        Event Evento(ByVal Valor As Int16)  
  
        Public Sub New()  
            ' Asociación del Evento de MiClase al código del evento.  
            AddHandler Evento, AddressOfCodigoDelEvento  
            ' Se lanza el evento, se provoca, en la generación  
            RaiseEvent Evento(22)  
        End Sub  
  
        Public Sub Inicio()  
            Console.WriteLine("Hola, lanzando el evento")  
            ' Se lanza el evento, se provoca  
            RaiseEvent Evento(24)  
        End Sub  
  
        Public Sub Metodo()  
            Console.WriteLine("Hola desde método, lanzando otra vez el evento")  
            ' Se lanza el evento, se provoca  
            RaiseEvent Evento(26)  
        End Sub  
  
        Sub CodigoDelEvento(ByVal Valor As Int16)  
            ' Código a ejecutar en el evento,  
            ' valor es el valor que se envía desde su generación  
            ' con la instrucción RaiseEvent  
            Console.WriteLine("Texto desde el código del evento {0} ", Valor)  
        End Sub  
    End Class  
End Namespace ' Ambito  
  
Module Principal  
    Sub Main()  
        ' Definición de la clase  
        Dim MiClase As Ambito.Ejemplo = New Ambito.Ejemplo  
        MiClase.Inicio()  
        MiClase.Metodo()  
        Console.ReadLine()  
    End Sub  
End Module
```

## 2. Como programar una clase.

### 2.1 Introducción.

No se puede negar que las clases son una evolución natural de la forma de programar, o sea que negar la evidencia es del todo inútil

En principio es muy atrevido escribir un capítulo con éste tema, pero no se pretende decir a nadie como tiene que hacer las cosas, sino simplemente indicar cuales son los criterios que se pueden seguir, o en que orden se ejecutan los métodos y se usan las variables, propiedades, para a partir de ahí crear lo que podría ser un orden más o menos adecuado, evidentemente desde lo que llevamos visto, el tiempo muchas veces te desdice, o te reafirma.

Tampoco vamos a entrar en todas las posibilidades de las clases, sino se convertiría en un apartado muy extenso, y la pretensión es la de sentar un punto de partida, para que a partir de ahí se siga progresando.

Tal como se ha desarrollado Studio Net el uso de clases escritas en otros lenguajes no debe resultar ningún problema.

### 2.2 Utilización de las clases.

La utilización de las clases con los pequeños ejemplos que hemos visto quizás no queda muy clara, pero hay que resaltar que su destino es el de crear objetos con las mismas, y que esos objetos son autónomos unos con respecto a otros.

¿Cuando definir una clase?, en principio eso va a ser un problema, pero desde luego en Studio Net todo esta agrupado o integrado por clases, otra cosa es el código que tengamos que escribir nosotros. Eso es otra historia. Pero ya lo iremos aprendiendo con el tiempo.

En principio una clase sería práctico crearla cuando deseamos crear lo que anteriormente definíamos como una librería.

También si deseamos crear una agrupación de código relacionado entre si, y que el fuente quede protegido. Hay que tener en cuenta que las empresas de programación de esta forma pueden tener objetos desarrollados que se reutilizan de una a otra aplicación, y que son o están depurados.

Una de las ventajas de las clases es que en contra de las librerías, o en algunos casos de las librerías, según el lenguaje, el código no es accesible por el usuario de la misma.

Hay más casos en los que se puede definir una clase, como son las denominadas Interface, o las equivalente a los tipos, que por cuestiones de rendimiento así se aconseja. Pero no es este el motivo principal del tema, por lo que solo dejar constancia de dicha situación para que quien lo desee pueda buscar ampliar dichos contenidos.

### 2.3 Como escribir una clase.

Cuando usamos una clase, lo que se hace es siempre seguir los siguientes pasos.

Declarar un objeto de esa clase.

Eso implica que se ejecute por defecto un método New, Constructor.

Por lo tanto los datos que nos interese controlar o las acciones a ejecutar en esa fase deben estar escritas en ese método.

Asignar las características a las propiedades de la clase.

Pero puede que no se le asigne ninguna propiedad de las existentes.

Desarrollar los métodos de la clase.

Utilizar la clase con los objetos que se declaren de la misma.

Finalizar el uso del objeto.

Se ejecuta el método destructor de la clase, Finalize, y se liberan recursos.

En base a esos pasos debemos programar el contenido de nuestra clase.

### 2.4 Declarar un objeto de esa clase.

Ese paso es el más sencillo de todos.

```
Dim Objeto as New ClaseDefinidaExistente
```

O bien se puede escribir

```
Dim Objeto as ClaseDefinidaExistente  
Objeto = New ClaseDefinidaExistente
```

Cualquiera de las dos sintaxis es válida, aunque la más utilizada es la primera.

Ahora hay que escribir la clase, y hemos de escribir una clase eficiente y robusta contra el mal uso de la misma.

Si por el motivo que fuese deseamos que no se pueda declarar un objeto de la clase que nosotros escribimos sin que se ejecute el control adecuado en su creación hay que proteger el método New.

Para ello lo declaramos Private sin argumentos y solo podrá ejecutarse en la versión que nosotros escribamos.

El ejemplo que sigue no permite declarar un objeto de la clase, sin que se le pase como argumento un valor de tipo string en el momento de la definición de dicho objeto.

Por lo tanto es imposible crear un objeto de esa clase sin más.

```
Public Class Clase  
    Dim Variable As String = "Variable inicializada"  
    Dim VariablePropiedad As String = "Propiedad inicializada"  
  
    Private Sub New()  
        ` esto impide la creación de un objeto .  
    End Sub  
  
    Public Sub New(ByVal Valor As String)  
        VariablePropiedad = Valor  
    End Sub  
End Class
```

Este ejemplo nos obliga a declarar el objeto con la siguiente sintaxis.

```
Dim Objeto As New Clase("Iniciación")
```

Impide que se pueda utilizar

```
Dim Objeto As New Clase
```

Por lo que si nosotros añadimos en el método New de la clase el siguiente código

```
Private Sub New()  
End Sub  
  
Public Sub New(ByVal Valor As String)  
    Select Case Valor <> ""  
        Case True  
            VariablePropiedad = Valor  
    End Select  
End Sub
```

Si al inicializar la clase no se recibe un valor adecuado la clase quedaría inicializada con el valor por defecto de la propiedad, sin asumir el valor recibido.

Si la clase es heredada desde otra clase la condición se mantiene, vemos un ejemplo.

```
Public Class Heredera
    Inherits Clase    ` Heredar la clase del ejemplo anterior

    Public Sub New()
        ` El primer paso es llamar al New de la clase origen, y proporcionar
        ` un valor de inicialización.
        MyBase.New("Valor desde la clase heredera")
    End Sub
End Class
```

Todo lo anterior se puede probar desde el código de éste formulario que tiene dos Label y un Button

```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs) _
        Handles Button1.Click
        ` Button para probar
        Dim Objeto As New Clase("Valor de inicialización")
        Label1.Text = Objeto.Propiedad
        Dim Otro As New Heredera
        Label2.Text = Otro.Propiedad
    End Sub

    Private Sub Button2_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs) _
        Handles Button2.Click
        ` Button para finalizar la ejecución.
        Me.Dispose()    ` Libera los recursos asignados a los objetos
        Me.Finalize()
        Me.Close()
    End Sub
End Class
```

Como vemos en los ejemplos, la adecuada utilización del método New es importante.

Por lo tanto ya se ha cubierto una primera etapa que es la de controlar la creación de un objeto de nuestra clase de forma incontrolada.

El siguiente paso es la declaración de las variables.

En el momento de declarar las variables de la clase nos encontraremos con dos tipos de variables.

Aquellas que se refieren a datos de la clase, y a las que se refieren a las propiedades que se ven desde el exterior de la clase.

Las variables normales, les podemos dar el tratamiento que habitualmente le damos a las variables de cualquier programa, pero las que se emparejan con las propiedades lo lógico es inicializarlas al valor que nos interesa, con el fin de que la clase a la hora de inicializarse no sea imprescindible asignar valor a las propiedades, si no que estas ya dispongan de un valor que pueda ser operativo, y solo sea necesario ajustar para personalizar el objeto.

En el ejemplo anterior podemos observar los dos tipos de variables.

## 2.5 Asignar las características a las propiedades de la clase.

Como hemos comentado, a la hora de inicializar la clase, las variables que se emparejan con las propiedades las vamos a inicializar, pero evidentemente el usuario puede querer personalizar o necesitar ajustar los valores del objeto.

Por lo tanto necesitará cambiar los valores de las propiedades.

Los valores de las propiedades se asignan en el apartado Set de la propiedad en cuestión y es ahí donde hay que escribir el código adecuado para su validación, como de costumbre es un apartado que nunca hay que olvidar, no solo se ha de asignar el dato, sino que también hay que validarlo previamente.

```
Public Class Clase
    Dim Variable As String = "Variable inicializada"
    Dim VariablePropiedad As String = "Propiedad inicializada"

    Private Sub New()
    End Sub

    Public Sub New(ByVal Valor As String)
        VariablePropiedad = Valor
    End Sub

    Public Property Propiedad() As String
        Get
            Return VariablePropiedad
        End Get

        Set(ByVal value As String)
            Select Case value <> ""
                Case True
                    VariablePropiedad = value
            End Select
        End Set
    End Property
End Class
```

Vemos en el ejemplo como se recibe o se trata una propiedad, esa propiedad es de lectura y escritura, pero podemos declararlas como ReadOnly o WriteOnly.

La parte del Get es lectura por parte del Objeto.

La parte del Set es de escritura por parte del Objeto.

Por lo tanto nosotros hemos de escribir nuestro código de control en la parte del Set, que es donde recibimos los datos desde el objeto declarado con la clase.

Cuando declaramos una propiedad de solo lectura, la parte de escritura se elimina, y al revés cuando es de solo escritura.

## 2.6 Desarrollar los métodos de la clase.

Está claro que éste no es un orden inamovible, y que se irán creando los métodos y las propiedades a medida que se vaya desarrollando la clase.

Los métodos nos vamos a encontrar que necesitarán a su vez de datos para su ejecución.

Que criterio seguir a la hora de asignar los datos de los métodos, usar una propiedad o un argumento en la línea de llamada al método.

Un criterio puede ser el siguiente.

Cuando se usa una propiedad y se valida la asignación del dato, no se está ejecutando ningún código que podamos decir que sea ejecutivo, que realiza alguna acción, solo se controla su dato y ya está.

Cuando se asigna un valor en la llamada a un método, ese método va a ejecutar una acción, por lo tanto, ya podemos tener un criterio.

Así que en el método solo tenemos que recabar aquella información que sea imprescindible para la ejecución de ese método.

Otro apartado a la hora de crear los métodos, es si estos han de ser procedimientos a funciones.

Un criterio a seguir es que si el método solo es una acción a ejecutar y no devuelve nada, podemos crear un procedimiento, pero si ese método como consecuencia de su ejecución ha de devolver un dato, entonces debemos crear una función.

Por lo tanto, tenemos otro tema resuelto.

La siguiente cuestión que nos aparece, es cuando un método puede resolver un problema desde distintos puntos de partida.

Eso nos va a llevar a que seguramente tengamos también distintos juegos de datos iniciales, con lo cual la solución es evidente, utilizar la sobrecarga.

Creamos distintos métodos con el mismo nombre y con las líneas de entrada de argumentos adecuada a cada situación prevista.

## 2.7 Utilización de los objetos de una clase.

La utilización de los objetos declarados de una clase, dependerá de la implementación de la clase en cuestión, que es la que le proporciona los recursos a los objetos.

La utilización del objeto, una vez declarado, será

Asignar valores a las propiedades, si estas lo requieren o lo necesita el código, y si no se ha hecho ya en la declaración del objeto.

Hacer uso de los métodos disponibles de la clase en el momento adecuado.

Finalizar el objeto, si procede, o dejar que la salida del procedimiento acabe con el mismo.

El uso dependerá evidentemente del tipo de clase que se haya implementado.

La codificación del uso de la clase puede hacerse bajo dos estilos de código.

El primero podría decirse que es el habitual.

Definir

Usar

Y no realizar acciones específicas de finalización, o utilizar la clase GC.

```
Dim Objeto As New Clase(6)
Objeto = Nothing
GC.Collect()
```

Con el método Collet de la clase GC, se obliga a la recuperación de los recursos libres del sistema.

La clase GC, controla el recolector de elementos no utilizados del sistema, es un servicio que reclama de forma automática la memoria que no se utiliza.

Los métodos de esta clase influyen en el momento en que se realiza la recolección de elementos no utilizados de un objeto y en el momento en que se liberan los recursos asignados por un objeto.

Las propiedades de esta clase proporcionan información sobre la cantidad de memoria total disponible en el sistema y la categoría de edad, o generación, de la memoria asignada a un objeto.

El segundo, y sobre todo si se consumen muchos recursos del sistema, es utilizar la instrucción Using, End Using.

Con esta instrucción se garantiza la recuperación de los recursos al finalizar el uso de los objetos.

```
Using Objeto As New Clase(7)
    MsgBox("El valor de la propiedad es " & Objeto.Propiedad & ".")
End Using
```

## 2.8 Secuencia de uso de un objeto.

El objeto creado como instancia de una clase, cuando se libera ejecuta el método finalize de la clase.

Ese método se ejecuta siempre de forma predeterminada antes de liberar el objeto.

Por lo tanto todas aquellas acciones que sean imprescindibles realizar antes de que se destruya el objeto se deben realizar en ese método.

Por ejemplo realizar el cierre de archivos, o de una conexión con base de datos.

Uno de los apartados es la liberalización de recursos, conviene que probemos el siguiente ejemplo con y sin la instrucción de Me.Dispose, y observar la diferencia en tiempo al finalizar el cierre del formulario cuando se usa éste método.

Hay que pulsar el Button1 para crear el objeto y después el Button2.

Este es el código de la clase, mínima expresión de clase.

```
Public Class Clase
    Dim Variable As String = "Variable inicializada"
    Dim VariablePropiedad As String = "Propiedad inicializada"

    Private Sub New()
    End Sub

    Public Sub New(ByVal Valor As String)
        VariablePropiedad = Valor
    End Sub
End Class
```

Y este el código del formulario, solo hay un Label y dos Button.

```
Public Class Form1
    Private Sub Button1_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs) _
        Handles Button1.Click
        Dim Objeto As New Clase("Inicialización")
    End Sub

    Private Sub Button2_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs)_
        Handles Button2.Click
        Me.Dispose()
        Me.Finalize()
        Me.Close()
    End Sub
End Class
```

Esto es referente a la finalización del programa en si.

Para finalizar el uso de un objeto se establece a Nothing, y a partir de ahí la referencia al mismo no es posible.

Podemos hacer lo siguiente.

```
Objeto = Nothing
```

También podemos preguntar por

```
If Objeto IsNot Nothing Then ...
```

Lo que nos permite saber si un objeto sigue haciendo referencia a algún recurso.

El ejemplo que sigue expresa de forma bastante efectiva la secuencia y el funcionamiento de los métodos Dispose y Finalize, es un ejemplo en modo consola, para su correcto funcionamiento hay que poner en las propiedades de proyecto como elemento inicial el Sub Main.

Este es el código principal del ejemplo, donde se crean los objetos para la prueba.

```
Module Principal
    Sub Main()
        ' Demostración de como Using llama al método dispose
        Using Objeto As New ClaseHeredera(6)
            MsgBox("El valor de la propiedad después de la " & _
                "inicialización es " & Objeto.Propiedad & ".")
        End Using
    End Sub
End Module
```

```

MsgBox("Punto intermedio entre demos.", MsgBoxStyle.Information)

' Demostración de como el Recolector de objetos
' del CLR, Common Language Runtime, llama al método finalize.

Dim OtroObjeto As New ClaseHeredera(6)
OtroObjeto = Nothing
GC.Collect()
End Sub
End Module

```

Y este es el módulo con el código de las clases del ejemplo.

```

Module ClaseDos
    Public Class ClaseBase
        Sub New()
            MsgBox("ClaseBase inicializándose con el método New.")
        End Sub

        Protected Overrides Sub Finalize()
            MsgBox("ClaseBase haciendo shutdown con el método Finalize.")
            MyBase.Finalize()
        End Sub
    End Class

    Public Class ClaseHeredera
        Inherits ClaseBase
        Implements IDisposable
        Private ValorPropiedad As Integer

        Sub New(ByVal Valor As Integer)
            ' La llamada al método New de la clase base es obligado.
            MyBase.New()
            MsgBox("Clase heredera se inicializa con el método New.")

            ValorPropiedad = Valor
        End Sub

        Property Propiedad() As Integer
            Get ' Entrada
                CheckIfDisposed()
                Propiedad = ValorPropiedad
            End Get
            Set(ByVal Value As Integer) ' Salida
                CheckIfDisposed()
                ValorPropiedad = Value
            End Set
        End Property

        Protected Overrides Sub Finalize()
            MsgBox("Clase heredera haciendo shutdown con el método Finalize.")
            Dispose(False)
        End Sub

        ' No añadir Overridable a éste método.
        Public Overloads Sub Dispose() Implements IDisposable.Dispose
            MsgBox("Clase heredera haciendo shutdown con el método Dispose.")
            Dispose(True)
            GC.SuppressFinalize(Me)
        End Sub
    End Class
End Module

```

```

Private disposed As Boolean = False
Public Sub CheckIfDisposed()
    If Me.disposed Then
        Throw New ObjectDisposedException(Me.GetType().ToString, _
            "Este objeto ha sido liberado.")
    End If
End Sub

Protected Overridable Overloads Sub Dispose( _
    ByVal disposing As Boolean)
    MsgBox("Clase heredera haciendo shutdown con el " & _
        "método Dispose sobrecargado, OverLoad")

    ' Código de tareas finales
    If Not Me.disposed Then
        If disposing Then
            ' Dispose de cualquier recurso utilizado.
        End If
        ' Dispose de cualquier recurso.

        ' Llamar a MyBase.Finalize si es una clase derivada ,
        ' y la clase base no ejecuta Dispose.
        MyBase.Finalize()
    End If
    Me.disposed = True
End Sub
End Class
End Module

```

## 2.9 Recuperar los recursos utilizados que están libres, clase GC.

La clase GC está diseñada para la gestión de los recursos liberados en memoria sin utilizarse, dispone de métodos para la gestión y recuperación de dichos recursos.

El recolector de elementos no utilizados realiza un seguimiento de los objetos asignados en la memoria administrada, y los reclama. De forma periódica, el recolector de elementos no utilizados reclama la memoria asignada a los objetos para los que no existen referencias válidas.

La recolección de elementos no utilizados se produce de forma automática, cuando una solicitud de memoria no puede satisfacerse utilizando la memoria libre que queda disponible.

La recolección de elementos no utilizados consta de los siguientes pasos:

El recolector de elementos no utilizados busca los objetos administrados a los que se hace referencia en el código administrado.

El recolector de elementos no utilizados intenta finalizar los objetos a los que no se hace referencia.

El recolector de elementos no utilizados libera los objetos a los que no se hace referencia y reclama la memoria utilizada por estos objetos.

Durante la recolección de elementos no utilizados, el recolector no liberará un objeto si encuentra una o varias referencias al mismo en el código administrado.

El recolector de elementos no utilizados no reconoce, sin embargo, las referencias a objetos desde el código no administrado y puede liberar objetos que se estén utilizando exclusivamente en código no administrado, a menos que se le impida hacerlo de forma explícita.