

Comencemos a programar con
VBA - Access

Entrega **20**

Más sobre Clases y Objetos
(1)

Recordemos lo expuesto hasta ahora sobre las clases

Allá por la entrega 07 definimos una clase en VBA como un conjunto de código, que contiene además de datos, procedimientos para manejarlos y que sirve “como molde” para la creación de Objetos.

En un “gran esfuerzo” algorítmico, definimos nuestra primera clase **CPersona**, que incluía unas variables y dos funciones de tipo público.

Esta era una clase muy elemental, trabajando con propiedades que simplemente eran variables declaradas como públicas y planteada únicamente para “abrir boca” en el mundo de las clases, lo que ya fue suficiente para ver algunas de las ventajas de la utilización de clases frente a otro tipo de estructuras, como las variables registro **Type**.

En la entrega 08 avanzamos un paso más y jugamos con las asignaciones de los objetos creados con las clases, y vimos que podíamos asignar el mismo objeto a dos variables objeto diferentes.

También trabajamos con el objeto **Collection**, añadiendo objetos al mismo.

Pero ¿qué es realmente una clase?

Ya lo hemos dicho, una clase es algo tan sencillo como un conjunto de código, que contiene además de datos, procedimientos para manejarlos, y con el que se pueden crear Objetos.

La clase es el código, y el Objeto creado es el llamado **Ejemplar de la clase**.

Desde el punto de vista de Visual Basic, una clase es un módulo de código que sirve para crear unas estructuras a las que llamamos Objetos.

Estas estructuras, no sólo contendrán datos, a los que se podrá acceder, además pueden poseer funciones y procedimientos, e incluso producir eventos.

El conjunto de datos, procedimientos y eventos recibe el nombre de **Miembros de la clase**.

En esta entrega, vamos a trabajar más a fondo con las clases.

- Aprenderemos a crear propiedades de lectura y escritura, aprovechando el concepto de **Encapsulación**.
- Veremos cómo generar y capturar **Eventos**

En la siguiente entrega veremos más ejemplos sobre la utilización de clases y además

- Tocaremos el tema de las **Interfaces**.
- Analizaremos conceptos como **Herencia y Polimorfismo** y, aunque no estén implementados en las clases de VBA, veremos cómo podemos en cierta forma emularlos.
- También veremos cómo crear un **Constructor** de Objetos, emulando la **Sobrecarga** del mismo.

Vemos que han aparecido una serie de conceptos, marcados en negrita, y que hasta ahora nos pueden resultar totalmente nuevos.

Encapsulación

Es la capacidad de los objetos para ocultar una serie de datos y procedimientos, de forma que para utilizarlos se obliga a acceder a ellos mediante unos procedimientos específicos, llamados **Propiedades** y otros procedimientos llamados **Métodos de la clase**.

Como veremos, la gran ventaja que tienen las propiedades es que permite filtrar los datos que van a manejarse pudiendo evitarse la asignación de datos incorrectos y teniendo la posibilidad de reaccionar y avisar al usuario si se ha cometido una incorrección..

Una parte de los datos se guardan en el propio objeto sin que sea posible acceder a ellos directamente. Sólo es posible manejarlos mediante el uso de las propiedades, teniendo que sujetarse a los filtros que incluya en ellas el programador.

A esta capacidad de proteger y ocultar los datos básicos del objeto es a lo que se llama **Encapsulación**.

Una clase podemos abordarla desde dos puntos de vista:

- Como programador de la clase
- Como usuario de la clase

El programador de la clase es el que define qué propiedades va a tener, de qué tipo y cómo se va a comportar.

El usuario de la clase, ya sea el que la ha diseñado u otra persona, va a utilizarla para crear objetos que le faciliten una serie de tareas de programación asignando valores concretos a las propiedades y utilizando sus procedimientos, llamados **métodos de la clase**.

Antes de seguir: consideraciones previas sobre las clases

Puede que al enfrentarse a la utilización de las clases, lo haga sin tener clara su verdadera utilidad. Incluso puede que le resulten farragosas, y que la curva de aprendizaje de las mismas, le haga perder el interés por ellas.

Si vd. fuera una de las muchas personas que no creen en la utilidad del uso de las clases, debería considerar los siguientes hechos.

- Es difícil que miles de programadores eficientes y convencidos de sus ventajas, estén totalmente equivocados. Sí; ya se que éste es un argumento muy endeble, y me viene ahora a la memoria las moscas y sus aficiones culinarias...
- Las clases permiten enfocar el desarrollo de un programa utilizando elementos más próximos y asimilables al mundo real.
- Permiten racionalizar el desarrollo de un proyecto pudiendo utilizar una planificación más metódica y racional, así como facilitar el mantenimiento posterior.
- La Programación Orientada a Objetos nació teniendo entre sus objetivos la reutilización del código, esto permite que cada vez los desarrollos sean más eficientes, de mejor calidad y más competitivos.
- No es algo tan novedoso, hacia principios de los años 70 ya existían lenguajes que trabajaban con este Paradigma, superando los conceptos de la Programación Estructurada que había sido hasta entonces la metodología a seguir.
- Al día de hoy, y a corto / medio plazo, VBA es un lenguaje que se seguirá utilizando en empresas que mantengan a Visual Studio 6 como plataforma de desarrollo, en las herramientas de Office, en desarrollos con Access, y en todo un abanico de software de multitud de fabricantes, y a pesar de las limitaciones que VBA presenta en la Programación Orientada a Objetos, la utilización de estas técnicas le supondrá una mejora considerable en sus desarrollos.
- No obstante podemos decir que la plataforma Net se está imponiendo sobre el resto de sus competidores. Por ejemplo, Visual Basic dejará pronto de ser soportada por Microsoft, y los programadores que quieran tener un futuro asegurado deben ya

empezar a plantearse muy seriamente el pasar a la plataforma Net. La buena noticia es que VB.Net, en su sintaxis, guarda un fuerte parecido con VBA.

- ❑ Los conceptos que rodean a la utilización de las clases y objetos, es probablemente el obstáculo más difícil para un programador, digamos “estándar” de Visual Basic, que apenas hace uso de las limitadas, pero a pesar de ello potentes, posibilidades de VBA en la programación orientada a objetos. Me atrevería a afirmar que casi todo lo que pueda aprender en esta entrega le servirá en el futuro con VB.Net, haciendo que su curva de aprendizaje sea mucho más suave. El dominio en el manejo de las clases y la Programación Orientada a Objetos, le tiende un puente que le facilitará el salto a lenguajes como VB.Net y C#.
- ❑ Tenga también en cuenta que la plataforma Net trabaja de forma exhaustiva con las clases. Hay que usarlas de forma obligatoria para el desarrollo de cualquier programa. En .Net hasta los módulos son en sí mismos clases, aunque no se declaren como tales.

Podría argumentar muchas más razones, pero prefiero centrarme en los temas que atañen a esta entrega.

Propiedades

Para estudiar qué son las propiedades, veamos la clase **CPersona** que vimos en el capítulo 7 (he eliminado las líneas de control de fecha, de la clase original, para ver mejor el ejemplo didáctico).

Os recuerdo que para crear una clase, usaremos la opción de menú **[Insertar] > [Módulo de clase]**, y en este caso, pondremos a la propiedad **Name** de la ventana Propiedades el valor **CPersona**.

```
Option Explicit

Public Nombre As String
Public Apellido1 As String
Public Apellido2 As String
Public FechaNacimiento As Date
Public Telefono As String

Public Function Edad() As Long
    Edad = (Date - FechaNacimiento) / 365.2425
End Function

Public Function NombreCompleto() As String
    NombreCompleto = Nombre _
        & " " & Apellido1 _
        & " " & Apellido2
End Function
```

Podemos, por ejemplo crear un objeto llamado **Empleado**.

Para ello, en un módulo estándar escribimos lo siguiente:

```
Public Sub PruebaCPersona()  
    Dim Empleado As New CPersona  
    With Empleado  
        .Nombre = "Antonio"  
        .Apellido1 = "López"  
        .Apellido2 = "Iturriaga"  
        .FechaNacimiento = #12/24/1980#  
        Debug.Print "Empleado: " & .NombreCompleto  
        Debug.Print "Fecha de nacimiento: " _  
            & .FechaNacimiento  
        Debug.Print "Edad: " & .Edad & " años"  
    End With  
End Sub
```

El resultado de todo esto aparecerá en la ventana **Inmediato**

```
Empleado: Antonio López Iturriaga  
Fecha de nacimiento: 24/12/1980  
Edad: 25 años
```

Tal y como está el código de la clase, nada nos impide poner en la fecha de asignación a la fecha de nacimiento el valor **1780**, con lo que tendríamos un empleado de **225** años, algo cuando menos, chocante.

```
.FechaNacimiento = #12/24/1780#
```

Igualmente podríamos haber escrito

```
.FechaNacimiento = #12/24/2020#
```

Lo que, a fecha de hoy, nos daría un empleado de -15 años.

¿Cómo podemos evitar que suceda esto? es decir, que introduzcan una fecha de nacimiento absurda.

Aquí es donde vamos a ver qué es y cómo se programa una propiedad.

Examinemos este código:

```
Private Const conEdadMaxima As Long = 100  
Public Nombre As String  
Public Apellido1 As String  
Public Apellido2 As String  
Private m_datFechaNacimiento As Date  
Public Telefono As String  
  
Public Property Let FechaNacimiento(ByVal Fecha As Date)
```

```
Dim datFechaMinima As Date

' Ponemos como fecha mínima la de hoy hace 100 años
datFechaMinima = DateAdd("yyyy", -conEdadMaxima, Date)

If Fecha < datFechaMinima Or Fecha > Date Then
    ' Si la fecha introducida es menor que _
    datFechaMinima o mayor que la fecha de hoy _
    generará el error de rango incorrecto
    Err.Raise Number:=106, _
              Source:="Clase CPersona", _
              Description:="Rango de edad inadecuado"
End If

If m_datFechaNacimiento <> Fecha Then
    m_datFechaNacimiento = Fecha
End If

End Property

Public Property Get FechaNacimiento() As Date
    FechaNacimiento = m_datFechaNacimiento
End Property
```

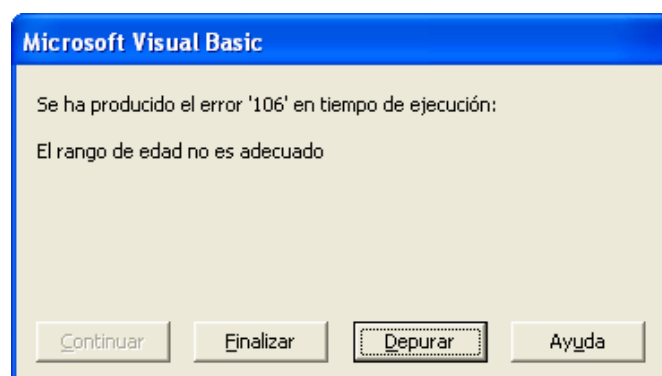
Se supone que el resto del código permanece igual.

Vemos que ha desaparecido la variable pública **FechaNacimiento**, siendo sustituida por la variable privada **m_datFechaNacimiento**.

Si ahora tratamos de ejecutar el procedimiento **PruebaCPersona** en el que la línea de asignación de fecha sea

```
.FechaNacimiento = #12/24/2020#
```

Nos dará un bonito mensaje genérico de error:



Lo mismo pasaría si hubiéramos intentado asignar

```
.FechaNacimiento = #12/24/1880#
```

Vemos que si la fecha es normal el código se ejecuta como antes.

Recordemos que la línea del procedimiento de prueba:

```
Debug.Print "Fecha de nacimiento: " _  
           & .FechaNacimiento
```

Llama también a la propiedad **FechaNacimiento**.

En el primer caso que hemos probado, esa propiedad la devolvía directamente la variable de tipo Date, pero en el segundo caso, la variable ha desaparecido, habiendo sido sustituida por los dos procedimientos **FechaNacimiento**.

Fijémonos en el encabezado de los mismos:

```
Public Property Let FechaNacimiento (ByVal Fecha As Date)  
    y  
Public Property Get FechaNacimiento () As Date
```

Vemos que el primero, **Property Let**, es equivalente a un procedimiento **Sub**, ya que no devuelve ningún valor, y el segundo, **Property Get**, es equivalente a un procedimiento **Function**, ya que devuelve un valor, en este caso del tipo Date.

Además vemos que los dos procedimientos **Property**, son públicos.

Otra cosa que llama la atención es que tienen el mismo nombre **FechaNacimiento**.

El primero, **Property Let**, sirve para asignar un valor, pasado como parámetro, a una variable, normalmente de tipo **Private**, y por tanto inaccesible (encapsulada).

El segundo, **Property Get**, sirve para devolver el valor de una variable, en esta clase la variable es **m_datFechaNacimiento**.

En ambos casos la variable a la que se tiene acceso es de tipo privado.

Fíjese en la notación que estoy empleando:

Pongo primero el prefijo **m_** para indicar que es una variable privada, con validez a nivel del módulo de clase.

A continuación pongo el prefijo **dat**, para indicar que es de tipo **Date**.

La variable acaba con el nombre "humanizado" de la misma **FechaNacimiento**.

El procedimiento **Property Let**, antes de asignar un valor a la variable comprueba que esté dentro del rango establecido; en este caso entre la fecha de hoy y la de hace 100 años.

Si supera este filtro comprueba que sea diferente a la fecha almacenada en la variable **m_datFechaNacimiento**; caso de serlo le asigna el nuevo valor de Fecha.

Veremos que esto último tiene su importancia si queremos generar un evento cuando cambie en un objeto el valor de una propiedad. Además no tiene sentido asignar un valor a una variable igual al que ya posee.

En resumidas cuentas, hemos creado dos procedimientos **Property**:

Uno de **Lectura**, **Property Get**, que suministra el valor de una variable interna de tipo privado. Su forma de escritura es equivalente a la de una función.

Un segundo de **Escritura**, **Property Let**, que graba un valor en una variable interna de tipo privado. Su forma de escritura es equivalente a la de un procedimiento.

Existe un tercer tipo de procedimiento **property** que se utiliza para grabar propiedades de tipo Objeto. Es el **Property Set**.

Este podría ser un ejemplo de uso:

```
Public Property Set Objeto(NuevoObjeto As Object)
    Set m_Objeto = NuevoObjeto
End Property
```

La propiedad **FechaNacimiento**, del ejemplo decimos que es de **Lectura / Escritura**, ya que tiene los dos procedimientos **Property Get y Property Let**.

Si quisiéramos crear una propiedad **de solo lectura**, por ejemplo que devolviera el número de registros existentes en una tabla, crearíamos sólo el procedimiento **Property Get**.

Igualmente si quisiéramos una propiedad de **solo escritura**, aunque no hay muchos casos que se me ocurran que lo necesiten, escribiríamos sólo el procedimiento **Property Let**.

A la hora de crear una clase, lo que se suele hacer es definir los **Atributos de la clase**.

Para todos aquellos atributos sobre los que se quiere tener un control, ya sea porque

- se quieren controlar los rangos de entrada o salida
- se vayan a generar eventos cuando cambien, vayan a cambiar o alcancen determinados valores
- se quiera hacer que sean sólo de Lectura o solo de Escritura

el método a seguir es el siguiente

- Se crea una variable de alcance Private para cada uno de los atributos
- Se genera un **Property Let** para la asignación de valores, o un **Property Set** si el dato puede admitir algún tipo de objeto.
- De forma paralela se genera un **Property Get** que devolverá el dato de la variable fuera del objeto.

En el caso de las funciones (métodos) Edad y NombreCompleto, podríamos haberlas definido como propiedades de sólo lectura, de la siguiente manera:

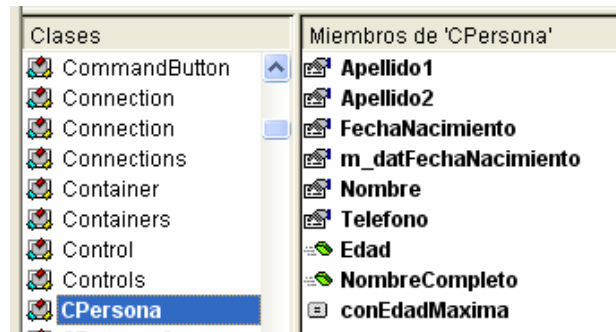
```
Public Property Get Edad() As Long
    Edad = (Date - FechaNacimiento) / 365.2425
End Property
```

```
Public Property Get NombreCompleto() As String
    NombreCompleto = Nombre _
        & " " & Apellido1 _
        & " " & Apellido2
End Property
```

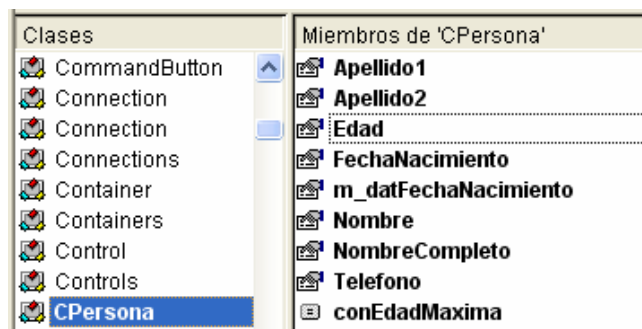
Antes de efectuar este cambio, estos procedimientos

Si abrimos el examinador de Objetos veremos lo siguiente:

Antes de hacer el cambio:



Después de hacerlo



Vemos que ha cambiado el símbolo que aparece delante, tanto de **Edad** como de **NombreCompleto**, y ha pasado de ser el símbolo de un **Método** a otro que representa una **Propiedad**.

Métodos de clase

A las funciones y procedimientos públicos de una clase se les llama **Métodos de clase**.

En el caso de la clase **CPersona**, tenemos dos métodos de clase, que devuelven valores, es decir son del tipo **Function**. Estos métodos son **Edad**, que devuelve un número del tipo **Long** y **NombreCompleto**, que devuelve una cadena **String**.

Podrían haberse creado métodos que no devolvieran ningún valor, es decir del tipo **Sub**.

Todos estos métodos públicos podrían, a su vez, haber utilizado para su ejecución interna otros métodos privados de la propia clase, que no fuesen visibles desde fuera de la misma.

Como ya hemos dicho, a la capacidad de ocultar determinados atributos y procedimientos de la clase, es a lo que se llama **Encapsulación**.

Clases que hacen referencia a sí mismas

En determinadas circunstancias, nos puede interesar que una clase contenga un miembro que sea a su vez un ejemplar del mismo tipo de clase.

Podría ser el caso de una clase **Persona** que contuviera una variable, por ejemplo llamada **Conyuge**, también del tipo **Persona**. Otro caso sería si queremos emular una estructura de Punteros, sin hacer accesos directos a memoria, por ejemplo una estructura del tipo Pila, Fila e incluso Árbol, en las que cada eslabón de la estructura sea un objeto de una clase, y

que tenga, por ejemplo métodos como Anterior y Siguiente, Derecha e Izquierda, que devuelvan objetos de la misma clase.

Vamos crear una clase a la que llamaremos **CIndividuo**.

Esta clase va a ser muy sencilla, y sólo va a tener una propiedad llamada **Nombre** que devolverá un **String** y otra propiedad llamada **Pareja** que va a devolver un objeto del tipo **CPareja**.

Vamos a ello. Como siempre creamos un nuevo módulo de clase al que pondremos por nombre **CIndividuo**.

Nota:

Lo de poner una **C** delante del nombre “en cristiano” de la clase es una convención de nombres que se había extendido en el mundillo de Visual Basic.

Ahora, en Vb.Net parece que la forma a adoptar es poner delante el prefijo **cls**.

Su código sería el siguiente.

```
Option Explicit

Private m_Pareja As CIndividuo
Private m_strNombre As String

Public Property Get Pareja() As CIndividuo
    Set Pareja = m_Pareja
End Property

Public Property Set Pareja(Persona As CIndividuo)
    Set m_Pareja = Persona
End Property

Public Property Get Nombre() As String
    Nombre = m_strNombre
End Property

Public Property Let Nombre(NuevoNombre As String)
    If m_strNombre <> NuevoNombre Then
        m_strNombre = NuevoNombre
    End If
End Property
```

En este código podemos observar varias cosas

En primer lugar el procedimiento para la escritura del valor de la propiedad, **Property Let** ha sido sustituido por un procedimiento **Property Get**. Esto es así porque se va a asignar un objeto a la variable **m_strNombre**.

Para probar esta clase vamos a escribir una “historia cotidiana” en la que se entremezclan tres individuos.

- *Antonio y María se conocen desde hace tiempo y como el roce genera el cariño, acabaron haciéndose novios.*
- *Durante una temporada mantienen una relación estable y feliz.*
- *Un buen día María conoció a Juan. La atracción mutua fue instantánea y fulgurante, lo que les llevó a vivir un apasionado romance.*
- *A todo esto Antonio no se entera de nada.*

¿Qué tenemos aquí?

Tenemos tres objetos de la clase **CIndividuo**, de los que la propiedad **Nombre** es **María, Antonio y Juan**.

Pido disculpas por llamar objetos a personas, aunque algunas se lo lleguen a merecer. Vemos que al principio **Antonio** es la **Pareja** de **María** y **María** lo es a su vez de **Antonio**.

En un momento dado, la **Pareja** de **María** pasa a ser **Juan**, y la de **Juan** **María**. **Antonio** sigue creyendo que su **Pareja** sigue siendo **María**.

Para representar esta historia, creamos el procedimiento **HistoriaDeTres ()**.

```
Public Sub HistoriaDeTres()  
    Dim Hombre As New CIndividuo  
    Dim Mujer As New CIndividuo  
    Dim UnTercero As New CIndividuo  
  
    ' Les ponemos nombres  
    Hombre.Nombre = "Antonio"  
    Mujer.Nombre = "María"  
    UnTercero.Nombre = "Juan"  
  
    ' Antonio y María son novios  
    Set Hombre.Pareja = Mujer  
    Set Mujer.Pareja = Hombre  
  
    Debug.Print Hombre.Nombre _  
        & " lleva mucho tiempo saliendo con " _  
        & Mujer.Nombre  
    Debug.Print "La pareja de " & Hombre.Nombre _  
        & " es: " & Hombre.Pareja.Nombre  
    Debug.Print "La pareja de " & Mujer.Nombre _  
        & " es: " & Mujer.Pareja.Nombre
```

```
Debug.Print Mujer.Nombre _
        & " conoce a " & UnTercero.Nombre _
        & " y ..."
Set Mujer.Pareja = UnTercero
Set UnTercero.Pareja = Mujer

Debug.Print "La pareja de " & Mujer.Nombre _
        & " es: " & Mujer.Pareja.Nombre
Debug.Print "La pareja de " & UnTercero.Nombre _
        & " es: " & UnTercero.Pareja.Nombre

' Antonio no se entera de nada
Debug.Print "Tras los cuernos " _
        & Hombre.Nombre _
        & " ni se ha enterado"
Debug.Print Hombre.Nombre _
        & " cree que su pareja es " _
        & Hombre.Pareja.Nombre

End Sub
```

Si ejecutamos el procedimiento, el resultado, en la ventana de Depuración será:

```
Antonio lleva mucho tiempo saliendo con María
La pareja de Antonio es: María
La pareja de María es: Antonio
María conoce a Juan y ...
La pareja de María es: Juan
La pareja de Juan es: María
Tras los cuernos Antonio ni se ha enterado
Antonio cree que su pareja es María
```

Vemos que para acceder al nombre de la pareja, podemos hacerlo de la siguiente forma

```
Individuo.Pareja.Nombre
```

Esto es así porque `Individuo.Pareja` devuelve un objeto del tipo `CIndividuo`, por lo que podremos acceder a su propiedad `Nombre`.

Supongamos que hubiéramos puesto

```
Debug.Print Mujer.Pareja.Pareja.Nombre
```

Esto nos devolvería **María**, ya `Mujer.Pareja` devuelve al Novio de María, por lo que la pareja del Novio de María, es la propia María.

De la misma forma

```
Mujer.Pareja.Pareja.Pareja
```

Nos devolverá al Individuo que es en ese momento Novio de María.

Creación de estructuras con clases.

Supongamos que queremos crear una estructura de tipo Árbol.

Definimos una estructura de tipo Árbol como aquella que tiene las siguientes características:

- Hay un tronco común, que sería la primera Rama.
- Del extremo de cada Rama pueden partir 0 ó 2 Ramas nuevas
- Al extremo del que parten nuevas ramas le llamaremos Nodo.

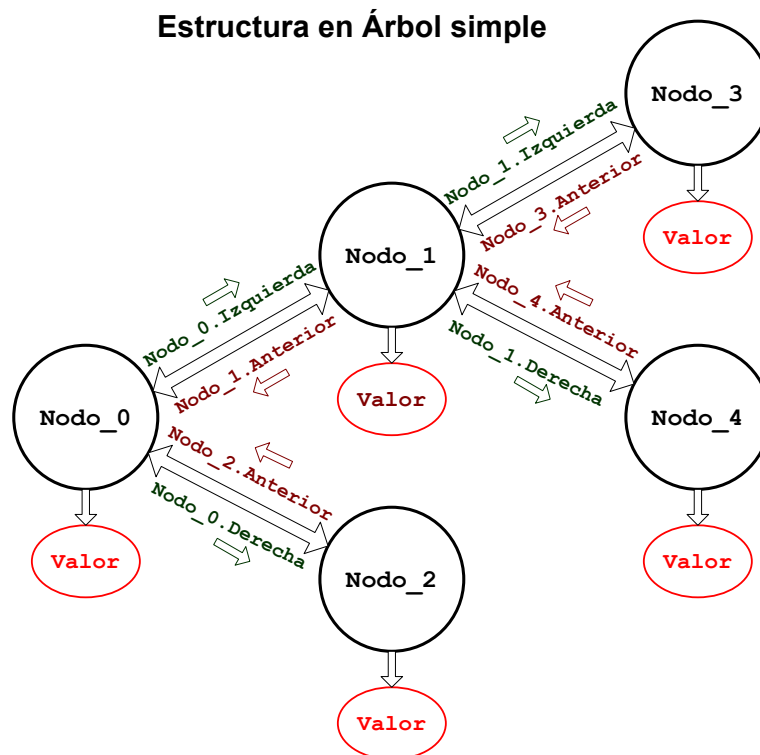
Estableceremos la comunicación entre Nodos, de forma que será el nodo el eslabón que servirá para desplazarse por el Árbol.

Definiremos unos métodos

- Anterior Devolverá el nodo anterior en el organigrama del árbol
- Izquierda Devolverá el nodo situado en el extremo de la rama izquierda
- Derecha Devolverá el nodo situado en el extremo de la rama derecha

Cada nodo podrá almacenar un dato en la propiedad **Valor** y tendrá una propiedad **Nombre**.

Vamos a tratar de construir la siguiente estructura:



Sería equivalente a un árbol con dos ramas principales, de una de las cuales a su vez parten otras dos ramas.

Vamos a crear una clase a la que llamaremos **CNodoArbolSimple**.

Primero insertaremos un módulo de clase con el nombre **CNodoArbolSimple**.

En este módulo escribiremos lo siguiente

```
Option Explicit
Private m_varValor As Variant
Private m_strNombre As String
Private m_ndoAnterior As CNodoArbolSimple
Private m_ndoIzquierda As CNodoArbolSimple
Private m_ndoDerecha As CNodoArbolSimple

Public Property Get Valor() As Variant
    If Not IsObject(m_varValor) Then
        Valor = m_varValor
    Else
        Set Valor = m_varValor
    End If
End Property

Public Property Let Valor(NuevoValor As Variant)
    m_varValor = NuevoValor
End Property

Public Property Set Valor(NuevoValor As Variant)
    Set m_varValor = NuevoValor
End Property
```

En las primeras líneas definimos tres variables privadas del tipo **CNodoArbolSimple**, que nos servirán para hacer referencia a los nodos **Anterior**, **Izquierda** y **Derecha**.

También definiremos una variable de tipo **Variant** que nos servirá para almacenar al **Valor** asignado al nodo.

Y ahora fijémonos en la propiedad **Valor**.

¿Por qué he puesto una propiedad **Property Get**, otra **Property Let** y una tercera **Property Set**?

La razón es muy sencilla.

Al manejar **Valor** una variable de tipo **Variant**, puede ocurrir que ésta haga referencia a un objeto.

No se puede asignar un objeto de la forma `m_varValor = NuevoValor`, como lo hace **Property Let**, ya que daría un error.

Es necesario hacer `Set m_varValor = NuevoValor` de lo que se encarga **Property Set**.

Vamos a hora a definir la propiedad **Valor** para almacenar datos en el nodo:

Vamos a probar este esbozo de clase para comprobar si lo dicho funciona correctamente.

En un módulo estándar escribimos:

```
Public Sub PruebaNodo()  
    Dim Nodo As New CNodoArbolSimple  
    Dim col As New Collection  
  
    col.Add "Lunes"  
    col.Add "Martes"  
    col.Add "Miércoles"  
  
    Nodo.Valor = 1000  
    Debug.Print Nodo.Valor  
  
    Set Nodo.Valor = col  
    Debug.Print Nodo.Valor(1)  
    Debug.Print Nodo.Valor(2)  
    Debug.Print Nodo.Valor(3)  
End Sub
```

Si hemos escrito correctamente todas las líneas y ejecutamos el procedimiento vemos que, como era de esperar, nos muestra en la ventana de depuración:

```
1000  
Lunes  
Martes  
Miércoles
```

Vamos ahora a completar la clase, **CNodoArbolSimple**, definiendo las propiedades **Anterior**, **Izquierda**, **Derecha**, y **Nombre** con lo que tendríamos:

```
Option Explicit  
  
Option Explicit  
Private m_varValor As Variant  
Private m_strNombre As String  
Private m_ndoAnterior As CNodoArbolSimple  
Private m_ndoIzquierda As CNodoArbolSimple  
Private m_ndoDerecha As CNodoArbolSimple  
  
Public Property Get Valor() As Variant  
    If Not IsObject(m_varValor) Then  
        Valor = m_varValor  
    Else  
        Set Valor = m_varValor
```

```
        End If
    End Property

    Public Property Let Valor(NuevoValor As Variant)
        m_varValor = NuevoValor
    End Property

    Public Property Set Valor(NuevoValor As Variant)
        Set m_varValor = NuevoValor
    End Property

    Public Property Get Nombre() As String
        Nombre = m_strNombre
    End Property

    Public Property Let Nombre(NuevoNombre As String)
        m_strNombre = NuevoNombre
    End Property

    Public Property Get Anterior() As CNodoArbolSimple
        Set Anterior = m_ndoAnterior
    End Property

    Public Property Set Anterior(NuevoNodo As
CNodoArbolSimple)
        Set m_ndoAnterior = NuevoNodo
    End Property

    Public Property Get Izquierda() As CNodoArbolSimple
        Set Anterior = m_ndoIzquierda
    End Property

    Public Property Set Izquierda(NuevoNodo As
CNodoArbolSimple)
        Set m_ndoIzquierda = NuevoNodo
    End Property

    Public Property Get Derecha() As CNodoArbolSimple
        Set Anterior = m_ndoDerecha
    End Property
```



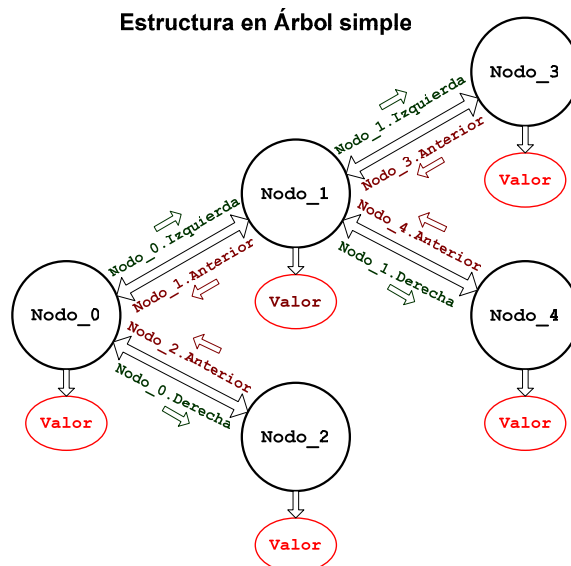
```

Public Property Set Derecha (NuevoNodo As CNodoArbolSimple)
    Set m_ndoDerecha = NuevoNodo
End Property

```

La clase está simplificada al máximo, sin incluir gestión de errores ni comprobaciones de valores previos; esto lo dejo para el lector.

Vamos ahora a ponerla a prueba generando un árbol con la estructura del árbol simple que pusimos al principio de esta sección



En el módulo estándar escribimos el siguiente procedimiento

```

Public Sub PruebaNodosArbol()
    ' La colección Nodos _
    ' contendrá los sucesivos nodos
    Dim Nodos As New Collection
    Dim i As Long
    Dim Nodo As New CNodoArbolSimple

    With Nodo
        .Nombre = "Nodo_0"
        .Valor = 0
        Set .Izquierda = New CNodoArbolSimple
        Set .Derecha = New CNodoArbolSimple
    End With

    ' Añado el nodo con sus datos y la clave "0" _
    ' a la colección
    Nodos.Add Nodo, "0"

    With Nodo.Izquierda

```

```
.Nombre = "Nodo_1"
.Valor = 10
Set .Izquierda = New CNodoArbolSimple
Set .Derecha = New CNodoArbolSimple
Set .Anterior = Nodo
End With
Nodos.Add Nodo.Izquierda, "1"

With Nodo.Derecha
.Nombre = "Nodo_2"
.Valor = 20
Set .Izquierda = New CNodoArbolSimple
Set .Derecha = New CNodoArbolSimple
Set .Anterior = Nodo
End With
Nodos.Add Nodo.Derecha, "2"

' Ahora trabajo con el nodo Nodo_3
With Nodo.Izquierda.Izquierda
.Nombre = "Nodo_3"
.Valor = 30
Set .Anterior = Nodo.Izquierda
End With
Nodos.Add Nodo.Izquierda.Izquierda, "3"

With Nodo.Izquierda.Derecha
.Nombre = "Nodo_4"
.Valor = 40
Set .Anterior = Nodo.Izquierda
End With
Nodos.Add Nodo.Izquierda.Derecha, "4"

Debug.Print
Debug.Print " Nodo Base "
' El procedimiento MuestraDatos está a continuación
MuestraDatos Nodos(CStr(0))

For i = 1 To 4
    Debug.Print
    Debug.Print " Nodo nº " & CStr(i)
```

```
MuestraDatos Nodos(CStr(i))
Next i

' para comprobar si todo está correcto _
  recorremos la estructura usando los enlaces

Debug.Print "Ahora la recorro por los enlaces"
Debug.Print "Siguiendo las propiedades"
Debug.Print "Izquierda, Derecha y Anterior"

Set Nodo = Nodos(1)
Debug.Print "Nodo Base " & Nodo.Nombre

Set Nodo = Nodo.Izquierda
Debug.Print "Nodo N° 1 " & Nodo.Nombre

Set Nodo = Nodo.Anterior.Derecha
Debug.Print "Nodo N° 2 " & Nodo.Nombre

Set Nodo = Nodo.Anterior.Izquierda.Izquierda
Debug.Print "Nodo N° 3 " & Nodo.Nombre

Set Nodo = Nodo.Anterior.Derecha
Debug.Print "Nodo N° 4 " & Nodo.Nombre
End Sub
```

El procedimiento MuestraDatos es el siguiente

```
Public Sub MuestraDatos(Nodo As CNodoArbolSimple)
  With Nodo
    Debug.Print Tab(5); "Nombre " & .Nombre
    Debug.Print Tab(5); "Valor " & .Valor
  End With
End Sub
```

Tras ejecutarse el procedimiento PruebaNodosArbol se mostrará en la ventana **Inmediato** lo siguiente:

```
Nodo Base
  Nombre Nodo_0
  Valor 0

Nodo n° 1
  Nombre Nodo_1
```

Valor 10

Nodo n° 2

Nombre Nodo_2

Valor 20

Nodo n° 3

Nombre Nodo_3

Valor 30

Nodo n° 4

Nombre Nodo_4

Valor 40

Ahora la recorro por los enlaces

Siguiendo las propiedades

Izquierda, Derecha y Anterior

Nodo Base Nodo_0

Nodo N° 1 Nodo_1

Nodo N° 2 Nodo_2

Nodo N° 3 Nodo_3

Nodo N° 4 Nodo_4

Para destruir la estructura podríamos ir eliminando cada uno de los elementos de la colección **Nodos**, mediante su método **Remove**, antes de salir del procedimiento.

Ya se que a primera vista no parece muy espectacular, pero hemos creado una estructura tipo árbol, en la que cada nodo puede estar enlazado con otros tres y es capaz de almacenar un valor.

En la clase **CNodoArbolSimple** hemos colocado una propiedad **Izquierda**, otra **Derecha** y una **Anterior**.

Si quisiéramos crear un Árbol más genérico, en el que un nodo podría tener **0**, **1** ó **n** ramas, deberíamos sustituir las propiedades **Derecha** e **Izquierda**, por una propiedad **Rama (Índice)** que trabajaría contra una colección privada de la propia clase.

Igualmente podríamos saltarnos una estructura de Árbol y definir una estructura de grafo más complejo.

Podríamos haber desarrollado una propiedad **Padre (Índice)** que enlazara con varios posibles Ancestros, de nivel **-1**, una propiedad **Hermano (Índice)** que enlazara con objetos del mismo nivel, y una propiedad **Hijo (Índice)** que enlazara con objetos de nivel **1**. A través de los objetos devueltos podríamos tener acceso a **Abuelos**, **Nietos**, etc...

Además podríamos hacer que cada nodo almacenara diferentes valores de diversos tipos.

Las referencias a **Hermanos**, **Padres** e **Hijos** podríamos guardarlos en Colecciones Privadas de la propia clase.

Las aplicaciones en las que una estructura de este calibre, podría servir como elemento clave, son innumerables:

- Desarrollo flexible de Escandallos de Producción
- Elaboración de Grafos
- Diseño de una hoja de cálculo
- Definir modelos de datos
- Diseño de juegos
- Todas las que tu imaginación pueda intuir. . .

Antes de seguir quiero volver a incidir en un aspecto que hemos tocado.

Si una clase incluye referencias a objetos la propia clase, y métodos para desplazarse entre las referencias, podemos encadenar estos métodos.

Por ejemplo si tenemos una clase que posee una propiedad **Siguiente** que devuelve otro objeto de esa misma clase, podríamos recorrer la estructura de objetos, utilizando el método **Siguiente** desde un Objeto base de forma reiterativa, hasta que lleguemos a un objeto cuya propiedad **Siguiente** esté sin asignar, es decir su valor sea **Nothing**.

```
Objeto.Siguiente.Siguiente.Siguiente. . .
```

Función Is Nothing

Para ver si hemos asignado un objeto a una variable objeto, podemos utilizar la función **Is Nothing**, que devolverá **True** si está sin asignar, y **False** si ya tiene asignado un objeto.

La forma de utilizarla es

```
MiObjeto Is Nothing
```

Por ejemplo, el siguiente código imprimirá primero **False**, ya que no hemos todavía asignado un objeto a la propiedad **Anterior**, del objeto **Nodo1**. A continuación imprimirá **True**, ya que el objeto **Nodo2** sí tiene asignado el objeto **Nodo1** en la propiedad **Anterior**.

```
Public Sub PruebaIsNothing()  
    Dim Nodo1 As New CNodoArbolSimple  
    Dim Nodo2 As New CNodoArbolSimple  
    Debug.Print Nodo1.Anterior Is Nothing  
    Set Nodo2.Anterior = Nodo1  
    Debug.Print Nodo2.Anterior Is Nothing  
End Sub
```

Eventos.

Un evento es un procedimiento de la clase que se ejecuta desde dentro del propio objeto creado con la clase, y que es posible captarlo en el objeto que contiene a la clase.

Como veremos próximamente, si en un formulario colocamos un botón de comando (**CommandButton**) cada vez que lo presionemos genera el evento **Click**.

Si quisiéramos que cuando se presione el botón de nombre **cmdSalir**, se cierre el formulario, podríamos poner en el módulo de clase del formulario el siguiente código.

```
Private Sub cmdSalir_Click()  
On Error GoTo HayError  
    DoCmd.Close  
  
Salir:  
    Exit Sub  
  
HayError:  
    MsgBox Err.Description  
    Resume Salir  
End Sub
```

Dentro del módulo de clase del objeto Botón, algún programador ha definido un evento al que ha puesto por nombre **Click()**

Nosotros podemos programar Eventos dentro de las clases, de forma que esos eventos puedan ser captados por otras clases que contengan a las primeras.

Para ello se utiliza las instrucciones **Event** y **RaiseEvent**, que veremos más adelante.

Qué es un Evento

Si vamos a la ayuda de Access, vemos que lo describe de la siguiente forma

Un evento es una acción específica que se produce en o con un objeto determinado. Microsoft Access puede responder a una variedad de eventos:

- clics del Mouse
- cambios en los datos
- formularios que se abren o se cierran
- muchos otros.

Los eventos son normalmente el resultado de una acción del usuario.

No se si aclara mucho, pero como complemento me atrevo a decir que un evento es algo así como una llamada de aviso que efectúa un objeto cuando suceden determinadas cosas.

Esa llamada puede, o no, ser recogida por un procedimiento, que actuará en consecuencia.

Vamos a ver un ejemplo clásico.

Pongamos en un formulario un botón de comando.

Ojo: desactiva antes el “Asistente para controles” de la barra de herramientas.

A su propiedad **Título**, le ponemos el texto **Saludo**.

Sin dejar de seleccionar el botón, activamos la hoja Eventos de la ventana de propiedades.

Seleccionamos el evento **Al hacer clic** y pulsamos en el botoncito con tres botones

Nos aparece una pequeña ventana con tres opciones. Seleccionamos **Generador de código** y pulsamos en el botón **Aceptar**.

Directamente nos abre el módulo de clase de ese formulario, probablemente con el siguiente código.

```
Option Compare Database
```

```
Option Explicit
```

```
Private Sub cmdSaludo_Click()
```

```
End Sub
```

Modificamos el procedimiento `cmdSaludo_Click` dejándolo así:

```
Option Compare Database
```

```
Option Explicit
```

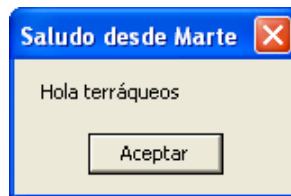
```
Private Sub cmdSaludo_Click()
```

```
    MsgBox "Hola terráqueos", , "Saludo desde Marte"
```

```
End Sub
```

Cerramos la ventana de edición, y ejecutamos el formulario.

Al pulsar en el botón, nos aparece el mensaje:



¿Por qué pasa esto?

Cuando hacemos clic sobre algunos objetos, éstos hacen una llamada, buscando en el objeto que los contiene, en este caso el formulario, un procedimiento que tenga como nombre **NombreDelObjeto_Click**.

Como el nombre del botón es `cmdSaludo`, examina el formulario para ver si existe algún procedimiento de nombre **cmdSaludo_Click**. Como en este caso sí existe, ejecuta dicho procedimiento.

Un procedimiento que se vaya a ejecutar como respuesta a un evento, puede incluir parámetros.

Vamos a poner en el mismo formulario un cuadro de texto (TextBox), al que llamaremos **txtPrueba**. Como en el caso del botón, seleccionaremos, en la ventana de propiedades, la hoja correspondiente a Eventos, y entre ellos, el evento **Al presionar una tecla**. En la ventana seleccionamos la opción Generador de código y nos abre el editor de código del módulo de clase del formulario, posicionándonos en un nuevo procedimiento de evento:

```
Private Sub txtPrueba_KeyPress (KeyAscii As Integer)
```

```
End Sub
```

Podemos ver que el nuevo procedimiento tiene el nombre del objeto **txtPrueba**, seguido de la barra baja `_` y el nombre, en inglés del evento **KeyPress**.

En este caso, además incluye el parámetro `KeyAscii`, de tipo `Integer`.

el evento es llamado cada vez que se pulsa una tecla en el cuadro de texto, pasándole un valor entero (`KeyAscii`) que contiene el código ASCII de la letra correspondiente a la tecla pulsada.

Por ejemplo, el código ASCII correspondiente a la letra **A** es el **65**, y el de la letra **a** es el **97**. Podríamos cambiar el valor de ese código. Por ejemplo, si en el procedimiento pusiéramos

```
Private Sub txtPrueba_KeyPress(KeyAscii As Integer)
    KeyAscii = Asc(UCase(Chr(KeyAscii)))
End Sub
```

Y ejecutáramos el código, veríamos que sólo podremos poner letras mayúsculas, aunque pulsemos minúsculas.

La razón es la siguiente

Cuando pulsamos una tecla, y antes de que realmente se llegue a escribir nada en el cuadro de texto, como ya hemos dicho, llama al procedimiento del evento, pasándole el código ASCII correspondiente a la letra de la tecla pulsada. Supongamos que hemos pulsado la letra **a** minúscula. Por tanto el código ASCII pasado será el **97**.

A continuación convierte el código **97** en la propia letra **a**, mediante la función `Chr` `Chr(97)`.

Mediante la función `Ucase`, `UCase(Chr(97))`, convierte la **a** minúscula en **A** mayúscula.

Finalmente extrae el código ASCII correspondiente a la **A** mayúscula, **65**, y lo pasa al parámetro `ASCII`.

En una fase posterior, el formulario muestra la **A** mayúscula en el cuadro de texto.

Resumen:

Un evento se coloca en el módulo del objeto contenedor del objeto que genera el evento. En el caso anterior en el módulo de clase del formulario, que es el que contiene al botón `cmdSaludo` y al cuadro de texto `txtPrueba`.

Los procedimientos que gestionan los eventos correspondientes tienen como nombre

```
NombreDelObjeto_NombreDelEvento([PosiblesParámetros])
```

Ejemplos de nombres de los procedimientos gestores de los eventos:

```
cmdSaludo_Click()
txtPrueba_KeyPress(KeyAscii As Integer)
txtPrueba_KeyDown(KeyCode As Integer, Shift As Integer)
```

Como podemos ver, un evento puede tener (o no) uno ó más parámetros.

Esos parámetros, normalmente pasados "Por Referencia", pueden ser cambiados dentro del procedimiento gestor de eventos.

Un mismo objeto puede generar diferentes tipos de eventos.

Crear clases en las que se definan Eventos.

Para que el objeto de una clase genere eventos, hay que escribir una serie de instrucciones que son semejantes a la cabecera de un procedimiento.

Para ello se utiliza la instrucción **Event**.

Instrucción Event

Sirve para declarar un evento definido por un usuario, dentro de una clase.

Su sintaxis es la siguiente

```
[Public] Event NombreDelProcedimiento [(listaDeParámetros)]
```

Dentro de la instrucción **Event** podemos considerar los siguientes elementos:

La declaración de un evento como **Public** es opcional, pero sólo porque éste es el valor por defecto. **Event** no permite su declaración como **Private**, lo que es totalmente lógico ya que el evento debe ser poder ser visible para los elementos que manejen el objeto de la clase.

NombreDelProcedimiento es el nombre que vamos a dar al evento en sí. Si fuéramos a programar un botón que reaccione cuando lo pulsemos, lo lógico sería poner como nombre de ese evento **Clic**.

Nota:

*Si existen eventos habituales con nombres específicos, normalmente en inglés, se suelen usar éstos, ya que normalmente son conocidos por los programadores, y por sí mismos resultan descriptivos. Por ejemplo, frente a un evento de nombre **Clic**, cualquier programador intuye cómo y cuándo se genera, sin necesidad de destripar el código de la clase.*

Como lista de parámetros podemos definir ninguno, uno ó varios, y los podemos pasar tanto **Por Referencia (By Ref)** como **Por Valor (By Val)**.

Hay una serie de parámetros cuyo nombre suele estar preestablecido, , como **Source**, que sería el elemento generador del evento, **Cancel** que se suele utilizar para cancelar la causa del evento, **X** e **Y** para devolver la posición del cursor, etc... El asignarles esos nombres, o sus equivalentes en el idioma local no es obligatorio, pero se considera una buena práctica de programación. Por ejemplo, en estas líneas vamos a utilizar el parámetro **Cancelar**, por lo claro y descriptivo que resulta su propio nombre, aunque quizás hubiera sido más aconsejable utilizar la palabra **Cancel**.

Igualmente Microsoft aconseja la utilización, para los eventos que se generan inmediatamente antes de que se realice una cosa, usar el gerundio de su verbo, en inglés **ing**, y para los eventos generados inmediatamente después de producirse un hecho, el participio **ed**.

Por ejemplo, si tuviéramos que desarrollar una clase que controlara el llenado de un depósito, podríamos definir en un evento que se generara justo antes de comenzar el llenado, al que podríamos poner como nombre **Filling**, y como parámetros, la cantidad a traspasar y el contenido actual del depósito. Después de haber volcado todo el líquido en el depósito podríamos generar un evento de nombre **Filled**, al que pasáramos la cantidad traspasada y el contenido del depósito después del llenado.

Vamos a usar un criterio semejante, pero utilizando el idioma español.

Si con lenguajes como **Visual Basic**, **VB.Net**, **C#**, **Object Pascal**, etc construyéramos componentes susceptibles de ser utilizados por programadores que usen diferentes idiomas, lo lógico sería que para dar nombres a los atributos de esos componentes, se utilizaran las palabras equivalentes del inglés.

Manos a la obra. Clase Depósito

Supongamos que ha transcendido al mundo empresarial nuestras habilidades como programadores para todo tipo de terrenos; es sólo una hipótesis...

Un buen día recibimos la visita del ingeniero jefe de una importante refinería petrolífera, y nos cuenta su problema.

La empresa tiene su propio equipo de programadores, pero hasta ahora, no han demostrado la capacidad suficiente como para justificar su elevado sueldo. Tal es así, que para cumplir con las nuevas normas de seguridad, necesitan desarrollar unas interfaces que permitan controlar el trasiego de líquidos que se efectúe en los diferentes depósitos, pudiéndose establecer unos niveles de seguridad mínimo y máximo en cuanto al contenido de los mismos, así como su contenido actual y el contenido máximo real.

El planteamiento es diseñar una clase que permita definir y gestionar esos parámetros y que, cuando se pretenda introducir más cantidad ó menos de la posible físicamente genere un error que pueda ser captado por el objeto que maneje el objeto depósito.

Además si se van a superar los límites de seguridad deberá generar los correspondientes eventos de aviso, que permitan interrumpir el proceso de llenado o vaciado.

También sería interesante que antes de que se vaya a añadir o a extraer líquido de un depósito, incluso sin que se superen los niveles de seguridad, se genere un evento que permita interrumpir la operación.

Cuando se haya terminado el proceso de llenado y vaciado, se deberá generar un evento, informando de la cantidad trasvasada.

Manos a la obra.

Si analizamos los requerimientos, vemos que para controlar las propiedades del depósito tenemos estos cuatro miembros fundamentales.

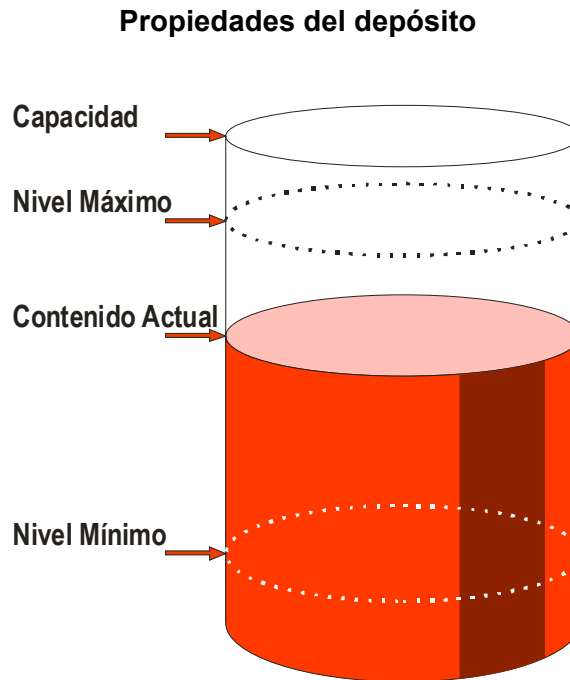
- Capacidad del depósito
- Contenido actual del depósito
- Nivel máximo de seguridad
- Nivel mínimo de seguridad

Otros elementos que deberemos controlar son

- Cantidad a extraer
- Cantidad a introducir

Los eventos que podemos generar serán los siguientes

- Introduciendo
- Introducido
- Extrayendo
- Extraído
- TrasvaselMposible
- VioladoNivelDeSeguridad



Para controlar cada una de las propiedades definiremos las variables privadas, a las que se accederá mediante las correspondientes propiedades.

También definiremos los eventos que hemos definido.

Insertamos un nuevo módulo de clase y escribimos

```
Dim m_sngCapacidad As Single
Dim m_sngNivelMaximo As Single
Dim m_sngContenido As Single
Dim m_sngNivelMinimo As Single

Public Event Introduciendo( _
    ByVal Volumen As Single, _
    ByRef Cancelar As Boolean)

Public Event Introducido( _
    ByVal Volumen As Single)

Public Event Extrayendo( _
    ByVal Volumen As Single, _
    ByRef Cancelar As Boolean)

Public Event Extraido( _
    ByVal Volumen As Single)
```

```
Public Event TrasvaseImposible( _  
    ByVal ContenidoActual As Single, _  
    ByVal VolumenATrasvasar As Single)  
  
Public Event ViolandoNivelDeSeguridad( _  
    ByVal ContenidoActual As Single, _  
    ByVal VolumenATrasvasar As Single, _  
    ByVal NivelDeSeguridad As Single)
```

Las variables **Private** definidas, no deberían representar para nosotros ningún problema por lo que no voy a comentarlas.

Para manejar estas variables, escribiremos las correspondientes propiedades de lectura y escritura **Property Get** y **Property Let**.

En cuanto a la declaración de los eventos, vemos el proceso que responde a la estructura:

```
Public Event NombreDelEvento (ListaDeParámetros)
```

En los eventos **Introduciendo** y **Extrayendo** hemos definido **Cancelar** como un parámetro por referencia **ByRef**. Con ello se podría permitir suspender la operación de trasvase desde el exterior del objeto, antes de que se llegara a efectuar.

¿Qué deberemos hacer para lanzar un evento cuando nos interese?

Simplemente utilizar la instrucción **RaiseEvent**.

Instrucción RaiseEvent

La instrucción **RaiseEvent** desencadena un evento declarado a nivel de un módulo de clase.

Este módulo puede ser un módulo específico de clase, o el módulo asociado a un formulario o a un informe de Access.

La forma de hacerlo es una mezcla entre la forma de llamar a una función y la de llamar a un procedimiento **Sub**. Su sintaxis es:

```
RaiseEvent NombreDelEvento (ListaDeParámetros)
```

Hay que usar paréntesis como en las funciones.

Para los procesos de trasvase de líquido, definiremos dos procedimientos Sub públicos dentro del módulo de clase Añadir y Extraer.

Estos Métodos de la clase serán los encargados de desencadenar los eventos cuando las condiciones lo requieran. Incluso la propiedad **Contenido**, cuando haya que variarlo, utilizará estos métodos para ajustar el contenido actual del depósito.

A continuación desarrollaremos el código completo y comentaremos los diferentes temas.

```
Dim m_sngCapacidad As Single  
Dim m_sngNivelMaximo As Single  
Dim m_sngContenido As Single  
Dim m_sngNivelMinimo As Single
```

```
Public Event Introduciendo ( _
```

```
        ByVal Volumen As Single, _
        ByRef Cancelar As Boolean)

Public Event Introducido( _
        ByVal Volumen As Single)

Public Event Extrayendo( _
        ByVal Volumen As Single, _
        ByRef Cancelar As Boolean)

Public Event Extraido( _
        ByVal Volumen As Single)

Public Event TrasvaseImposible( _
        ByVal ContenidoActual As Single, _
        ByVal VolumenATrasvasar As Single)

Public Event ViolandoNivelDeSeguridad( _
        ByVal ContenidoActual As Single, _
        ByVal VolumenATrasvasar As Single, _
        ByVal NivelDeSeguridad As Single)

' Capacidad del depósito
Public Property Get Capacidad() As Single
    Capacidad = m_sngCapacidad
End Property

Public Property Let Capacidad(ByVal Volumen As Single)
    If m_sngCapacidad <> Volumen Then
        m_sngCapacidad = Volumen
    End If
End Property

' Nivel Máximo de seguridad
Public Property Get NivelMaximo() As Single
    NivelMaximo = m_sngNivelMaximo
End Property

Public Property Let NivelMaximo(ByVal Volumen As Single)
    If m_sngNivelMaximo <> Volumen Then
```

```
        Select Case Volumen
        Case Is < 0
            MsgBox "El volumen máximo no puede ser negativo"
        Case Is > m_sngCapacidad
            MsgBox "El volumen máximo no puede ser mayor" _
                & vbCrLf _
                & "que la capacidad del depósito"
        Case Is < m_sngNivelMinimo
            MsgBox "El volumen máximo no puede ser menor" _
                & vbCrLf _
                & "que el volumen mínimo"
        Case Else
            m_sngNivelMaximo = Volumen
        End Select
    End If
End Property

' Nivel Mínimo de seguridad
Public Property Get NivelMinimo() As Single
    NivelMinimo = m_sngNivelMinimo
End Property

Public Property Let NivelMinimo(ByVal Volumen As Single)
    If m_sngNivelMinimo <> Volumen Then
        Select Case Volumen
        ' Control de casos incorrectos
        Case Is < 0
            MsgBox "El volumen mínimo no puede ser negativo"
        Case Is > m_sngCapacidad
            MsgBox "El volumen mínimo no puede ser mayor" _
                & vbCrLf _
                & "que la capacidad del depósito"
        Case Is > m_sngNivelMaximo
            MsgBox "El volumen mínimo no puede ser mayor" _
                & vbCrLf _
                & "que el volumen máximo"
        Case Else
            m_sngNivelMinimo = Volumen
        End Select
    End If
End Property
```

```
End Property
' Contenido actual del depósito
Public Property Get Contenido() As Single
    Contenido = m_sngContenido
End Property

Public Property Let Contenido(ByVal Volumen As Single)
    ' Para hacer que el depósito _
    ' Contenga un determinado volumen _
    ' introducimos lo que falte _
    ' o extraemos lo que sobre
    Select Case Volumen
    Case Is > m_sngContenido
        Introducir Volumen - m_sngContenido
    Case Is < m_sngContenido
        Extraer m_sngContenido - Volumen
    End Select
End Property

' Método para introducir en el depósito
Public Sub Introducir(ByVal Volumen As Single)
    On Error GoTo HayError

    Dim blnCancelar As Boolean
    ' Si vamos a introducir una cantidad negativa _
    ' asume que se quiere extraer
    If Volumen < 0 Then
        Extraer -Volumen
        Exit Sub
    End If

    ' Desencadenamos el primer evento
RaiseEvent Introduciendo(Volumen, blnCancelar)
    ' Si en el gestor del evento _
    ' se ha decidido cancelar la operación
    If blnCancelar Then
        ' Cancelamos el proceso
        Exit Sub
    Else
        ValidaTrasvase Volumen
    End If
End Sub
```

```
        m_sngContenido = m_sngContenido + Volumen
        RaiseEvent Introducido (Volumen)
    End If

Salir:
Exit Sub

HayError:
    MsgBox "Se ha producido el error nº " & Err.Number _
        & vbCrLf _
        & Err.Description, _
        vbCritical + vbOKOnly, _
        "Error en la clase CDeposito " & Err.Source
    Resume Salir
End Sub

' Método para extraer del depósito
Public Sub Extraer(ByVal Volumen As Single)
    On Error GoTo HayError

    Dim sngVolumenFinal As Single
    Dim sngVolumenATrasvasar As Single
    Dim blnCancelar As Boolean

    ' Si vamos a extraer una cantidad negativa _
    ' asume que se quiere introducir
    If Volumen < 0 Then
        Introducir -Volumen
        Exit Sub
    End If

    ' Desencadenamos el primer evento
    RaiseEvent Extrayendo (Volumen, blnCancelar)

    If blnCancelar Then
        ' Cancelamos el proceso
        Exit Sub
    Else
        ' Pasamos -Volumen a ValidaTrasvase
        ValidaTrasvase -Volumen
    End If
End Sub
```



```
        m_sngContenido = m_sngContenido - Volumen
        RaiseEvent Extraído(Volumen)
    End If

Salir:
Exit Sub

HayError:
    MsgBox "Se ha producido el error nº " & Err.Number _
        & vbCrLf _
        & Err.Description, _
        vbCritical + vbOKOnly, _
        "Error en la clase Deposito " & Err.Source
    Resume Salir
End Sub

' Comprobación de si el trasvase es posible
Private Sub ValidaTrasvase(ByVal Volumen As Single)
    ' Este procedimiento se encarga de comprobar _
    ' si es posible extraer o introducir en el depósito _
    ' y generar los eventos y errores, en su caso.
    Dim sngVolumenFinal As Single

    sngVolumenFinal = m_sngContenido + Volumen

    Select Case sngVolumenFinal
    Case Is > m_sngCapacidad
        ' Si no cabe en el depósito _
        ' desencadenamos el evento y generamos un error
        RaiseEvent TrasvaseImposible( _
            m_sngContenido, _
            Volumen)
        Err.Raise 1100, "Proceso de trasvase", _
            "Se pretende introducir más volumen" _
            & " del que cabe en el depósito"
    Case Is < 0
        ' Si no hay suficiente volumen en el depósito _
        ' desencadenamos el evento y generamos un error
        RaiseEvent TrasvaseImposible( _
            m_sngContenido, _
```

```
                Volumen)
    Err.Raise 1110, "Proceso de trasvase", _
        "Se pretende extraer más volumen" _
        & " que el que hay en el depósito"
Exit Sub

Case Is > m_sngNivelMaximo
    ' Si cabría en el depósito _
    pero superara el nivel de seguridad máximo
    RaiseEvent ViolandoNivelDeSeguridad( _
        m_sngContenido, _
        Volumen, _
        m_sngNivelMaximo)
Case Is < m_sngNivelMinimo
    ' Si se puede extraer, pero al final quedara _
    una cantidad inferior _
    al nivel de seguridad mínimo
    RaiseEvent ViolandoNivelDeSeguridad( _
        m_sngContenido, _
        Volumen, _
        m_sngNivelMaximo)

End Select
End Sub
```

Comentemos este código.

En general los procedimientos **Property Get**, que devuelven valores, tienen una estructura bastante elemental, por lo que no voy a hacer comentarios sobre ellos.

Como casi siempre, los procedimientos que presentan algo más de complicación son los **Property Let**, por las validaciones a efectuar antes de admitir los nuevos datos.

Suelo tener por costumbre que, si el valor a introducir es idéntico al existente anteriormente, simplemente hago salir del procedimiento sin cambiar nada. Este es un tipo de actuación que no siempre será la adecuada. En determinadas circunstancias es preciso que se registre un intento de asignar un valor a una propiedad idéntico al que tenía previamente.

Veamos la propiedad **NivelMaximo**:

Esta propiedad devuelve y establece el nivel máximo de seguridad.

Normalmente se asignará después de definir la capacidad del depósito, y lógicamente no debe ser ni negativa ni mayor que la capacidad total del depósito. Tampoco puede ser inferior al nivel de seguridad mínimo del mismo.

Para la propiedad **NivelMinimo**, podemos considerar lo dicho en la propiedad anterior, solo que no debe ser mayor que el nivel máximo de seguridad.

Por todo esto, lo normal sería primero definir la capacidad total del depósito, a continuación las propiedades de los niveles de seguridad Máximo y Mínimo, y finalmente el contenido

actual, que como inicialmente será cero, al llamar a la propiedad **Contenido** ejecutará el procedimiento **Introducir**.

Podríamos haber definido eventos que se dispararan en el momento que cambiáramos la capacidad del depósito, o sus niveles de seguridad, pero como éstas serán propiedades que normalmente se definirán en el momento de la creación del depósito, y no cambiarán durante su vida útil, no he considerado pertinente escribirlos. En una clase desarrollada para su explotación real, seguramente sí habría que considerarlos.

La propiedad **Contenido** devuelve o establece la cantidad contenida ahora en el depósito.

Si en el mundo real queremos establecer una cantidad para el contenido de un depósito, realizaríamos los siguientes pasos.

- Comprobaríamos el volumen existente en el depósito
- Si la cantidad es idéntica a la que queremos que contenga, se acaba el problema.
- Si la cantidad que queremos que tenga es mayor que la existente en el depósito, añadiremos la diferencia entre lo que queremos que haya y lo que realmente hay.
- Si es mayor que la que queremos extraeremos la diferencia.

Es lo que hace el segmento de código

```
Select Case Volumen
    Case Is > m_sngContenido
        Introducir Volumen - m_sngContenido
    Case Is < m_sngContenido
        Extraer m_sngContenido - Volumen
End Select
```

Llama al método **Introducir** ó al método **Extraer** pasándole como parámetro el volumen que será necesario trasvasar.

Es una de las ventajas de trabajar con objetos, ya que podemos aplicar al código una lógica de procesos muy cercana a lo que ocurriría en el mundo real, con lo que el código se hace más estructurado, modular y comprensible.

Vamos a analizar primero el método **Introducir**.

Lo primero que hace es comprobar si el Volumen a introducir es negativo, lo que equivaldría a que tendríamos que extraer. Por tanto, si fuera así llamaría al método **Extraer**.

```
If Volumen < 0 Then
    Extraer -Volumen
Exit Sub
End If
```

Seguidamente dispara el evento **Introduciendo**, al que le pasa como parámetros el **Volumen** a trasvasar, y por referencia la variable Boleana **blnCancelar**.

```
' Desencadenamos el primer evento
RaiseEvent Introduciendo (Volumen, blnCancelar)
```

En el objeto contenedor del objeto Depósito, que podría ser por ejemplo un formulario, en el procedimiento manejador del evento podría haberse establecido la propiedad **Cancelar** al

valor **True**, con lo que se interrumpiría el proceso de introducción. Este proceso se realiza en el segmento de código:

```

If blnCancelar Then
    ' Cancelamos el proceso
    Exit Sub
Else
    ValidaTrasvase Volumen
    m_sngContenido = m_sngContenido + Volumen
    RaiseEvent Introducido (Volumen)
End If

```

Si no se ha interrumpido el proceso, llama a procedimiento **ValidaTrasvase**, que se encarga de comprobar si las cantidades están en los márgenes correctos.

El procedimiento **ValidaTrasvase**, comprueba si es posible efectuar la operación de trasvase en su totalidad, es decir que durante el trasvase no vamos a intentar meter más volumen del que cabe en el depósito, ni vamos a intentar extraer más volumen del que actualmente hay.

Si ocurriera uno de estos dos casos, dispararía el evento **TrasvaseImposible** informando del contenido actual del depósito y del volumen que se pretende trasvasar.

A continuación, si intentara introducir más cantidad de la que cabe en el depósito, generaría el error nº 1100, o el error 1110 si lo que se pretendiera es extraer más cantidad de la existente. Estos errores serían captados por el procedimiento que ha hecho la llamada de comprobación **Introducir** o **Extraer**.

La siguiente tarea es comprobar si volumen final es mayor que el nivel máximo, o menor que el nivel mínimo, caso en el que disparará el evento **ViolandoNivelDeSeguridad**.

A este evento le pasa como parámetros el contenido actual del depósito, el volumen a trasvasar y el nivel de seguridad, Máximo ó Mínimo, violado.

Si no se ha generado ningún error significa que la operación de trasvase es posible, aunque estuviese fuera de los límites de seguridad. Por ello, al retomar el control, el procedimiento **Introducir**, realiza el trasvase y genera el evento **Introducido**.

```

m_sngContenido = m_sngContenido + Volumen
    RaiseEvent Introducido (Volumen)

```

El procedimiento **Extraer** tiene una estructura semejante al procedimiento **Introducir**; de hecho los dos podrían haber sido sustituidos por un procedimiento con un nombre tal como **Trasvasar**.

Si el parámetro **Volumen** pasado a **Introducir** fuera negativo, sería equivalente a llamar al procedimiento **Extraer** con el **Volumen** positivo, y viceversa.

Si abrimos el examinador de objetos, nos mostrará todos los miembros, (atributos, métodos y eventos) que hemos definido en la clase.



Vamos ahora a ver cómo podemos llegar a manejar los eventos generados, y cómo les podríamos dar una utilidad práctica.

A la hora de declarar un objeto de forma que se puedan manejar sus eventos, es preciso hacerlo con la palabra **WithEvents** .

Palabra clave WithEvents.

Sirve para hacer que en un objeto contenedor en el que se ha declarado otro objeto capaz de generar eventos, podamos definir procedimientos manejadores de esos eventos del objeto contenido. Quizás suene un poco a galimatías, pero si queremos que un formulario sea capaz de manejar los eventos de un objeto **Depósito**, de la clase **CDeposito** deberemos declararlo, en el formulario, precedido de la palabra clave **WithEvents** .

La forma de utilizar **WithEvents** es

```
Dim|Public|Private WithEvents NombreObjeto As NombreDeLaClase
```

En nuestro caso podríamos declarar un objeto de nombre **Deposito** de la clase **CDeposito**. Para ello escribimos en el módulo de clase de un formulario

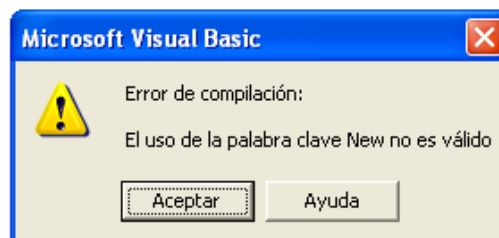
```
Dim WithEvents Deposito As CDeposito
```

Cuando queremos declarar un objeto con capacidad para manejar sus eventos, el objeto no podremos declararlo con la palabra clave **New**.

Por ejemplo, esta línea de código, colocada en la cabecera de un formulario producirá un error de compilación en tiempo de diseño.

```
Option Explicit
```

```
Dim WithEvents Deposito As New CDeposito
```



Tampoco podemos declararlo en un módulo normal; debe ser **dentro de un módulo de una clase**, ya sea una clase “normal”, o en el módulo de clase de un formulario o informe.

Cuando declaramos un objeto con **WithEvents** , debemos simplemente declarar su variable, y después crear el objeto, o asignarle un objeto del tipo declarado.

Una forma correcta de hacerlo sería, por ejemplo, usando el evento **Al cargar (Load)** del formulario. También podríamos usar el mismo evento para asignarle los valores iniciales.

Por ejemplo abrimos un formulario nuevo y en su módulo de clase escribimos:

```
Option Explicit
```

```
Dim WithEvents Deposito As CDeposito
```

```
Private Sub Form_Load()
```

```

' Le ponemos un título al formulario
Caption = " Depósito marca ACME"
' Creamos el objeto Deposito
Set Deposito = New CDeposito
' Le asignamos los valores iniciales
With Deposito
    .Capacidad = 1000
    .NivelMaximo = 900
    .NivelMinimo = 100
End With
' Limitamos la cantidad a trasvasar
sngTrasvaseMaximo = 500
End Sub

```

Vemos que creamos el objeto de tipo **CDeposito** y se lo asignamos a la variable **Deposito** en la línea

```
Set Deposito = New CDeposito
```

Probando la clase CDeposito

En ese mismo formulario vamos a poner los siguientes objetos

Objeto	Nombre
Cuadro de texto	txtCantidad
Botón	cmdExtraer
Botón	cmdIntroducir
Etiqueta	lblCapacidad
Etiqueta	lblMaximo
Etiqueta	lblContenido
Etiqueta	lblMinimo

También podríamos poner otras etiquetas para indicar qué es lo que contienen. Es lo que se ha hecho en el ejemplo.

En el modo diseño, el formulario podría adquirir una apariencia semejante a ésta:

The screenshot shows a Windows form titled "Detalle" with a grid layout. The grid contains the following elements:

- Row 1: A label "Volumen" followed by a text box containing "volumen". To the right is a label "Cantidad a trasvasar" followed by a text box containing "Independiente".
- Row 2: A label "Volumen Máx. Seguridad" followed by a text box containing "Máximo". To the right is a button labeled "Extraer".
- Row 3: A label "Contenido" followed by a text box containing "Contenido". To the right is a button labeled "Introducir".
- Row 4: A label "Volumen Mín. Seguridad" followed by a text box containing "Mínimo".

La intención es que al presionar un botón, se realicen las operaciones de **Introducir** y **Extraer** en el **depósito** el volumen definido en el cuadro de texto **txtCantidad** mostrando en las etiquetas el contenido actual del depósito.

Como la capacidad del depósito la hemos definido como de **1000** unidades, una de las restricciones que vamos a poner es que no se puedan trasvasar en ningún sentido más de **500** unidades de una vez. Requisito solicitado a última hora por los técnicos de la refinería.

El código del formulario sería este:

```
Option Explicit

Dim WithEvents Deposito As CDeposito

Dim sngTrasvaseMaximo As Single

Private Sub Form_Load()
    ' Le ponemos un título al formulario
    Caption = " Depósito marca ACME"
    ' Creamos el objeto Deposito
    Set Deposito = New CDeposito
    ' Le asignamos los valores iniciales
    With Deposito
        .Capacidad = 1000
        .NivelMaximo = 900
        .NivelMinimo = 100
        ' Mostramos los datos
        lblCapacidad.Caption = CStr(.Capacidad)
        lblMaximo.Caption = CStr(.NivelMaximo)
        lblMinimo.Caption = CStr(.NivelMinimo)
        ' Limitamos la cantidad a trasvasar
        sngTrasvaseMaximo = .Capacidad / 2
    End With
    MuestraContenido
End Sub

' Creamos los manejadores de los eventos _
del objeto Deposito

Private Sub Deposito_Introduciendo( _
    ByVal Volumen As Single, _
    ByRef Cancelar As Boolean)
    ControlDeVolumen Volumen, Cancelar
End Sub
```

```
Private Sub Deposito_Extrayendo ( _
    ByVal Volumen As Single, _
    ByRef Cancelar As Boolean)
    ControlDeVolumen Volumen, Cancelar
End Sub

Public Sub Deposito_ViolandoNivelDeSeguridad( _
    ByVal ContenidoActual As Single, _
    ByVal VolumenATrasvasar As Single, _
    ByVal NivelDeSeguridad As Single)
    Dim strMensaje As String
    If ContenidoActual + VolumenATrasvasar _
        > NivelDeSeguridad Then
        strMensaje = "superará el nivel de seguridad"
    Else
        strMensaje = "no alcanzará el nivel de seguridad"
    End If
    MsgBox "Tras el trasvase, el depósito" _
        & vbCrLf _
        & strMensaje, _
        vbInformation + vbOKOnly, _
        "Violación de niveles de seguridad"
End Sub

Private Sub ControlDeVolumen ( _
    ByVal Volumen As Single, _
    ByRef Cancelar As Boolean)
    ' Con este procedimiento controlamos _
    ' que no se trasvase más de lo indicado
    If Abs(Volumen) > sngTrasvaseMaximo Then
        MsgBox CStr(Volumen) & " es demasiado volumen" _
            & vbCrLf _
            & "para trasvasarlo de una vez", _
            vbCritical + vbOKOnly, _
            "Cantidad a trasvasar inadecuada"
        Cancelar = True
    End If
End Sub
```



```
Private Sub cmdExtraer_Click()  
    Dim sngVolumen As Single  
    sngVolumen = Val(Nz(txtVolumen, 0))  
    If sngVolumen <> 0 Then  
        Deposito.Extraer sngVolumen  
    Else  
        Exit Sub  
    End If  
    MuestraContenido  
End Sub  
  
Private Sub cmdIntroducir_Click()  
    Dim sngVolumen As Single  
    sngVolumen = Val(Nz(txtVolumen, 0))  
    If sngVolumen <> 0 Then  
        Deposito.Introducir sngVolumen  
    Else  
        Exit Sub  
    End If  
    MuestraContenido  
End Sub  
  
Private Sub MuestraContenido()  
    lblContenido.Caption = CStr(Deposito.Contenido)  
End Sub
```

Es interesante fijarse en el procedimiento **ControlDeVolumen** que comprueba si la cantidad a trasvasar es excesiva.

Este procedimiento es llamado desde los manejadores de eventos del depósito

Deposito_Introduciendo y **Deposito_Extrayendo**.

Los dos manejadores le pasan el parámetro **Cancelar**, que han recibido desde los correspondientes eventos **Introduciendo** y **Extrayendo**.

Si en **ControlDeVolumen** se hace que **Cancelar** tome el valor **True**, como así sucede si la cantidad a trasvasar es mayor de lo permitido por la cantidad asignada a la variable **sngTrasvaseMaximo**, ese valor se transmite al manejador del evento, y desde éste al propio evento, y por tanto a los métodos **Extraer** e **Introducir** del objeto **Deposito**. Cuando esto sucede se interrumpe el proceso de trasvase.


Podemos comprobar que se muestran los correspondientes mensajes de advertencia si los valores de volumen están fuera de los valores admitidos, o si se violan los límites de seguridad.

También vemos que el propio objeto genera los mensajes de error, como podemos apreciar en los siguiente gráficos:

Depósito marca ACME

Volumen	1000	Cantidad a trasvasar:	<input type="text" value="350"/>
Volumen Máx. Seguridad	900		<input type="button" value="Extraer"/>
Contenido	600		<input type="button" value="Introducir"/>
Volumen Mín. Seguridad	100		


Violación de niveles de seguridad

 Tras el trasvase, el depósito superará el nivel de seguridad

Depósito marca ACME

Volumen	1000	Cantidad a trasvasar:	<input type="text" value="550"/>
Volumen Máx. Seguridad	900		<input type="button" value="Extraer"/>
Contenido	0		<input type="button" value="Introducir"/>
Volumen Mín. Seguridad	100		

Cantidad a trasvasar inadecuada

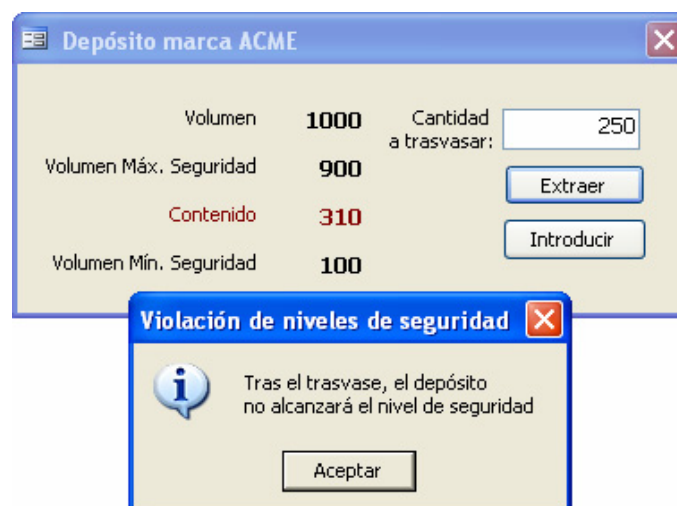
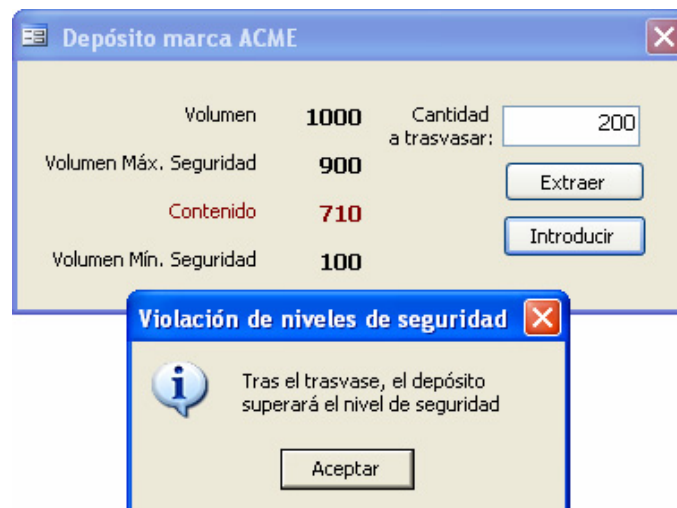
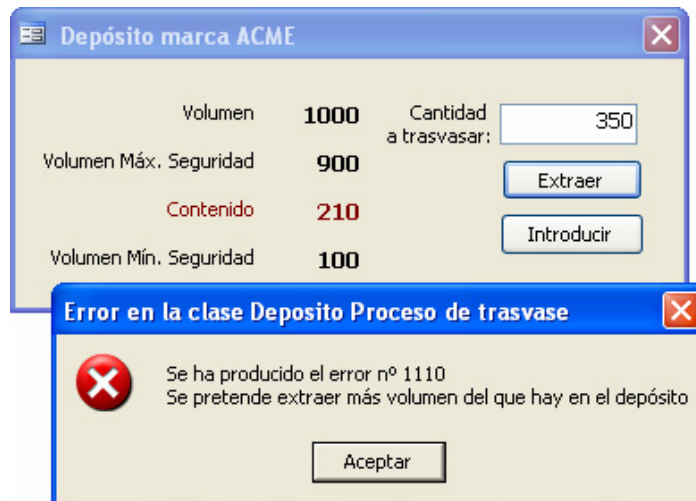
 550 es demasiado volumen para trasvasarlo de una vez

Depósito marca ACME

Volumen	1000	Cantidad a trasvasar:	<input type="text" value="300"/>
Volumen Máx. Seguridad	900		<input type="button" value="Extraer"/>
Contenido	710		<input type="button" value="Introducir"/>
Volumen Mín. Seguridad	100		

Error en la clase CDeposito Proceso de trasvase

 Se ha producido el error nº 1100
Se pretende introducir más volumen del que cabe en el depósito



Vamos que los mensajes de advertencia funcionan como está previsto.

El procedimiento **MuestraContenido** muestra la cantidad existente tras cada trasvase.

Como sugerencia, la clase **CDeposito**, se podría tomar como base, con pequeñas modificaciones, por ejemplo para controlar la validez de entradas y salidas de un artículo en un almacén, e incluso para gestionar su stock.

Eventos Initialize y Terminate de la clase.

Cuando creamos una clase VBA, ésta nos suministra los procedimientos **Initialize** y **Terminate**.

Initialize se genera justo después de que se cree el objeto, o lo que es lo mismo, tras crearse un ejemplar de la clase o una instancia de la misma.

Esto sucede tras crear un objeto con la palabra clave **New**, ya sea mediante

```
Dim MiObjeto as New MiClase
```

ó

```
Set MiObjeto = New MiClase
```

Me permito recordar aquí que no se puede crear directamente un objeto precedido por la palabra clave **WithEvents**, directamente con **As New**.

El evento **Terminate** se genera justo antes de que se destruya el objeto.

El evento **Initialize** puede usarse para asignar propiedades por defecto al objeto en un inmediateamente después de haber sido creado.

Hay que recordar que VBA, al contrario que otros lenguajes como VB.Net o C#, no posee los llamados **Constructores**, que permiten crear un objeto asignándole valores concretos a sus propiedades a la vez que éstos se crean.

Por ejemplo en VB.Net, sería factible asignar las propiedades **Nombre**, **Apellido** y **FechaDeNacimiento** al objeto **Alumno** de una clase tipo **ClsPersona**, en el momento de crearse.

```
Set Alumno = New clsPersona("Antonio", "Pérez", #3/6/65#)
```

Más adelante veremos cómo podemos superar esta carencia de los objetos de VBA y crear pseudo-constructores.

Para utilizar el evento **Initialize**, se haría de la siguiente forma

```
Private Sub Class_Initialize()  
    m_datFechaContrato = Date  
End Sub
```

En este caso asignaríamos a la variable **m_datFechaContrato** de la clase el valor de la fecha actual. Como hemos dicho, esto sucederá inmediateamente después de crearse el objeto de la clase.

El evento **Terminate**, es muy útil, por ejemplo para eliminar referencias del propio objeto a otros objetos, para cerrar enlaces con bases de datos, etc... justo antes de que se destruya el objeto. Por ejemplo

```
Private Sub Class_Terminate()  
    m_cnnMiconexion.Close  
    set m_cnnMiconexion = Nothing  
End Sub
```

Vamos a programar una sencilla clase a la que pondremos como nombre **CTrabajador** y a instanciarla, para ver el efecto de los eventos **Initialize** y **Terminate**.

```
Option Explicit

Private m_strNombre As String
Private m_datFechaContrato As Date

Private Sub Class_Initialize()
    ' Ponemos como fecha de contrato la de hoy
    FechaContrato = Date
    MsgBox "Contratado un nuevo trabajador" _
        & vbCrLf _
        & "Fecha provisional del contrato: " _
        & Format(FechaContrato, "d/m/yyyy"), _
        vbInformation + vbOKOnly, _
        "Class_Initialize"
End Sub

Public Property Get Nombre() As String
    Nombre = m_strNombre
End Property

Public Property Let Nombre(ByVal NuevoNombre As String)
    If m_strNombre <> NuevoNombre Then
        m_strNombre = NuevoNombre
    End If
End Property

Public Property Get FechaContrato() As Date
    FechaContrato = m_datFechaContrato
End Property

Public Property Let FechaContrato( _
    ByVal NuevaFecha As Date)
    If m_datFechaContrato <> NuevaFecha Then
        m_datFechaContrato = NuevaFecha
    End If
End Property

Private Sub Class_Terminate()
```

```

MsgBox "El trabajador " & Nombre _
      & vbCrLf _
      & "contratado el " _
      & Format(FechaContrato, "d/m/yyyy") _
      & vbCrLf _
      & "ha sido despedido.", _
vbInformation + vbOKOnly, _
"Class_Terminate"

End Sub

```

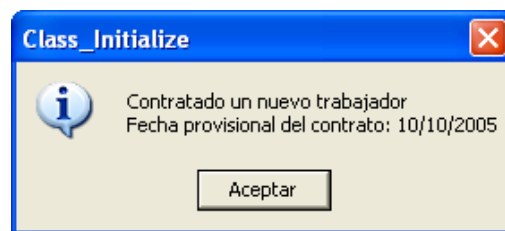
Para probar la clase y comprobar los efectos de los eventos, escribimos en un módulo normal el siguiente código:

```

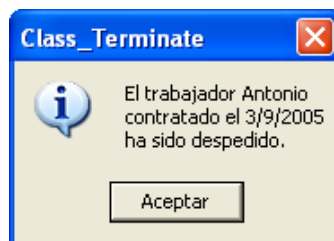
Public Sub PruebaCTrabajador()
    Dim Empleado As New CTrabajador
    Empleado.Nombre = "Antonio"
    Empleado.FechaContrato = #9/3/2005#
End Sub

```

Al ejecutar el procedimiento **PruebaCTrabajador**, nos muestra en pantalla primero



y a continuación, cuando se va a eliminar la instancia al objeto **Empleado**



Propiedades en los módulos estándar.

Hay un tema que quizás no es muy conocido.

Al igual que podemos definir propiedades de lectura y/o escritura en una clase, o en el módulo de clase de un formulario o informe, también podemos definir las en los módulos, llamémoslos así, "estándar".

Por ejemplo, sería perfectamente válido escribir el siguiente código en un módulo "normal".

```

Option Explicit

Private strNombre As String

```

```
Public Property Get Nombre() As String
    Nombre = strNombre
End Property

Public Property Let Nombre(ByVal NuevoNombre As String)
    strNombre = NuevoNombre
End Property
```

Con ello podríamos escribir desde cualquier parte del código, incluso desde módulos diferentes, lo siguiente

```
Public Sub PruebaProperty()
    Nombre = "Pepe"
    Debug.Print Nombre
End Sub
```

Y funcionará perfectamente.

Si hubiéramos definido varios módulos con esa misma propiedad, para acceder a ella, como en el caso de los objetos, podemos hacerlo mediante **NombreDelMódulo.Propiedad**.

Por ejemplo la hubiéramos escrito en el módulo **MóduloConPropiedades**, la propiedad **Nombre**, podríamos haber hecho

```
Public Sub PruebaProperty()
    MóduloConPropiedades.Nombre = "Pepe"
    Debug.Print MóduloConPropiedades.Nombre
End Sub
```

Por lo tanto, tenemos plena libertad para aprovechar las **Property** en módulos y definir propiedades, con las ventajas que éstas tienen y sin necesidad de efectuar instancias de clases.

No se interprete esto como una sugerencia para no utilizar las clases, sino más bien todo lo contrario.

Nota sobre la próxima entrega

Inicialmente me había planteado cubrir todo el tema de las clases en este capítulo.

Como habréis podido apreciar, fundamentalmente por la extensión de las líneas de código de los ejemplos, esta entrega está a punto de superar lo que se consideraría una extensión llamemos “razonable”, habiendo alcanzado las 47 páginas.

Por ello habrá una nueva entrega de nombre **Más sobre Clases y Objetos (2)**, antes de meternos a fondo con los formularios e informes.

En la próxima entrega abordaremos los temas más importantes que restan de las clases y pondré una buena cantidad de código de ejemplo.

Algunos de vosotros los consideraréis como temas avanzados, más considerando el título genérico de estas entregas “**Comencemos a programar con VBA - Access**” pero veréis que no son tan complicados como inicialmente pueden parecer.