

Comencemos a programar con
VBA - Access

Entrega 21

Más sobre Clases y Objetos
(2)

Continuamos con las clases

En la entrega anterior

- Repasamos los conceptos básicos de las clases
- Aprendimos a crear **Propiedades** de lectura y escritura
- Vimos cómo Generar y capturar **Eventos**
- Creamos clases que se llamaban a sí mismas, lo que nos permite la creación de estructuras y grafos.

En esta entrega veremos cómo

- Instanciar clases desde otras clases
- Declarar y usar Interfaces
- Emular un constructor de clase
- Emular la Herencia y el Polimorfismo

Procedimientos Friend

Hasta ahora, los procedimientos de un módulo los hemos podido definir como Privados mediante la palabra **Private** o **Dim**, o públicos mediante la palabra reservada **Public**.

En las clases hay otro tipo de alcance para definir un método o propiedad; este nuevo alcance viene definido por la palabra clave **Friend**.

Friend protege a los métodos y propiedades de una clase, de forma que sólo pueden ser invocados por procedimientos del mismo proyecto en el que está definida la clase.

Para los procedimientos del propio proyecto es como si estuviera declarado como **Public**.

Para el código externo al proyecto los métodos y propiedades declarados como **Friend** están ocultos.

Podemos incluso declarar el procedimiento **Property Get** de una propiedad **Public** y el **Property Let**, o **Property Set** como **Friend**.

Un aspecto muy práctico de esta forma de definir las propiedades es que así conseguimos que la aplicación del propio proyecto pueda leer o establecer una propiedad, mientras que al código externo, de otras aplicaciones o proyectos, esa propiedad sería de sólo lectura.

En el siguiente ejemplo la propiedad **Nombre** sería de sólo lectura para un código externo, y de Lectura / Escritura para el código del propio proyecto.

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

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

Gestión de colores - 2º Caso práctico:

Como continuación a lo visto hasta ahora, y como base para otros ejemplos de esta misma entrega vamos a desarrollar un nuevo caso práctico.

Ésta es una clase que podría llegar a ser muy útil en el trabajo diario de desarrollo.

Supongamos que recibimos el siguiente encargo:

Diseñar una clase que va a ser utilizada para controlar los colores de todos los objetos de formularios e informes que posean, una o varias propiedades que definan el color de alguno de sus elementos.

Nos piden además que podamos tener un absoluto control de los valores que tomen los componentes **Rojo**, **Verde** y **Azul** del propio color.

*Como aclaración a esto último, comentaré que el color que se ve por el monitor, es la suma de tres colores primarios, el **Rojo**, el **Verde** y el **Azul**.*





Dependiendo de la intensidad de cada uno de ellos podremos obtener todo el resto de la gama de colores.

En **VBA**, en **Access** y en **Visual Basic**, los colores se expresan mediante un valor numérico de tipo **Long**.

Algunos colores tienen expresado su valor mediante unas constantes predefinidas.

En la siguiente tabla se muestran los valores para algunos colores, sus componentes de Rojo, Verde y Azul, su valor numérico, y la constante **VBA** equivalente, si la tuvieran.

Nombre	Color	Rojo	Verde	Azul	Valor Long	Constante
Blanco		255	255	255	16777215	vbWhite
Gris claro		204	204	204	13421772	--
Gris medio		128	128	128	8421504	--
Gris oscuro		64	64	64	4210752	--
Nombre	Color	Rojo	Verde	Azul	Valor Long	Constante
Negro		0	0	0	0	vbBlack
Rojo		255	0	0	255	vbRed
Verde vivo		0	255	0	65280	vbGreen
Azul		0	0	255	16711680	vbBlue
Azul Mar Claro		0	128	255	16744192	
Amarillo		255	255	0	65535	vbYellow
Magenta		255	0	255	16711935	--
Azul turquesa		0	255	255	16776960	vbCyan

Naranja		255	128	0	33023	--
Violeta		128	0	128	8388736	--
Rojo oscuro		128	0	0	128	--
Verde		0	128	0	32768	--

El sistema que usa VBA, permite trabajar con **más de 16 millones de colores** diferentes, en concreto **16.777.216** ó lo que es lo mismo doscientos cincuenta y seis elevado al cubo.

Una cantidad más que razonable de posibles valores para nuestra vida diaria.

Para obtener el valor **Long** de un color, sabiendo las proporciones de cada color primario Rojo, Verde y Azul, VBA nos provee de la función **RGB**.

Función RGB

Devuelve un valor de tipo **Long**, correspondiente a un color expresado en sus colores primarios Rojo (**Red**), Verde (**Green**) y Azul (**Blue**).

Sintaxis

RGB(rojo, verde, azul)

Cada parámetro **rojo**, **verde** y **azul** puede tomar cualquier valor entre 0 y 255.

Por ejemplo, **RGB(0,255,255)**, que corresponde al **Azul turquesa**, llamado también **Azul Cian**, nos dará el valor **16776960**, que es a su vez el valor de la constante **vbCyan**.

La forma como calcula el valor del color la función **RGB** es la siguiente:

RGB → rojo + verde * 256 + azul * 256^2

Por ejemplo **RGB(64,128,230)** nos da **15106112**.

Comprobémoslo

64 + 128 * 256 + 230 * 256^2 = 64 + 32768 + 15073280

Lo que efectivamente suma un total de: **15106112**.

Para obtener los valores de los componentes del color, a partir del valor Long total del color, haremos los siguientes cálculos

Tenemos que **RGB(64,128,230) → 15106112**

Partiendo de esa cifra podemos obtener los valore **Byte** de los colores básicos:

15106112 mod 256 → 64 (rojo)

Int(15106112 / 256) mod 256 → 128 (verde)

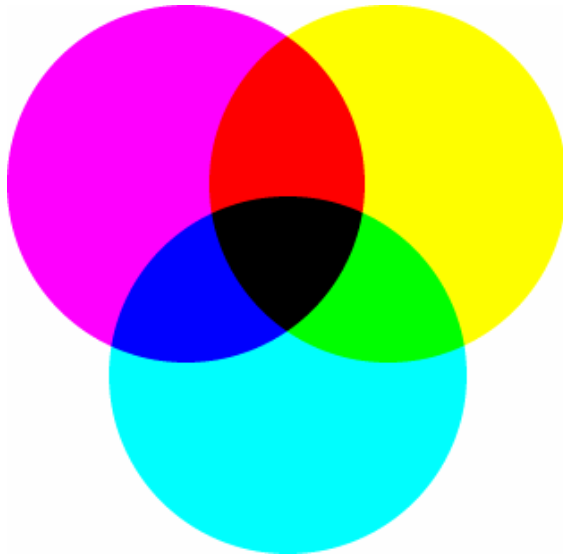
Int(15106112 / 65536) → 230 (azul)

Nota:

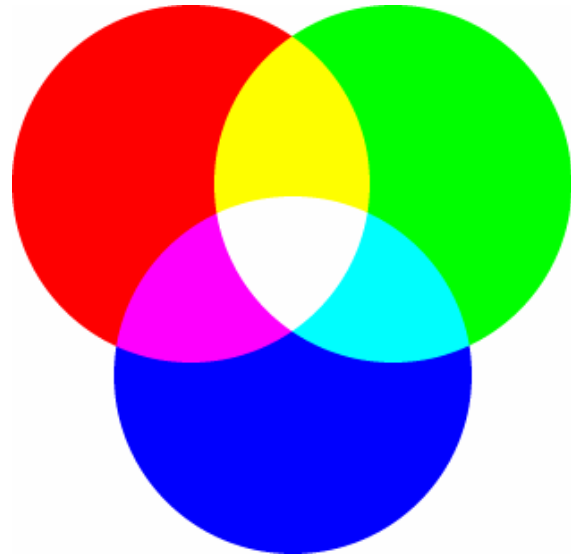
*Los monitores de ordenador, así como en general los monitores de televisión, usan el sistema de color **Aditivo**, sumándose los valores correspondientes a los tres colores luz primarios.*

Por el contrario, en la industria gráfica, así como en las pinturas o las impresoras de color, se usa el llamado **Método Sustractivo**, en el que los colores primarios son el **amarillo**, el **magenta**, el **azul cian** y el **negro**. Esto es así en los sistemas que usan el color blanco del papel como un color más y para tintes no opacos o en distribución de las tintas mediante pequeños puntos de pigmento.

En caso contrario habría que añadir además el color Blanco como un primario más.



Método Sustractivo



Método Aditivo

En la imagen superior tenemos los dos tipos de métodos y los resultados de su uso simultáneo

Clase básica para gestionar el color.

Partimos de una variable privada, `m_lngColor` que será la que guarde el color en formato `Long`.

Vamos a crear por tanto una propiedad llamada `Color`, de lectura y escritura, que leerá y escribirá en la variable `m_lngColor`.

Además queremos que el valor de la variable `m_lngColor` se pueda actualizar usando los colores primarios, y que a su vez podamos obtener el valor de cada componente de color **Rojo**, **Verde** y **Azul** de un valor concreto de `m_lngColor`.

Para ello crearemos las propiedades `ColorRojo`, `ColorVerde` y `ColorAzul` de lectura y escritura.

Código de la clase

Insertaremos un nuevo módulo de clase al que llamaremos `ClsColor`.

```
Option Explicit
```

```
Private Const con8Bits As Long = 256
```

```
Private Const con16Bits As Long = 65536 ' 256*256
```

```
Private m_lngColor As Long
Public Property Get Color() As Long
    Color = m_lngColor
End Property

Public Property Let Color(NuevoColor As Long)
    Dim blnCancelar As Boolean

    If m_lngColor <> NuevoColor Then
        m_lngColor = NuevoColor
    End If
End Property

Public Property Get ColorRojo() As Byte
    ColorRojo = m_lngColor Mod con8Bits
End Property

Public Property Let ColorRojo(NuevoColor As Byte)
    Dim BytRojo As Byte
    Dim lngColor As Long

    BytRojo = m_lngColor Mod con8Bits
    If BytRojo <> NuevoColor Then
        Color = RGB(NuevoColor, ColorVerde, ColorAzul)
    End If
End Property

Public Property Get ColorVerde() As Byte
    ColorVerde = Int(m_lngColor / con8Bits) Mod con8Bits
End Property

Public Property Let ColorVerde(NuevoColor As Byte)
    Dim BytVerde As Byte
    Dim lngColor As Long

    BytVerde = Int(m_lngColor / con8Bits) Mod con8Bits
    If BytVerde <> NuevoColor Then
        Color = RGB(ColorRojo, NuevoColor, ColorAzul)
    End If
End Property
```

```
Public Property Get ColorAzul() As Byte
    ColorAzul = Int(m_lngColor / con16Bits)
End Property

Public Property Let ColorAzul(NuevoColor As Byte)
    Dim BytAzul As Byte
    Dim lngColor As Long

    BytAzul = Int(m_lngColor / con16Bits)
    If BytAzul <> NuevoColor Then
        Color = RGB(ColorRojo, ColorVerde, NuevoColor)
    End If
End Property
```

Como se puede apreciar es una clase relativamente sencilla, que no tiene implementado ningún tipo de evento. Posee las propiedades **Color**, **ColorRojo**, **ColorVerde** y **ColorAzul** que acceden a la variable privada **m_lngColor**.

Veamos cómo la podríamos utilizar.

Utilización de la clase Color en un informe

Vamos a producir unos curiosos efectos de color en un informe.

Creemos un nuevo informe en modo diseño.

Al informe le asignamos una anchura de 14,5 cms ¡Controle los márgenes de página!

A la altura de la sección Detalle la ponemos unos 14 cms.

Lo que vamos a hacer es dibujar una serie de pequeños cuadraditos que de izquierda a derecha vayan tomando los colores de la gama del rojo al amarillo, y que hacia abajo los colores vayan hacia el azul.

Para dibujar un rectángulo en un informe, podemos usar el método **Line** del objeto **Report**. (el propio informe). Este método permite dibujar líneas rectas o rectángulos ya sean sin color interior o rellenos de un color determinado. Le recomiendo que para mayor información vea la ayuda correspondiente al método **Line** de Access.

Supongamos que lo queremos dibujar con la esquina superior izquierda a 100 unidades de distancia desde el margen izquierdo y a 200 unidades desde la parte superior de la sección.

Además queremos que tenga una anchura de 400 unidades y una altura de 300.

El rectángulo tendrá un color rojo, no sólo el perímetro, también el contenido.

La forma de hacer todo esto sería tan simple como:

```
Line(100,200)-(100 + 400, 200 + 300), vbRed, BF
```

O lo que es lo mismo

```
Line(100,200)-(500, 500), vbRed, BF
```

En el primer paréntesis están las coordenadas **x** e **y** de la **esquina superior izquierda**.

En el segundo las coordenadas **x** e **y** de la **esquina inferior derecha**.

Hay que tomar en cuenta que la coordenada **y** crece hacia abajo.

Cuando en una entrega más avanzada estudiemos la programación de los informes, veremos cómo se puede cambiar todo esto.

vbRed indica que se debe pintar en **rojo**.

B indica que en vez de una línea se dibujará un **rectángulo** y **F** que se dibujará como un rectángulo sólido.

Hasta que lleguemos al capítulo de los métodos gráficos, en informes, no voy a dar pormenores de esta instrucción, por lo que, como he comentado más arriba, si desea más información le recomiendo que consulte en la ayuda de VBA, el método **Line**.

Nota importante:

En Access sólo se pueden usar los métodos gráficos dentro de los informes.

*Al contrario que en Visual Basic, no se pueden usar ni en formularios ni en otros tipos de objetos, como los objetos **Image**.*

Simplemente para abrir boca vamos a ver cómo podemos usar este método.

Seleccionamos la sección **Detalle** del informe, mostramos la ventana de **Propiedades**, activamos la pestaña de **Eventos**, hacemos clic en el evento **Al imprimir**, pulsamos el botón con los tres botoncitos y seleccionamos el Generador de código.

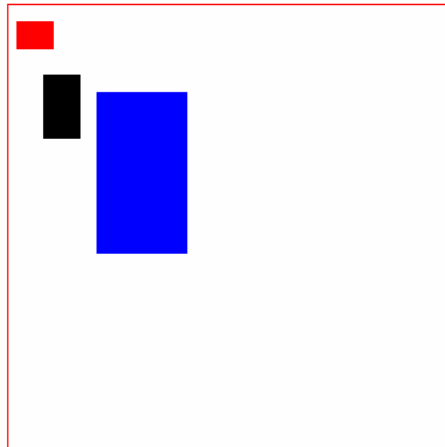
Se nos abrirá el módulo de clase del formulario en el manejador del evento **Format**

```
Private Sub Detalle_Format( _  
    Cancel As Integer, _  
    FormatCount As Integer)  
  
End Sub
```

En este evento vamos a escribir las sentencias **Line** para varios rectángulos diferentes

```
Private Sub Detalle_Format( _  
    Cancel As Integer, _  
    FormatCount As Integer)  
    Line(100,200)-(500, 500), vbRed, BF  
    Line(400,800)-(800, 1500), vbBlack, BF  
    Line(2000,2800)-(1000, 1000), vbBlue, BF  
    Line (0, 0)-(5000, 5000), vbRed, B  
End Sub
```

Si abrimos el informe, nos mostrará en pantalla lo siguiente:



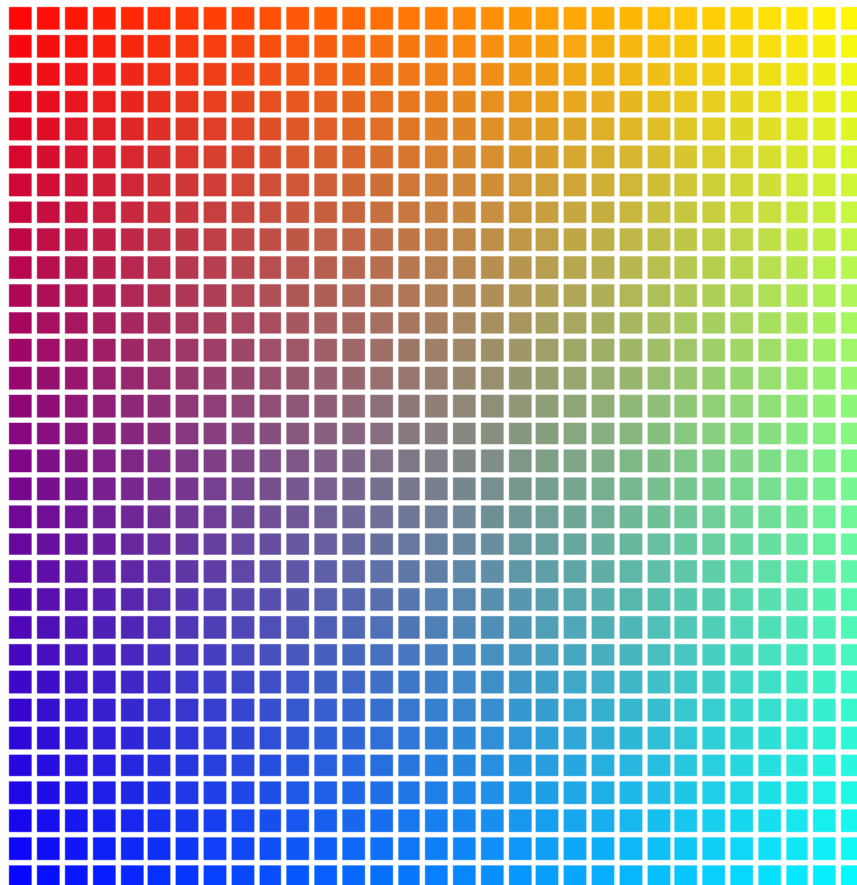
Como lo anterior era una prueba, eliminamos las sentencias **Line** anteriores y escribimos:

`Option Explicit`

```
Private Sub Detalle_Format( _  
    Cancel As Integer, _  
    FormatCount As Integer)  
    ' Lado del cuadrado  
    Const conLado As Long = 200  
    ' Separación entre cuadrados  
    Const conSeparador As Long = 60  
    Dim Color As New ClsColor  
    Dim i As Long, j As Long  
    Dim X As Long    ' Distancia al margen izquierdo  
    Dim Y As Long    ' Distancia al borde superior  
  
    For i = 1 To 32  
        ' Transición de colores en las líneas de cuadrados  
        Color.ColorRojo = 256 - 8 * (i - 1) - 1  
        Color.ColorAzul = 8 * i - 1  
        Y = conLado * i + conSeparador * (i - 1)  
        For j = 1 To 32  
            ' Transición en las columnas de cuadrados  
            Color.ColorVerde = 8 * j - 1  
            X = conLado * j + conSeparador * (j - 1)  
            Line (X, Y)-(X + conLado, Y + conLado), _  
                Color.Color, BF  
        Next j  
    Next i
```

```
Next i  
End Sub
```

Si abrimos el informe, nos mostrará una imagen que podríamos calificar de “Vistosa”. Calificarla como obra de arte sería, además de exagerado, petulante.



Si recorremos el perímetro vemos que se producen estas transiciones de color

- Del rojo al amarillo
- Del amarillo al azul turquesa
- Del azul al azul turquesa
- Del rojo al azul

Pasando por colores intermedios como los violetas, naranjas, toda una gama de verdes y algunos colores tierra.

Este resultado, en cierto modo espectacular, lo hemos conseguido simplemente escribiendo unas pocas líneas de código. Y a mí personalmente me gusta.

Si hubiéramos intentado conseguir este resultado trabajando de la forma convencional, probablemente hubiéramos necesitado más código, o un código menos intuitivo.

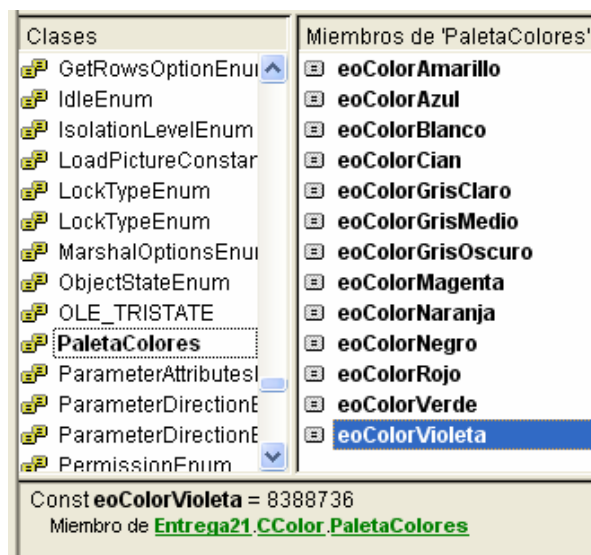
Enumeraciones en un módulo de clase

Si en el encabezado de la clase Color escribiéramos:

```
Option Explicit
```

```
Enum PaletaColores
    eoColorRojo = vbRed           ' RGB(255, 0, 0)
    eoColorVerde = vbGreen        ' RGB( 0, 255, 0)
    eoColorAzul = vbBlue          ' RGB( 0, 0, 255)
    eoColorAmarillo = vbYellow    ' RGB(255, 255, 0)
    eoColorMagenta = 16711935     ' RGB(255, 0, 255)
    eoColorCian = vbCyan          ' RGB( 0, 255, 255)
    eoColorNaranja = 33023        ' RGB(255, 128, 0)
    eoColorAzulMar = 16744192     ' RGB( 0, 128, 255)
    eoColorVioleta = 8388736      ' RGB(128, 0, 128)
    eoColorBlanco = vbWhite       ' RGB(255, 255, 255)
    eoColorGrisClaro = 13421772   ' RGB(204, 204, 204)
    eoColorGrisMedio = 8421504    ' RGB(128, 128, 128)
    eoColorGrisOscuro = 4210752   ' RGB( 64, 64, 64)
    eoColorNegro = vbBlack        ' RGB( 0, 0, 0)
End Enum
```

Y fuéramos al examinador de objetos podríamos ver lo siguiente



Vemos que, por ejemplo la constante **eoColorVioleta**, de valor **8388736**, es un miembro del tipo enumerado **PaletaColores**, perteneciente a la clase **ClsColor**.

Si pulsamos en **PaletaColores**, nos informa que es una constante de tipo **Enum**, miembro de la clase **ClsColor** y con un alcance **Público**.



¿Cómo es posible que se comporte como un conjunto de constantes públicas si no se ha utilizado delante la palabra clave **Public**?

Si miramos la ayuda de VBA, veremos que indica literalmente lo siguiente:

*Los tipos **Public Enum** de un módulo de clase no son miembros de la clase; sin embargo, se escriben en la biblioteca de tipos.*

Aparentemente hay una contradicción con lo que dice el explorador de objetos.

En realidad podríamos interpretar que es como si pasara a formar parte de la **biblioteca** de **VBA**, por lo que no sería necesario crear un ejemplar de la clase **ClsColor** para tener acceso a los valores de cada una de las constantes.

Este comportamiento tiene su lógica, ya que es muy habitual utilizar constantes enumeradas como tipos de parámetro para funciones, procedimientos o variables, tanto públicas como privadas.

Con ello podemos directamente definir parámetros para procedimientos del tipo enumerativo definido en cualquiera de las clases que tengamos en el proyecto, como si estos tipos enumerados hubiesen sido declarados **Public** en un módulo estándar.

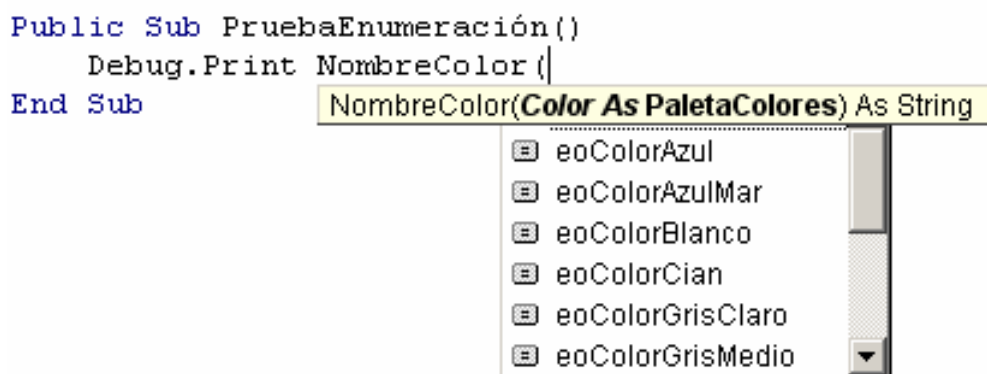
Supongamos que hemos ya escrito la clase **ClsColor** y que en un módulo normal escribimos la siguiente función:

```
Public Function NombreColor( _
    Color As PaletaColores _
) As String
    Select Case Color
        Case eoColorRojo
            NombreColor = "Rojo"
        Case eoColorVerde
            NombreColor = "Verde"
        Case eoColorAzul
            NombreColor = "Azul"
        Case eoColorAmarillo
            NombreColor = "Amarillo"
        Case eoColorMagenta
            NombreColor = "Magenta"
        Case eoColorCian
            NombreColor = "Cian"
        Case eoColorNaranja
            NombreColor = "Naranja"
```

```
Case eoColorAzulMar
    NombreColor = "Azul marino claro"
Case eoColorVioleta
    NombreColor = "Violeta"
Case eoColorBlanco
    NombreColor = "Blanco"
Case eoColorGrisClaro
    NombreColor = "Gris Claro"
Case eoColorGrisMedio
    NombreColor = "Gris Medio"
Case eoColorGrisOscuro
    NombreColor = "Gris Oscuro"
Case eoColorNegro
    NombreColor = "Negro"
Case Else
    NombreColor = "Color sin tabular"
End Select
End Function
```

A partir de este momento podemos llamar a esta función, y la ayuda en línea nos mostrará las diferentes opciones de color que podemos seleccionar.

Si escribimos unas líneas de código que llamen a la función `NombreColor` veremos lo siguiente:



Un tema que, según mi criterio, podría haberse mejorado en Visual Basic es que en un caso de estos, el compilador nos obligara a introducir sólo valores incluidos en el tipo que se le ha asignado al parámetro; en este caso cualquier valor definido en `PaletaColores`.

Por desgracia no es así. A la función `PruebaEnumeracion` podríamos pasarle no sólo valores definidos en la enumeración `PaletaColores`, definida en la clase `ClsColor`, sino que "tragaría" cualquier valor del tipo `Long`.

Supongo que los programadores de Microsoft consideraron que la sintaxis de VB debería tener este comportamiento ya que una constante enumerada no deja de ser una constante numérica de tipo `Long`.

Podemos por lo tanto escribir un procedimiento que llame a la función `NombreColor` y que utilice cualquier valor no definido en `PaletaColores`.

```
Public Sub PruebaEnumeracion()  
    Debug.Print "eoColorRojo: " & NombreColor(eoColorRojo)  
    Debug.Print "vbRed: " & NombreColor(vbRed)  
    Debug.Print "vbWhite: " & NombreColor(vbWhite)  
    Debug.Print "vbGreen: " & NombreColor(vbGreen)  
    Debug.Print "-1: " & NombreColor(-1)  
    Debug.Print "0: " & NombreColor(0)  
    Debug.Print "8388736: " & NombreColor(8388736)  
    Debug.Print "RGB(255, 128, 0): " _  
        & NombreColor( RGB(255, 128, 0) )  
    Debug.Print "eoColorGrisMedio: " _  
        & NombreColor(eoColorGrisMedio)  
  
End Sub
```

El resultado de ejecutar este procedimiento sería:

```
eoColorRojo: Rojo  
vbRed: Rojo  
vbWhite: Blanco  
vbGreen: Verde  
-1: Color sin tabular  
0: Negro  
8388736: Violeta  
RGB(255, 128, 0): Naranja  
eoColorGrisMedio: Gris Medio
```

Vemos que podemos pasar los valores:

- Directamente del tipo `PaletaColores` (`eoColorRojo`)
- Un color incluido en `PaletaColores` y también definido en la clase de VBA `ColorConstants` (`vbRed`)
- Un valor no incluido en `PaletaColores` (`-1`)
- Un color incluido en `PaletaColores` con su valor numérico (`0`) (`8388736`)
- Lo mismo pero mediante una función (`RGB(255, 128, 0)`)

Un utilidad adicional e interesante de la función **PruebaEnumeracion()** es que nos muestra un ejemplo de cómo podemos construir nuestra propia función para devolver un valor que queremos asociar a una constante numérica.

Como ya vimos anteriormente, a una clase se le puede añadir una serie de **Funciones y procedimientos Sub** públicos a los que llamamos **métodos**.

Por ejemplo, para aumentar la potencia de la clase **clsColor**, le vamos a añadir los métodos :

- **AclaraColor**, que aclarará el color que tenga almacenado, o lo que es lo mismo, aumentará sus componentes de Rojo, Azul y Verde.
- **OscureceColor**, que oscurecerá el color almacenado; lo que equivale a reducir sus componentes de Rojo, Azul y Verde.
- **InvierteColor**, que lógicamente invierte el color almacenado. Esto se consigue haciendo que cada componente de color Rojo, Azul y Verde sea igual a 255 menos el color del componente actual. Para aclararlo un poco, por ejemplo el color naranja, tal y como se ha definido en esta clase, tiene de componente Rojo 255, de Verde 128 y de Azul 0. Su complementario tendrá 0 127 y 255 por lo que su valor será el correspondiente a la función **Rgb(0, 127, 255)** lo que nos da el valor 16744192 que es un valor muy parecido al definido como **eoAzulMar**.
- **ConvierteAGris** que convertiría el color original a un tono gris. Un tono gris tiene igual cantidad de Rojo, Verde y Azul. Para simplificar el código lo que hará será convertir un color a su promedio de los tres componentes; advirtiéndolo de antemano que ese no sería el procedimiento ortodoxo para hacerlo correctamente.

Podríamos crear todo un juego de “filtros gráficos”, al estilo de los de cualquier programa de edición gráfica, pero no es el objeto de este documento.

Veamos cuál podría ser el código de estos tres métodos que vamos a añadir a la clase:

```
Public Sub AclaraColor(Optional ByVal Porcentaje As Single = 0.2)
    Dim intRojo As Integer
    Dim intVerde As Integer
    Dim intAzul As Integer
    ' Definimos como aclarado mínimo el 2%
    If Porcentaje < 0.02 Or Porcentaje > 1 Then
        Exit Sub
    Else
        intRojo = ColorRojo + (255 - ColorRojo) * Porcentaje
        If intRojo > 255 Then
            intRojo = 255
        End If
        intVerde = ColorVerde + (255 - ColorVerde) *
Porcentaje
        If intVerde > 255 Then
            intVerde = 255
```

```
        End If
        intAzul = ColorAzul + (255 - ColorAzul) * Porcentaje
        If intAzul > 255 Then
            intAzul = 255
        End If
    End If
    m_lngColor = RGB(intRojo, intVerde, intAzul)
End Sub

Public Sub OscureceColor( _
    Optional ByVal Porcentaje As Single = 0.2)
    Dim intRojo As Integer
    Dim intVerde As Integer
    Dim intAzul As Integer
    ' Definimos como oscurecimiento mínimo el 2%
    If Porcentaje < 0.02 Or Porcentaje > 1 Then
        Exit Sub
    Else
        intRojo = ColorRojo * (1 - Porcentaje)
        If intRojo < 0 Then
            intRojo = 0
        End If
        intVerde = ColorVerde * (1 - Porcentaje)
        If intVerde < 0 Then
            intVerde = 0
        End If
        intAzul = ColorAzul * (1 - Porcentaje)
        If intAzul < 0 Then
            intAzul = 0
        End If
    End If
    m_lngColor = RGB(intRojo, intVerde, intAzul)
End Sub

Public Sub InvierteColor()
    Dim bytRojo As Byte
    Dim bytVerde As Byte
    Dim bytAzul As Byte

    bytRojo = ColorRojo
```



```
    bytVerde = ColorVerde
    bytAzul = ColorAzul
    m_lngColor = RGB(bytRojo, bytVerde, bytAzul)
End Sub

Public Sub ConvierteAGris()
    Dim bytRojo As Byte
    Dim bytVerde As Byte
    Dim bytAzul As Byte
    Dim bytGris As Byte

    bytRojo = ColorRojo
    bytVerde = ColorVerde
    bytAzul = ColorAzul

    bytGris = CByte((ColorRojo + ColorVerde + ColorAzul) / 3)
    m_lngColor = RGB(bytGris, bytGris, bytGris)
End Sub
```

Más adelante vamos a realizar un ejemplo en el que utilizaremos alguno de estos métodos.

Herencia y Polimorfismo

Además de la **Encapsulación**, ocultación de algunos miembros de una clase, con acceso controlado a los mismos, que ya vimos en la entrega anterior, comentamos que en el paradigma de la **Programación Orientada a Objetos**, se utilizan otras dos características:

La **Herencia** y el **Polimorfismo**. Hay una cuarta característica, la **Herencia Múltiple**, pero ésta no se considera como estrictamente necesaria.

Por herencia se entiende la capacidad de crear una clase, simplemente declarando que una clase nueva hereda de otra previamente existente.

La clase creada por herencia, automáticamente debería tener los mismos métodos y propiedades que la clase "Padre", o al menos aquéllos que no estén especificados como "no heredables".

Esto ocurre, por ejemplo, con VB.Net.

Si tenemos una clase `clsPersona` que tiene las propiedades

```
Nombre
Apellido1
Apellido2
FechaNacimiento
```

Y los métodos

```
Edad
NombreCompleto
```

Si creáramos la clase `clsTrabajador` de esta manera

```
Friend Class clsTrabajador
    Inherits clsPersona
End Class
```

¡Ojo! que el anterior es código de VB.Net, no de VBA ni de Visual Basic.

La clase `clsTrabajador` automáticamente heredarán las propiedades y los métodos que tiene la clase `clsPersona`.

La palabra reservada `Inherits`, puesta delante de `clsPersona` indicará al compilador que la clase `clsTrabajador` se va a heredar desde la clase `clsPersona`.

Otra particularidad de las clases heredadas en VB.Net es que en ellas se pueden redefinir o **sobrecargar** los métodos de la clase, con lo que cuando se llame a un método redefinido, se ejecutará el método que se ha escrito en la clase hija, en vez del de la clase padre. En este caso, si se deseara que fuese el método de la clase Padre el que se ejecutara, se podría hacer invocando a la clase Padre, a través de la clase Hija.

Si tenemos dos clases que heredan de una clase común podría ser que aunque un método se llame de la misma manera en las dos clases, en realidad ejecuten código muy diferente.

También podría ocurrir que dos métodos con el mismo nombre en dos clases diferentes, se distingan externamente sólo por el número ó el tipo de los parámetros.

También se podría definir el mismo método varias veces en una misma clase, con código diferente, cambiando los tipos o el número de parámetros del mismo.

Estos cuatro últimos párrafos hacen referencia a lo que se llaman **métodos Polimórficos**.

Por desgracia, algo que está implementado en lenguajes como **VB.Net** y **C#**, estrictamente hablando, no lo está en VBA ni en Visual Basic.

Es decir, **VBA no tiene ni Polimorfismo ni Herencia**.

La buena noticia es que podemos diseñar mecanismos que, de alguna manera, **emulen** el **Polimorfismo** y la **Herencia** en VBA. Uno de estos mecanismos es la utilización de **Interfaces**. Pero sin la potencia propia de lenguajes como **VB.Net** o **C#**,

Interfaces

Una interfaz es el conjunto de propiedades y métodos de una clase a los que se puede acceder mediante el código. En otras palabras, aquellos atributos de la clase a los que podemos acceder directamente desde una instancia, o ejemplar de la clase, porque tienen un “alcance” de tipo **Public**. En resumidas cuentas es “la cara (faz)” que nos muestra la propia clase.

En el caso concreto de la clase **ClsColor**, tenemos

- Propiedad **Color** de tipo Long de Lectura / Escritura
- Propiedad **ColorRojo** de tipo Byte de Lectura / Escritura
- Propiedad **ColorVerde** de tipo Byte de Lectura / Escritura
- Propiedad **ColorAzul** de tipo Byte de Lectura / Escritura

Si por ejemplo hubiéramos definido un método público de tipo **sub** llamado **AclaraColor** y un método público de tipo **Function** llamado **ColorInverso**, que devolviera un valor **Long**, ambos métodos también formarían parte de la interfaz de **ClsColor**.

El respetar la interfaz de las clases, es importante, ya que un programador espera que una clase se comporte de una forma determinada.

Por ejemplo, si hemos definido una nueva versión de una clase concreta, deberemos respetar el interfaz de la clase anterior.

Un programador espera que si una clase tenía la propiedad **Nombre**, la nueva versión de la clase, la siga manteniendo, y que si esa propiedad tenía un parámetro de tipo **String**, el parámetro de la propiedad **Nombre** en la nueva versión de la clase, sea también de tipo **string**.

Una nueva versión de la clase, debe mantener las propiedades, los métodos y los parámetros en propiedades y métodos, que tenía la clase anterior. Debe haber una “compatibilidad hacia atrás” entre una versión y las anteriores..

Si esto no ocurriera así, podrían darse un gran número de errores en cascada, convirtiendo la labor de depuración y mantenimiento de la aplicación en un verdadero infierno.

¿Ha oído hablar del “**infierno de las dlls**”, cuando no se han respetado las interfaces entre versión y versión?.

Podría llegar a aceptarse que una nueva versión de la clase tuviera más propiedades y métodos, o que incluso alguna de las propiedades tuviera más parámetros, siempre que estos fuesen de tipo opcional, aunque esto, estrictamente hablando, sería romper el “contrato” de la interfaz.

Lo que nunca debería ocurrir es que las llamadas a los procedimientos o propiedades de la clase base, sean incompatibles con las llamadas a la nueva versión de la clase.

Por eso se suele decir que la Interfaz de una clase es un contrato que debe respetarse.

Existe un tipo especial de clases que, en VBA, podríamos definir como **Clases Abstractas** que declaran las propiedades y los métodos que van a “publicar” al exterior, pero sin tener desarrollado, ni en las propiedades ni en los métodos, código para implementarlos.

A estas clases se les llama **Interfaces**.

Se suelen grabar con el nombre precedido de la letra **I**.

Por ejemplo **IColor**, **IElementoGráfico**, **IDireccionable**...

Creación de una Interfaz

Supongamos que queremos hacer un programa para controlar Bichos, en concreto gatos, perros y leones, que pueden comerse entre ellos, o al menos intentarlo.

Primero vamos a crear la interfaz correspondiente al tipo genérico **IBicho**.

Para ello insertamos un nuevo módulo de clase al que le ponemos como nombre **IBicho**.

```
Option Explicit

Function Tipo() As String
    ' Código del método a implementar en la clase
End Function

Sub Comer(Bicho As Object)
    ' Código del método a implementar en la clase
End Sub

Function Sonido() As String
    ' Código del método a implementar en la clase
End Function
```

Como vemos, la interfaz **Bicho** va a mostrar al resto del mundo, los métodos **Tipo**, **Comer** y **Sonido**. Los métodos **Tipo**, y **Sonido** devuelven una cadena **String**.

La clase **clsGato** mostrará la propiedad **Nombre**, y los métodos **Correr**, **Cazar** y **Maullar**

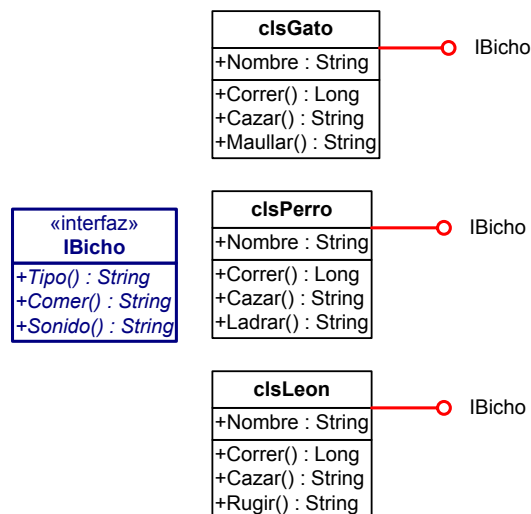
La clase **clsPerro** mostrará **Nombre**, **Correr**, **Cazar** y **LadRAR**

La clase **clsLeon** mostrará **Nombre**, **Correr**, **Cazar** y **Rugir**

Las tres clases implementan la interfaz **IBicho**.

Aunque Access acepta métodos sin ninguna implementación, conviene al menos poner una línea comentada, ya que si quisiéramos pasar el código a Visual Basic, su compilador podría eliminar los métodos sin implementar, considerando que no tienen ninguna utilidad, y dar problemas a la hora de compilar y ejecutar el código.

Diagrama UML de las clases

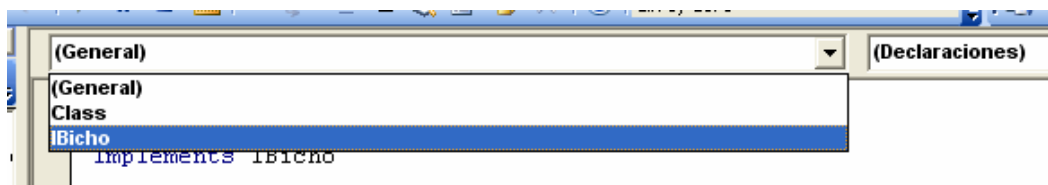


Vamos ahora a crear la clase **clsGato**, para lo que insertamos el correspondiente módulo de clase.

Para que pueda implementar la interfaz **IBicho**, en el encabezado de la clase ponemos la instrucción:

Implements IBicho

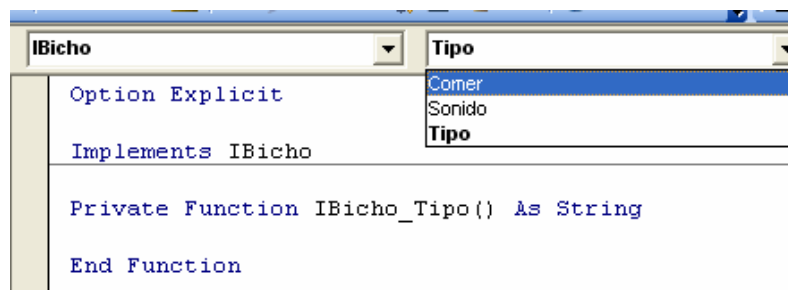
Tras esto, en el editor de código podemos seleccionar la interfaz **IBicho** del gato, y el editor nos escribirá la estructura del método **IBicho_Tipo**.



Private Function IBicho_Tipo() As String

End Function

Podemos seleccionar el resto de los métodos de **IBicho**, y nos los irá poniendo en el editor



Tras seleccionarlasm todas tendremos los tres métodos

Option Explicit

```
Implements IBicho

Private Sub IBicho_Comer(Bicho As Object)

End Sub

Private Function IBicho_Tipo() As String

End Function

Private Function IBicho_Sonido() As String

End Function
```

Como adelanto, diré que cuando hagamos que el objeto **clsGato** muestre la interfaz **IBicho**, mostrará sólo estos métodos. Si la interfaz **IBicho** tuviera alguna propiedad o método **Sub**, también los mostraría. De todo esto veremos un ejemplo más adelante.

Si observamos los métodos anteriores, tienen la siguiente estructura

```
Private Function NombreInterfaz_Método(Parámetros) As Tipo
```

Si hubieran sido métodos de tipo **Sub**

```
Private Sub NombreInterfaz_Método(Parámetros)
```

De forma similar, si las tuviera, mostraría las propiedades.

```
Private Property Let Interfaz_Propiedad (parámetros)
Private Property Set Interfaz_Propiedad (parámetros)
Private Property Get Interfaz_Propiedad (parámetros) _
    As TipoDevuelto
```

Esto nos puede recordar algo a la construcción de los manejadores de eventos que hemos visto con anterioridad.

Puede sorprender que a pesar de que éstos son los métodos y propiedades que va a mostrar la interfaz, se declaren como **Private**. Esto es así para que no sean visibles cuando creamos una instancia de la clase que implementa la interfaz.

Para comprobarlo, añadiremos a la clase la estructura correspondiente al resto de los miembros públicos de **clsGato**; es decir la propiedad **Nombre**, y los métodos **Correr**, **Cazar** y **Mauullido**.

Tras todo ello lo tendremos el siguiente código:

```
Option Explicit

Implements IBicho

Public Property Get Nombre() As String
```

```
End Property

Public Property Let Nombre (ByVal NuevoNombre As String)

End Property

Public Function Correr() As Long

End Function

Public Function Cazar (Bicho As Object) As String

End Function

Public Function Maullido() As String

End Function

Private Sub IBicho_Comer (Bicho As Object)

End Sub

Private Function IBicho_Tipo() As String

End Function

Private Function IBicho_Sonido() As String

End Function
```

Hasta aquí sólo tenemos el esqueleto de la clase, que posteriormente completaremos.

Para ver los efectos de todo esto, vamos a crear un módulo estándar con un procedimiento que creará una instancia de la clase **clsGato**.

En ese módulo creamos el procedimiento **ProbarInterfaz()**.

Conforme vamos escribiendo el código vemos que, como era de esperar, nos muestra los atributos públicos de la clase

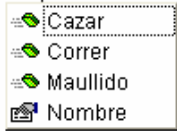
```

Option Compare Database
Option Explicit

Public Sub ProbarInterfaz()
    Dim gtoMicifuz As New clsGato

    gtoMicifuz.|
End Sub

```



Vamos a ver ahora cómo se invoca la interfaz **IBicho**.


```

Public Sub ProbarInterfaz()
    Dim gtoMicifuz As New clsGato
    Dim Bicho As IBicho

    gtoMicifuz.Nombre = "Micifuz"

    Set Bicho = gtoMicifuz
    Debug.Print Bicho.
End Sub

```



Para ello hemos declarado una variable del tipo de la interfaz **IBicho**:

```
Dim Bicho As IBicho
```

A continuación le asignamos la variable que hace referencia a la clase **clsGato** (el gato en sí mismo).

```
Set Bicho = gtoMicifuz
```

Ahora **Bicho** hace referencia al gato, sin embargo oculta todos los atributos propios del gato y sólo muestra los atributos de la interfaz **IBicho**.

¿Qué ventaja tenemos con esto?

Si creáramos las clases **clsPerro** y **clsLeon**, ambas implementando la interfaz **IBicho**. Podríamos asignarlas a variables declaradas como **IBicho**, y manejaríamos los métodos y propiedades de la interfaz.

Con eso conseguiríamos utilizar objetos de clases diferentes mediante los mismos métodos y propiedades en todos los casos.

Las posibilidades que proporciona esta técnica son muy amplias, máxime teniendo en cuenta que podemos implementar diferentes interfaces con la misma clase.

Por ejemplo podríamos haber hecho

```

Option Explicit

Implements ISerVivo
Implements IBicho

```

Esto suponiendo que hubiéramos definido una interfaz de nombre **ISerVivo**.

A continuación podríamos haber hecho


```
Dim gtoMicifuz As New clsGato
Dim SerVivo As ISerVivo
Dim Bicho As IBicho
```

```
Set SerVivo = gtoMicifuz
Set Bicho = gtoMicifuz
```

Ahora **gtoMicifuz** mostraría la interfaz de **clsGato**, **SerVivo** mostraría la interfaz de **ISerVivo** y **Bicho** la de **IBicho**.

Las interfaces permiten hacer que objetos procedentes de clases distintas, muestren una serie de métodos y propiedades iguales. Con ello conseguimos emular de una forma coherente el polimorfismo con Visual Basic y VBA.

¿Cómo podríamos utilizar esto?

Vamos a crear las clases **clsGato**, **clsPerro** y **clsLeon** definitivas

Clase clsGato:

```
Option Explicit

Implements IBicho

Private Const conlngDistancia As Long = 200
Private m_Nombre As String
Private m_lngDistancia As Long

Private Sub Class_Initialize()
    ' Asignamos una capacidad aleatoria _
    ' de desplazamiento al gato
    Randomize Timer
    m_lngDistancia = conlngDistancia + Rnd *
conlngDistancia
End Sub

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

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

Public Function Correr() As Long
    Correr = m_lngDistancia
```

```
End Function
```

```
Public Function Cazar(Bicho As Object) As String
```

```
    Cazar = "El gato " & Nombre _  
        & vbCrLf _  
        & "ha cazado al " _  
        & IBicho_Tipo _  
        & " " _  
        & Bicho.Nombre
```

```
End Function
```

```
Public Function Maullido() As String
```

```
    Maullido = "¡Miau!, ¡Miau!"
```

```
End Function
```

```
Private Sub IBicho_Comer(Bicho As Object)
```

```
    MsgBox "El gato " & Nombre _  
        & vbCrLf _  
        & "se está comiendo a " _  
        & Bicho.Nombre
```

```
End Sub
```

```
Private Function IBicho_Tipo() As String
```

```
    IBicho_Tipo = "gato"
```

```
End Function
```

```
Private Function IBicho_Sonido() As String
```

```
    IBicho_Sonido = Maullido
```

```
End Function
```

Podemos observar que desde dentro de un procedimiento de la Interfaz se puede llamar a un método de la clase, y a la inversa.

Esto ocurre, por ejemplo en

```
Private Function IBicho_Sonido() As String
```

```
    IBicho_Sonido = Maullar
```

```
End Function
```

Desde el método `IBicho_Sonido` de la interfaz se invoca el método `Maullar` de la clase.

Aunque los métodos propios de la clase no sean visibles desde una interfaz, un método de la interfaz, desde dentro de la clase, sí podrá ver los métodos de la clase, y viceversa.

Clase clsPerro

```
Option Explicit

Implements IBicho

Private Const conlngDistancia As Long = 2000
Private m_Nombre As String
Private m_lngDistancia As Long

Private Sub Class_Initialize()
    Randomize Timer
    m_lngDistancia = conlngDistancia + Rnd *
conlngDistancia
End Sub

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

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

Public Function Correr() As Long
    Correr = m_lngDistancia
End Function

Public Function Cazar(Bicho As Object) As String
    Cazar = "El perro " & Nombre _
        & vbCrLf _
        & "ha cazado a " _
        & Bicho.Nombre
End Function

Public Function Ladrido() As String
    Ladrido = "¡¡Guau guau!!"
End Function

Private Sub IBicho_Comer(Bicho As Object)
    MsgBox "El perro " & Nombre _
```

```
        & vbCrLf _  
        & "se está comiendo a " _  
        & Bicho.Nombre  
End Sub  
  
Private Function IBicho_Tipo() As String  
    IBicho_Tipo = "perro"  
End Function  
  
Private Function IBicho_Sonido() As String  
    IBicho_Sonido = Ladrido  
End Function
```

Y este será el código de la **Clase clsLeon**:

```
Option Explicit  
  
Implements IBicho  
  
Private Const conlngDistancia As Long = 1000  
Private m_Nombre As String  
Private m_lngDistancia As Long  
  
Private Sub Class_Initialize()  
    ' Asignamos una capacidad aleatoria _  
    ' de desplazamiento al gato  
    Randomize Timer  
    m_lngDistancia = conlngDistancia + Rnd *  
    conlngDistancia  
End Sub  
  
Public Property Get Nombre() As String  
    Nombre = m_Nombre  
End Property  
  
Public Property Let Nombre(ByVal NuevoNombre As String)  
    m_Nombre = NuevoNombre  
End Property  
  
Public Function Correr() As Long  
    Correr = m_lngDistancia  
End Function
```

```
Public Function Cazar(Bicho As Object) As String
    Cazar = "El león " & Nombre _
        & vbCrLf _
        & "ha cazado al " _
        & IBicho_Tipo _
        & " " _
        & Bicho.Nombre
End Function

Public Function Rugido() As String
    Rugido = "¡¡Grrrrrrrr Grrrrrrrrrr!!"
End Function

Private Sub IBicho_Comer(Bicho As Object)
    MsgBox "El león " & Nombre _
        & vbCrLf _
        & "se está comiendo a " _
        & Bicho.Nombre
End Sub

Private Function IBicho_Tipo() As String
    IBicho_Tipo = "león"
End Function

Private Function IBicho_Sonido() As String
    IBicho_Sonido = Rugido()
End Function
```

Ejemplo sobre la creación y uso de interfaces.

Vamos a crear la interfaz correspondiente a un elemento gráfico.

Para los nombres de las propiedades y métodos de la interfaz, voy a elegir sus equivalentes en inglés. Esto es algo muy habitual, ya que facilita la comprensión del código, sobre todo si éste va a tener una distribución internacional.

Estos son los atributos básicos que vamos a definir para los objetos que implementen la interfaz **iGraficElement**.

Color, es el color que va a tener la línea que define el perímetro, por ejemplo de un rectángulo o una elipse. En los elementos gráficos reales esta propiedad es **ForeColor**, y la que define el color del relleno o del fondo es **BackColor**, pero en este ejemplo vamos a simplificar sus atributos.

Left, es la distancia respecto del borde izquierdo, de la esquina superior izquierda, o del extremo izquierdo de un segmento.

Top, es equivalente a la propiedad **Left**, pero respecto al borde superior.

Whidth, es la anchura del elemento gráfico.

Height, es su altura.

Estas cuatro últimas propiedades las poseen todos los controles visibles.

Draw hace que se dibuje, el objeto en el lienzo **Canvas**.

¡Manos a la obra!

Insertamos un nuevo módulo de clase, le ponemos como nombre **IGraficElement**.

El código de la Interfaz que plasma los anteriores atributos sería tan simple como:

```
Option Explicit

Property Get Color () As Long
    ' Código de la propiedad a implementar en la clase
End Property

Property Let Color (NewColor As Long)
    ' Código de la propiedad a implementar en la clase.
End Property

Property Get Left() As Integer
    ' Código de la propiedad a implementar en la clase.
End Property

Property Let Left(ByVal X As Integer)
    ' Código de la propiedad . . .
End Property

Property Get Top() As Integer
    ' Código de la propiedad . . .
End Property

Property Let Top(ByVal Y As Integer)
    ' Código de la propiedad . . .
End Property

Property Get Whidth() As Integer
    ' Código de la propiedad . . .
End Property
```

```
Property Let Whidth(ByVal NewValue As Integer)
    ' Código de la propiedad . . .
End Property

Property Get Height() As Integer
    ' Código de la propiedad . . .
End Property

Property Let Height(ByVal NewValue As Integer)
    ' Código de la propiedad . . .
End Property

Property Get Canvas() As Object
    ' Código de la propiedad . . .
End Property
    ' Como la propiedad Canvas devuelve un objeto _
    ' que es el "lienzo" sobre el que se va a pintar _
    ' Será del tipo Property Set
Property Set Canvas(NewCanvas As Object)
    ' Código de la propiedad . . .
End Property

Sub Draw()
    ' Código del método para Dibujar . . .
End Sub
```

Como ya hemos comentado, no hemos desarrollado ninguna propiedad ni método, Éstos deben ser desarrollados en las clases que implementen esta interfaz.

Es tradicional poner una línea comentada, en las propiedades y métodos de las interfaces ya que, si estuviéramos desarrollando una interfaz con Visual Basic de Visual Studio, el compilador podría llegar a eliminar un procedimiento en una clase que no tuviera escrito nada en su interior.

En Access, el único objeto que admite dibujar en su interior es el objeto **Report**, pero por ejemplo en Visual Basic de la plataforma Visual Studio, podemos dibujar en un objeto **Form**, en un control **Picture Box**, en un control **Image**, etc. Estos objetos tienen desarrollados métodos gráficos equivalentes en todos ellos, funcionando de una forma polimórfica.

Por ejemplo, para dibujar una línea, tanto en un objeto **Report** de Access, como en un **Picture** o un **Form** de Visual Basic usaremos el método **Line**.

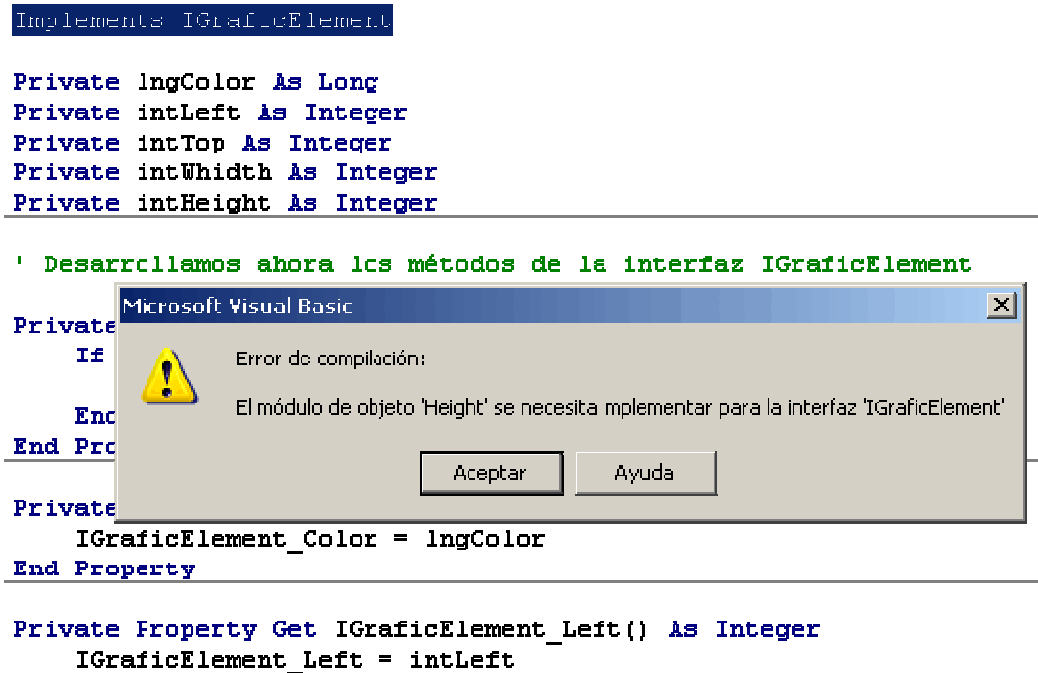
Para dibujar una elipse, un círculo, un sector circular ó elíptico, usaremos el método **Circle**.

Hay además una multitud de objetos ActiveX que también implementan esos métodos, e incluso, utilizando Visual Studio, podríamos desarrollar nuestros propios controles ActiveX que implementaran métodos gráficos.

Nota:

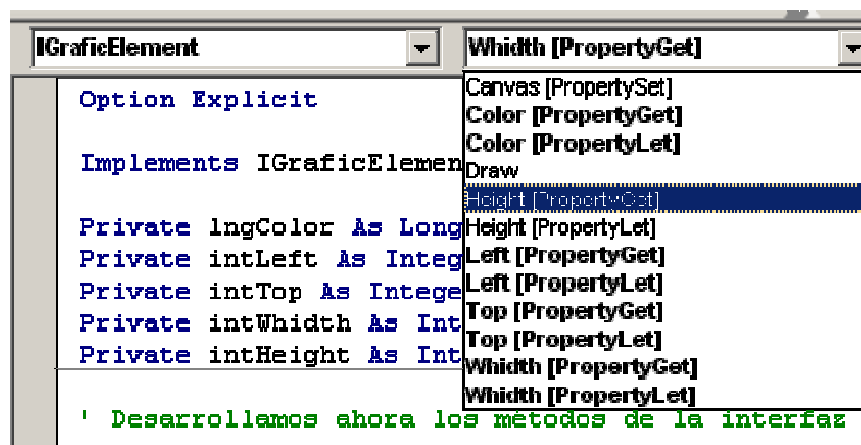
No entra en el objetivo de este trabajo mostrar cómo desarrollar controles ActiveX con Visual Basic, con C++ o con cualquier otra plataforma de desarrollo de software..

Cuando diseñemos una clase que implemente esta interfaz deberá incluir todos los métodos de esta; caso contrario dará un error de compilación, como podemos ver en la siguiente imagen:



En este caso nos avisa que todavía que faltan métodos por desarrollar:

Para desarrollar cada uno de los métodos, en los cuadros combinados ubicados en la parte superior del editor de código, seleccionamos el objeto **IGraficElement** y el método a escribir:



También podríamos escribirlo de forma directa, poniendo el nombre de la interfaz seguida del carácter línea inferior y el nombre del método.

Vamos a crear tres clases que implementarán la interfaz **IGraficElement**.

- `clsRecta`
- `clsRectangulo`
- `clsElipse`

En cada una de ellas desarrollaremos los métodos declarados en la interfaz.

Lo que realmente va a cambiar entre las tres clases será el método `Draw` que las dibuja en el Lienzo (`Canvas`).

Este sería el código básico de la clase `clsRecta`:

```
Option Explicit
```

```
Implements IGraficElement
```

```
Private lngColor As Long  
Private intLeft As Integer  
Private intTop As Integer  
Private intWhidth As Integer  
Private intHeight As Integer  
Private objCanvas As Object
```

```
' Desarrollamos ahora los métodos de IGraficElement
```

```
Private Property Let IGraficElement_Color(NewColor As Long)  
    If lngColor <> NewColor Then  
        lngColor = NewColor  
    End If  
End Property
```

```
Private Property Get IGraficElement_Color() As Long  
    IGraficElement_Color = lngColor  
End Property
```

```
Private Property Get IGraficElement_Left() As Integer  
    IGraficElement_Left = intLeft  
End Property
```

```
Private Property Let IGraficElement_Left( _  
    ByVal NewPosition As Integer)  
    If intLeft <> NewPosition Then  
        intLeft = NewPosition
```

```
        End If
    End Property

    Private Property Get IGraficElement_Top() As Integer
        IGraficElement_Top = intTop
    End Property

    Private Property Let IGraficElement_Top( _
        ByVal NewPosition As Integer)
        If intTop <> NewPosition Then
            intTop = NewPosition
        End If
    End Property

    Private Property Get IGraficElement_Whidth() As Integer
        IGraficElement_Whidth = intWhidth
    End Property

    Private Property Let IGraficElement_Whidth( _
        ByVal NewValue As Integer)
        If intWhidth <> NewValue Then
            intWhidth = NewValue
        End If
    End Property

    Private Property Get IGraficElement_Height() As Integer
        IGraficElement_Height = intHeight
    End Property

    Private Property Let IGraficElement_Height( _
        ByVal NewValue As Integer)
        If intHeight <> NewValue Then
            intHeight = NewValue
        End If
    End Property

    Private Property Get IGraficElement_Canvas() As Object
        Set IGraficElement_Canvas = objCanvas
    End Property
```

```
Private Property Set IGraficElement_Canvas( _  
    NewCanvas As Object)  
    Set objCanvas = NewCanvas  
End Property  
  
Private Sub IGraficElement_Draw()  
    Dim X0 As Integer  
    Dim Y0 As Integer  
    Dim X1 As Integer  
    Dim Y1 As Integer  
  
    ' Para dibujar la recta usaremos el método _  
    Line(Coords. Origen)-(Coords. Final) Color  
    X0 = intLeft  
    If intHeight < 0 Then  
        Y0 = intTop  
        Y1 = intTop - intHeight  
    Else  
        Y0 = intTop  
        Y1 = Y0 + intHeight  
    End If  
    X1 = X0 + intWhidth  
End Sub
```

De forma semejante desarrollaríamos las clases correspondientes a **clsRectangulo** y **clsElipse**.

Lo que cambiaría en ellas sería el método **Draw**, aunque en la clase **clsRectangulo** el cambio sería muy pequeño, ya que bastaría con añadirle el parámetro **B** al método **Line**.

```
Public Sub Draw()  
    Dim X0 As Integer  
    Dim Y0 As Integer  
    Dim X1 As Integer  
    Dim Y1 As Integer  
  
    ' Para dibujar el rectángulo usaremos el método _  
    Line(Coords. Origen)-(Coords. Final), Color, B  
    X0 = intLeft  
    If intHeight < 0 Then  
        Y0 = intTop  
        Y1 = intTop - intHeight
```

```
Else
    Y0 = intTop
    Y1 = Y0 + intHeight
End If
X1 = X0 + intWhidth
objCanvas.Line (X0, Y0)-(X1, Y1), lngColor, B
End Sub
```

En el caso de la clase **clsElipse** los cambios serían más sustanciales, ya que usará el método **Circle**, al que se le pasan las coordenadas del **centro**, el **radio**, **color** y **aspecto**.

(Ver el método **Circle** en la ayuda de VBA)

Las coordenadas del centro y el aspecto (relación entre la altura y la anchura), las podemos extraer de las variables

```
intLeft
intTop
e
intHeight
```

Podría ser algo así como esto:

```
Public Sub Draw()
    Dim X0 As Integer
    Dim Y0 As Integer
    Dim X1 As Integer
    Dim Y1 As Integer
    Dim CentroX As Integer
    Dim CentroY As Integer
    Dim sngAspecto As Single

    ' Para dibujar la Elipse usaremos el método _
    Circle(Coords.Centro), Color, Aspecto
    X0 = intLeft
    If intHeight < 0 Then
        Y0 = intTop
        Y1 = intTop - intHeight
    Else
        Y0 = intTop
        Y1 = intTop + intHeight
    End If
    X1 = intLeft + intWhidth
    CentroX = X0 + intWhidth / 2
    CentroY = Y0 + intHeight / 2
    sngAspecto = intHeight / intWhidth
```

```
objCanvas.Circle (CentroX, CentroY), _  
                intWhidth / 2, _  
                lngColor, , , sngAspecto
```

```
End Sub
```

Uso de las clases y de la interfaz

Vamos a ver cómo utilizar estas clases en Access.

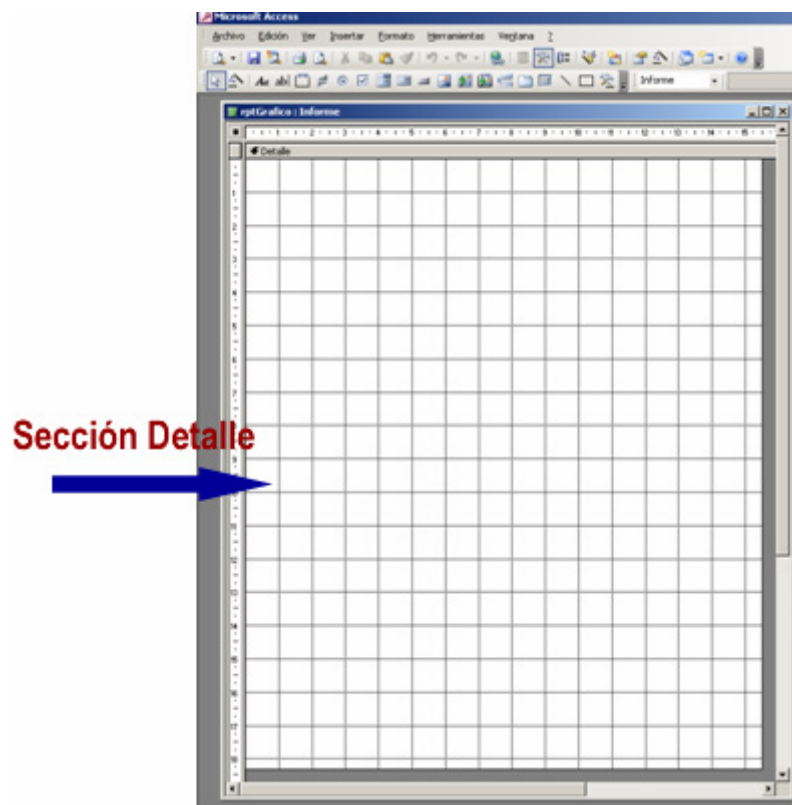
Ya he comentado que, en Access, el único objeto que admite métodos gráficos es el objeto Report.

Primero crearemos un informe que no esté ligado a ningún conjunto de datos () y al que llamaremos **rptGrafico**.

Haremos que la sección Detalle adquiera un tamaño adecuado para que se pueda dibujar en ella, ya que vamos a utilizar el evento Al imprimir (**Print**) de la misma.

```
Private Sub Detalle_Print( _  
    Cancel As Integer, _  
    PrintCount As Integer)
```

Yo la he diseñado tal y como aparece en el gráfico:



En la cabecera del módulo de clase del informe declararemos las clases que vamos a utilizar. Lógicamente esas clases, así como la interfaz **IGraficElement** deberán estar en módulos del mismo fichero Access.

Option Explicit

```
Dim objGrafico As New IGraficElement
Dim Recta As New clsRecta
Dim Rectangulo As New clsRectangulo
Dim Elipse As New clsElipse
Dim ColorObjeto As New clsColor
```

Al declararlas con **New**, desde ese mismo instante ya hemos creado una instancia de cada una de las cuatro clases.

En el evento **Al abrir (Open)** del informe, asignaremos como lienzo (**Canvas**) el propio informe (recordemos que **Me** hace referencia al propio objeto la clase actual que en este caso es el informe en cuyo módulo de clase está escrito el código que incluye **Me**):

```
Private Sub Report_Open(Cancel As Integer)
    ' Asignamos el Lienzo a cada clase
    Set Recta.Canvas = Me
    Set Rectangulo.Canvas = Me
    Set Elipse.Canvas = Me
End Sub
```

Vamos a utilizar el evento **Al imprimir (Print)** de la sección Detalle del informe para dibujar en él

```
Private Sub Detalle_Print( _
    Cancel As Integer, _
    PrintCount As Integer)
    Const Incremento As Integer = 30
    Const Bucles As Integer = 40
    Const Oscurecimiento As Single = 0.025
    Const CambioLuz As Single = 0.25
    Dim i As Integer
    Dim intPuente As Integer

    ' Primero usamos las clases, propiamente dichas _
    ' para dibujar una línea, un rectángulo y una elipse
    With Recta
        .Top = 0
        .Left = 0
        .Height = 2500
        .Width = 5000
        .Color = eoColorAzul
        .Draw
    End With
```

```
With Rectangulo
    .Top = 0
    .Height = 2500
    .Whidth = 5000
    .Color = eoColorRojo
    .Draw
End With

With Elipse
    .Top = 0
    .Height = 2500
    .Whidth = 5000
    .Color = eoColorNegro
    .Draw
End With

' Vamos a trabajar ahora con la Interfaz
Set objGrafico = Recta
With objGrafico
    ColorObjeto.Color = eoColorVioleta
    .Left = 1000
    .Whidth = 0
    .Height = 2500
    For i = 1 To Bucles / 4
        .Color = ColorObjeto.Color
        .Left = .Left + Incremento
        .Draw
        ColorObjeto.AclaraColor
    Next i
    ColorObjeto.Color = eoColorVioleta
    .Left = .Left + 10 * Incremento
    For i = 1 To Bucles / 4
        .Color = ColorObjeto.Color
        .Left = .Left - Incremento
        .Draw
        ColorObjeto.AclaraColor CambioLuz / 2
    Next i
End With
```

```
Set objGrafico = Rectangulo
ColorObjeto.Color = eoColorAzul
ColorObjeto.OscureceColor CambioLuz
With objGrafico
    ColorObjeto.AclaraColor CambioLuz
    .Left = 6000
    .Whidth = 2500
    .Height = 2500
    For i = 1 To Bucles
        .Color = ColorObjeto.Color
        .Top = .Top + Incremento
        .Left = .Left + Incremento
        .Whidth = .Whidth - 2 * Incremento
        .Height = .Height - 2 * Incremento
        .Draw
        ColorObjeto.AclaraColor 0.05
    Next i
End With
```

```
Set objGrafico = Elipse
With objGrafico
    ColorObjeto.Color = eoColorRojo
    ColorObjeto.AclaraColor 2 * CambioLuz
    .Left = 2000
    .Whidth = 2500
    .Height = 2500
    For i = 1 To 2 * Bucles
        .Color = ColorObjeto.Color
        .Top = .Top + Incremento / 6
        .Left = .Left + Incremento / 6
        .Whidth = .Whidth - Incremento / 3
        .Height = .Height - Incremento / 3
        .Draw
        ColorObjeto.OscureceColor Oscurecimiento
    Next i
End With
With objGrafico
    ColorObjeto.Color = eoColorAzul
    ColorObjeto.AclaraColor 2 * CambioLuz
    For i = 1 To 2 * Bucles
```

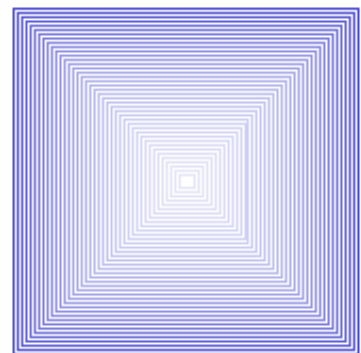
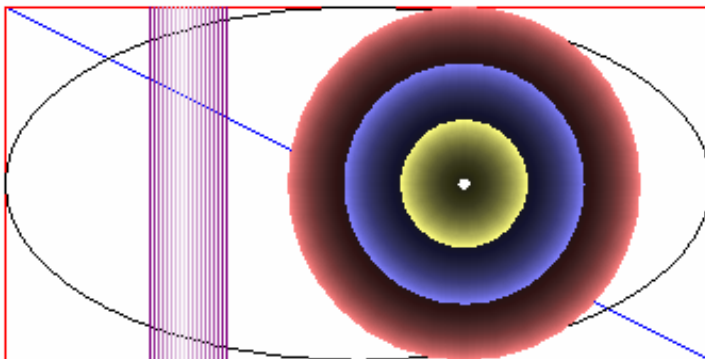


```

        .Color = ColorObjeto.Color
        .Top = .Top + Incremento / 6
        .Left = .Left + Incremento / 6
        .Width = .Width - Incremento / 3
        .Height = .Height - Incremento / 3
        .Draw
        ColorObjeto.OscureceColor Oscurecimiento
    Next i
End With
With objGrafico
    ColorObjeto.Color = eoColorAmarillo
    ColorObjeto.AclaraColor 2 * CambioLuz
    For i = 1 To 2 * Bucles
        .Color = ColorObjeto.Color
        .Top = .Top + Incremento / 6
        .Left = .Left + Incremento / 6
        .Width = .Width - Incremento / 3
        .Height = .Height - Incremento / 3
        .Draw
        ColorObjeto.OscureceColor Oscurecimiento
    Next i
End With
End Sub

```

Al ejecutar este código, se dibujan estas figuras gráficas:



Como ya se comenta en el propio código, podemos apreciar que en la primera parte usamos directamente las clases y sus métodos, para simplemente trazar una línea, un rectángulo y una elipse mediante el método **Draw** De cada una de ellas.

En la segunda parte del evento utilizamos una instancia de la interfaz, **IGraficElement** que asociamos a la variable **objGrafico**. A este objeto le vamos asignando sucesivamente los sucesivos objetos **Recta**, **Rectangulo** y **Elipse**, que a su vez son instancias de las clases **clsRecta**, **clsRectangulo** y **clsElipse**, y cada una de ellas implementa la interfaz **IGraficElement**.

¿Demasiado código para un resultado tan pobre?

No lo voy a negar, pero lo que realmente me interesa es que el lector capte la esencia de la utilización de interfaces, que lo que permiten es usar objetos diferentes mediante los mismos métodos y propiedades.

En definitiva aprovechar las ventajas del **Polimorfismo**.

Herencia

En un punto anterior hemos dicho que las clases en VBA no poseen la capacidad de heredar unas de otras, es decir, la programación con VBA no tiene posibilidad de implementar la herencia de clases. Ésta es una de las razones por las que se dice que Visual Basic no es un verdadero lenguaje orientado a objetos.

Planteada esta premisa inicial, al igual que con el polimorfismo, existe la posibilidad de “emular” la herencia mediante la llamada **Herencia mediante delegación**.

El truco consiste en lo siguiente:

Supongamos que queremos crear la clase **clsCuadrado**, que fácilmente podemos deducir que es un caso particular de la clase **clsRectangulo** que hemos creado con anterioridad.

Si, como explicábamos en el punto anterior, VBA implementara la herencia como sí es en el caso de **VB Net**, sería tan sencillo como declararla así

```
Friend Class clsCuadrado
    Inherits clsRectangulo
    .....
End Class
```

¡Ojo! que éste es también código de VB.Net, no de VBA ni de Visual Basic.

Sólo con la línea

```
Inherits clsRectangulo
```

La clase **clsCuadrado** tendría todos los métodos y propiedades de la clase **clsRectángulo**.

Podríamos a su vez redefinir el método **Draw** e incluso eliminar las propiedades **Width** y **Height** y sustituirlas por una única propiedad **Lado** a la que llamaríamos **Side**.

Por desgracia con VBA la cosa no es tan fácil, pero como he dicho podemos emular este comportamiento con algo más de código.

Para ello crearemos la clase **clsCuadrado**, y dentro de ella instanciaremos una clase del tipo **clsRectangulo** que la declararemos como **Private** para que no se tenga acceso a ella desde fuera de la clase **Cuadrado**..

Ese objeto de tipo **Rectángulo** creado dentro de la clase **clsCuadrado** será el que haga todo el trabajo.

El código de la clase **clsCuadrado** podría ser el siguiente:

Empezaremos declarando un objeto del tipo **clsRectangulo**; objeto que instanciaremos en el evento **Initialize** de la clase

```
Private Rectangulo As clsRectangulo

Private Sub Class_Initialize()
    Set Rectangulo = New clsRectangulo
End Sub
```

A continuación vamos a escribir la destrucción del objeto **Rectangulo** en el evento **Terminate** de la clase, y algo que no lo hemos hecho en el código de las clases anteriores, destruir la propiedad **Canvas** de la clase antes de que ésta desaparezca.

```
Private Sub Class_Terminate()
    Set Rectangulo.Canvas = Nothing
    Set Rectangulo = Nothing
End Sub
```

El resto de las propiedades y el método **Draw**. Se implementarían así:

```
Public Property Get Color() As PaletaColores
    Color = Rectangulo.Color
End Property

Public Property Let Color(NewColor As PaletaColores)
    Rectangulo.Color = NewColor
End Property

Public Property Get Left() As Integer
    Left = Rectangulo.Left
End Property

Public Property Let Left(ByVal X As Integer)
    Rectangulo.Left = X
End Property

Public Property Get Top() As Integer
    Top = Rectangulo.Top
End Property

Public Property Let Top(ByVal Y As Integer)
    Rectangulo.Top = Y
End Property

Property Get Side() As Integer
    Side = Rectangulo.Whidth
End Property
```

```
Property Let Side(ByVal NewValue As Integer)
    Rectangulo.Whidth = NewValue
    Rectangulo.Height = NewValue
End Property

Property Get Canvas() As Object
    Set Canvas = Rectangulo.Canvas
End Property

Property Set Canvas(NewCanvas As Object)
    Set Rectangulo.Canvas = NewCanvas
End Property

Sub Draw()
    Rectangulo.Draw
End Sub
```

Frente a las anteriores clases, ésta tiene la peculiaridad de que no implementa las propiedades **Whidth** y **Height**, y en cambio implementa una nueva propiedad **Side** no incluida en las otras clases.

Además tampoco implementa la interfaz `IGraficElement` aunque podríamos llegar a implementársela con algo más de código, si éste fuera nuestro propósito.

Vamos a ver cómo podemos usar la clase

Al igual que en el caso anterior creamos un nuevo informe en el que aumentaremos la altura de la sección detalle y aprovechamos los eventos del mismo.

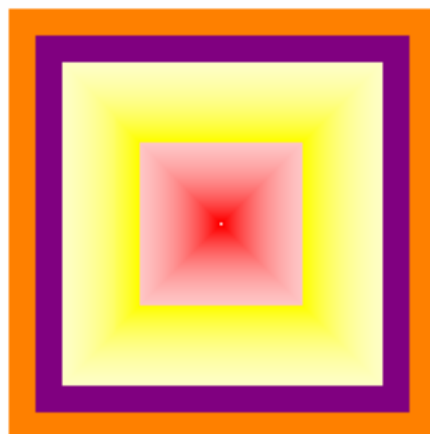
```
Private Sub Detalle_Print( _
    Cancel As Integer, _
    PrintCount As Integer)

    Const Escala As Integer = 30
    Dim intLado As Integer
    Dim i As Integer

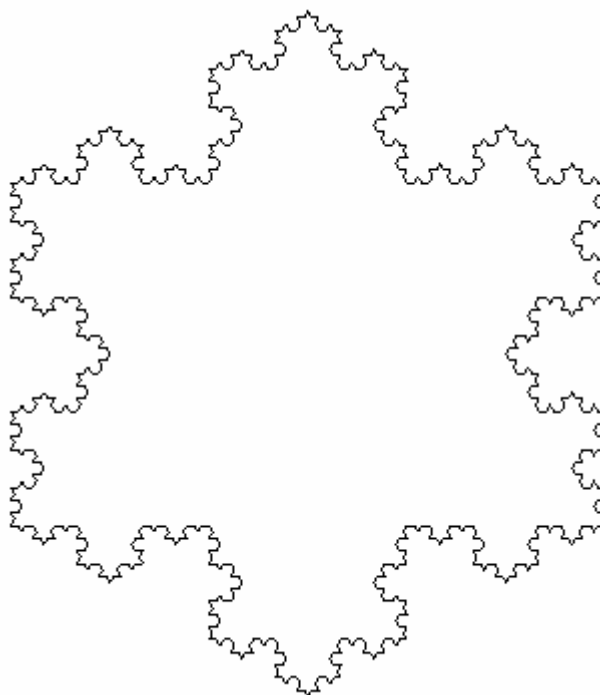
    objColor.Color = eoColorRojo
    For i = 1 To 30
        intLado = Escala * i
        With Cuadrado
            .Color = objColor.Color
            .Side = intLado
            .Left = 2500 - intLado / 2
            .Top = .Left
```

```
        .Draw
        objColor.AclaraColor 0.05
    End With
Next i
objColor.Color = eoColorAmarillo
For i = 31 To 60
    intLado = Escala * i
    With Cuadrado
        .Color = objColor.Color
        .Side = intLado
        .Left = 2500 - intLado / 2
        .Top = .Left
        .Draw
        objColor.AclaraColor 0.05
    End With
Next i
For i = 61 To 70
    intLado = Escala * i
    Cuadrado.Color = eoColorVioleta
    With Cuadrado
        .Side = intLado
        .Left = 2500 - intLado / 2
        .Top = .Left
        .Draw
    End With
Next i
For i = 71 To 80
    intLado = Escala * i
    Cuadrado.Color = eoColorNaranja
    With Cuadrado
        .Side = intLado
        .Left = 2500 - intLado / 2
        .Top = .Left
        .Draw
    End With
Next i
End Sub
```

Tras ejecutarse este código, el resultado final sería algo así como lo siguiente:



Debo reconocer que siento cierta debilidad por el diseño gráfico.
Las clases anteriores u otras similares se pueden usar para muchas cosas.
Este es otro ejemplo realizado con la clase `clsRecta`.



Para los que sientan curiosidad les diré que es una figura fractal que representa al **Copo de Koch**. Es un fractal de superficie limitada pero con un perímetro de longitud infinita.

Si he utilizado los elementos gráficos es simplemente porque me resulta divertido, sobre todo considerando las pocas posibilidades que vienen implementadas en Access.

Más adelante utilizaremos lo aprendido en estos capítulos para realizar clases que por ejemplo nos sirvan para gestionar datos de clientes o proveedores.

Constructor de la clase

De entrada debo decir que en las clases de VBA no podemos definir constructores, al menos de la forma como se entienden en la Programación Orientada a Objetos.

No obstante en el capítulo anterior comentamos la posibilidad de emularlos.

Primero vamos a definir qué podemos entender por un constructor de una clase:

Un constructor “ortodoxo” debería poder introducir los parámetros que definan completamente la clase en el momento de su creación.

Por ejemplo con un constructor “De verdad” podríamos definir

```
Dim MiCuadrado as New clsCuadrado 1200, 1400, VbRed, 1000, Me
```

Vuelvo a recordar que esto no funciona con VBA, aunque sí se pueda hacer con VB.Net

Esto haría que se creara el objeto **MiCuadrado** de la clase **clsCuadrado** a 1200 unidades del margen izquierdo, a 1400 unidades del margen superior, de color **Rojo** y con un lado de tamaño 1000 unidades. Además se supone que se llama desde el módulo de clase de un informe, con lo que asigna el propio **Informe** como Lienzo.

Esto supone que se ha de crear un método. En VB.Net este método recibe el nombre de **New** y se ejecuta en el momento de crearse la clase.

Además estos lenguajes permiten crear simultáneamente diferentes versiones del método **New**, con lo que podríamos crear el objeto con diferentes combinaciones de parámetros para su creación.

VBA posee el evento **Initialize**, pero si lo utilizáramos como constructor, sólo podríamos definir valores por defecto para las propiedades de la clase.

Podríamos por ejemplo añadir a la clase **clsCuadrado** el método Constructor, cuyo desarrollo podría ser el siguiente

```
Public Sub Constructor( _  
    Optional ByVal X As Integer, _  
    Optional ByVal Y As Integer, _  
    Optional ByVal NewColor As PaletaColores, _  
    Optional ByVal NewSide As Integer, _  
    Optional NewCanvas As Object)  
    Left = X  
    Top = Y  
    Color = NewColor  
    Side = NewSide  
    If TypeName(NewCanvas) <> "Nothing" Then  
        Set Canvas = NewCanvas  
    End If  
End Sub
```

Por cierto, la función **TypeName**, devuelve una cadena con información sobre el tipo de variable que se ha pasado como parámetro.

En este caso, si no pasamos ningún valor al parámetro `NewCanvas`, su contenido será **Nothing**, por lo que la función `TypeName` devolvería la cadena **"Nothing"**.

Si no es así, significa que se le ha pasado algún objeto, por lo que se lo asignamos a la propiedad `Canvas`.

Siguiendo con el planteamiento anterior podríamos hacer

```
Private Sub Report_Open(Cancel As Integer)
    Set Cuadrado = New clsCuadrado
    Cuadrado.Constructor 1200, 1400, vbRed, 1000, Me
End Sub
```

De esta forma hemos inicializado de un golpe y totalmente los valores del objeto **Cuadrado**.

Lástima que no se pueda hacer simultáneamente con la creación del objeto.

¿O sí se puede hacer?.

Podríamos emularlo auxiliándonos de un procedimiento externo que admitiera los parámetros del procedimiento `Constructor` y que al llamarlo creara una instancia de la clase y ejecutara su método `Constructor`.

Veamos cómo se hace:

En un módulo “normal” creamos una función a la que llamaremos `clsCuadrado_New`, que incluirá los mismos parámetros que el método `Constructor` de la clase.

Éste sería su código:

```
Public Function clsCuadrado_New( _
    Optional ByVal X As Integer, _
    Optional ByVal Y As Integer, _
    Optional ByVal NewColor As PaletaColores, _
    Optional ByVal NewSide As Integer, _
    Optional NewCanvas As Object _
) As clsCuadrado
    Dim Cuadrado As New clsCuadrado
    If TypeName(NewCanvas) <> "Nothing" Then
        Cuadrado.Constructor X, Y, _
                               NewColor, NewSide, _
                               NewCanvas
    Else
        Cuadrado.Constructor X, Y, _
                               NewColor, NewSide
```



```

End If
Set clsCuadrado_New = Cuadrado
End Function

```

Cierto que es un subterfugio, ya que necesita código externo a la propia clase para emular un método que debería estar dentro de la propia clase, lo que va en contra de la filosofía de la P.O.O. pero... ¡menos es nada!

Para llamar a esta función lo podríamos hacer como en los casos precedentes desde el informe:

```

Option Explicit
Dim Cuadrado As clsCuadrado

Private Sub Report_Open(Cancel As Integer)
    Set Cuadrado = clsCuadrado_New(1200, 1400, _
                                   vbRed, 1000, Me)
End Sub

Private Sub Detalle_Print( _
    Cancel As Integer, _
    PrintCount As Integer)

    Cuadrado.Draw
End Sub

```

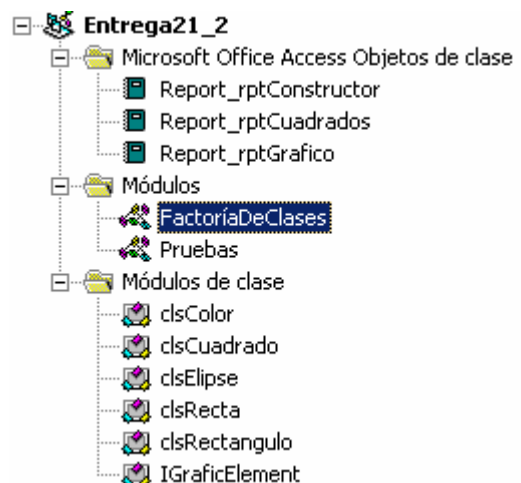
Con esto simulamos un constructor, ya que funciona como si a la vez que creáramos un objeto le asignáramos sus atributos.

Factoría de Clases

No es una mala costumbre definir un método **Constructor** cada vez que diseñamos una clase, e incluir las respectivas funciones constructoras, que en nuestro caso serían

- **clsColor_New**
- **clsRecta_New**
- **clsRectangulo_New**
- **clsCuadrado_New**
- **clsElipse_New**

dentro de un módulo estándar
al que podríamos poner como nombre
FactoríaDeClases.



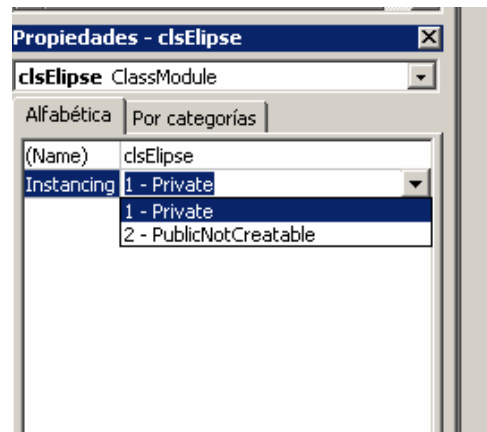
Propiedad Instancing de la clase

Si miramos la ventana de propiedades de las clases, vemos que tienen una propiedad llamada Instancing.

Esta propiedad puede tomar dos valores

1 - **Private**

2 - **PublicNotCreatable**



El valor de la propiedad **Instancing** determina si una clase es privada o si está disponible para su uso por parte de otras aplicaciones.

En el caso concreto de VBA para Access la propiedad **Instancing** sólo puede tomar los dos valores comentados con anterioridad.

Private hace que no se permita el acceso de otras aplicaciones a la información de la biblioteca de tipos de la clase y que no se pueden crear, desde fuera, instancias de la misma.

PublicNotCreatable permite que las demás aplicaciones puedan usar los objetos de la clase sólo después de que hayan sido creadas.

En el software de desarrollo Visual Basic 6.0, no en el VBA de Access, se permiten otro tipo de valores para esta propiedad

3 - **Multiuse**

4 - **GlobalMultiuse**

6 - **SingleUse**

7 - **Global SingleUse**

Pero no siempre es así para todo tipo de proyectos. Por ejemplo, si se está desarrollando un componente **ActiveX**, al igual que en VBA de Access, sólo se permiten los valores **Private** y **PublicNotCreatable**, lo que resulta lógico.