

Comencemos a programar con VBA - Access

Entrega 00

Índice

Entrega 01	Introducción	10 páginas
Planteamiento		2
Objetivos.....		2
¿A quién va dirigido?		2
¿Quién soy yo?.....		2
¿Qué es VBA?		3
Los módulos		4
Primer formulario con código.....		6
Un poco de teoría		7
Entrega 02	Entorno de Desarrollo	7 páginas
Entorno de desarrollo		2
Ventanas.....		3
Uso de la ventana Inmediato (ventana de Depuración)		4
Entrega 03	Primeros conceptos	7 páginas
Primeros conceptos		2
Ámbito ó Alcance de las Constantes y variables.....		3
Procedimientos Sub.....		4
Entrega 04	Primeros conceptos II	12 páginas
Funciones		2
Funciones en formularios		5
Entrega 05	Tipos de datos y Declaraciones	13 páginas
Declaración de variables		2
Construcción del nombre de una variable (o constante)		2
Tipos de datos		3
Datos numéricos.....		3
Prefijos.....		4
Números de Coma Flotante.....		5
Tipo Decimal.....		5
Tipo Date		5
Tipo Boolean (booleano)		8
Tipo Variant		10
Empty - Null - Cadena vacía.....		10
Declaraciones múltiples en una línea		11
Valores por defecto.....		12
Ámbito ó alcance de una declaración.....		12
Dim		12
Private.....		12
Public.....		12
Global		13
Entrega 06	Estructuras de datos	11 páginas
Matrices ó Arrays.....		2

Matrices de varias dimensiones	4
Matrices dinámicas	4
Instrucción Erase	7
Redim con varias dimensiones	7
Índices superior e inferior de una matriz	7
Registros (Estructuras de variables definidas por el usuario)	8
La instrucción With	10
Matrices de Registros	11
Entrega 07	Colecciones y Objetos
	12 páginas
Introducción a las Colecciones	2
Antes de seguir: For Each - - - Next	4
Colecciones de Controles	5
Introducción a las Clases	7
Entrega 08	Continuando con las Clases
	9 páginas
Aclaraciones sobre el código del último capítulo	2
Sigamos analizando el código	8
Entrega 09	Estructuras de Control
	15 páginas
Estructuras de Control	2
Estructuras de Decisión	2
La Instrucción If	2
La Función If	5
La Instrucción Select Case	5
Estructuras de Bucle	9
Las Instrucciones For - - - Next	9
La Instrucción For Each - - - Next	14
Entrega 10	Estructuras de Control II
	12 páginas
Estructuras de Control, segunda parte	2
Las Instrucciones While - - - Wend	2
Las Instrucciones Do - - - Loop	4
Entrega 11	Gestión de errores
	12 páginas
Errores	2
La gestión de los errores en procedimientos	2
Errores en Tiempo de Ejecución	3
Instrucciones de salto	5
La Instrucción Goto	5
Gosub - - Return	5
Capturar Errores	6
El objeto Err	6
La instrucción Resume	7
Gestionando errores	10
Generación directa de errores (Err.Raise)	12

Entrega 12	Trabajando con Procedimientos	22 páginas
Procedimientos		2
Variables Static		2
Paso de parámetros Por Valor y Por Referencia.....		3
Parámetros Opcionales		5
Puntualizaciones sobre parámetros opcionales.		5
Procedimientos Recursivos frente a Iterativos		7
Pasar un parámetro, tipo matriz, mediante ParamArray		9
Uso de parámetros Con Nombre		10
Constantes Enumeradas		11
Uso de las Constantes Enumeradas		12
Funciones para intercambio de información		15
Función MsgBox		15
Función InputBox.....		19
Entrega 13	Funciones de VBA	12 páginas
Funciones propias de VBA		2
Función Choose.....		2
Función Switch		3
Función Format.....		4
Utilización con cadenas String.....		4
Utilización con fechas		5
Cadenas con nombre		7
Utilización de Format con números		9
Cadenas con nombre para números		10
Formato compuesto		11
Configuración Regional		11
Entrega 14	Funciones de VBA II	21 páginas
Funciones adicionales para formato.....		2
Función FormatNumber		2
Función FormatCurrency		3
Función FormatPercent		3
Función FormatDateTime		3
Función MonthName		5
Función WeekdayName		5
Manipulación de cadenas		7
Función Left.....		8
Función LeftB.....		8
Function Right.....		9
Function Mid		9
Instrucción Mid.....		9
Funciones LTrim, Rtrim y Trim		11
Funciones Len y LenB		11
Buscar y sustituir cadenas.....		13
Función InStr		13
Función InStrReverse		14

Función StrComp.....	14
Función Replace.....	14
Función StrReverse.....	15
Función Filter.....	15
Función Split.....	16
Función Join.....	18
Operador Like.....	18
Funciones Asc y AscB.....	19
Funciones Chr y Chr\$.....	19
Diferencia entre funciones que trabajan en modo Carácter y en modo Byte.....	20

Entrega 15**Operadores**

22 páginas

Operadores.....	2
Tipos de Operadores.....	2
Operadores aritméticos.....	2
Operador Suma.....	2
Operador Resta.....	5
Operador Producto.....	5
Operador División.....	6
Operador Elevar a potencia.....	6
Operador División entera.....	7
Operador Módulo o Resto.....	8
Operadores de Comparación.....	8
Operador Like.....	9
Operador Is.....	11
Operadores de concatenación.....	15
Operadores lógicos.....	15
Operador And.....	15
Operador Or.....	17
Operador Xor.....	18
Operador Not.....	19
Operador Eqv.....	19
Operador Imp.....	20
Prioridad de los operadores.....	20

Entrega 16**Código vs. Macros - Objeto DoCmd**

30 páginas

Código frente a macros.....	2
¿Cuándo usar Macros y cuándo código VBA?.....	7
El objeto DoCmd.....	7
Uso de DoCmd en los Asistentes para controles.....	29

Entrega 17**Trabajar con ficheros**

17 páginas

Trabajar con Ficheros.....	2
Trabajar con carpetas.....	2
Función Dir.....	3
Función CurDir.....	5

Instrucción ChDir	6
Instrucción ChDrive	6
Instrucción Mkdir	6
Instrucción Rmdir	7
Instrucción Kill.....	7
El objeto FileSearch.....	8
Propiedades y métodos de FileSearch.....	8
Propiedad LookIn.....	8
Propiedad Filename.....	8
Propiedad SearchSubFolders.....	8
El objeto FileSearch.....	8
Método Execute.....	8
Propiedad LastModified	10
Objeto FoundFiles	11
Método NewSearch	12
Propiedad FileType.....	12
Otras propiedades y métodos.....	14
Colección SearchFolders.....	14
Objeto ScopeFolder.....	14
Colección ScopeFolders.....	14
Colección SearchScopes.....	14
Objeto SearchScope.....	14
Método RefreshScopes	15

Entrega 18**Trabajar con ficheros II**

18 páginas

Trabajando con Ficheros	2
Instrucción Open.....	2
Función FreeFile.....	3
Instrucción Print #.....	3
Función Tab.....	5
Función Spc.....	6
Instrucción Width #	7
Instrucción Write #.....	8
Instrucciones Input # y Line Input #.....	10
Ficheros de acceso Aleatorio	12
Instrucción Get	13
Instrucción Put.....	13
Función Seek.....	16
Ejemplos de apertura de ficheros con la instrucción OPEN	17
Notas sobre esta entrega	18

Entrega 19**Trabajar con ficheros III**

40 páginas

Exportar, importar y vincular ficheros de texto	2
Estructura de un fichero Schema.ini.....	3
Otros tipos de formato en ficheros Schema.ini.....	9
Varios esquemas en un único archivo Schema.ini	11
Abrir ficheros de texto como si fuera un Recordset.....	11

Ficheros ini	14
Leer y escribir un fichero ini.....	14
Introducción (necesaria) a la utilización de las funciones API.....	15
Información en Internet sobre las APIs de Windows	16
Declaración de las funciones API para el manejo de un fichero ini.....	17
Escritura y lectura en un fichero ini.....	19
Lista las Secciones de un fichero ini.....	26
Lectura y escritura en bloque de toda una sección en un fichero ini.....	29
El Registro de Windows.....	36
Escritura y lectura del registro con VBA	36
Instrucción SaveSetting	37
Función GetSetting	38
Función GetAllSetting	38
Instrucción DeleteSetting.....	39
Grabar, leer y borrar cualquier Sección – Clave del registro.....	40
Notas sobre este capítulo	40
Entrega 20	Más sobre Clases y Objetos (1)
	47 páginas
Recordemos lo expuesto hasta ahora sobre las clases	2
Antes de seguir: consideraciones sobre las clases.	3
Propiedades.....	4
Métodos de clase.....	9
Clases que hacen referencia a sí mismas	9
Creación de estructuras con clases.....	13
Función Is Nothing.....	21
Eventos.....	21
Qué es un Evento	22
Crear clases en las que se definan Eventos	24
Instrucción Event	25
Manos a la obra. Clase Depósito.....	26
Instrucción RaiseEvent	28
Palabra clave WithEvents.....	37
Probando la clase CDeposito	38
Eventos Initialize y Terminate de la clase.....	44
Propiedades en los módulos estándar	46
Nota sobre la próxima entrega	47
Entrega 21	Más sobre Clases y Objetos (2)
	50 páginas
Continuamos con las clases	2
Procedimientos Friend.....	2
Gestión de colores - 2º Caso práctico	3
Función RGB	4
Clase básica para gestionar el color.....	5
Utilización de la clase Color en un informe.....	7
Enumeraciones en un módulo de clase.....	11
Herencia y Polimorfismo.....	18

Interfaces	19
Creación de una Interfaz	20
Ejemplo sobre la creación y uso de interfaces	29
Uso de las clases y de la interfaz	37
Herencia	42
Constructor de la clase	47
Factoría de Clases.....	49
Propiedad Instancing de la clase.....	50

Apéndice 01	Tipos numéricos, Bytes y bits	10 páginas
Byte, bit, Bytes, bits ¿Qué es eso?.....		2
Kas, Megas, Gigas		2
Cómo afecta esto a la memoria del PC		3
Lectura / Escritura de los datos en memoria (Tipos numéricos)		3
Estándar IEEE-754		7
Simple precisión		7
Doble precisión		7
Tipo Decimal.....		8
Manejo de textos		9

Comencemos a programar con VBA - Access

Entrega 01

Introducción

Planteamiento

Este “cursillo” nace como respuesta a las continuas demandas por parte de los intervinientes en los foros de Access, de un manual que permita, desde cero, aprender los fundamentos de la programación con VBA.

La idea inicial es ir suministrando sucesivas entregas imprimibles y descargables desde uno ó varios enlaces de Internet.

Estos textos se complementarán, si fuese necesario, con ficheros mdb que contendrán el código y los ejemplos planteados en el curso.

Objetivos

El objetivo que me planteo, espero no ser demasiado ambicioso, es que el lector, tras seguir el cursillo, adquiera las bases de programación suficiente como para poder manejar las herramientas que le suministra VBA de Access, y sea capaz por sí mismo de ir evolucionando y completando sus conocimientos.

Por mi experiencia personal y con alumnos de cursos que he impartido, el problema fundamental para soltarse en la programación es conocer los elementos esenciales del lenguaje y las herramientas que el sistema te ofrece.

En nuestro caso vamos a utilizar el lenguaje Visual Basic para Aplicaciones y su entorno de desarrollo, orientado principalmente a su utilización con Access.

¿A quién va dirigido?

Va dirigido a todos aquellos que quieran comenzar a programar con VBA.

Como requisitos necesarios, el lector deberá haber manejado previamente Access y ser capaz de crear tablas, formularios, informes y consultas básicas.

¿Y quién diablos soy yo para atreverme a esto?

Me llamo Eduardo Olaz, y vivo en Pamplona, una bonita ciudad del norte de España, famosa por sus fiestas de San Fermín y por la casi absoluta falta de “marcha” el resto del año.

En el año 2003 Microsoft me concedió el nombramiento de MVP (Most Valuable Professional).

Varios de vosotros me conoceréis por mis intervenciones en el grupo de Access microsoft.public.es.access, o en el de Visual Basic

Desde el año 1992 me ha tocado trabajar con todo tipo de bases de datos Access, en local, en red, de tamaño enorme, pequeñas, etc.

Empecé con la versión Access 1.0 bajo Windows 3.1. Ahora utilizo la versión 2003.

He impartido varios cursos de programación con Access y con Visual Basic 6.

Ya vale de hablar de mí y vamos a lo que cuenta...

¿Qué es VBA?

VBA quiere decir Visual Basic para Aplicaciones.

Es un conjunto de librerías, (un tipo especial de programas), desarrollado por Microsoft que incluye, entre otras cosas, un entorno de desarrollo y un lenguaje de programación.

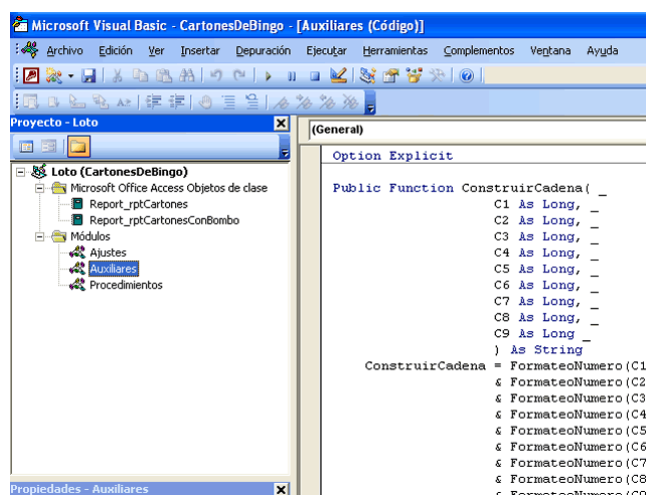
VBA no es exclusivo de Access; lo podemos encontrar también en todas las aplicaciones de Office como Word o Excel, en otro tipo de programas como Project o Visio, y en programas que no son de Microsoft y tan diversos como Corel Draw o AutoCad.

Dominando los fundamentos de VBA, se podría desarrollar aplicaciones en cualquiera de esos aplicativos.

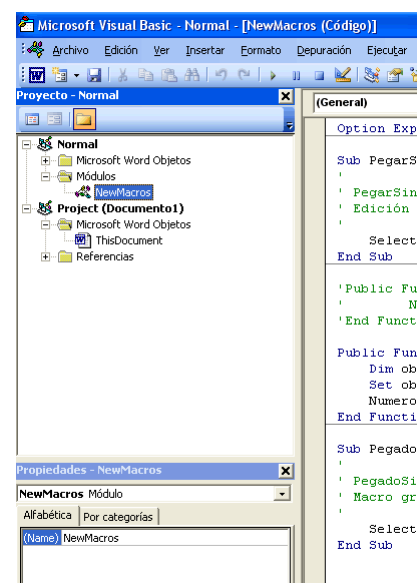
VBA tiene un entorno de programación que es semejante para todos estos programas.

Por ejemplo:

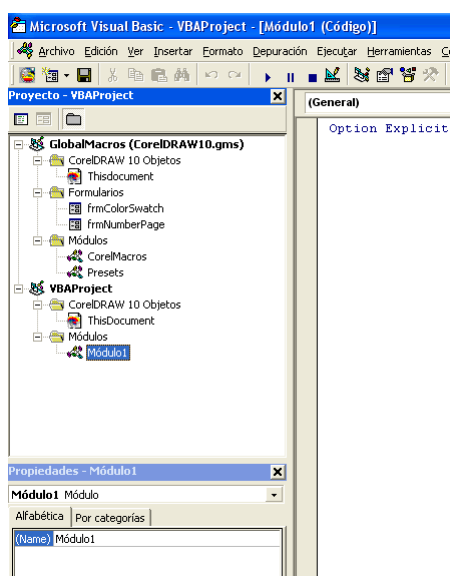
VBA para Access



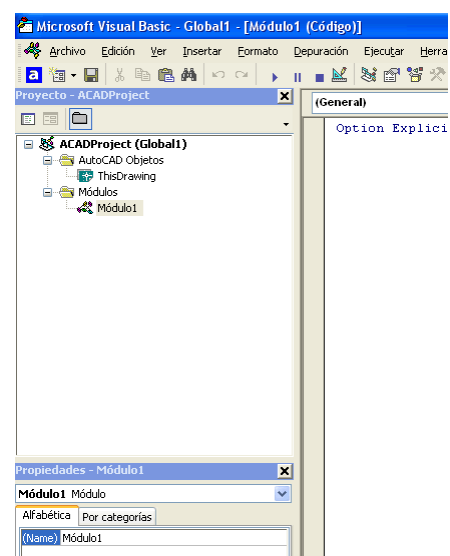
VBA para Word



VBA para Corel Draw



VBA para Auto Cad



Comencemos a programar con VBA - Access

-Eso está muy bien, pero ¿cómo empiezo a programar?-

-Vale. Un poco de paciencia.-

Los módulos

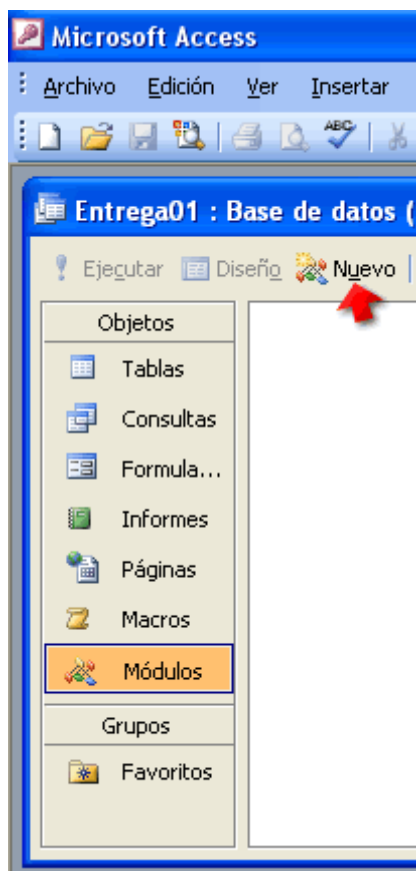
Los módulos son los objetos en los que se guarda el código que va a utilizar VBA.

Hay tres tipos de módulos.

- Módulos generales
- Módulos asociados a formularios e informes
- Módulos de Clase.

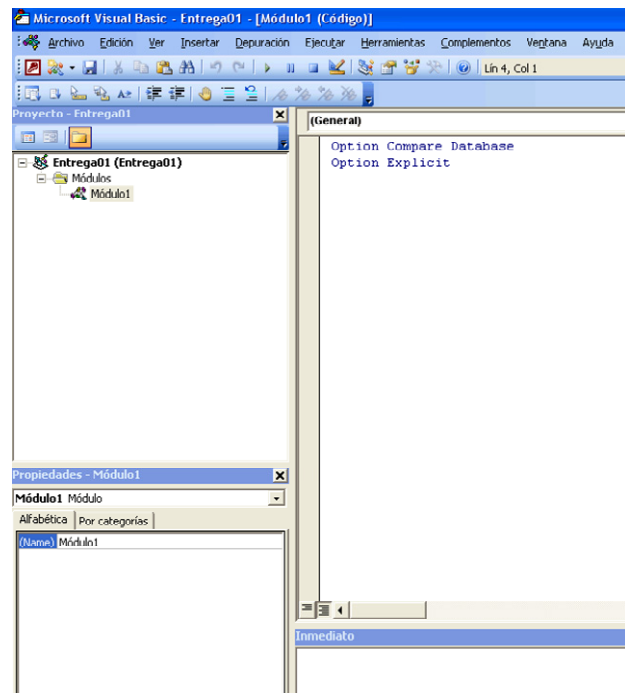
Manos a la obra: vamos a crear una nueva base de datos con el nombre Entrega01.mdb

Para acceder a los módulos generales debemos presionar en la pestaña Módulos de la Barra de Objetos de Access:



Para crear un módulo nuevo haga Clic en el botón [Nuevo].

Una vez hecho esto se abre el editor de VBA y nos crea un módulo con el original nombre de Módulo1.



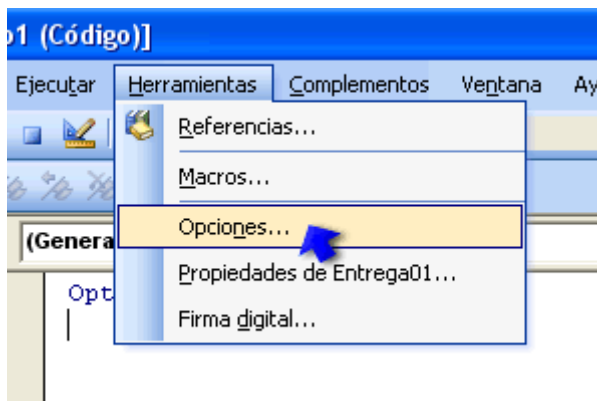
Este módulo está vacío, salvo las dos líneas

```
Option Compare Database
```

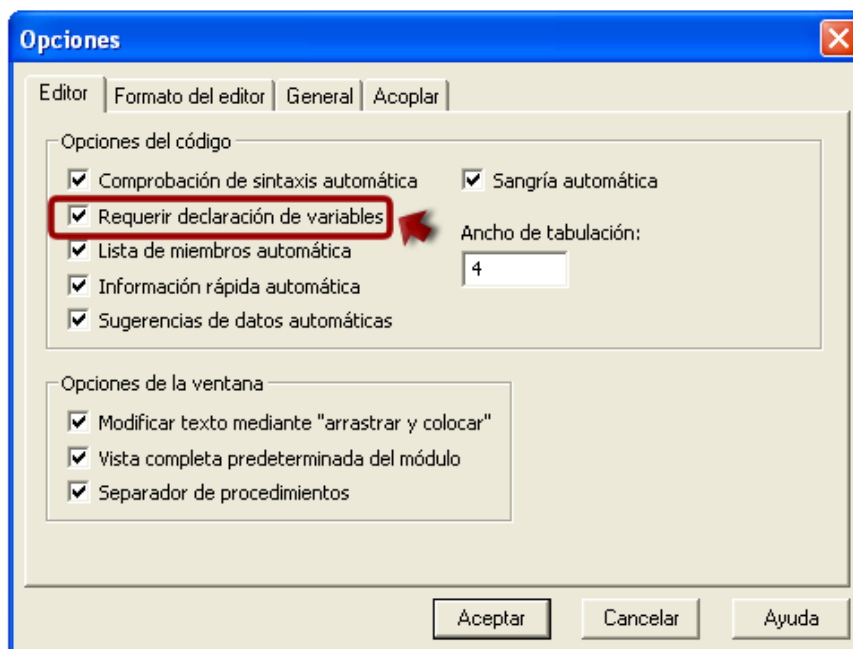
```
Option Explicit
```

Si no apareciera la línea **Option Explicit**, vamos a hacer lo siguiente:

Desde la ventana del editor que se ha abierto, presionamos la opción de menú:
Herramientas > Opciones



En el formulario que aparece activaremos la opción [Requerir declaración de variables].



Le damos a Aceptar.

-¿Pero qué es esto?-

Aunque adelantemos conceptos trataré de explicarme.

Cuando empecemos a escribir código, veremos que existen unos elementos que se llaman variables, a los que podremos asignar valores.

Si tenemos activada la opción [Requerir declaración de variables] nos obligará a declarar las variables antes de poder usarlas. Más adelante veremos que hay variables que pueden contener Texto, Números, Fechas, Objetos, etc.

En una entrega posterior aprenderemos a declarar una variable. Esto significa darle nombre y expresar qué tipo de dato va a contener.

Una vez hecho esto grabamos el módulo dándole al botón Guardar [Disquete] ó a la opción de menú Archivo>Guardar ó combinando las teclas [Ctrl] + [S]

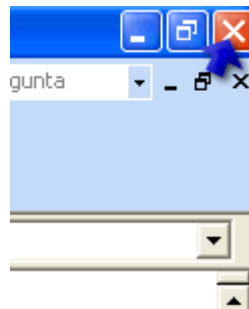
Aceptamos el nombre por defecto Módulo1 y cerramos la ventana.

Para cerrar la ventana del editor podemos usar Varios caminos:

Archivo > Cerrar y volver a Micro ...



Botón Cerrar



Combinación de teclas

[Ctrl] + [Q]

Si volvemos a crear un nuevo módulo veremos que ahora sí contiene la línea

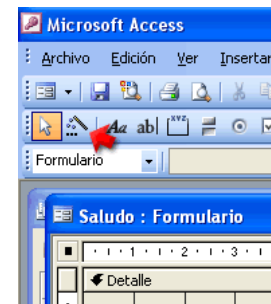
`Option Explicit`

Ahora cerramos este módulo.

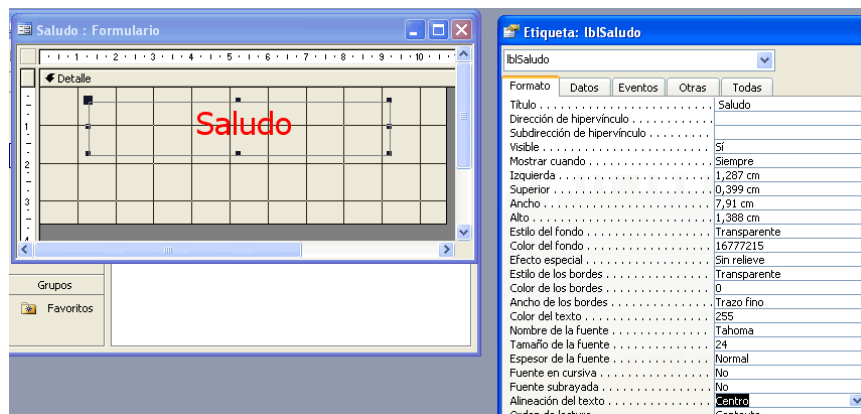
Nuestro primer formulario con código

Vamos a crear un nuevo formulario que lo vamos a llamar Saludo.

Para que nos de plena libertad de actuación, vamos a desactivar el botón del [Asistente para controles]



1. Nos vamos a la opción Formularios y presionamos en [Crear formulario en vista diseño]
2. Lo guardamos como Saludo
3. Añadimos una Etiqueta al formulario.



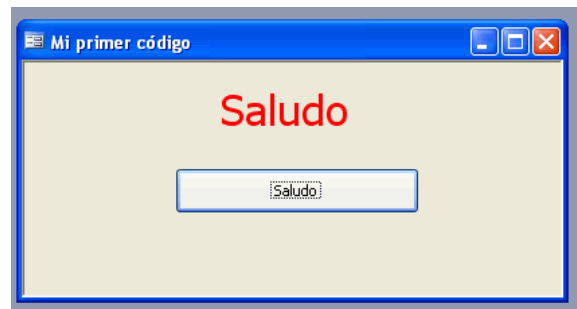
- a. Utilizando las Propiedades, ponemos a la propiedad [Nombre] de la etiqueta el valor lblSaludo (Pestaña [Otras] de la hoja de Propiedades) sustituyendo el que tenía por defecto: Etiqueta1.
 - b. Desde la pestaña [Formato] le ponemos a la propiedad [Título] el valor Saludo, a la propiedad [Tamaño de la fuente] el valor 24, y al [Color de texto] el valor 255 (Rojo) y a la [Alineación del texto] el valor Centro.
4. Añadimos un Botón y le ponemos como [Nombre], en vez de Comando2, cmdSaludo

5. Seleccionamos el Formulario y ponemos a No las propiedades [Selectores de registro], [Botones de desplazamiento] y Separadores de registro.
6. La propiedad [Centrado automático] la ponemos a Sí, y a la propiedad [Título] el valor Mi primer código.
 - a. Le ponemos como [Nombre] Saludo

Si visualizamos el formulario se verá algo así como esto:

Si presionamos el botón vemos que no pasa nada.

Vamos a hacer que cuando se presione el botón aparezca el mensaje ¡¡¡**Hola Mundo!!!** en la etiqueta lblSaludo, sustituyendo su contenido actual que es Saludo.



Un poquito de teoría

Windows funciona con las llamadas Ventanas. Esas ventanas contienen objetos, y tanto las ventanas como los objetos pueden mandar unos “mensajes” para indicar a la aplicación que usa Windows, o al propio Windows que han ocurrido determinadas cosas. Esos mensajes se llaman Eventos. *Por cierto; en realidad muchos de los objetos, como los botones, también son ventanas; pero esa es otra guerra.*

En nuestro caso tenemos los siguientes objetos:

- Formulario **Saludo**
- Etiqueta **lblSaludo**
- Botón **cmdSaludo**

Cada uno de esos **Objetos** posee una serie de **Propiedades**, generan determinados **Eventos** y pueden hacer determinadas cosas mediante unos procedimientos propios que se llaman **Métodos**.

Por ejemplo, al presionar el botón **cmdSaludo**, se genera el evento [Al hacer Clic] (**Click**).

Podríamos capturar ese evento y aprovecharlo para, por ejemplo, cambiar el texto de la etiqueta **lblSaludo**.

El texto que aparece en la etiqueta está en la propiedad [Título]; en inglés [**Caption**].

Ahora la pregunta: ¿dónde y cómo se hace todo eso?

Cada formulario, y veremos más adelante que también cada informe, lleva asociado un módulo especial. Se llama módulo de clase del formulario.

Vamos a verlo.

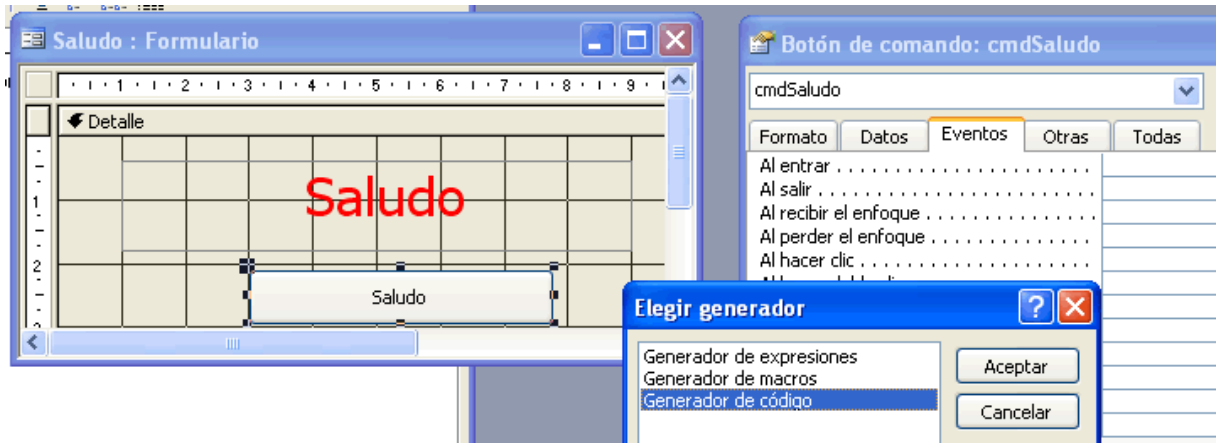
En el modo Diseño del formulario Saludo, seleccionamos el botón **cmdSaludo**.

Abrimos la Hoja de propiedades y seleccionamos la pestaña [Eventos], y dentro de éstos el evento [Al hacer clic].

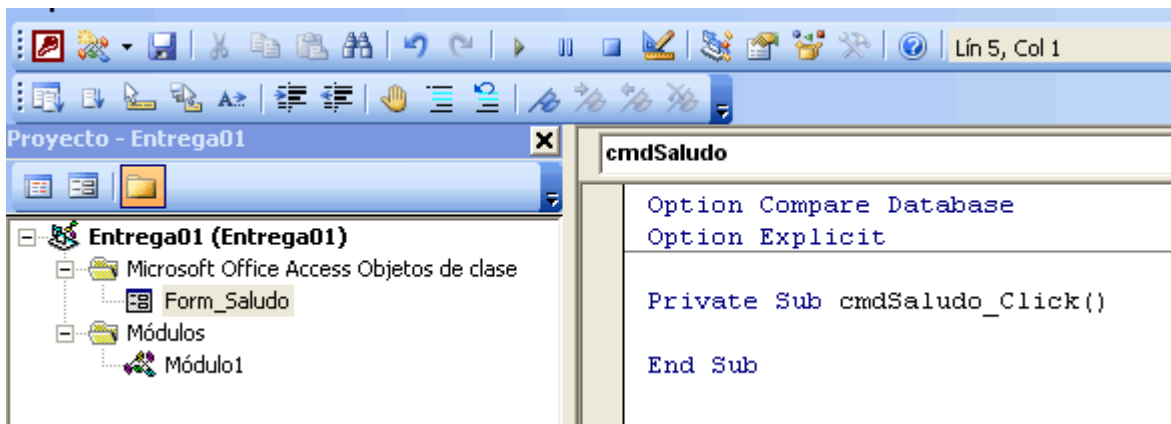
Al poner el cursor sobre él, nos muestra un botoncito con tres puntos.

Pulsamos en él, y seleccionamos [Generador de código] en la ventana que se nos abre.

A continuación pulsamos en el botón [Aceptar].



Tras esto nos abrirá el módulo de clase del formulario Saludo, con el esquema de código correspondiente al evento Clic del botón `cmdSaludo`:



Crea el siguiente código:

```
Option Compare Database
Option Explicit

Private Sub cmdSaludo_Click()

End Sub
```

Lo que ahora nos importa es el conjunto

```
Private Sub cmdSaludo_Click()

End Sub
```

Este código se corresponde al procedimiento que recoge el evento Clic del botón.

Consta de varias partes:

`Private Sub` indica que empieza un procedimiento del tipo Sub, más adelante veremos en detalle qué es esto.

`cmdSaludo_Click()` Indica que es el procedimiento al que llama el evento Click del botón `cmdSaludo`.

`End Sub` Indica el punto donde acaba el procedimiento

Entre esas dos líneas podemos escribir el código que le dirá a Access qué es lo que tiene que hacer.

Vamos a hacer 2 cosas:

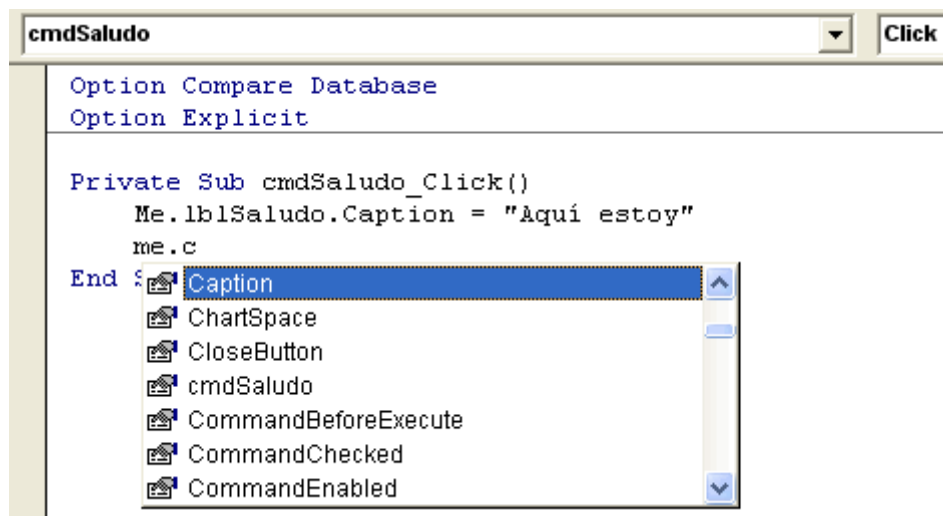
1. Escribiremos en la propiedad [Título] (Caption) de la etiqueta el texto
Hola Mundo.
2. En la propiedad [Título] (Caption) del formulario mostraremos el texto.
Aquí estoy.

Debo modificar el texto del código para que ponga:

```
Option Compare Database
Option Explicit

Private Sub cmdSaludo_Click()
    Me.lblSaludo.Caption = "¡¡¡Hola Mundo!!!"
    Me.Caption = "¡Aquí estoy!"
End Sub
```

Al escribir el texto, se abre una ventana de Ayuda contextual, lo que simplifica en gran medida la escritura correcta del texto.



Esta ventana te va mostrando las propiedades y métodos. Para seleccionar uno en concreto podemos utilizar la tecla [Enter], o mejor aún, la tecla [tab] ó de tabulación (es la tecla que suele estar encima de la de Bloqueo de Mayúsculas).

```
Me.lblSaludo.Caption = "¡¡¡Hola Mundo!!!"
```

Lo que hace es pones el texto ¡¡¡Hola Mundo!!! En la etiqueta lblSaludo.

Las comillas son para indicar a VBA que lo que hay dentro es un texto.

```
Me.Caption = "¡Aquí estoy!"
```

Pone el texto ¡Aquí estoy! en el título del formulario.

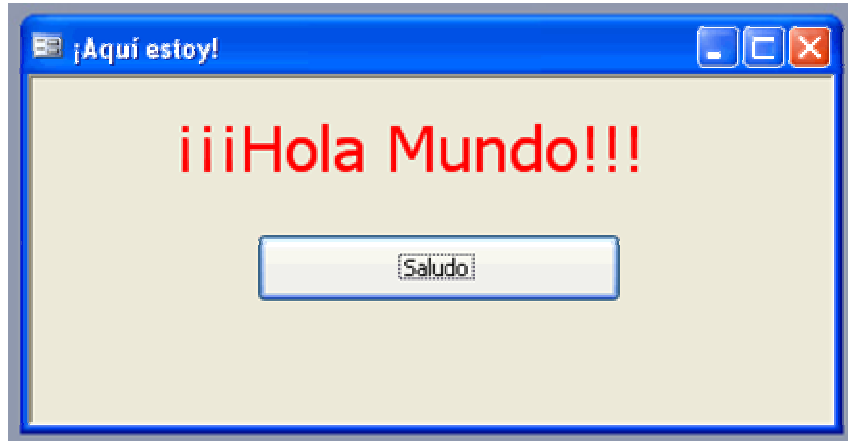
Me es el propio formulario.

Si nos fijamos en la sentencia `Me.lblSaludo.Caption` está indicando la propiedad **Caption** del objeto **lblsaludo** del **formulario** actual.

Fijaros que cada objeto ó propiedad está separada por un punto.

Hablaremos más de esto.

Cerramos la ventana de edición del código, y si hemos seguido correctamente los pasos, al presionar el botón formulario debería tener un aspecto semejante a éste:



Seguiremos en el siguiente capítulo.

Comencemos a programar con
VBA - Access

Entrega **02**

Entorno de desarrollo

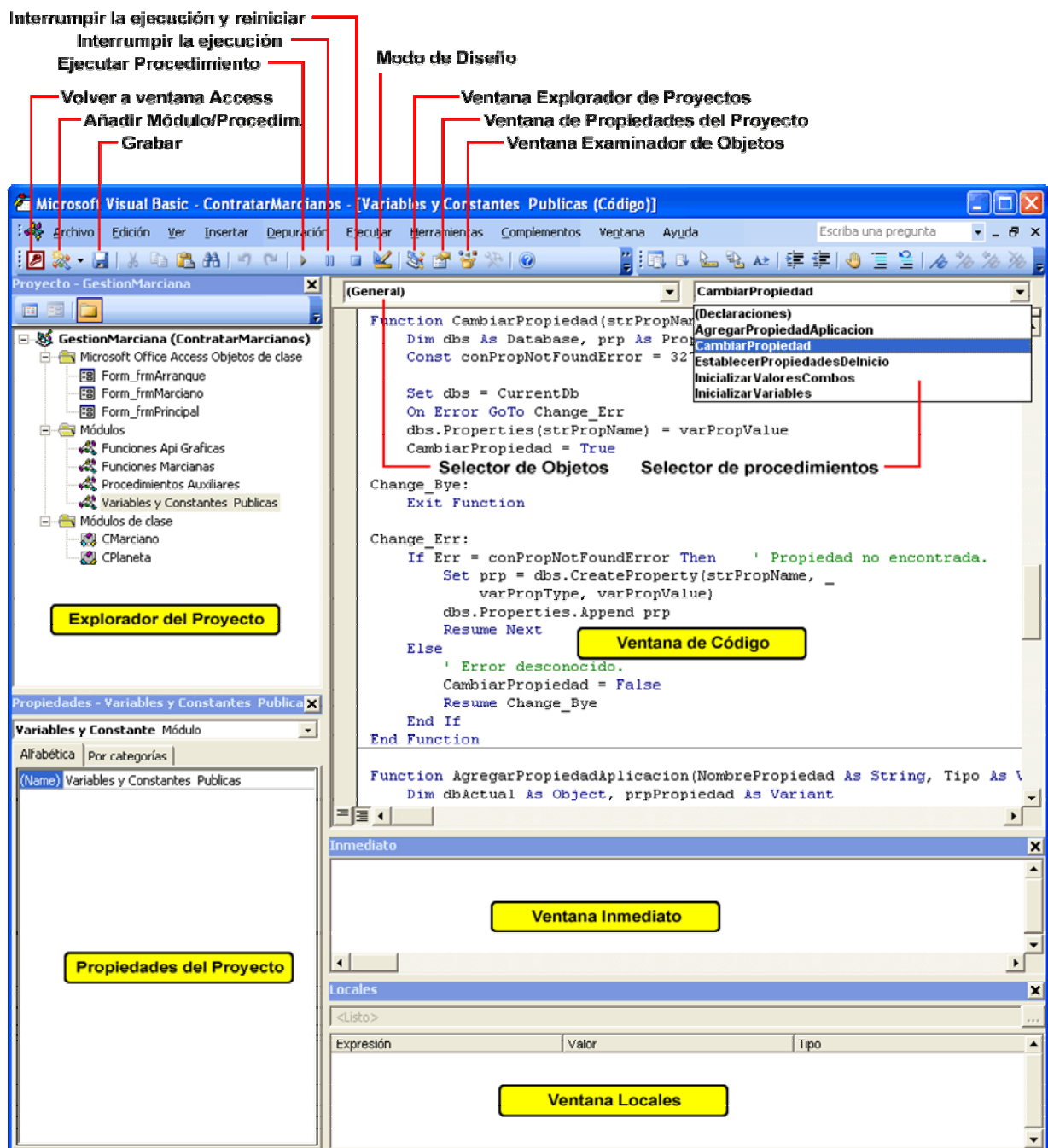
Entorno de desarrollo

Ya hemos comentado que VBA para Access tiene un entorno de desarrollo similar al de otras aplicaciones con VBA.

Este entorno recibe el nombre de IDE, que significa algo así como Entorno de Desarrollo Integrado. En el "idioma bárbaro" se escribe Integrated Development Environment.

Este entorno consta de una serie de ventanas y botones, algunos de los cuales se muestran en el siguiente gráfico.

Ventanas y botones del Editor de código VBA



Las ventanas, por el momento más importantes para nosotros, son

- **Ventana de Código**

Es la ventana en la que se escribe todo el código que va a gestionar nuestra aplicación.

Podemos seleccionar el módulo que se mostrará en cada momento, por ejemplo seleccionándolo con un doble clic en la ventana del Explorador del Proyecto.

- **Ventana del Explorador del Proyecto**

La ventana del proyecto nos muestra todos los objetos que poseen módulos.

Entre ellos tenemos los módulos de Formularios e Informes, los módulos estándar, y los módulos de clase.

En esta primera parte del cursillo vamos a trabajar con los módulos asociados a formularios y los módulos estándar.

Si en la ventana del proyecto hacemos doble clic sobre cualquiera de los elementos que aparecen, abriremos su módulo correspondiente.

Por ejemplo, si en la base de datos correspondiente al gráfico presionáramos (doble clic) el formulario **Form_frmMarciano** nos abriría el módulo asociado al formulario frmMarciano.

Si presionamos sobre **Funciones Marcianas** nos abriría el módulo estándar Funciones Marcianas y si presionáramos sobre **CMarciano**, nos abriría el módulo de clase CMarciano.

Fijaros que el icono asociado a cada módulo cambia en función del tipo que sea.

- **Ventana Inmediato**

Esta es una ventana desde la que podemos ordenar que se ejecuten determinadas órdenes, o imprimir desde el código determinados valores para efectuar comprobaciones. Recibe también el nombre de **Ventana de Depuración**.

Su mayor utilidad es para la depuración del código, es decir para efectuar comprobaciones de errores. De hecho es un objeto que recibe el nombre de **Debug**.

- **Ventana Locales**

Esta es una ventana que también sirve para la comprobación y el seguimiento de los valores que van tomando determinados parámetros, variables, etc. durante la ejecución del código.

Por tanto comparte la función de depuración con la ventana Inmediato.

- **Ventana Propiedades**

En esta ventana podemos ver las propiedades de los diferentes objetos que intervienen en el proyecto, al estilo de la ventana de propiedades del Formulario ó Informe en el modo diseño.

Los módulos, como Objetos de Access, también tienen sus propiedades.

Existen otras ventanas no comentadas aquí, pero que no dejan de ser importantes. Entre ellas podemos destacar la Ventana del **Examinador de Objetos**, o la Ventana de **Inspección**, de las que hablaremos más adelante.

Uso de la ventana Inmediato

Antes de nada vamos a hacer lo siguiente:

Creamos una nueva base de datos a la que llamamos Entrega02.mdb

Vamos a Módulos y presionamos en el botón [Nuevo].

Nos creará un módulo con el original nombre de **Módulo1**.

Vamos a examinar las ventanas que se abren y las comparamos con el gráfico de la página anterior 02-2.

¿Identificas lo que se ha explicado en los puntos anteriores?

Si por un casual no apareciera la ventana Inmediato, haz lo siguiente:

Vete al menú Ver y presiona en la opción [Ventana Inmediato]; o más fácil aún, presiona la combinación de teclas [Ctrl] + [G].

Vamos a mantener las ventanas de **Código**, la de **Inmediato** y la del **Explorador del proyecto**. Las otras, si te molestan, puedes cerrarlas. Más tarde las podrás abrir con los botones o las opciones de menú correspondientes (Menú Ver > y seleccionar su opción).

¿Para qué sirve la ventana Inmediato?

Vamos a hacer un ejemplo.

Pincha con el cursor en ella, escribe lo siguiente:

```
Print 4 + 5
```

 y dale a la tecla [Enter]

Y ¡oh sorpresa! Escribe **9** en la siguiente línea.

Ahora escribe **? 4 + 5** más [Enter]

Y vuelve a escribir la cifra **9**

Escribe **? 4 * 5** más [Enter]

¿Adivinas qué va a escribir? ¡Premio! **20**

Ahora vamos a hacerlo un poco más complicado

```
a = 2 : b = 4 : Print a * b
```

Pues sí, te escribe **8** en la siguiente línea.

¿Qué pasará si escribo **? a^b** y aprieto [Enter]?

¡Sorpresa! Ha escrito **16** que es 2 elevado a 4

Si ahora presiono la tecla de la barra de menú que tiene la forma de un cuadradito azul y que en el gráfico la identifico con Interrumpir la ejecución y reiniciar, si vuelvo a escribir

```
? a^b
```

 como resultado me da **1**.

Pensemos un poco sobre estos sorprendentes resultados.

Tras la primera línea

```
a = 2 : b = 4 : Print a * b
```

parece que Access recuerda que existe “algo” llamado **a**, que se le ha dado el valor **2** y algo llamado **b** que tiene el valor **4**, como se demuestra tras el resultado de escribir la segunda vez **? a^b**

Pero resulta que tras presionar el botón de Reiniciar (el cuadradito azul) resulta que Access se equivoca y da como resultado **1**.

Pues no, Access no se ha equivocado. El exponente **b** tiene el valor **0**, y cualquier número elevado al exponente **0** da como resultado **1**.

En realidad éste no es un gran ejemplo, ya que estaríamos en el caso singular de 0 elevado a cero, que no está definido en el campo de los números reales.

Pero no vamos a ser más “papistas que el Papa”.

Vamos a hacer ahora otro experimento.

Escribimos en la ventana Inmediato:

```
a = "2" : b = "4" : ? a + b
```

Tras presionar la tecla Enter el resultado es ¡chan, chan....!

Si se te ha ocurrido que sería 6 te puedo decir que estás equivocado.

Si tu respuesta ha sido 24, has acertado; pero yo me pregunto ¿No estás perdiendo el tiempo leyendo este capítulo? ¡so listo...!

Bromas aparte expliquemos el resultado.

Como introducción hemos usado unos elementos que se llaman **Operadores**.

Los que hemos usado por ahora son:

- + Suma
- * Producto
- ^ Exponenciación

El símbolo **:** no es el operador de división. En realidad es un simple separador de instrucciones. El operador de división es **/**.

Hay otros muchos más que ya iremos viendo poco a poco.

¿Por qué al escribir `a = "2" : b = "4" : ? a + b`, nos da **24**?

Vamos a fijarnos en estos pequeños detalles `a = "2"` y `b = "4"`

La cifra **2** está entre 2 comillas, lo mismo que la cifra **4**.

Esto hace que VBA considere el 2 y el 4, no como números, sino como texto, también llamado cadena.

Tras esto el operador **+** no efectúa una suma numérica, sino que enlaza dos cadenas de texto. Si enlazamos la cadena `"2"` con la cadena `"4"`, nos da la cadena `"24"`.

Para los “listos” que ya se sabían el resultado, el resto que no haga ni caso de este párrafo, les informo que esto, en lenguajes más avanzados, se llama Sobrecarga de Operadores, lo que equivale a decir que el operador **+** está **Sobrecargado**.

En resumidas cuentas, que con números hace la operación de **Suma numérica** y con cadenas de texto la operación de **enlazar cadenas**.

Vamos a seguir con “pequeñas sorpresas”.

Escribamos algo tan inocente como:

```
Print 2,4 * 2
```

¡Sorpresa!. ¿Qué ha pasado?

Que nos imprime

2

8

¿Por qué no nos ha dado como resultado 4,8 que es lo que le daría a un buen cristiano, judío ó musulmán, (o lo que sea siempre que sea bueno)?.

De toda la vida la operación 2,4 * 2 ha dado como resultado 4,8

¡La culpa de esto la tienen que tener los americanos!

Y no es broma.

Sólo un pequeño matiz:

Los americanos utilizan la coma como separador de miles.

En cambio el separador decimal, en vez de la coma, es el punto.

En una sentencia Print, cuando encuentra una coma entre dos valores, intercala una tabulación.

Probemos ahora:

`Print 2.4 * 2` y nos da 4,8

¡Nueva sorpresa!

-Ahora lo hace bien pero ¿no hemos quedado que el separador decimal es el punto?-.

-¿Por qué ahora me muestra el resultado decimal con una coma como separador?-.

-Cosas de Windows-.

Como sabe que al otro lado de la pantalla hay un hispano, le muestra el resultado "en cristiano"; Excepto para algunos PCs. De México, Florida y otras "colonias".

-Borro lo de colonias-

-¿En qué estaría yo pensando?-

La pregunta ahora es

-¿Y cómo lo sabe?-.

-Fácil; para eso está la Configuración Regional de Windows-.

Vamos a dejar de divagar y sigamos experimentando.

Escribimos en la ventana Inmediato lo siguiente:

`a = 2 : b = "Peras" : Print a + b`

-¡No salimos de sustos!-

-¿Qué he roto?-



Tranquilo que no has roto nada.

¿No nos enseñaron en el colegio que no había que sumar Peras con Manzanas?

Le estamos diciendo a VBA que sume un número con un texto, y lógicamente se queja.

Cambiamos la línea

```
a = "2" : b = " Peras" : ? a + b
```

Ahora sí imprime bien

```
2 Peras
```

Al poner el 2 entre comillas, lo hemos cambiado al modo texto y entonces ha funcionado.

Si hacemos

```
a = "Dos" : b = " Peras" : ? a + b
```

Nos imprimirá

```
Dos Peras
```

-A ver profe si me aclaro-

Tenemos “algo” llamado **a** que le da lo mismo ser **Número** que **Texto**.

¿Cómo se come esto?

Tranquilo “pequeño saltamontes”. No seas impaciente. La respuesta en las próximas entregas.

Comencemos a programar con
VBA - Access

Entrega **03**

Primeros conceptos

Primeros conceptos

Ya hemos visto que en la ventana **Inmediato** podemos efectuar cálculos, con valores directos o con unos elementos a los que les asignamos valores.

Todo esto es interesante, pero todavía no podemos hacer grandes cosas, de hecho con una simple calculadora seríamos más eficientes.

Un poco de paciencia; todo a su tiempo...

Si no tenemos abierto un fichero Access, es el momento para hacerlo, por ejemplo crea un nuevo fichero con el nombre **Entrega03.mdb**.

Abrimos el apartado [**Módulos**] y presionamos el botón [**Nuevo**].

Pulsamos con el cursor en la **ventana de Código** y escribimos lo siguiente:

```
Option Compare Database
Option Explicit

Const Pi As Double = 3.14159265358979
```

¿Qué significa esta última línea?

A falta de explicar otros detalles, declaramos una **Constante** llamada Pi del tipo Double (un valor numérico de coma flotante de 8 bytes) y le asignamos el valor del famoso Pi.

Os recuerdo que Pi es algo más que el simple 3,1416 que nos enseñaron en el colegio...

Al declarar Pi como Constante nos impide que en otro punto del módulo podamos hacer algo así como

```
Pi = 25.36
```

-¿Por qué?-

-Porque el valor de una constante es en definitiva Constante, y no se puede cambiar.-

-¿Y cómo podemos declarar algo que se pueda cambiar?-

-Ese algo, en contraposición a una constante, se llama **Variable**.

Para declarar variables se podría hacer algo así como

```
Dim Precio As Currency
Dim Nombre As String
```

Enseguida veremos qué es eso de **Currency** y **String**.

Ahora vamos a intentar usar el valor que hemos declarado de Pi.

Pulsamos en la ventana **Inmediato** y escribimos

```
? Pi y presionamos [Enter]
```

¡Y no pasa nada!

Se suponía que seríamos muy felices si nos imprimiera el valor 3,14159265358979

¿Por qué no lo ha hecho?

Ámbito ó Alcance de las Constantes y variables

Antes que nada aclarar que se suelen utilizar indistintamente ambos términos

¿Qué entendemos por **ámbito** o por **Alcance**?

Algo tan simple, y a la vez tan complejo, como los lugares en los que VBA puede “Ver”, a una variable ó constante lo que equivale a que se entera que existen, las constantes y variables que declaramos. Más adelante veremos que todo esto aún se puede generalizar para otros elementos de la programación.

En el punto anterior hemos escrito ? **Pi** en la ventana Inmediato y no ha pasado nada.

Al declarar en la cabecera de un módulo una variable con **Dim**, o una constante con **Const** hacemos que esa variable, y esa constante sólo sean visibles desde dentro del código del módulo.

Para que puedan verse desde fuera es necesario ponerles la palabra **Public**.

Esto se hace de forma diferente para las **Constantes** y **Variables**.

En el caso de las constantes se pone **Public** delante de la palabra **Const**.

En el caso de las variables **Public** sustituye a **Dim**.

El código del módulo quedaría así:

```
Option Compare Database
Option Explicit
```

```
Public Const Pi As Double = 3.14159265358979
Public Precio As Currency
Public Nombre As String
```

Si ahora pulsamos con el ratón en la ventana **Inmediato**, escribimos ? **Pi** y [Enter]

Nos escribirá 3,14159265358979.

Vamos a hacer otra prueba

Escribe en la ventana Inmediato **Pi = 3.1416** y pulsa [Enter]. ¿Qué ha pasado?



```
Pi = 3.1416
```

Simplemente que estamos intentando asignar un valor a una constante que ya lo tenía.

En definitiva una constante sólo puede tomar el valor una vez, “y a ti te encontré en la calle”.

Por definición una constante no puede cambiar.

Sólo mantiene relaciones y se casa una vez.

En cambio las variables son mucho más promiscuas.

Todo lo anterior, entendedlo como una introducción a la declaración de variables y Constantes, que en realidad presenta más posibilidades que las aquí vistas.

En puntos posteriores, estudiaremos el tema con más profundidad.

Aquí han aparecido cosas nuevas; por ejemplo la palabra `Currency` y la palabra `String`.

En el ejemplo, con `Currency`, hacemos que la variable `Precio` sea del tipo `Moneda`; un tipo de datos “aparentemente” de coma flotante creado para manejar datos monetarios sin errores de “Redondeo”.

Podéis obtener ayuda sobre estos tipos de datos poniendo el cursor, por ejemplo en cualquiera de sus nombres, por ejemplo `Currency` y pulsando [F1].

Con `String` hacemos que la variable `Nombre` sea tratada como una variable de tipo `Cadena`, que permite manejar cadenas de `Caracteres`.

En realidad podríamos haber hecho las declaraciones sin especificar su tipo pero, por el momento sin más explicaciones, os diré que esto genera código menos optimizado e incluso puede dar problemas a la hora de encontrar posibles errores.

Por ejemplo podríamos haber hecho:

```
Public Const Pi = 3.14159265358979
Public Precio
Public Nombre
```

Y la cosa funcionaría...

En realidad el declarar así una variable ó constante, equivale a haber hecho esta otra declaración

```
Public Const Pi As Variant = 3.14159265358979
Public Precio As Variant
Public Nombre As Variant
```

-¿Y qué es eso de `Variant`?-

El `Variant` es un tipo de dato al que se le puede asignar casi cualquier cosa.

Esto lo hace a costa de consumir más recursos que otros tipos de variables y por consiguiente una velocidad más lenta en la ejecución.

Ya tenemos las bases para el siguiente paso:

- Procedimientos `Sub`
- Procedimientos `Function` ó `Funciones`

Procedimientos `Sub`

Un procedimiento `Sub` llamado también `Procedimiento` a secas es un conjunto de código que realiza determinadas tareas.

Suele estar contenido entre las expresiones `Sub` y `End Sub`

El término Sub puede ir precedido de otras expresiones, por ejemplo para delimitar el ámbito en el que puede ser llamado el procedimiento.

Al procedimiento se le pueden pasar una serie de datos para que los use internamente.

A estos datos se les llama **Parámetros** ó **Argumentos**.

Si en la ventana de código escribimos la palabra **Sub**, le ponemos encima el cursor y presionamos la tecla [F1], Access nos mostrará la ayuda aplicable a **Sub**.

En la ayuda podemos entre otras cosas podemos ver:

```
[Private | Public | Friend] [Static] Sub nombre [(lista_argumentos)]
```

```
[instrucciones]
```

```
[Exit Sub]
```

```
[instrucciones]
```

End Sub

Por cierto, estas líneas indican cómo es la estructura de un procedimiento **Sub**.

Los elementos que están entre Paréntesis Cuadrados [] son opcionales.

Cuando hay varias palabras separadas por Barras Verticales |, nos está indicando que podemos seleccionar una de ellas.

Según esto serían igual de válidas las sentencias:

```
Sub Procedimiento_01()
```

```
End Sub
```

```
Public Sub Procedimiento_02()
```

```
End Sub
```

```
Private Sub Procedimiento_03()
```

```
End Sub
```

```
Public Sub Procedimiento_04(Argumento1 As Double)
```

```
End Sub
```

Por supuesto que cada opción genera un código que se comportará de forma diferente.

Es igualmente válido el que ya hemos usado en la primera entrega.

```
Private Sub cmdSaludo_Click()  
    Me.lblSaludo.Caption = "¡¡¡Hola Mundo!!!"  
    Me.Caption = "¡Aquí estoy!"  
End Sub
```

Con una salvedad, este último es un tipo especial de `Sub`. Y es especial porque captura el evento que se produce cuando se presiona el botón `cmdSaludo` de su formulario. Recordemos que un Evento es un mensaje que envía Windows y en este caso es capturado por Access.

En el módulo que hemos creado vamos a escribir el siguiente código:

```
Public Sub Circunferencia()
    Dim Radio As Single
    Radio = 2.5
    Debug.Print "Circunferencia = " & 2 * Pi * Radio
    Debug.Print "Círculo = " & Pi * Radio ^ 2 & " m2"
End Sub
```

Ahora, en la ventana Inmediato escribe `Circunferencia` y presiona [Enter]. Efectivamente, en esa ventana se escribirá:

```
Circunferencia = 15,7079632679489
Círculo = 19,6349540849362 m2
```

Vamos a analizar un poco esto.

Hemos creado un procedimiento llamado `Circunferencia`.

Este procedimiento es de los de tipo `Sub`.

La primera línea es el encabezado.

Al hacerlo `Public`, dentro de un módulo estándar, permitimos que se pueda llamar desde cualquier parte de la base de datos. Es un procedimiento público.

En la segunda línea declaramos la variable `Radio`, de tipo `Single`.

El tipo `Single` es un tipo de coma flotante, que ocupa 4 bytes, o lo que es lo mismo, 32 bits.

En la siguiente línea asignamos a la variable el valor `2.5`.

Recuerdo que dentro del código, las comas separadores de decimales se ponen como puntos.

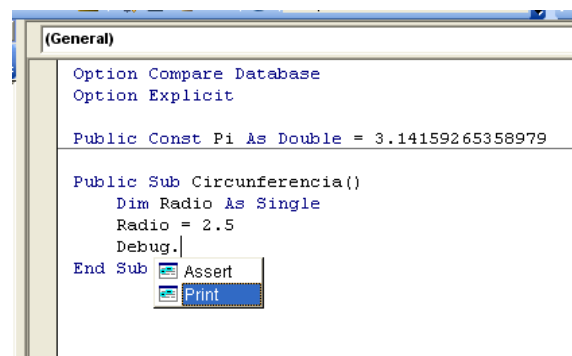
La siguiente línea utiliza el método `Print` del objeto `Debug`.

La forma de llamarlo es poniendo primero el objeto, a continuación un punto y después el método.

Fijaros que mientras lo escribís se os muestra una "Ayuda en línea" que os permite seleccionar lo que queréis escribir.

El objeto `Debug` es la propia ventana Inmediato.

El método `Print` imprime lo que se ponga a continuación.



Lo primero que imprime es una cadena, en nuestro caso "Circunferencia = "

A continuación vemos el operador & seguido de la expresión $2 * \text{Pi} * \text{Radio}$

Este operador hace de "Pegamento" entre dos expresiones.

En este caso enlaza la cadena primera con el resultado de la operación $2 * \text{Pi} * \text{Radio}$

Dando como resultado **Circunferencia = 15,7079632679489**

Lo mismo es aplicable a la siguiente línea.

Con `End Sub` se acaba el código del procedimiento.

-Esto está bien, pero sólo puede imprimir los datos de la circunferencia de radio 2,5, lo que no es gran cosa-

-Bueno, sí. Es un inicio-

-Ahora, me voy a sacar un conejo de la chistera y modifico el procedimiento Sub, de la siguiente forma:

```
Public Sub Circunferencia(Radio As Single)
    Debug.Print "Circunferencia = " & 2 * Pi * Radio
    Debug.Print "Círculo = " & Pi * Radio ^ 2 & " m2"
End Sub
```

Fijaros que la declaración de la variable Radio ya no se hace en el cuerpo del procedimiento, sino en su cabecera.

Además ha desaparecido la línea en la que asignábamos el valor del radio.

-¿Y cómo se utiliza esto?-

No tiene ninguna complicación especial

Vamos a modificar la forma como llamamos desde la ventana Inmediato al procedimiento, y escribimos

```
Circunferencia 2.5
```

El resultado es el mismo que en el caso anterior.

Pero si escribimos

```
Circunferencia 7.8
```

Nos dará los valores correspondientes a una circunferencia de radio 7.8.

En este caso hemos utilizado un procedimiento con un único argumento.

El argumento es precisamente 7.8.

Si tuviera, por ejemplo 2, se escribirían separados por una coma

```
NombreProcedimiento Argumento1, Argumento2
```

Hay otro tipo de procedimientos de los que hemos hablado.

Se llaman procedimientos **Function**, las famosas **Funciones**.

Tienen la particularidad de que pueden devolver un valor, pero de eso hablaremos en la próxima entrega.

Comencemos a programar con
VBA - Access

Entrega **04**

Primeros conceptos 2

Funciones

En la entrega anterior hemos visto una introducción a los procedimientos `Sub`.

Otro de los tipos de procedimientos son los procedimientos `Function`, que al igual que los procedimientos `Sub` están delimitados por `Function` y `End Function`.

La principal diferencia entre un procedimiento `Sub` y un procedimiento `Function` es que este último devuelve un valor.

Entre los dos delimitadores se escribe el código que va a realizar las tareas que deseemos, y al final devolverá algún valor.

Son las llamadas **Funciones**.

Veamos la forma como lo explica la ayuda de Access.

Si vamos a la ventana Inmediato, escribimos `Function`, ponemos el cursor en lo escrito y presionamos [F1], entre otras cosas nos aparece la sintaxis de su declaración:

```
[Public | Private | Friend] [Static] Function nombre [(lista_argumentos)] [As tipo]
[instrucciones]
[nombre = expresión]
[Exit Function]
[instrucciones]
[nombre = expresión]
```

End Function

Vemos que para declarar una función empezaríamos, por ejemplo poniendo la palabra `Public` (si quisiéramos que la función sea accesible desde cualquier parte de Access) a continuación la palabra `Function`, después abriríamos un paréntesis y pondríamos los parámetros que necesitáramos, cerraríamos el paréntesis y finalmente pondríamos el tipo de datos que devolvería la función.

Supongamos que quisiéramos escribir una función en la que pasándole el ángulo y el radio, nos devolviera la longitud de un arco de circunferencia.

La cabecera de esta función sería:

```
Public Function Arco(Angulo As Single, Radio As Single) As Single
```

Hay un truco para que una línea con mucho texto se pueda dividir en varias líneas, sin que deje de ser una única línea.

Se coloca al final de la línea que se quiere dividir un espacio en blanco y la raya de subrayado.

Con ello la línea anterior se podría poner así:

```
Public Function Arco( _
    Angulo As Single, _
    Radio As Single _
) As Single
```

Aquí declaramos la función como pública, para que se pueda utilizar desde cualquier parte de la aplicación, siempre que se escriba en un módulo normal.

A continuación escribimos la palabra `Function`, seguida de su nombre, en este caso `Arco`.

Abrimos paréntesis y escribimos los dos parámetros que necesitamos, Angulo y Radio, declarándolos del tipo Single (tipo numérico de coma flotante de 4 Bytes).

Cerramos el paréntesis y declaramos la función Arco también del tipo Single.

Si escribís la cabecera de la función en un módulo normal, veréis que VBA, al igual que pasaba con los procedimientos Sub, os añade automáticamente el final correspondiente, es decir

```
End Function
```

Con esto la función quedaría así:

```
Public Function Arco( _  
    Angulo As Single, _  
    Radio As Single _  
    ) As Single
```

```
End Function
```

Por ahora no es gran cosa, ya que no hace nada.

Permitidme que escriba el código completo de esta función y os la explique paso a paso.

Como en la entrega anterior, voy a declarar la constante Pi como pública en la cabecera del módulo, para que pueda ser utilizada por la aplicación.

Si ya la tuviera declarada en otro módulo este paso no sería necesario.

El código sería el siguiente:

```
Option Compare Database  
Option Explicit  
  
Public Const Pi as Double = 3.14159265358979  
  
Public Function Arco( _  
    Angulo As Single, _  
    Radio As Single _  
    ) As Single  
    Arco = Pi * Radio * Angulo / 180  
End Function
```

Aparte de la fórmula matemática de Trigonometría elemental, vemos lo siguiente.

Asignamos al nombre de la función Arco el resultado de la fórmula de la izquierda.

Esto hará que la función Arco devuelva el resultado de la operación matemática.

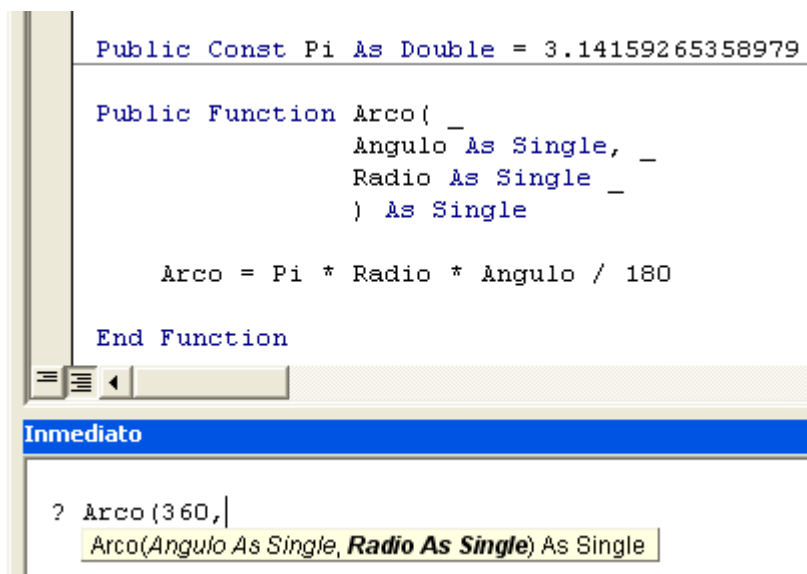
Vamos a centrarnos en lo que aparece entre **Function** y el final de la línea.

Vamos a probar la función.

Para ello escribiremos en la ventana Inmediato.

```
? Arco(360,1)
```

Supongo que ya no será sorpresa para vosotros la ayuda en línea que os va prestando el propio VBA de Access:



```
Public Const Pi As Double = 3.14159265358979

Public Function Arco( _
    Angulo As Single, _
    Radio As Single _
) As Single

    Arco = Pi * Radio * Angulo / 180

End Function
```

Inmediato

```
? Arco(360,|
Arco(Angulo As Single, Radio As Single) As Single
```

Tras completar la expresión, y darle a [Enter] nos escribirá: **6,283185** que es la longitud de un arco de radio 1 y ángulo 360°, lo que equivale a la circunferencia completa.

Vamos a hacer un experimento: ¿Qué pasa si cambiamos la declaración de tipo de esta función?

Vamos a cambiar el Single de la declaración de la cabecera, por Double.

Double es un número de coma flotante de más precisión que single (8 Bytes de Double frente a 4 de single). La cabecera quedará así:

```
Public Function Arco( _
    Angulo As Single, _
    Radio As Single _
) As Double
```

Para ello escribiremos **? Arco(360,1)** en la ventana Inmediato, y presionamos [Enter].

Efectivamente la precisión ha aumentado, ya que el valor escrito ha sido:

```
6,28318530717958
```

Por ello si cambiamos ahora la cabecera:

```
Public Function Arco( _
    Angulo As Double, _
    Radio As Double _
) As Double
```

Se supone que los cálculos serán más precisos.

Efectivamente así es, pero en una aplicación crítica deberíamos jugar que el tipo de precisión adecuada, para hacer la aplicación lo más ligera posible.

Si probamos a hacer el cambio vemos que en nuestra función, con los datos anteriores apenas cambia el resultado. Pero ¿qué pasaría si estuviéramos programando la órbita de una nave con destino a Marte? En este caso la precisión sería un elemento clave.

Funciones en formularios

Vamos a hacer una pequeña y elemental calculadora que nos permita sumar dos cifras.

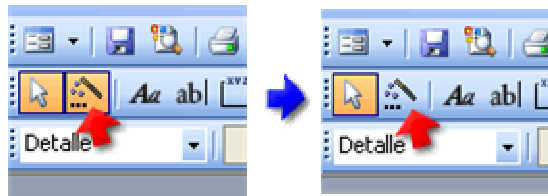
Cerramos los módulos y creamos, ó abrimos un formulario en modo diseño.

Este formulario no lo enlazamos con ninguna tabla ni consulta.

Ahora vamos a deshabilitar el asistente para controles.

En la barra de controles, pulsamos en la “Varita mágica” para deshabilitarla.

Deberá quedar sin resaltar, como se indica en el gráfico con las flechas rojas.



Ponemos dos cuadros de texto en el formulario.

Por defecto, Access les pondrá un nombre semejante a Texto2 y Texto2.

Vamos a cambiar esos nombres.

Abrimos la hoja de propiedades, os recuerdo Menú **Ver** > **Propiedades**, ó pulsando las teclas **[Alt] + [Enter]**.

En la pestaña **[Otras]** ponemos el cursor en la primera propiedad, que “casualmente” es la propiedad **Nombre**. Ahí es donde, pulsando en los correspondientes cuadros de texto, vamos a cambiar su nombre.

Les ponemos, al primer Cuadro de texto el nombre de **txtSumando_1**, y al segundo **txtSumando_2**.

Ahora le vamos a poner un botón, cambiando su nombre, algo así como Comando5, por **cmdSumar**, y su título ó texto del botón, por **Sumar**.

Al programar es muy importante que los objetos, variables, constantes, procedimientos, etc. tengan nombres con cierto sentido.

Si no fuese así, si más adelante tuvieras que revisar tu código, o si lo tuviese que hacer otra persona, tendríais que invertir una gran cantidad de tiempo en averiguar qué es y qué hace cada cosa en el código.

Más adelante hablaremos sobre normalización de código, pero fijate que el nombre de los cuadros de texto está precedido por las letras **txt** y el del botón por **cmd**.

Así si en un pedazo de código veo una palabra que empieza por **cmd** sé que se refiere a un botón, y si empieza por **txt** sé que se refiere a un cuadro de texto.

Otro ejemplo: Los nombres de las etiquetas los suelo empezar por **lbl**.

Estas son **convenciones de normalización** “más ó menos” estándares.

A partir de este momento las iré aplicando al código y veréis como su uso se irá convirtiendo en algo cada vez más natural.

A lo que íbamos.

Tenemos en un formulario los siguientes elementos:

<code>txtSumando_1</code>	Cuadro de texto
<code>txtSumando_2</code>	Cuadro de texto
<code>cmdSumar</code>	Botón de comando

Probablemente al poner los cuadros de texto se nos hayan colocado dos etiquetas.

De momento no vamos a cambiar su nombre, pero les vamos a poner como Título

Operando 1

Operando 2

Ahora vamos a poner una etiqueta nueva, con título Resultado, y de nombre `lblResultado`.

Más de uno se habrá dado cuenta que `lbl` viene de **Label** (etiqueta en inglés), `txt` de **Text** y `cmd` de **Command**.

A esta etiqueta le vamos a poner un texto un poco más grande y como color de texto el rojo.

¿Pero qué quieres que hagamos con todo esto?

Tan simple como tener un formulario en el que escribir dos números en las correspondientes casillas, y que al pulsar un botón nos muestre su suma.

¿Y cómo sabe Access cuando se ha pulsado el botón y qué es lo que tiene que hacer?

Para eso utilizaremos el Evento **Clic** del botón `cmdSumar`.

Ya he comentado que un Evento es un “*mensaje*” que manda Windows, *cuando pasa algo*, que puede ser captado por Access y a su vez reenviado a Windows con un código que se tiene que ejecutar.

Seleccionamos el botón, y en la [**Hoja de Propiedades**] pulsamos en la pestaña Eventos.

Seleccionamos el cuadro correspondiente a “Al hacer clic” y pulsamos en el pequeño botón con tres puntos que nos aparece.

A continuación veremos una pequeña pantalla que nos permite seleccionar entre

- Generador de expresiones
- Generador de macros
- Generador de código

Lógicamente seleccionamos **Generador de código**, y pulsamos [**Aceptar**] ó doble clic en [Generador de código].

Inmediatamente se nos abre el editor de código con el siguiente código

```
Option Compare Database
Option Explicit

Private Sub cmdSumar_Click()

End Sub
```

El cursor estará seguramente posicionado entre las dos líneas.

¿Qué es esto?

Es el procedimiento que captura el evento Clic del botón **cmdSumar**.

Ahora lo que quiero es que tras poner un valor en cada uno de los cuadros de texto, si pulsamos el botón, nos muestre el resultado de la suma en la etiqueta **lblResultado**.

El contenido de un cuadro de texto lo maneja su propiedad **Value**, que se puede leer y escribir.

El contenido de una etiqueta (su título) lo maneja su propiedad **Caption**, y también es de lectura y Escritura.

Vamos al curro.

Escribimos lo siguiente

```
Private Sub cmdSumar_Click()  
Me.lblResultado.Caption = Me.txtSumando_1.Value + Me.txtSumando_2.Value  
End Sub
```

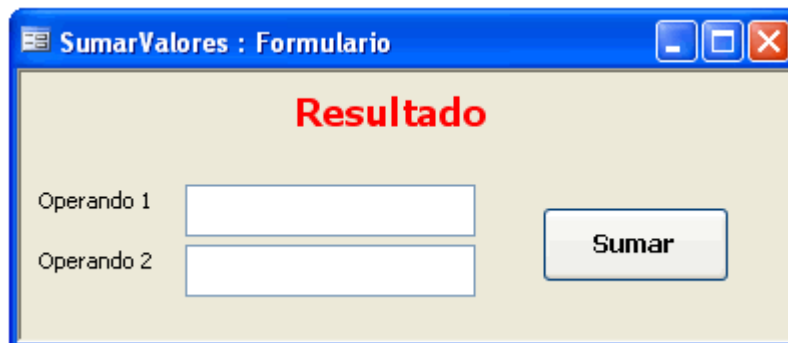
Que traducido “al cristiano” quiere decir:

Cuando se presione sobre el botón **cmdSumar** (evento clic) pon como título de la etiqueta **lblResultado** (propiedad **caption**) el resultado de sumar el contenido (propiedad **Value**) del cuadro de texto **txtSumando_1** y del cuadro de texto **txtSumando_2**.

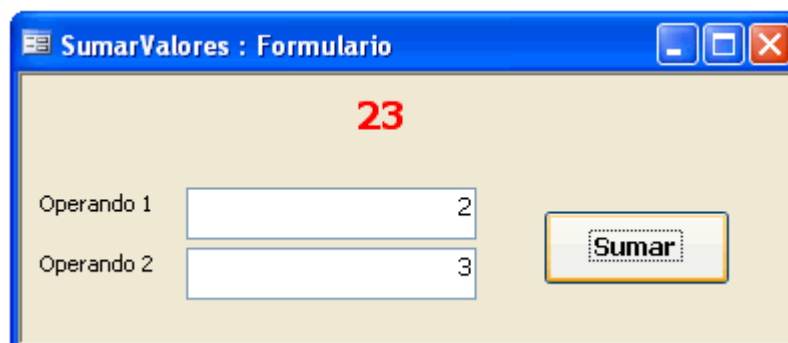
El control Cuadro de texto es un **TextBox** y la Etiqueta un control **Label**.

Bueno, grabamos el código y el formulario y lo abrimos.

Nos quedará algo así como



Si ahora introducimos, por ejemplo el valor 2 en el primer cuadro de texto, el valor 3 en el segundo y presionamos el botón Sumar, nos llevamos una pequeña sorpresa:



¿Qué ha pasado?.

¿Por qué en vez de un 5 nos da 23? al fin y al cabo 2 más 3 son 5

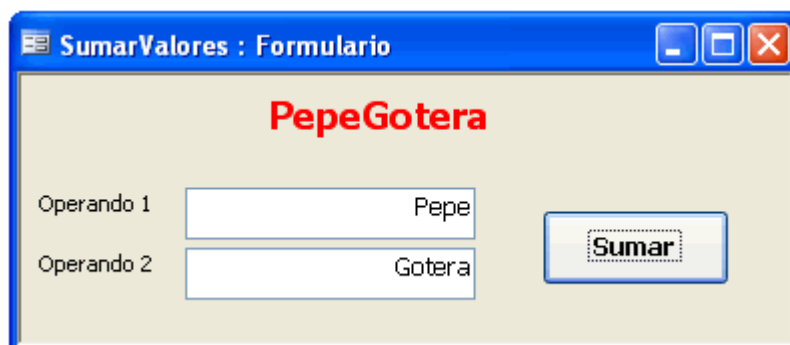
Estas son algunas de las cosillas que desaniman a los principiantes...

No es que Access nos esté tomando el pelo. Recordemos que estamos hablando del contenido de un [Cuadro de Texto], y que como tal todo lo que contiene lo interpreta como texto.

Aunque hemos introducido un 2 para el control es como si introdujéramos el texto "2".

Vamos a probar a introducir en el primer Cuadro de texto **Pepe**, y en el segundo **Gotera**.

Al presionar el botón nos da



Un par de párrafos más adelante veremos cómo solucionar este problema.

Primero un par de matizaciones sobre el código.

Tomemos el texto

```
Me.lblResultado.Caption
```

Vamos a desmenuzarlo.

Me representa el **formulario** al que pertenece el código.

```
Me.lblResultado
```

lblResultado representa a la **etiqueta** lblResultado del formulario actual.

```
Me.lblResultado.Caption
```

Caption representa a la propiedad **Título** de la etiqueta.

La utilización de Me no es necesaria; pero facilita a la ayuda en línea el que te presente las diferentes opciones a escribir.

Tomemos ahora el texto

```
Me.txtSumando_1.Value
```

Ya hemos visto que me representa al formulario.

```
Me.txtSumando_1
```

txtSumando_1 representa al **cuadro de texto** txtSumando_1

```
Me.txtSumando_1.Value
```

Value representa el contenido del cuadro de texto ó la propiedad **Value**.

La propiedad Value de los cuadros de texto tiene dos peculiaridades:

La primera es que es la propiedad por defecto del control.

Eso quiere decir, que si se escribe en el código el nombre del cuadro de texto, sin especificar nada más, hará implícitamente referencia a la propiedad Value.

Por ello son equivalente es igual de válidos los siguiente códigos:

```
Me.lblResultado.Caption = Me.txtSumando_1.Value + Me.txtSumando_2.Value
```

```
lblResultado.Caption = txtSumando_1 + Me.txtSumando_2
```

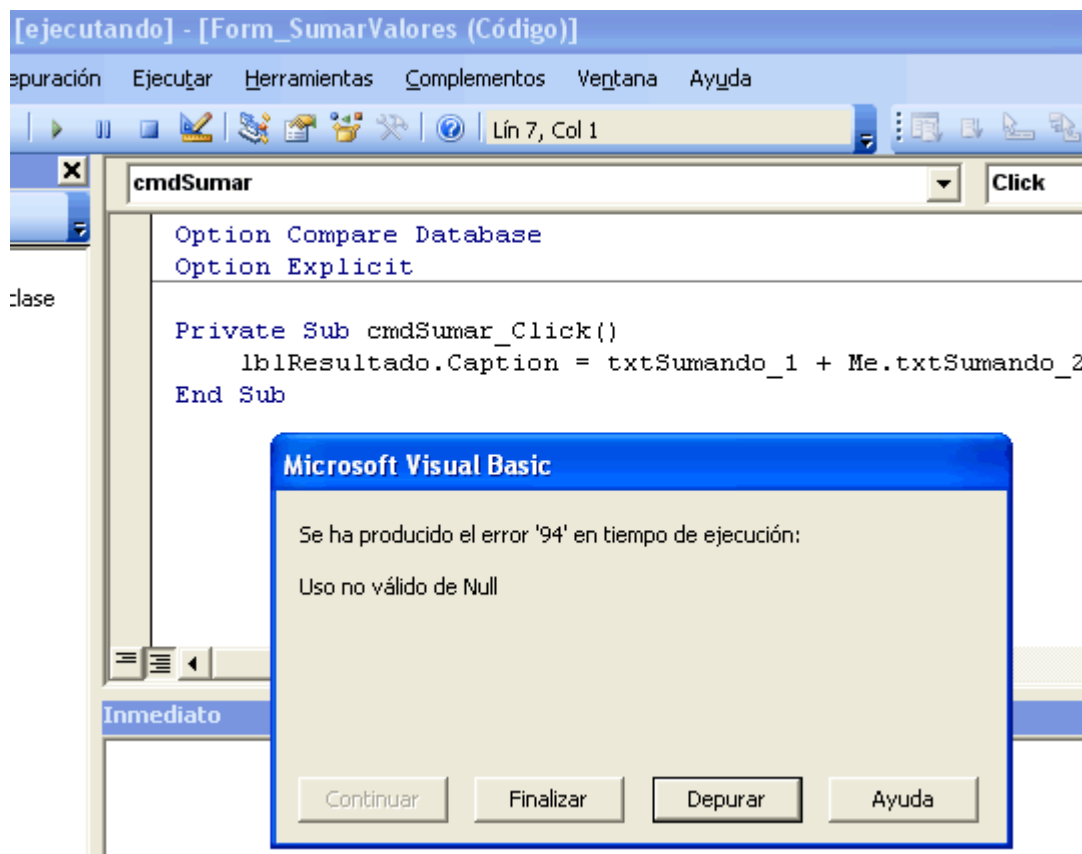
He quitado las referencias al formulario **Me** y la propiedad **Value**.

Cambiamos el código anterior por éste:

```
Private Sub cmdSumar_Click()  
    lblResultado.Caption = txtSumando_1 + Me.txtSumando_2  
End Sub
```

Ahora vamos a ejecutar el formulario, pero antes de presionar el botón de sumar dejamos uno de los cuadros de texto sin nada escrito en él.

¡Un nuevo problema!...



Esto hace referencia a la segunda peculiaridad de los cuadros de texto:

Si el cuadro de texto contiene algún valor, nos devolverá un texto con ese valor, pero si no contiene nada nos devolverá el valor **Nulo (Null)**.

Para poder funcionar así, la propiedad Value del cuadro de texto tiene que ser del tipo **VARIANT**. Sí ese tipo de dato capaz de tragarlo todo.

Un valor nulo no es cero, ni una cadena de texto vacía. Un valor nulo en realidad se corresponde al resultado del código ASCII 0. Por el momento no os importe no saber qué significa esto.

Como uno de los cuadros de texto tenía el valor **Nulo (Null)**, VBA no puede sumarlo al contenido del otro cuadro de texto. Podríamos decir que un nulo no se puede sumar con nada.

¿Pero qué tenemos que hacer para poder sumar los datos?

Primero deberíamos convertir los valores contenidos en los cuadros de texto a valores numéricos.

Para efectuar esa conversión existe una función de VBA llamada **Val (Valor)**

Si escribimos en la ventana inmediato la palabra Val, ponemos en ella el cursor y presionamos **[F1]** nos aparece la ayuda de la función Val.

Con la siguiente información:

Devuelve los números contenidos en una cadena como un valor numérico del tipo adecuado.

Sintaxis

Val(cadena)

El argumento obligatorio *cadena* es cualquier expresión de cadena válida.

Vamos a hacer pruebas en la ventana Inmediato.

Escribimos las sucesivas expresiones y presionamos [Enter]

Expresión	Resultado impreso
? "2" + "3"	23
? 2 + 3	5
? Val("2") + Val("3")	5
? Val("12.234")	12,234
? Val(" 23 Caballos")	23
? Val(" 3,1416")	3
? Val("Pepe")	0
? Val(Null)	Genera un error (Nulo no tiene un valor numérico)
? Val(Nz(Null,""))	0
? Nz(Null,"Valor Nulo")	Valor Nulo
? Nz("Pepe","Valor Nulo")	Pepe

Después de estos pequeños experimentos dejo que saquéis vuestras propias conclusiones.

Vemos aquí una cosa nueva, la función Nz (Valor).

Vamos a escribir Nz en la ventana Inmediato y pulsamos **[F1]**.

Pone algo semejante a esto

Nz(ValorVariant[,ValorSiEsNulo])

El resumen de la información es que la función Nz devuelve uno de esto dos tipos de datos.

Si el primer argumento no es nulo, devuelve el primer argumento.

Si fuera nulo, devolvería el valor del segundo argumento.

Y con esto ¿qué conseguimos?.

Vamos a efectuar un nuevo cambio en el evento Clic del botón y escribimos lo siguiente:

```
Private Sub cmdSumar_Click()  
    Dim dblSumando1 As Double  
    Dim dblSumando2 As Double  
  
    dblSumando1 = Val(Nz(txtSumando_1, ""))  
    dblSumando2 = Val(Nz(txtSumando_2, ""))  
  
    lblResultado.Caption = CStr(dblSumando1 + dblSumando2)  
End Sub
```

Si ahora no escribimos nada en alguno, o los dos cuadros de texto, ya no nos genera el error.

Igualmente si escribimos valores numéricos, nos muestra el dato correcto.

Si escribimos un dato no numérico lo cambia al valor Cero, ó al resultado de Val(Expresión).

Tenemos una nueva función.

En la penúltima línea del código vemos:

```
lblResultado.Caption = CStr(dblSumando1 + dblSumando2)
```

La función **CStr(Valor)** convierte un valor numérico a un texto, para asignarlo a la propiedad **caption** de la etiqueta **lblResultado**., que sólo admite textos.

Este último paso no es estrictamente necesario, ya que VBA efectúa implícitamente la conversión.

Fijaros ahora que en el código repetimos dos veces la expresión Val(Nz(...

Si pensamos un poco vemos que nos resultaría interesante crearnos una función más avanzada que Val, que nos devolviera un valor numérico como ésta pero que no diera error si se le pasa un valor nulo.

Podríamos llamarla ValorNumerico(Valor) y devolvería, por ejemplo un valor de tipo coma flotante Doble.

Como deseamos que se pueda utilizar desde cualquier parte de Access la haremos pública y la escribiremos en un módulo Estándar y el código sería algo así:

```
Public Function ValorNumerico ( _  
    Valor As Variant _  
    ) As Double  
  
    ValorNumerico = Val(Nz(Valor, ""))  
End Function
```

Por cierto, las **dos comillas** seguidas "" equivalen a una cadena de texto de longitud 0.

No confundir con Null. Ya se que es lioso, pero por ahora creedme.

Esta función devuelve el valor Cero cuando le pasamos un valor que sea texto no numérico, o un valor nulo.

El parámetro `Valor` se declara como `Variant` para que pueda admitir tanto Cadenas de Texto, como Números e incluso el valor `Null`.

Os recuerdo que la función la debéis escribir en un módulo estándar si la queréis usar desde cualquier otra parte de Access.

Tras esto podríamos simplificar el código del formulario y cambiarlo por el siguiente:

```
Private Sub cmdSumar_Click()  
    Dim dblSumando1 As Double  
    Dim dblSumando2 As Double  
  
    dblSumando1 = ValorNumerico(txtSumando_1)  
    dblSumando2 = ValorNumerico(txtSumando_2)  
    lblResultado.Caption = CStr(dblSumando1 + dblSumando2)  
End Sub
```

Ya se que todo esto os puede resultar un poco indigesto, pero haciendo pruebas es como mejor lo podréis asimilar.

En la próxima entrega haremos un repaso y ampliación de lo visto hasta ahora.

Comencemos a programar con
VBA - Access

Entrega **05**

**Tipos de datos y
Declaraciones**

Declaración de variables

En entregas anteriores hemos visto cómo se declaran las variables. En esta entrega vamos a ampliar conceptos.

Una variable es un elemento del código que apunta a una dirección de memoria en donde se almacena un dato.

Haciendo referencia a la variable se puede devolver el dato al que apunta e incluso modificarlo.

Las **constantes** son similares a las variables, sólo que su contenido se le asigna en el momento en el que se declaran, y después no es posible cambiarlo.

Hay tres temas que debemos considerar en una variable

- El **nombre** de la variable
- El **tipo** de dato al que apunta
- El **ámbito** en el que es visible.

Construcción del nombre de una variable (o constante).

Nombre

El **nombre** de una variable está compuesto por caracteres ASCII.

Para su construcción hay que ceñirse a las siguientes reglas:

- No se pueden usar caracteres que tienen un uso especial en Access, como son el Punto ".", los paréntesis "(", ")", la barra vertical "|", o los caracteres que se pueden utilizar como operadores; entre ellos
+ - / * < >.

- Una variable debe empezar por una letra ó por el signo de subrayado
Estos nombres serían correctos, lo que no quiere decir que sean aconsejables.
A123456 _Pepe R2P2

- El nombre de una variable no puede tener espacios en blanco.
Por ejemplo no sería válido el nombre Apellidos Nombre.
En cambio sí sería válido Apellidos_Nombre ó ApellidosNombre.

- Una variable puede terminar con un carácter de declaración de tipo
% & ! # @ \$

Estos caracteres sólo se pueden usar al final del nombre de la variable.

Nota: Estos caracteres también se pueden usar para declarar el tipo de dato que devuelve una función.

Por ejemplo esta cabecera de función sería válida:

```
Public Function Nombre$()
```

Que sería equivalente a:

```
Public Function Nombre() As String
```

- No se puede usar como nombre de variable una palabra reservada de VBA.
Por ejemplo no se pueden usar String, integer, For, If como nombres de variable.
- El nombre de una variable puede tener hasta 255 caracteres - aunque yo de ti no lo intentaría forastero -.

- No se pueden declarar dos variables con el mismo nombre dentro del mismo procedimiento o en la cabecera de un mismo módulo.

Tipos de datos

Además de las **Variables** hay otra serie de elementos que manejan datos, como son las **Constantes**, Procedimientos **Sub** y procedimientos **Function** que son capaces de manejar datos de distintos tipos, e incluso las funciones que devuelven datos.

Pero ¿qué tipos de datos podemos manejar? Y ¿qué características tienen?

Hay varios tipos de datos.

Entre ellos podemos definir

Numéricos

Booleanos

Fecha / Hora

De texto (cadenas)

Variant

De objeto

Registros de datos definidos por el usuario, . . .

Datos numéricos

Existen dos familias de datos numéricos.

Datos numéricos de **número entero**

Datos numéricos de **coma flotante**.

Como **datos enteros** tenemos los siguientes tipos:

Nombre del Tipo	Tamaño	Valor inferior	Valor Superior	Sufijo	Prefijo
Byte	1 Byte	0	255		byt
Integer	2 Bytes	-32.768	32.767	%	int
Long	4 Bytes	-2.147.483.648	2.147.483.647	&	lng

Por **Sufijo** entendemos un **carácter de definición de tipo** que se puede añadir a la Variable ó Constante, para definir el tipo al que pertenece.

Por ejemplo `Dim ValorLong&` declara implícitamente la variable `ValorLong` como **Long**.

Esta declaración equivale a la siguiente

```
Dim ValorLong as Long
```

Si hubiéramos hecho `Dim ValorLong` a secas, la variable `ValorLong` se hubiera declarado como **Variant**.

El uso de estos **caracteres de definición de tipo**, es una herencia del Basic primitivo.

Esos caracteres Sufijo se pueden también aplicar a las declaraciones de constantes

```
Public Const MesesAño As Integer = 12
```

Equivale a `Public Const MesesAño% = 12`

Si se utilizan estos caracteres, no hay que utilizar la declaración explícita de tipo.

Por ejemplo

```
Dim Asistentes% As Integer
```

Darí error ya que sería una declaración redundante porque `Dim Asistentes%` ya ha definido `Asistentes` como `Integer`.

El uso del sufijo es aplicable también a los valores. Esto puede ser muy útil en determinadas circunstancias.

Por ejemplo intenta hacer lo siguiente:

Abre el editor de código, ponte en la ventana Inmediato y escribe lo siguiente:

```
a = 256 * 256 : Print a
```

Un comentario: los dos puntos actúan como separador de las dos sentencias.

Al presionar la tecla **[Intro]** nos aparece un bonito mensaje de error de desbordamiento.

Modifiquemos la línea anterior y pongamos lo siguiente (siempre en la ventana inmediato).

```
a = 256& * 256 : Print a
```

```
Ahora sí nos imprime 65536
```

El error se produce porque VBA deduce que en la primera expresión debe producir como máximo un valor del tipo `Integer` y `65536`, que es su resultado, está por encima del máximo admitido para un `Integer`.

No me preguntéis por qué lo hace, alguien de Microsoft tomó esta decisión en su momento.

En la segunda expresión, al poner uno de los términos como `Long` usando `256&`, hace que la expresión la considere como `Long`, que admite perfectamente el valor resultante.

Para ver que el error no lo produce a, sino la expresión, escribe directamente en la ventana inmediato lo siguiente:

```
Print 256 * 256
```

Se produce exactamente el mismo error.

En cambio si escribimos

```
Print 256 * 256&
```

El error desaparece.

Prefijo

Por prefijo entenderemos el conjunto de letras que “es aconsejable” poner delante del nombre de una variable, para indicarle a la persona que escribe, ó lee, el código, el tipo de dato que contiene una variable.

Aquí voy a tratar de seguir, no de forma absoluta, algunas de las llamadas **“Convenciones de nombres de Leszynski para Microsoft Access”**.

Incluyo un resumen de las normas de Leszynski en el **Apéndice 02**.

También puedes encontrar una descripción sucinta del convenio de Leszynski entre las páginas 19 y 27 de este documento.

<http://quajiros.udea.edu.co/fnsp/Documentos/Direccion/SII/Normalizacion.pdf>

Además incluye interesantes reglas sobre normas para el desarrollo de programas.

También podéis encontrar información en el sitio de McPegasus:

<http://www.iespana.es/mcpegasus/CONTENT/leszynski.htm>

En las páginas del MSDN de Microsoft podéis encontrar información muy interesante sobre la normativa de codificación:

<http://msdn.microsoft.com/library/spa/default.asp?url=/library/SPA/vbcn7/html/vbconProgrammingGuidelinesOverview.asp>

La ventaja de utilizar estos métodos se aprecia inmediatamente. Así, sólo con leer que una tiene como nombre **lngDiasTrabajados** podríamos deducir que es una variable de tipo Long que probablemente sirva para manejar los días trabajados.

Si en una línea de código aparece el nombre **cmdSalir**, podríamos deducir que es un botón de comando que probablemente sirva para salir de un formulario u otro sitio.

Igualmente **lblNombre** es una etiqueta y **txtApellido1** es un cuadro de texto. Ya iremos viendo estas cositas poco a poco.

Repito que los prefijos, **int**, **lng**, **txt**, **lbl**, **cmd**, etc. no modelan el tipo de contenido de la variable, sino que sirven como información adicional para la persona que escribe ó lee el código.

Números de Coma Flotante

Los números de coma flotante son unos tipos de datos que admiten valores no enteros.

Por ejemplo estos son valores de coma flotante:

3.14159265358979 2.71828182845905 2856.1# 4.00!

Como **datos de coma flotante** tenemos los siguientes tipos:

Nombre	Tamaño	Negativos	Positivos	Sufijo	Prefijo
Single	4 Bytes	De -3,402823E38 a -1,401298E-45	De 1,401298E-45 a 3,402823E38	!	sng
Double	8 Bytes	-1.79769313486231E308 -4,94065645841247E-324	4,94065645841247E-324 1,79769313486232E308	#	dbl
Currency	8 Bytes	-922337203685477,5808 a 0	0 a 922337203685477,5807	@	cur
Decimal	12 Bytes	-79.228.162.514.264.337 .593.543.950.335	79.228.162.514.264.337 .593.543.950.335	&	dec

Por **Sufijo** entendemos un **carácter de definición de tipo** que se puede añadir a la Variable

Para el manejo de valores monetarios se suele utilizar el tipo **Currency**.

Este tipo no da errores de redondeo y permite manejar hasta magnitudes de 15 dígitos exactos en su parte entera. En posteriores capítulos haremos mención a algunos errores de redondeo que pueden llegar a dar los tipos de coma flotante.

Tipo Decimal

El tipo **Decimal** es un tipo muy especial.

Permite trabajar con hasta 29 dígitos enteros exactos ó hasta con 28 dígitos exactos en su parte decimal.

Es un tipo con muy poco uso en Access, ya que normalmente no se necesita ese tipo de precisión, y además resulta engorroso su uso.

Algunas de sus peculiaridades se explican en el Apéndice 01.

Tipo Date

El tipo **Date** es un tipo de dato adecuado para manejar datos de tipo Fecha / Hora.

El valor 0 representa las 0 horas del 30 de diciembre de 1899.

La parte entera representa el número de días que han pasado desde esa fecha.

La parte decimal representa el tanto por 1 de día adicional que ha pasado.

Por ejemplo, ahora son las 18 horas 15 minutos 30 segundos del 20 de enero del 2005

Internamente este dato lo guarda como 38372,7607638889.

¿Qué quiere decir?

Han pasado 38372 días desde las 0 horas del 30 de diciembre de 1899

Pero además ha transcurrido 0,7607638889 días, lo que equivale a 18 horas 15 minutos 30 segundos frente a 24 horas

Si pasamos la hora actual a segundos nos da $3600 * 18 + 15 * 30 + 30 \rightarrow 65730$ segundos

Un día completo tiene $3600 \text{ seg/hora} * 24 \text{ horas} = 86400$

Si dividimos 65730 entre 86400 nos da aproximadamente 0,7607638889

Para ver cómo utilizar un tipo date, vamos a crear un procedimiento Sub en un módulo estándar que nos permita ver lo explicado en las líneas anteriores.

En realidad un tipo **Date** es un tipo **Double** y como él utiliza 8 Bytes.

Por cierto, el prefijo para indicar que es tipo **Date** es **dat**.

Su código será el siguiente

```
Public Sub PruebaTipoDate()  
    Dim datFechaActual As Date  
    Dim datSemanaProxima As Date  
    Dim datAyer As Date  
    Dim datMañana As Date  
  
    datFechaActual = #1/20/2005 6:15:30 PM#  
    datSemanaProxima = datFechaActual + 7  
    datAyer = datFechaActual - 1  
    datMañana = datFechaActual + 1  
  
    Debug.Print "Tipo Doble", CDBl(datFechaActual)  
    Debug.Print "Ahora", datFechaActual  
    Debug.Print "Próx. Semana", datSemanaProxima  
    Debug.Print "Ayer", datAyer  
    Debug.Print "Mañana", datMañana  
End Sub
```

Si desde la ventana inmediato escribimos

PruebaTipoDate y presionamos [**Enter**] nos mostrará:

```
Tipo Doble      38372,7607638889  
Ahora          20/01/2005 18:15:30
```

Próx. Semana	27/01/2005	18:15:30
Ayer	19/01/2005	18:15:30
Mañana	21/01/2005	18:15:30

Puntualizaciones:

Ya os habréis dado cuenta que VBA utiliza el Inglés como lenguaje, y más concretamente el Inglés americano. Esto hace que los datos usen ese mismo modelo.

Por ejemplo, la coma flotante nuestra, ellos la cambian por el punto.

Algo así pasa con las fechas. El formato americano de fecha usa el siguiente orden:

Mes / Día / Año

Por eso en la línea

```
datFechaActual = #1/20/2005 6:15:30 PM#
```

se pone 1/20/2005, en vez de 20/1/2005 como hacemos nosotros.

Esto es importante recordarlo ya que nos evitará futuros errores.

Además VBA tiene un comportamiento que puede despistar.

Si hubiéramos hecho

```
datFechaActual = #20/1/2005 6:15:30 PM#
```

Hubiera funcionado exactamente igual, ya que VBA deduciría que el 20 no puede referirse al mes, pues existen sólo 12 meses, por lo que deduce “por su cuenta” que lo que en realidad queríamos era introducir el 20 como día y el 1 como mes.

Estos comportamientos que a veces tiene VBA de tomar decisiones por su cuenta es algo “que me saca de quicio” (nada ni nadie es perfecto).

Resumiendo

La forma de construir una fecha / hora es

Mes / Día / Año Horas:Minutos:Segundos

Dependiendo de la configuración de Windows que tengas, si por ejemplo introduces

```
datFechaActual = #1/20/2005 18:15:30#
```

VBA te lo puede convertir automáticamente a

```
datFechaActual = #1/20/2005 6:15:30 PM#
```

añadiéndole por su cuenta el PM

Y no sólo toma decisiones de ese tipo.

Si escribes

```
datFechaActual = #1-20-2005#
```

VBA te cambia automáticamente a

```
datFechaActual = #1/20/2005#
```

Para construir una fecha no es necesario introducirle la hora.

Tampoco es necesario introducirle el año entero.

Estas fechas son perfectamente válidas

```
#1/20/2005 18:15:30#  
#1/20/2005 6:15:30 PM#  
#1/20/2005#  
#1/20/05#  
#01/20/05#  
#18:15:30#
```

Puedes obtener más información en la ayuda de Access.

Hay toda una batería de funciones de VBA para el manejo y manipulación de fechas.

Estas funciones las veremos más adelante.

Tipo Boolean (booleano)

El tipo **Boolean** (booleano) es un tipo de dato adecuado para manejar datos de tipo **Sí / No**, **True / False**.

Sorprendentemente el tipo **Boolean** ocupa 2 Bytes, supongo que por compatibilidad con otros tipos.

Es como si estuviera definido como de tipo **Integer**, y como veremos, algo de esto hay.

El prefijo para un booleano es **bln**.

A una variable Booleana se le puede asignar un valor de varias formas

Por ejemplo esto sería correcto:

```
Public Sub PruebaTipoBooleano()  
    Dim blnDatoBooleano As Boolean  
  
    blnDatoBooleano = True  
    Debug.Print "Valor True " & blnDatoBooleano  
    blnDatoBooleano = False  
    Debug.Print "Valor False " & blnDatoBooleano  
    blnDatoBooleano = -1  
    Debug.Print "Valor -1 " & blnDatoBooleano  
    blnDatoBooleano = 0  
    Debug.Print "Valor 0 " & blnDatoBooleano  
    blnDatoBooleano = 4  
    Debug.Print "Valor 4 " & blnDatoBooleano  
    Debug.Print "Valor entero de Verdadero " & CInt (blnDatoBooleano)  
End Sub
```

Este código nos mostrará en la ventana Inmediato

```
Valor True Verdadero  
Valor False Falso  
Valor -1 Verdadero  
Valor 0 Falso  
Valor 4 Verdadero
```

Valor entero de Verdadero -1

Nota: La instrucción `CInt (blnDatoBoleano)` convierte a un valor Entero el contenido de la variable `blnDatoBoleano`.

Para asignar a una variable booleana el valor **falso** le podemos asignar `False` ó `0`.

Para asignarle cierto lo podemos hacer pasándole directamente `True` ó **cualquier valor numérico diferente de Cero**, incluso aunque sea de coma flotante.

Si convertimos el contenido de una variable Boleana que contenga `True` a un valor numérico, nos devolverá `-1`.

Otra forma de asignarle un valor sería asignarle una expresión que devuelva `True` ó `False`.

Por ejemplo

```
blnDatoBoleano = 8<7
```

Asignaría `False` a `blnDatoBoleano` (es falso que 8 sea menor que 7).

```
blnDatoBoleano = 8>7
```

Asignaría `True` a `blnDatoBoleano`.

Tipo String (cadena)

El tipo `String` está especializado en almacenar datos de tipo Cadena (Texto).

El carácter para definir el tipo es el signo de Dólar `$`.

Hay dos tipos String

- String de longitud variable
- String de longitud fija

Para asignar una cadena de **longitud variable**, la declaración se hace de forma similar al resto de los tipos

```
Dim strNombre As String
Dim strNombre$
Public strNombre As String
Private strNombre As String
```

Para declarar una variable como de longitud fija se utiliza el asterisco (*) y a continuación el número de caracteres que va a contener

```
Dim strCuentaCorriente As String * 20
```

Para declarar cadenas de longitud fija no se puede usar el carácter `$`.

Por ejemplo, esto daría error

```
Dim strCuentaCorriente$ * 20
```

Una cadena de longitud variable podría en teoría almacenar hasta 2^{31} caracteres, es decir unos 2000 millones de caracteres. Lógicamente siempre que el PC tuviera suficiente memoria.

Una cadena de longitud fija puede contener hasta 65536 caracteres (64 KBytes).

Si por ejemplo hacemos:

```
Dim strCliente As String * 40
strCliente = "Pepe Gotera"
```

strCliente contendrá la cadena **Pepe Gotera** seguida de 29 **espacios en blanco** (para completar los 40 que debe contener la variable).

Tipo Variant

El tipo **Variant** es un tipo de dato que puede contener prácticamente cualquier tipo de datos. El prefijo que se suele utilizar para identificar una variable **Variant** es **var**.

Hay excepciones, por ejemplo **no puede contener una cadena de longitud fija**.

Cuando declaramos una variable o escribimos la cabecera de una función, sin especificar el tipo que va a contener ó devolver, implícitamente las estamos declarando como de tipo Variant.

Si declaramos una constante, sin especificar su tipo, el tipo que tome lo hará en función del dato que le asignemos en la propia declaración.

Por ejemplo si declaramos en la cabecera de un módulo estándar

```
Global Const conEmpresa = "Manufacturas ACME s.l."
```

*Hace que la constante conEmpresa se configure como del tipo **String**.*

*He usado aquí el modificador de alcance **Global**. Este modificador del Alcance de la variable sería, en este caso, equivalente a haber utilizado **Public**.*

Las declaraciones de constante siguientes serían equivalentes a la anterior

```
Public Const conEmpresa = "ACME s.l."
```

```
Public Const conEmpresa As String = "ACME s.l."
```

```
Public Const conEmpresa$ = "ACME s.l."
```

La forma de declarar una variable **explícitamente** como Variant es la siguiente.

```
ModificadorDeAlcance Nombre As Variant
```

```
Dim varMiVariable As Variant
```

Ya hemos dicho que si se declara una variable sin especificar su tipo, implícitamente la declara como Variant.

La última declaración sería equivalente a

```
Dim varMiVariable
```

Empty, Null, Cadena vacía

Existen unos valores especiales que no son ni números ni texto.

Por ejemplo **Empty** (vacío) es valor que toma por defecto el tipo Variant después de ser declarado y antes de que se le asigne otro valor.

Para ver si una variable de tipo Variant ha sido inicializada, podemos utilizar la función **IsEmpty**(variable) que devolverá **True** ó **False**.

Para comprobarlo, vamos a crear un procedimiento que utilice la función **IsEmpty** sobre una variable, antes y después de asignarle un valor; en este caso **Null**.

```
Public Sub PruebaEmpty()  
    Dim varVariant As Variant  
    Debug.Print "Variable vacía = "; IsEmpty(varVariant)  
    varVariant = Null  
    Debug.Print "Variable vacía = "; IsEmpty(varVariant)  
    Debug.Print "Valor actual = "; varVariant  
End Sub
```

Si llamamos a este procedimiento desde la ventana Inmediato, nos imprimirá en ella lo siguiente:

```
Variable vacía = Verdadero  
Variable vacía = Falso  
Valor actual = Nulo
```

La primera vez que se ejecuta `IsEmpty(varVariant)` devuelve `True` porque todavía no se le ha asignado ningún valor. Tras hacer `varVariant = Null` devuelve `False`.

`Null` es un dato nulo. Equivale al código ASCII 0.

Por ejemplo, un cuadro de texto que no contenga ningún valor, devolverá el valor `Null`.

El valor `Null` sólo se puede asignar a una variable del tipo `Variant`, o a una variable del tipo `Objeto`; tipo de variable que veremos en otra entrega.

Una Cadena Vacía es un dato del tipo `String` que no contiene ningún carácter.

La cadena vacía se puede asignar a una variable del tipo `Variant` y a una variable de tipo `String`.

Para asignar la cadena vacía a una variable se escribe la variable seguida del operador de asignación `=` y a continuación 2 comillas dobles.

```
strCadenaVacía = ""
```

Declaraciones múltiples en una línea.

VBA permite declarar múltiples variables en una misma línea de código.

La forma correcta de hacerlo sigue esta estructura:

```
Alcance Variable1 As Tipo, Variable2 As Tipo, Variable3 As Tipo
```

Por ejemplo, esto sería correcto

```
Dim lngUnidades As Long, strNombre As String
```

Otros lenguajes de programación permiten hacer una declaración del siguiente estilo:

```
Dim strNombre, strApellido1, strApellido2 As String
```

Tras esto las tres variables serían del tipo `String`.

En cambio en VBA, si hubiéramos declarado así las variables, `strNombre` y `strApellido1` serían del tipo `Variant`, y `strApellido2` sería del tipo `String`.

Valores por defecto

Cuando declaramos una variable, ésta se inicializa con un valor por defecto.

Dependiendo del tipo de dato de la variable el valor que tome por defecto será diferente.

Las variables de tipo **numérico** toman el valor 0

Las de tipo **fecha** también el valor 0, que se corresponde a las 0 horas del 30 de diciembre de 1899.

Las de tipo **booleano** el valor **False**, ó 0.

Las de tipo **cadena** la cadena vacía "".

Las de tipo **variant**, el valor **Empty**.

Ámbito ó alcance de una declaración

Hemos visto varias instrucciones que afectan al alcance ó visibilidad de una Variable, Constante o Procedimiento.

Estas instrucciones son:

Dim

Public

Global

Private

Hay otras formas de declarar Variables como son las instrucciones:

Static

Friend

Dim

Si la instrucción **Dim** se utiliza para declarar una variable en la cabecera de un módulo, esa variable sólo será "**visible**" por los procedimientos que estén dentro de ese módulo.

De forma semejante, si la instrucción **Dim** se utiliza dentro de un procedimiento, esa variable sólo podrá ser vista por el código del interior del procedimiento.

Private

La instrucción **Private** se suele utilizar para definir de forma explícita que una constante, variable o procedimiento sólo van a ser visibles desde dentro de un módulo.

Se suele utilizar para la declaración de variables y constantes en las cabeceras de módulos, así como para definir el alcance de los procedimientos de ese módulo.

Dentro de un procedimiento se usa **Dim** en vez de **Private**.

Public

La instrucción **Public** se suele utilizar para definir que una constante, variable o procedimiento van a ser visibles desde cualquier parte de la aplicación.

Se suele utilizar para la declaración de variables y constantes en las cabeceras de módulos, así como para definir el alcance de los procedimientos de ese módulo.

Por ello no es factible declarar una variable como **Public** dentro de un procedimiento.

Sí se puede en cambio, declararla en la cabecera de cualquier módulo **estándar** o **de clase**.

Nota: Una aclaración respecto a los elementos declarados en un módulo de clase.

Si declaramos una variable, constante ó procedimiento como **Public** en un módulo de clase, por ejemplo en un formulario, para poder usarlo hay que hacerlo a través de una **Instancia** de ese módulo de clase.

A las variables y constantes públicas de un módulo de clase, así como a los procedimientos Property, se les llama **Propiedades** de la clase.

Al resto de los procedimientos de la clase que sean públicos se les llama **Métodos** de la clase.

Cuando veamos las clases, veremos que una **instancia** es un **ejemplar** de una clase.

La clase como tal es el propio código, pero una instancia es un objeto creado mediante la clase. Se que es un poco críptico, pero más adelante lo entenderéis sin problema.

Por ejemplo una instancia de un formulario es el propio formulario, o si tenemos la clase Persona y creamos una persona llamada Pepe, la instancia de la clase es Pepe.

Si hemos declarado una variable, por ejemplo **Nombre** como **Public** para poder usarla deberemos hacer referencia a ella mediante la instancia de la clase, que es Pepe

```
Pepe.Nombre
```

Lo mismo ocurriría si hemos declarado como **public** la función **Edad**, es decir el método **Edad**.

```
Pepe.Edad
```

Así mismo, si en el formulario **Pedidos** declaramos **datFechaUltimoPedido** como **Public**, para acceder a ese dato lo tenemos que hacer a través del propio formulario.

```
Pedidos.datFechaUltimoPedido
```

Desde el código del formulario, no es necesario hacer referencia al mismo formulario, aunque sí puede hacerse utilizando la palabra clave **Me**, que es una variable declarada implícitamente y que representa a la instancia de la clase u objeto creado mediante la clase.

Desde el propio formulario sería lo mismo

```
Me.datFechaUltimoPedido  
    que  
    datFechaUltimoPedido
```

No tenéis por qué entender todo esto todavía. Lo iré comentando poco a poco y lo desarrollaré en la entrega en la que hablemos de las **Clases**.

Global

La instrucción **Global** se utiliza de forma semejante a **Public**, dentro de módulos estándar.

Yo personalmente no la suelo usar; prefiero utilizar la instrucción **Public**.

Comencemos a programar con
VBA - Access

Entrega **06**

Estructuras de datos

Matrices ó Arrays

Una matriz en VBA es un conjunto de variables del mismo tipo, a las que se puede acceder mediante un índice, que indica su posición en ella.

Imaginemos que queremos almacenar en el código, para su posterior utilización, el número de días de cada mes del año.

Por ejemplo, podemos hacer esto

```
Public Mes01 As Integer, Mes02 As Integer
Public Mes03 As Integer, Mes04 As Integer
Public Mes05 As Integer, Mes06 As Integer
Public Mes07 As Integer, Mes08 As Integer
Public Mes09 As Integer, Mes10 As Integer
Public Mes11 As Integer, Mes12 As Integer
```

```
Public Sub CargarMeses ()
    Mes01 = 31
    Mes02 = 28
    Mes03 = 31
    Mes04 = 30
    Mes05 = 31
    Mes06 = 30
    Mes07 = 31
    Mes08 = 31
    Mes09 = 30
    Mes10 = 31
    Mes11 = 30
    Mes12 = 31
End Sub
```

Frente a esta declaración de variables, un tanto engorrosa, vamos a considerar estas alternativas:

```
Public Mes(12) As integer
```

```
Public Sub CargarMeses ()
    Mes(1) = 31
    Mes(2) = 28
    Mes(3) = 31
    Mes(4) = 30
    Mes(5) = 31
    Mes(6) = 30
    Mes(7) = 31
    Mes(8) = 31
    Mes(9) = 30
```

```
Mes (10) = 31
```

```
Mes (11) = 30
```

```
Mes (12) = 31
```

```
End Sub
```

Para asignar los valores a los elementos de la matriz, ejecutaremos el procedimiento `CargarMeses`

La tercera forma aún resulta más directa

```
Dim Mes () As Variant
```

```
Public Sub CargarMeses ()
```

```
    Mes = Array(0, 31, 28, 31, 30, 31, 30, _
               31, 31, 30, 31, 30, 31)
```

```
End Sub
```

A veces, por facilidad de lectura del código, interesa distribuir la escritura de una única línea de código, en varias líneas físicas. Para ello se pone al final de la línea un espacio en blanco seguido de la barra baja, como se ve en el ejemplo anterior.

Para acceder a los días de un mes, por ejemplo Julio, en el ejemplo primero tenemos que utilizar directamente la variable

```
Mes07
```

Para hacer lo mismo en los ejemplos 2º y 3º, tenemos que tomar en cuenta que la variable `Mes` contiene los doce valores de los días del mes.

```
Mes (7)
```

Este método es mucho más práctico y da muchas más posibilidades para usar estructuras de bucle, como veremos más adelante.

Pero, ¿por qué he puesto `Mes = Array(0, 31, . . .)?`

Por defecto las matrices, si no se indica el rango de sus índices, empiezan con el índice 0.

Lo de añadir un valor más al principio como valor de `Mes`, en este caso 0 aunque podría haber puesto cualquier otro valor, es para que haya una coherencia entre los casos 2º y 3º, y que por ejemplo `Mes (7)` sea el valor de **Julio** en las dos matrices.

La declaración `Dim Mes (12) As integer` genera trece variables, accesibles desde el índice 0 `Mes (0)` al índice 12 `Mes (12)`.

Una segunda puntualización:

La línea de código

```
Mes = Array(0, 31, 28, 31, 30, 31, 30, _
           31, 31, 30, 31, 30, 31)
```

hace que, `Mes (i)`, para cualquiera de los valores de `i`, sea del tipo `integer`.

Si quisiéramos que `Mes (i)` devolviera un tipo `Long`, deberíamos poner el sufijo de declaración de tipo `Long` "&" detrás de cada número:

```
Mes = Array(0&, 31&, 28&, 31&, 30&, 31&, 30&, _
           31&, 31&, 30&, 31&, 30&, 31&)
```

Si quisiéramos que las matrices por defecto comenzaran con el índice 1, deberíamos escribir en uno de los módulos, antes que cualquier procedimiento o declaración de matriz, la instrucción **Option Base 1**

Si hubiéramos escrito en la cabecera del módulo

```
Option Base 1
```

Y a continuación `Dim Mes (12) As integer` se puede acceder a la variable `Mes` mediante índices que van del **1** al **12**.

Existe la posibilidad de incluir en la declaración el rango de índices que va a manejar una matriz. La forma de hacerlo es

```
Delimitador NombreDeMatriz (IndiceInferior To IndiceSuperior)
```

En nuestro caso podríamos haber hecho

```
Public Mes (1 to 12) As integer
```

En los casos que hemos visto hasta ahora, estamos utilizando Matrices unidimensionales, cuyos elementos son accesibles mediante 1 único índice.

Son del tipo `Matriz (Indice)`

Matrices de varias dimensiones

Podemos declarar matrices de varias dimensiones; es decir que utilicen más de un índice.

Supongamos que nos encargan hacer un programa, en uno de cuyos puntos tenemos que controlar el número de personas que viven en un conjunto de bloques de vivienda numerados del 1 al 20. Cada bloque tiene 4 plantas que van de la planta baja (0) a la planta 3. Cada planta tiene 4 viviendas numeradas de la 1 a la 4.

Podríamos hacer una declaración de este tipo:

```
Dim Vivienda (1 To 20, 0 To 3, 1 To 4) As Integer
```

Si en la vivienda nº 2 de la planta baja del bloque 13 vivieran 3 personas, para asignar este valor a la variable sería así.

```
Vivienda (13, 0, 2) = 3
```

Si después en algún punto del código hacemos

```
intPersonas = Vivienda (13, 0, 2) = 3
```

la variable `intPersonas` contendrá el valor 3.

Matrices dinámicas

Supongamos que estamos haciendo un programa de ventas y que uno de sus condiciones es poder tener controladas en memoria, una vez seleccionado un tipo de producto, las referencias que existen del mismo.

Nos podremos encontrar tipos de producto con 1 referencia, otros con 4 ó con cualquier número de ellas.

A priori desconocemos el número de referencias que puede tener un tipo de producto, es más su número puede cambiar con el tiempo.

Para solucionar este tipo de problemas existen las llamadas Matrices Dinámicas.

Una matriz dinámica debe ser declarada, a nivel de módulo o de procedimiento, sin índices.

En nuestro caso se haría

```
Dim Referencias () As String
```

Más adelante, en un procedimiento, podríamos asignarle el número de elementos.

Si vamos a trabajar con un tipo de producto que tuviera 8 referencias, podremos redimensionar la matriz mediante la instrucción **Redim**.

```
ReDim Referencias(1 to 8)
```

Supongamos que posteriormente cambiamos de tipo de producto y pasamos a uno con sólo 2 referencias. En el código haremos

```
ReDim Referencias(1 to 2)
```

Tras redimensionar una matriz con **Redim**, los valores que contenía la matriz se **reinician**, tomando el valor por defecto del tipo de dato declarado.

En el caso de las cadenas el valor por defecto es la cadena vacía "", en el de los números el valor es 0 y en el de los Variant el valor **Empty**.

Si por necesidades de programación deseáramos conservar los valores que tenía una matriz dinámica antes de su redimensionado, hay que utilizar la instrucción **Preserve** entre **Redim** y el nombre de la variable matriz.

Para ver esto vamos a analizar este código:

```
Public Sub PruebaRedim()  
    Dim n As Long  
    Dim Referencias() As String  
  
    Debug.Print  
    n = 5  
    ReDim Referencias(1 To n)  
    Referencias(5) = "Referencia 05"  
    Debug.Print Referencias(5)  
  
    n = 8  
    ReDim Referencias(1 To n)  
    Debug.Print  
    Debug.Print "Tras Redim"  
    Debug.Print "Los datos se han borrado"  
    Debug.Print """" & Referencias(5) & """"  
    Debug.Print  
    Referencias(5) = "Referencia 05"  
    Referencias(8) = "Referencia 08"  
    Debug.Print "Los datos se han cargado"  
    Debug.Print """" & Referencias(5) & """"  
    Debug.Print """" & Referencias(8) & """"  
    Debug.Print  
    n = 10  
    ReDim Preserve Referencias(1 To n)  
    Debug.Print "Tras Redim con Preserve"  
    Debug.Print "los datos se han conservado"  
    Debug.Print """" & Referencias(5) & """"
```

```
    Debug.Print """" & Referencias(8) & """"  
End Sub
```

Antes de seguir, voy a hacer unas aclaraciones.

*La línea de código `Debug.Print` realiza solamente un salto de línea en la ventana **Inmediato**.*

Las cuatro comillas seguidas equivalen a una comilla en modo texto. Es decir

```
    Debug.Print """" & "MiTexto" & """"
```

*hace que se imprima "MiTexto" en la ventana **Inmediato**.*

En el caso del ejemplo: "Referencia 05".

Si ejecutamos el procedimiento `PruebaRedim`, nos imprimirá en la ventana **Inmediato**:

```
Referencia 05
```

```
Tras Redim  
Los datos se han borrado  
""
```

```
Los datos se han cargado  
"Referencia 05"  
"Referencia 08"
```

```
Tras Redim con Preserve  
los datos se han conservado  
"Referencia 05"  
"Referencia 08"
```

Tras las líneas

```
n = 5  
ReDim Referencias(1 To n)
```

Redimensiona la matriz `Referencias` como si hubiéramos hecho

```
ReDim Referencias(1 To 5)
```

Posteriormente asigna un valor al elemento de la matriz de índice 5 y lo imprime.

Lo siguiente que hace es redimensionar la matriz a 8 elementos.

Tras ello el elemento 5º de la matriz ha desaparecido

Vuelve a asignar valores, en este caso a los elementos 5º y 8º y los imprime.

Redimensiona otra vez la matriz, esta vez con `preserve`, y se comprueba que no han desaparecido los valores anteriores.

Nota:

Lógicamente, aunque usemos `Preserve`, si redimensionamos una matriz a un número menor de elementos que la matriz anterior, los elementos superiores al nuevo índice máximo desaparecerán.

Instrucción Erase

Si tenemos declarada una matriz dinámica, VBA reserva una zona de memoria para guardar sus datos.

Si quisiéramos dejar libre, de forma explícita esa memoria una vez utilizada esa matriz, podemos usar la instrucción **Erase**.

Si consultamos la ayuda de VBA indica que **Erase** Vuelve a inicializar los elementos de matrices de tamaño fijo y libera el espacio de almacenamiento asignado a matrices dinámicas.

Esto quiere decir que si tenemos declarada una matriz de tamaño fijo por ejemplo:

```
Dim MatrizFija(10) As String
Dim MatrizFija2(10) As Long
```

La instrucción

```
Erase MatrizFija, MatrizFija2
```

No liberará la memoria ocupada, sólo reinicializará la matriz `MatrizFija` a cadenas vacías y la matriz `MatrizFija2` a ceros.

En cambio si tenemos `ReDim Referencias(1 To 5)`

`Erase Referencias`, libera la memoria ocupada por la matriz `Referencias`.

Redim con varias dimensiones

Supongamos ahora que además del caso de las viviendas del ejemplo tuviéramos que controlar otro grupo de viviendas compuesto de 4 bloques de 8 plantas (1 a 8) y con 6 puertas por plantas (1 a 6).

Para ello podríamos haber declarado inicialmente la matriz como

```
Dim Vivienda() As Integer
```

Posteriormente cuando vallamos a utilizarla con el primer conjunto de bloques de viviendas haríamos

```
Redim Vivienda(1 To 20, 0 To 3, 1 To 4)
```

Cuando tengamos que utilizarla con la segunda urbanización

```
Redim Vivienda(1 To 4, 1 To 8, 1 To 6)
```

Índices superior e inferior de una matriz.

En un punto del código nos puede ocurrir que necesitemos saber qué índices tiene como máximo y mínimo una matriz, ya sea ó no dinámica.

Para ello tenemos las funciones **UBound** y **LBound**.

UBound devuelve el índice máximo y **LBound** devuelve el índice mínimo.

Para probar estas funciones vamos a hacer lo siguiente

```
Public Sub PruebaIndicesMaxYMin()
    Dim Matriz(1 To 8) As Long
    Dim MatrizDinamica() As String
    Dim MultiDimensional(1 To 4, 1 To 8, 10) As Long
```



```

ReDim MatrizDinamica(-2 To 4) As integer

Debug.Print "Valor mínimo de Matriz()"
Debug.Print LBound(Matriz)
Debug.Print "Valor máximo de Matriz()"
Debug.Print UBound(Matriz)
Debug.Print "Número de elementos"
Debug.Print
Debug.Print UBound(MatrizDinamica) - LBound(Matriz) + 1
Debug.Print "Valor mínimo de MatrizDinamica()"
Debug.Print LBound(MatrizDinamica)
Debug.Print "Valor máximo de MatrizDinamica()"
Debug.Print UBound(MatrizDinamica)
Debug.Print "Número de elementos"
Debug.Print UBound(MatrizDinamica) - LBound(Matriz) + 1
Debug.Print
Debug.Print "Valor máximo índice 1° MultiDimensional()"
Debug.Print UBound(MultiDimensional, 1)
Debug.Print "Valor mínimo índice 2° MultiDimensional()"
Debug.Print LBound(MultiDimensional, 2)
Debug.Print "Valor mínimo índice 2° MultiDimensional()"
Debug.Print LBound(MultiDimensional, 3)
End Sub

```

Para obtener el valor máximo ó mínimo de los índices en una matriz de varias dimensiones, como se puede ver en el código, hay que hacer lo siguiente

```

UBound(NombreDeLaMatriz, N°DeIndice)
LBound(NombreDeLaMatriz, N°DeIndice)

```

Registros (Estructuras de variables definidas por el usuario)

Un registro ó estructura definida por el usuario está compuesta por un conjunto de datos, del mismo ó diferente tipo, que están relacionadas.

Supongamos que en un programa quisiéramos controlar los datos de diferentes personas.

Los datos a controlar son **Nombre**, **Apellido1**, **Apellido2**, **Fecha de nacimiento** y **Teléfono**.

Podríamos definir 5 variables, por ejemplo de esta forma:

```

Public strNombre As String
Public strApellido1 As String
Public strApellido2 As String
Public datNacimiento As Date
Public strTelefono As String

```

Incluso si tuviéramos que manejar varias personas simultáneamente podríamos crear esas variables como matrices dinámicas.

Pero ¿no sería una ventaja agrupar todos los datos en una misma variable?

Supongamos que lo pudiéramos hacer, y que esa variable se llamara `Amigo`.

Sería interesante que

`Amigo.Nombre` nos devolviera el nombre, ó que

`Amigo.Apellido1` nos devolviera su apellido.

Esto puede hacerse mediante la siguiente estructura:

```
Public Type Persona
    Nombre As String
    Apellido1 As String
    Apellido2 As String
    FechaNacimiento As Date
    Telefono As String
End Type
```

Tras esto podríamos hacer

```
Public Sub PruebaRegistro()
    Dim Cliente As Persona
    Dim Vendedor As Persona
    Cliente.Nombre = "Antonio"
    Cliente.Apellido1 = "Vargas"
    Cliente.Apellido2 = "Giménez"
    Cliente.Telefono = "979111111"
    Debug.Print
    Debug.Print Cliente.Nombre
    Debug.Print Cliente.Apellido1
    Debug.Print Cliente.Apellido2
    Debug.Print Cliente.Telefono
    Debug.Print
    'Ahora usando With
    With Vendedor
        .Nombre = "Pedro"
        .Apellido1 = "Jaizquíbel"
        .Apellido2 = "Gorráiz"
        .Telefono = "979222222"

        Debug.Print.Nombre _
            & " "; .Apellido1 _
            & " "; .Apellido2
        Debug.Print "Teléfono " _
            & .Telefono
    End With
End Sub
```

Si ejecutamos este procedimiento, nos mostrará lo siguiente en la ventana Inmediato:

```
Antonio
Vargas
Giménez
979111111
```

```
Pedro Jaizquíbel Gorráiz
Teléfono 979222222
```

Para crear la estructura `Persona`, he utilizado el par de sentencias

```
Type
    - - -
    - - -
End Type
```

Entre estas dos sentencias se declaran las variables que van a actuar como campos, ó miembros de ese registro, con el tipo al que pertenecen.

La estructura de tipo Registro puede ser declarada como `Public` ó `Private` en la cabecera de los módulos nunca dentro de un procedimiento.

Dentro de un módulo de clase ya sea propio, o asociado a un formulario ó informe, tanto las estructuras como las variables que las van a manejar sólo pueden declararse como de tipo `Private`.

Por ejemplo si hemos declarado en un módulo la estructura `Persona`, no puedo declarar una variable pública, dentro de un formulario ó informe, del tipo `Persona`

```
Public Comprador As Persona (daría error)
```

Pero sí podría hacer

```
Private Comprador As Persona
```

La instrucción `With`

Si observamos el código del procedimiento `PruebaRegistro` de la página anterior, vemos que, tanto para asignar valores a los atributos de la variable `Vendedor`, he utilizado la instrucción `With`, acompañada de su complemento `End With`.

Esto nos permite evitarnos tener que repetir de reiterativamente el nombre `Vendedor` en cada línea de asignación ó lectura.

Entre `With Vendedor` y `End With` al escribir el punto se supone que estamos utilizando atributos del registro `Vendedor`.

Este tipo de sintaxis ayudados por la instrucción `With`, se puede utilizar no sólo con Variables `Registro`, sino también con todo tipo de objetos, como veremos posteriormente.

Matrices de Registros

Además de los tipos de datos estudiados también podemos declarar matrices de Estructuras tipo Registro, incluso Matrices Dinámicas.

Por ejemplo podríamos declarar en la cabecera de un formulario lo siguiente:

```
Private Trabajadores() As Persona
```

Y en alguno de los procedimientos del formulario podríamos reinicializarla; por ejemplo en el evento **Al cargar**.

```
Private Sub Form_Load()  
    ReDim Trabajadores(100)  
End Sub
```

A partir de este momento podríamos cargar y leer datos en la matriz desde cualquiera de los procedimientos del formulario.

Para ello deberemos pasarle el índice de la matriz a la variable Trabajadores, y el nombre del atributo.

```
Trabajadores(1).Nombre = "Pepe"  
Trabajadores(1).Apellido1 = "Gotera"  
Trabajadores(2).Nombre = "Otilio"
```

E incluso podríamos utilizar la instrucción **With**

```
With Trabajadores(1)  
    Debug.Print .Nombre  
    Debug.Print .Apellido1  
End With
```

En el siguiente capítulo veremos unos Objetos muy utilizados internamente por Access.

Son las **Colecciones**.

Estas estructuras del tipo **Collection** aparecen en muchos elementos de **Access**:

- Formularios
- Informes
- Controles
- ADO
- DAO
- Etc...

Comencemos a programar con
VBA - Access

Entrega **07**

Colecciones y Objetos

Introducción a las Colecciones

Una colección, siguiendo la definición de la ayuda de Access, es un conjunto ordenado de elementos, a los que se puede hacer referencia como una unidad.

El objeto **Colección**, recibe el nombre de **Collection**.

Una colección se parece a una matriz, en el sentido de que podemos acceder a sus elementos mediante un índice, pero aquí acaban prácticamente sus semejanzas.

Algunas de las ventajas de las colecciones frente a las matrices son:

- Utilizan menos memoria que las matrices. Sólo la que necesitan en cada caso.
- El acceso a los elementos de la colección es más flexible. Como veremos hay varios métodos para hacerlo. Eso no significa que sea más rápido.
- Las colecciones tienen métodos para añadir nuevos elementos y eliminar los que ya contiene.
- El tamaño de la colección se ajusta de forma automática tras añadir ó quitar algún miembro.
- Pueden contener simultáneamente elementos de diferentes tipos

Son como un saco en los que podemos meter "casi todo". Una de las pocas cosas que no se pueden añadir a una colección son las estructuras **Registro** definidas por el usuario, que vimos en la entrega anterior.

Para poder meter en una matriz simultáneamente diferentes tipos de datos, habría que declararla como **Variant**, o lo que es lo mismo, no especificar su tipo en el momento de declararla.

En la declaración de una variable del tipo **Collection**, no hay que declarar ni el número de elementos que va a contener, ni su tipo.

Aunque más adelante hablaremos de los Objetos, para declarar una colección, como objeto que es, hay que dar 2 pasos.

1. Se declara una variable como del tipo **Collection**.
2. Se crea una **instancia** del objeto **Collection** utilizando la palabra clave **New** y la palabra clave **Set** que sirve para la Asignación de objetos.
3. Hay una alternativa que es declararla y realizar la instancia en un único paso, que sería:


```
Public NombreDeLaColección As New Collection
```

Para **añadir elementos** a una colección, se utiliza el método **Add**.

La forma es `NombreDeLaColección.Add Elemento`

Podemos añadir antes ó después de un elemento dado.

La forma es `NombreDeLaColección.Add Elemento, NumeroAntes`

`NombreDeLaColección.Add Elemento,, NumeroDespués`

Podemos incluso definir una palabra clave para acceder después a un elemento dado de la colección.

`NombreDeLaColección.Add Elemento, "Clave"`

Si utilizamos una clave, podemos también colocarlo antes ó después de un elemento dado.

Ejemplo:

Supongamos que tenemos la colección **Productos**.

Para añadir elemento "Llave fija de de 7-8 mm." podemos hacer:

```
Productos.Add "Llave fija de de 7-8 mm."
```

Si ahora queremos añadir "Destornillador Philips de 9 mm." con la clave "DestPhi009"

```
Productos.Add "Destornillador Philips de 9 mm.", "DestPhi009"
```

Vamos ahora a añadir un nuevo producto en la posición Nº 2

```
Productos.Add "Martillo Carpintero 4", "MrtCrp004", 2
```

Vamos ahora a añadir un nuevo producto después de la posición Nº 2

```
Productos.Add "Martillo Carpintero 6", "MrtCrp006", , 2
```

Para saber cuántos elementos hay en una colección tenemos el método **Count**.

```
lngElementos = Productos.Count
```

Podemos obtener un elemento de la colección mediante su índice ó su Clave.

Para eliminar un elemento de una colección se utiliza el método **Remove**, indicando su índice ó su clave.

```
Productos.Remove (3)  
Productos.Remove ("DestPhi009")
```

Vamos ahora a probar todo.

Escribimos en un módulo estándar lo siguiente

```
Public Sub PruebaColeccion1()  
    Dim Productos As Collection  
  
    Set Productos = New Collection  
  
    Productos.Add "Llave fija de de 7-8 mm."  
    Productos.Add "Destornillador Philips de 9 mm.", "DestPhi009"  
    Productos.Add "Martillo Carpintero 4", "MrtCrp004", 2  
    Productos.Add "Martillo Carpintero 6", "MrtCrp006", , 2  
  
    Debug.Print Productos.Count & " elementos"  
    Debug.Print Productos(1)  
    Debug.Print Productos(2)  
    Debug.Print Productos("MrtCrp006")  
    Debug.Print Productos("DestPhi009")  
  
    Productos.Remove (3)  
    Productos.Remove ("DestPhi009")  
  
    Debug.Print Productos.Count & " elementos"  
End Sub
```

Al ejecutarlo nos mostrará en la ventana **Inmediato**:

```
4 elementos
Llave fija de de 7-8 mm.
Martillo Carpintero 4
Martillo Carpintero 6
Destornillador Philips de 9 mm.
2 elementos
```

Si utilizamos la opción de añadir un elemento antes ó después de uno dado, deberíamos cerciorarnos de que ese elemento existe.

Si en el ejemplo anterior la línea 6 hubiera sido

```
Productos.Add "Martillo Carpintero 4", "MrtCrp004", 6
```

Nos hubiera dado un error de "Subíndice fuera de intervalo", ya que en ese momento la colección **Productos** no contenía 6 elementos.

Para averiguar si existe ese nº de elemento usaremos el método **Count** del objeto **Productos**.

Antes de seguir: For Each - - - Next

Como veremos más adelante, en VBA existen unas estructuras de código que permiten realizar bucles ó iteraciones.

Una de ellas es la estructura

For Each Elemento in Colección

(Tras este Each, Elemento pasa a ser uno de los elementos de esa colección y podemos usarlo)

Next Elemento

Elemento debe ser del tipo **variant** ó un objeto del tipo de los que están en la colección

Por ejemplo, vamos a crear un nuevo procedimiento:

```
Public Sub PruebaColeccion2()
    Dim Clientes As New Collection
    Dim Cliente As Variant

    With Clientes
        .Add "Antonio Urrutia Garastegui", "Urrutia"
        .Add "Ibón Arregui Viana", "Arregui"
        .Add "Pedro Martínez Vergara", "Martínez"
    End With

    For Each Cliente In Clientes
        Debug.Print Cliente
    Next Cliente

End Sub
```


Al ejecutarlo nos muestra:

```
Antonio Urrutia Garastegui
Ibón Arregui Viana
Pedro Martínez Vergara
```

Fijaros que Cliente que es del tipo Variant, automáticamente se convierte en un tipo String, al pasar por el bucle For Each Cliente In Clientes

Esto es porque la colección está conteniendo datos del tipo String.

Colecciones de Controles

Las colecciones las podemos encontrar en muchas partes de Access.

Por ejemplo los formularios e informe tienen una serie de controles; pues bien, también cada formulario, o informe, posee una colección que es la colección **Controls**.

Esta colección guarda las referencias de los controles del formulario, o informe.

Vamos a ver cómo podemos leer datos de los Objetos de esta colección.

Para ello vamos a utilizar la estructura

```
For Each Objeto in Me.Controls
Next Objeto
```

Manos a la obra:

Los objetos del formulario (controles) los almacenaremos en la colección **Objetos**.

Vamos a crear un nuevo formulario, y habiendo desactivado el **asistente de controles**, como se explicaba en la entrega 01, vamos a colocar varios controles. Da igual cuáles sean.

He colocado una Etiqueta, un Cuadro de Texto, un Botón de Opción, un Cuadro Combinado, un Cuadro de Lista y un Botón de Comando.

Los cinco últimos, excepto el botón de comando, llevan a su vez asociada una etiqueta.

El módulo de clase del formulario es el que se abre desde el modo diseño del formulario, al presionar en el menú **Ver** y a continuación la opción **Código**.

También se puede presionar el botón [**Código**]

Ahora, en el módulo de clase del formulario vamos a escribir lo siguiente

```
Private Sub Form_Load()
    Caption = " Controles en " & Me.Name
    MostrarControles
End Sub
```

Y a continuación escribiremos en el procedimiento `MostrarControles`, con lo que quedará así:

```
Option Compare Database
Option Explicit

Private Sub Form_Load()
    Caption = " Controles en el formulario"
```

```

        MostrarControles
End Sub

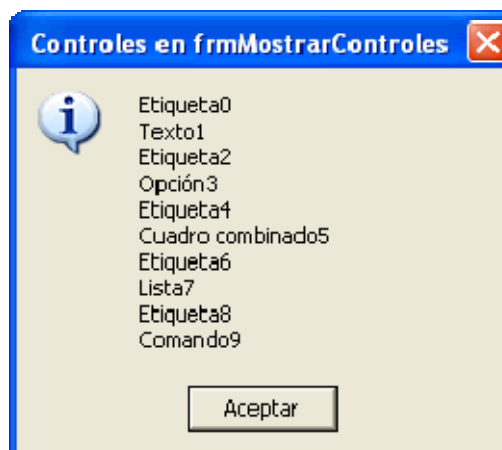
Private Sub MostrarControles ()
    Dim ctrl As Control
    Dim strMensaje As String
    For Each ctrl In Me.Controls
        strMensaje = strMensaje & ctrl.Name & vbCrLf
    Next ctrl
    MsgBox strMensaje, _
        vbInformation + vbOKOnly, _
        " Controles en " _
        & Me.Name
End Sub

```

Al arrancar el formulario, y antes de mostrarse aparecerá este Cuadro de mensaje.

*Como inciso, quiero indicar que un Cuadro de Mensaje se muestra mediante la función **MsgBox**.*

Permite mostrar mensajes con diversos tipos de iconos, botones y configurar el título de la barra superior.



El evento `Form_Load()` (**al cargar**) del formulario realiza 2 acciones

- Primero pone título al formulario Access.
En este caso **"Controles en el formulario"**. Para ello utiliza la línea `Caption = " Controles en el formulario"`
La propiedad **Caption**, en este caso se refiere al título del propio formulario.
- A continuación llama al procedimiento `MostrarControles` que está definido en el mismo formulario como un procedimiento `private`.

Vamos a fijarnos detenidamente en el procedimiento **MostrarControles**.

En él definimos dos variables, la primera de nombre `ctrl`, de tipo `Control`.

La segunda es la cadena de texto `strMensaje` de tipo `string`.

A continuación utiliza la estructura de código.

```
For Each ctrl In Me.Controls
```

Como hemos comentado antes, se podría traducir en algo así como:

Toma, uno a uno, todos los objetos **Control** que encuentres en la colección **Controls** del formulario actual y los vas asignando a la variable **ctrl**.

Con esto se consigue que **ctrl** vaya adquiriendo sucesivamente la posibilidad de manejar cada uno de los controles de la colección **Controls**, lo que equivale a poder manejar todos los controles del formulario.

En este código, cuando **ctrl** apunta a un control, puede leer ó modificar algunas de las propiedades del control al que apunta. Por ejemplo puede leer su propiedad **Name** y escribirla en la cadena **strMensaje**. A continuación llega a la siguiente línea que le hace buscar el siguiente control de la colección **Controls**.

Estas dos líneas son:

```
strMensaje = strMensaje & ctrl.Name & vbCrLf  
Next ctrl
```

Como elemento nuevo puedo mencionar **vbCrLf**.

vbCrLf es una constante definida por VBA. Es una constante de texto que contiene dos caracteres, en concreto el carácter que produce un **Retorno de Carro** (ASCII 13) y el carácter que genera un **Salto de Línea** (ASCII 10).

Forma parte de las llamadas **Constantes Intrínsecas**.

En resumen va asignando uno a uno los controles de **Controls** a la variable **ctrl**, leemos su propiedad **Name** que escribimos en la variable **strMensaje** y vamos repitiendo el proceso y escribiendo al principio de la línea siguiente.

A continuación mostramos el valor de la variable **strMensaje** en un cuadro de mensaje.

Este es un ejemplo que nos muestra cómo podemos ir asignando los elementos de una colección de controles a una variable del tipo control.

Las colecciones **Controls** las crea implícitamente Access en los formularios e informes.

Introducción a las Clases

Puede sorprender a más de uno que en el comienzo de un curso de introducción a VBA empecemos ya a hablar de las clases.

¿Y por qué no?

En el fondo, desde el principio estamos usando clases.

Un formulario es un objeto, pero tiene su módulo de clase asociado.

Cada vez que creamos un objeto, estamos usando su módulo de clase, la mayor parte de las veces sin saberlo.

¿Pero qué es una clase?

En VBA de Access, 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 Ejemplar de la clase.

Repasemos el Tipo Persona que habíamos definido en la entrega anterior

```
Public Type Persona  
    Nombre As String  
    Apellido1 As String
```

```
Apellido2 As String
FechaNacimiento As Date
Telefono As String
End Type
```

Está bien; pero estaría mejor si por ejemplo, tras introducir la fecha de nacimiento, pudiéramos obtener además la Edad de la persona de forma automática.

Para ello el tipo Persona, debería tener un procedimiento que nos devolviera la edad, en función de la fecha de nacimiento.

Esto no se puede hacer con los tipos **Registro**.

Para poder hacerlo podemos utilizar una clase.

Para crear una clase primero debemos crearnos un módulo de clase.

Para ello, desde la ventana del editor de código hacemos lo siguiente.

Desde la opción de menú **Insertar** seleccionamos la opción **Módulo de clase**.

También podríamos haber presionado la flecha Hacia abajo del botón [**Agregar módulo**] y seleccionar **Módulo de clase**.

Se nos abre el editor en la ventana del nuevo módulo, y aparece éste en la ventana del **Explorador del proyecto**. Esta ventana normalmente aparecerá en la parte izquierda de la pantalla.

Fijaros que aparece un icono con el nombre **Clase1** a su derecha.

Este es un icono especial que llevan los módulos de clase.

En VBA de Access, y también en el de Visual Basic, cada clase debe tener su propio módulo. Lo que equivale a decir que **en un módulo de clase sólo puede haber una clase**.

Vamos a cambiar el nombre de la clase y llamarla **CPersona**.

Para ello podemos, por ejemplo presionar el botón de [**Guardar**].

En esta entrega vamos a utilizar una parte muy pequeña de las posibilidades de las clases, pero es una forma de empezar.

Empezando:

Vamos a introducir este código en el módulo de clase:

```
Option Compare Database
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
    ' Que conste que esta función no es exacta
    If FechaNacimiento > 0 Then
        Edad = (Date - FechaNacimiento) / 365.2425
    End If
End Function
```

```
End If
End Function

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

Tampoco nos ha pasado nada por escribir esto.

Prácticamente la única parte que todavía no hemos visto es

```
If FechaNacimiento > 0 Then
    Edad = (Date - FechaNacimiento) / 365.2425
End If
```

De todas formas es fácil de entender:

Si `FechaNacimiento` es mayor que 0, entonces haz que la función devuelva la diferencia entre la fecha de hoy `Date`, y la `FechaNacimiento`, dividida entre 365.25.

Veremos las instrucciones **If...Then...Else** en próximas entregas.

Otra puntualización es la comilla simple

```
' Que conste que esta función no es exacta
```

La comilla simple permite escribir comentarios en el texto, para darnos información, o al que vaya a leer el código con posterioridad.

Los comentarios son ignorados durante la ejecución del código.

Ya tenemos la clase `CPersona`. ¿Cómo la podremos usar?.

Grabamos todo y nos abrimos un nuevo módulo, esta vez de los estándar.

Esta vez va a haber algo más de código:

```
Option Compare Database
Option Explicit

Public Empleados As Collection

Public Sub PruebaConClase()
    Set Empleados = New Collection
    Dim psnEmpleado As Cpersona
    Dim Empleado As New Cpersona
    Dim i As Long

    'Asignamos valores a la variable psnEmpleado
    Set psnEmpleado = New Cpersona
    With psnEmpleado
```

```
.Nombre = "Antonio"
.Apellido1 = "Urrutia"
.Apellido2 = "Garastegui"
.FechaNacimiento = #2/24/1965#
.Telefono = "998111111"
End With
' Añadimos el contenido de la variable a la colección
Empleados.Add psnEmpleado, "Urrutia"

'Asignamos valores del 2° empleado a psnEmpleado
Set psnEmpleado = New Cpersona
With psnEmpleado
.Nombre = "Ibón"
.Apellido1 = "Arregui"
.Apellido2 = "Viana"
.FechaNacimiento = #9/14/1985#
.Telefono = "998222222"
End With
' Añadimos el segundo empleado a la colección
Empleados.Add psnEmpleado, "Arregui"

'Asignamos valores de otro nuevo empleado
Set psnEmpleado = New Cpersona
With psnEmpleado
.Nombre = "Pedro"
.Apellido1 = "Martínez"
.Apellido2 = "Vergara"
.FechaNacimiento = #3/11/1979#
.Telefono = "998333333"
End With
' Añadimos el segundo empleado a la colección
Empleados.Add psnEmpleado, "Martínez"

For Each Empleado In Empleados
    With Empleado
        Debug.Print .Nombre
        Debug.Print .Apellido1
        Debug.Print .Apellido2
        Debug.Print .FechaNacimiento
        Debug.Print .Telefono
        Debug.Print .NombreCompleto _
            & ", " & .Edad & " años"
```

```
        End With
    Next Empleado

    Set Empleado = Empleados(2)
    With Empleado
        Debug.Print .Nombre
        Debug.Print .Apellido1
        Debug.Print .Apellido2
        Debug.Print .FechaNacimiento
        Debug.Print .Telefono
        Debug.Print .NombreCompleto _
            & ", " & .Edad & " años"
    End With

    Set Empleado = Empleados("Martínez")
    With Empleado
        Debug.Print .Nombre
        Debug.Print .Apellido1
        Debug.Print .Apellido2
        Debug.Print .FechaNacimiento
        Debug.Print .Telefono
        Debug.Print .NombreCompleto _
            & ", " & .Edad & " años"
    End With

    VaciaColeccion Empleados

End Sub

Public Sub VaciaColeccion(ByRef Coleccion As Collection)
    Dim i As Long
    For i = Coleccion.Count To 1 Step -1
        Coleccion.Remove (i)
    Next i
End Sub
```

Otro tema nuevo que aparece aquí es el bucle **For - - - Next** en el procedimiento **VaciaColección**.

Os adelanto que en este caso i va tomando valores que van del número de elementos de la colección, hasta 1, disminuyendo de 1 en 1.

Analizad este código y en la próxima entrega lo comentaré más a fondo.

Si ejecutamos el procedimiento **PruebaConClase** nos imprime, en la ventana **Inmediato**:

Antonio
Urrutia
Garastegui
24/02/1965
998111111
Antonio Urrutia Garastegui, 40 años
Ibón
Arregui
Viana
14/09/1985
998222222
Ibón Arregui Viana, 19 años
Pedro
Martínez
Vergara
11/03/1979
998333333
Pedro Martínez Vergara, 26 años
Ibón
Arregui
Viana
14/09/1985
998222222
Ibón Arregui Viana, 19 años
Pedro
Martínez
Vergara
11/03/1979
998333333
Pedro Martínez Vergara, 26 años

Comencemos a programar con
VBA - Access

Entrega **08**

Continuando con las Clases

Aclaraciones sobre el código del último capítulo.

Una clase no deja de ser un conjunto de Datos y Procedimientos dentro de un Módulo de Clase.

El nombre de la clase vendrá dado por el nombre con el que grabemos ese módulo.

Así en el ejemplo grabamos previamente el módulo de la clase **CPersona**, con ese nombre **Cpersona**.

Ya comenté que a los datos, accesibles de forma pública, se les llama **Propiedades** de la clase. A los procedimientos que son públicos se les llama **Métodos** de la clase.

Recordemos el código de la clase **Cpersona**

He modificado la línea : **If FechaNacimiento <> 0 Then**

```
Option Compare Database
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
    ' Que conste que esta función no es exacta
    If FechaNacimiento <> 0 Then
        Edad = (Date - FechaNacimiento) / 365.2425
    End If
End Function

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

Las dos primeras líneas son similares a las que podemos encontrar en todos los módulos de Access. Os recuerdo que la primera es para indicar que los criterios de comparación entre cadenas serán los definidos por Access, y la segunda es para exigir la declaración Explícita de las variables.

En cuanto a las Propiedades tenemos 5,

Nombre Apellido1 Apellido2 FechaNacimiento Teléfono

Y tenemos 2 procedimientos públicos (**Métodos**):

Edad NombreCompleto

Una consideración:

Tanto **Edad** como **NombreCompleto** también se podrían considerar como propiedades de la clase **CPersona**, de sólo lectura.

No obstante existen otros métodos específicos para crear las propiedades.

Estos métodos son los procedimientos **Property**. Hablaremos de ellos en otra entrega próxima cuando estudiemos las clases con más de profundidad.

Realmente estas propiedades, y métodos no revisten ninguna especial complicación. **Edad** tras comprobar que el dato pasado sea diferente que cero realiza un simple cálculo matemático de resta y división, devolviendo el resultado. Si el parámetro fuera igual a cero, devolverá el **valor por defecto** de un **Long**, que también es **cero**.

En cuanto a **NombreCompleto**, realiza una concatenación de los contenidos de las tres primeras propiedades, separándolas con Espacios en Blanco (" ") y devuelve su resultado.

Ya tenemos definida la clase **CPersona**, pero y ahora ¿qué podemos hacer con ella?.

Ya os he comentado que una clase sirve para crear **Objetos**.

Y ¿en qué se diferencia una clase de un objeto?.

Aunque no sea una comparación exacta, la que puede haber entre los planos de ingeniería de un automóvil, y el automóvil construido.

O entre el código genético de un ser y el ser vivo real.

Si mediante el código de la clase **CPersona**, creamos el objeto **Pepe**, y el objeto **Juan**, (se les llamaría **Ejemplares de la Clase CPersona**), ambos tienen en común el código en el que se basan para existir, me refiero a los objetos informáticos, pero en concreto estos dos **Ejemplares** cambian en al menos, la propiedad **Nombre**.

Por ejemplo, yo tengo un hermano gemelo, que se llama **Enrique**.

Si utilizáramos mis datos y los de mi hermano para atribuir las propiedades al Objeto **Yo** y al Objeto **MiHermano**, de la clase **CPersona**, diferirán en dos propiedades, el **Nombre** y el **Teléfono**, pero la clase en la que se basarán será la misma.

Es decir, el Objeto es la entidad creada, y la clase es el código en el que se basa ese objeto.

Para crear un objeto en VBA, se han de hacer dos cosas

1. Se declara la variable que va a manejar ese objeto, como del tipo de la clase del objeto. **Dim Yo As CPersona**

Tras esta declaración se define que la variable **Yo** va a ser un objeto del tipo del generado por la clase **CPersona**.

Pero en realidad todavía no es nada. Por ejemplo, si **CPersona** fueran los planos de un coche, tras esta declaración **Yo** sería algo así como la orden de fabricación de ese coche. Todavía sólo existe en un papel, aunque se sabe que se tiene intención de fabricarlo. ¿Y cómo se fabrica?

2. Utilizando la instrucción **New**. **Set Yo = New CPersona**

Esta línea sería equivalente a decir **Haz** que **Yo** sea una **CPersona Nueva**. Tras esto **Yo** saldría nuevecito y reluciente del taller de montaje de los modelos **CPersona**.

Pero **Yo** quién es. Es sólo un **Ejemplar** de la clase **CPersona**, También podemos decir que es una **Instancia** de la clase **CPersona** ó un **Objeto** de la clase. Lo

que hemos hecho, es mediante la instrucción **Set**, asignar una **Referencia de Objeto** a la Variable **Yo**.

¡Pobre **Yo**! Es todas estas cosas, en la jerga informática, pero en realidad ¿quién es?

¡Es sólo un **indocumentado**!

Y es un indocumentado porque no tiene ni nombre ni apellidos ni teléfono, y su fecha de nacimiento, así como su edad es Cero.

Vamos, que es un **Cero a la izquierda**.

Pero eso sí, es un flamante y nuevecito **objeto**.

Como he indicado la instrucción **Set** asigna una referencia de Objeto a la variable **Yo**. En realidad ¿qué hace?. Con **New CPersona** creamos una zona de memoria en la que ubicamos el objeto, y con **Set** conectamos la variable **Yo** a esa zona de memoria.

Si la declaración de la variable **Yo** la hubiéramos hecho de esta forma:

```
Dim Yo As New CPersona
```

Nos podremos ahorrar el paso de la asignación mediante **Set**.

Con **As New CPersona** se declara e instancia simultáneamente la variable **Yo**.

Pero no siempre es aconsejable declarar una variable objeto e instanciarla a la vez.

Por cierto, **Set** permite otras cosas, por ejemplo que haya **2 variables diferentes** apuntando al mismo objeto real.

Vamos a hacer un experimento:

Si no lo has hecho, créate un módulo de clase e introduce en él, el código de la clase **CPersona**.

Graba ese módulo de clase con el nombre **CPersona** y créate un módulo estándar.

En ese módulo estándar escribe los siguiente:

```
Option Compare Database
Option Explicit

Public Sub PruebaDeLaClaseCPersona ()
    Dim Friki As CPersona
    Dim Alienigena1 As CPersona
    ' Hay un segundo alienígena que lo vamos _
    ' a declarar y crear simultáneamente
    Dim Alienigena2 As New CPersona
    ' Creamos una instancia del Friki
    Set Friki = New CPersona
    'Todavía no es nadie, vamos a identificarlo
    With Friki
        .Nombre = "Carlos Jesús"
        .Apellido1 = "el Curandero"
        .Apellido2 = "Friki"
        .FechaNacimiento = #1/24/1945#
    End With
End Sub
```

```
        .Telefono = "696969696"
End With
' Ya no es un Sin Papeles
' está documentado y nos saluda.

Debug.Print " - - - - -"
Debug.Print "Hola terrícolas"
Debug.Print "Soy " & Friki.NombreCompleto
Debug.Print "Tengo " & Friki.Edad & " años"
Debug.Print "Y he venido a salvar al Mundo"

' Como todo el mundo sabe _
' los alienígenas son capaces _
' de apoderarse de las mentes de los Frikis
' Vamos a crear al Alienigenal _
' no sólo con los datos del Friki _
' sino siendo el verdadero Friki
Set Alienigenal = Friki
' y lo podemos comprobar
Debug.Print " - - - - -"
Debug.Print "Brrlp. Brrlp"
Debug.Print "Ahora habla el alienígena"
Debug.Print "que se ha hecho Uno con el Friki:"
Debug.Print "Soy un alienígena"
Debug.Print "en el cuerpo de " _
            & Alienigenal.NombreCompleto
' Al Alinígenal no le gusta _
' lo que ha encontrado en el Friki
' Por ello recupera su verdadera personalidad
With Alienigenal
    .Nombre = "Christopher"
    .Apellido1 = "el Mensajero"
    .Apellido2 = "de Raticulín"
    .FechaNacimiento = #1/1/1492#
End With

'Tras recuperar Alienigenal su personalidad _
¿Qué ha pasado con el Friki?
' que al ser la misma persona que el alienígena _
también ha cambiado
Debug.Print " - - - - -"
Debug.Print "Transferida la personalidad"
```

```

Debug.Print "del Alienígena al Friki"
Debug.Print "Nuevo mensaje del Friki:"
Debug.Print "Hola terrestres"
Debug.Print "Soy " & Friki.NombreCompleto
Debug.Print "Tengo " & Friki.Edad & " años"
Debug.Print "Y en el pié llevo una pila"
Debug.Print "de 2 millones de voltios"

' El Alienigena2 todavía está en el hiper-espacio
' Hagamos que pise la tierra
With Alienigena2
    .Nombre = "Micael"
    .Apellido1 = "el Vengador"
    .Apellido2 = "de Gamínedes"
    .FechaNacimiento = #12/31/999#
End With

' Y trasvasa su personalidad al Friki
Set Friki = Alienigena2
Debug.Print " - - - - -"
Debug.Print "Brrlp. Brrlp"
Debug.Print "Ahora el Friki es el segundo Alienígena:"
Debug.Print "Soy " & Friki.NombreCompleto
Debug.Print "Tengo " & Friki.Edad & " años"
Debug.Print "Y castigaré a los impíos"

' Harto de tanta tontería, apago el televisor _
  y destruyo las referencias de los objetos creados
Set Friki = Nothing
Set Alienigena1 = Nothing
Set Alienigena2 = Nothing
End Sub

```

Si ejecutamos el procedimiento **PruebaDeLaClaseCPersona** nos mostrará en la ventana inmediato los siguiente:

```

- - - - -
Hola terrícolas
Soy Carlos Jesús el Curandero Friki
Tengo 60 años
Y he venido a salvar al Mundo
- - - - -

Brrlp. Brrlp
Ahora habla el alienígena

```

```

que se ha hecho Uno con el Friki:
Soy un alienígena
en el cuerpo de Carlos Jesús el Curandero Friki
- - - - -
Transferida la personalidad
del Alienígena al Friki
Nuevo mensaje del Friki:
Hola terrestres
Soy Christopher el Mensajero de Raticulín
Tengo 513 años
Y en el pié llevo una pila
de 2 millones de voltios
- - - - -
Brrlp. Brrlp
Ahora el Friki es el segundo Alienígena:
Soy Micael el Vengador de Gamínedes
Tengo 1005 años
Y castigaré a los impíos

```

Antes de que se termine el procedimiento, existen tres variables que hacen referencias a objetos de la clase **CPersona**.

Tras las líneas

```

Set Friki = Nothing
Set Alienigena1 = Nothing
Set Alienigena2 = Nothing

```

Destruimos los objetos, al eliminar las referencias que se hacen a ellos.

*Un objeto existirá en memoria mientras quede alguna variable que haga referencia a él; es decir **Apunte** al objeto.*

Si tenemos declarado el objeto **Friki** y hacemos:

```

Set Alienigena1 = Friki
Set Alienigena2 = Friki

```

En ese momento, en realidad sólo habrá un objeto, el que hacía referencia la variable **Friki**, y tres variables que hacen referencia a él.

Cualquier cambio en las propiedades que hagamos en cualquiera de las variables, se reflejará inmediatamente en las otras dos, ya que estamos cambiando las propiedades del mismo objeto.

Recordemos que **es un único objeto con tres referencias diferentes** (no caigo qué, pero esta frase me recuerda a algo...)

Sigamos analizando el código del final de la entrega anterior

El plato fuerte del código era el procedimiento **PruebaConClase**, que ejecutaba una serie de operaciones con instancias de la clase **CPersona** (Objetos) que cargaba y descargaba en la colección **Empleados**.

Os recuerdo el argumento del código.

La variable **Empleados** del tipo colección estaba declarada como pública.

A continuación instanciábamos la variable **Empleados**, mediante la instrucción **New**, como hemos hecho en el caso de las clases.

Al fin y al cabo, una colección es un objeto y su existencia se rige por sus reglas.

Luego declarábamos **psnEmpleado** y creábamos con **New**, a la vez que declarábamos, la variable **Empleado** como del tipo **CPersona**.

Asignábamos datos a las propiedades de la variable **psnEmpleado** usando **With** y **End With**.

Añadíamos el objeto que instancia la variable **psnEmpleado** en la colección **Empleados**, mediante su método **Add**.

A continuación volvíamos a reconstruir el objeto al que instancia la variable **psnEmpleado**.

Para ello volvemos a utilizar **New**. Asignamos datos a sus propiedades y volvemos a añadirlo a la colección.

¿Por qué vuelvo a llamar a **Set psnEmpleado = New Cpersona**?

Si no hubiera hecho, el primer elemento de la colección, sería el mismo que el objeto que instancia la variable **psnEmpleado**. Por ello al cambiar los datos en la variable, también se cambiarían en el elemento de la colección. Es decir, **Empleados (1)** apuntaría al mismo objeto que el que es apuntado por **psnEmpleado**, por lo que, al ser el mismo, tendría las mismas propiedades, como hemos visto con el **Friki** y los **Alienígenas**.

En unas líneas posteriores, hacemos asignaciones de Objetos a Variables de objeto mediante **Set**, como hemos visto en el punto anterior.

Quiero resaltar que cuando hemos cargado los datos en la colección, con cada elemento le hemos asignado una clave. Esto nos permite acceder a los elementos de la colección **Empleados** mediante su **índice** o su **clave**.

Esto lo podemos ver en las líneas

```
Set Empleado = Empleados(2)
- - - -
ó
Set Empleado = Empleados("Martínez")
- - - -
```

Voy a fijarme en el último procedimiento, **VaciaColeccion**.

Fijaos que este procedimiento es llamado al final del procedimiento **PruebaConClase**.

```
Public Sub VaciaColeccion(ByRef Coleccion As Collection)
    Dim i As Long
    For i = Coleccion.Count To 1 Step -1
        Coleccion.Remove (i)
    Next i
End Sub
```


¿Qué hace este procedimiento?

Primero observad que el parámetro **Colección**, tiene delante la palabra **ByRef**.

ByRef hace que se pase al procedimiento la propia colección, al contrario que si se hubiera utilizado **ByVal**, que haría que lo que se pasara como parámetro al procedimiento fuera una copia del parámetro

El bucle **For . . . Next** hace que se repita el código que hay entre **For** y **Next**, mientras **i** sea mayor ó igual que **1**.

Me explico:

```
For i = Coleccion.Count To 1 Step -1
```

Hace que **i** vaya tomando como valores **n, n-1, n-2, . . . , 2 ,1** siendo **n** el número de elementos de la colección, devuelto por la propiedad **Count** de la misma.

La variable **i** va cambiando de valor con cada vuelta, y el incremento ó decremento lo establece la cantidad que sigue a **Step**; en este caso **-1**, con lo que decrece de uno en uno. Si hubiera sido **2**, la variable hubiera crecido de 2 en 2. después de alcanzar **i** el valor que se indica a continuación de **To**, en este caso **1**, el bucle se detiene.

Si no se indica **Step Incremento**, la variable que sigue a **For Variable = ...** irá creciendo de **1** en **1**.

En este ejemplo, el código que contiene el bucle es

```
Coleccion.Remove (i)
```

Remove es el método de la colección que elimina el elemento **i** de la misma.

En el ejemplo empieza por el último elemento y acaba el bucle cuando ha eliminado el primero.

Tras completar los ciclos la colección estará vacía, se saldrá del bucle y del procedimiento.

Comencemos a programar con VBA - Access

Entrega 09

Estructuras de Control

Estructuras de Control.

Las estructuras de control son segmentos de código que nos permiten tomar decisiones en base a unos datos dados, o repetir procesos (bucles) mientras sucedan determinadas condiciones en los parámetros controlados por el código.

Ya hemos comentado algunas de ellas en las entregas anteriores.

Estas estructuras vienen determinadas por una serie de instrucciones, entre las que destacaremos:

Estructuras de Decisión

- `If Then`
- `If Then Else`
- `IIF`
- `Select Case`

Estructuras de Bucle

- `For Next`
- `For Each . In Next`
- `While Wend`
- `Do Loop`

Instrucción de Salto

- `Goto`

Nota:

Antes de seguir adelante, adoptaré el sistema habitual para las expresiones de la sintaxis de una sentencia.

Las partes de código situadas entre corchetes [] son opcionales.

De las partes contenidas entre Llaves { } hay que seleccionar una de ellas.

Estructuras de Decisión.

La Instrucción If

Permite ejecutar un grupo de instrucciones de código, en función de que el valor de una expresión sea Cierta o Falsa **True** / **False**.

La forma más básica de esta instrucción es:

If *condición* **Then** [*instrucciones*]

Condición debe ser una expresión, numérica, relacional ó lógica que devuelva **True** ó **False**.

Por ejemplo:

```
If Divisor<>0 then Cociente = Dividendo/Divisor
```

Si el valor de la variable `Divisor` es diferente a Cero, entonces haz que la variable `Cociente` tome el valor de la división entre la variable `Dividendo` y la variable `Divisor`.

Esta forma de la instrucción **If** sólo se puede poner en una única línea de código, aunque admite múltiples instrucciones separadas por los dos puntos ":".

```
If Divisor<>0 then C = Dividendo/Divisor : Debug.print C
```

Segunda forma

```
If condición Then  
    [instrucciones]  
End If
```

El ejemplo anterior podría haberse escrito:

```
If Divisor<>0 then  
    Cociente = Dividendo/Divisor  
End If
```

Esta forma permite la ejecución de varias líneas de sentencias entre **Then** y **End If**.

Esta sintaxis es preferible a la primera, ya que genera un código más claro de interpretar.

La instrucción **If** permite ejecutar otro grupo de sentencias, si el resultado de la evaluación de la expresión fuera falso.

```
If condición Then  
    [instrucciones para el caso de que condición sea  
    True]  
Else  
    [instrucciones para el caso de que condición sea  
    False]  
End If
```

Ejemplo:

```
Public Sub PruebaIf01()  
    Dim Dividendo As Single  
    Dim Divisor As Single  
    Dim Cociente As Single  
  
    Dividendo = 4  
    Divisor = 2  
  
    If Divisor <> 0 Then  
        Cociente = Dividendo / Divisor  
        Debug.Print Cociente  
    Else  
        MsgBox "No puedo dividir entre cero", _  
            vbOKOnly + vbCritical, _  
            "División por cero"  
    End If  
  
End Sub
```

En este caso, como `Divisor <> 0` devuelve `False`, se ejecutará la línea que aparece entre `Else` y `End If`, con lo que mostrará el mensaje de error.

Estas sentencias admiten aún un modo adicional, y es usar `Else If`. Es una nueva evaluación tras una anterior que da como resultado falso.

Supongamos que queremos hacer una función que devolviera el Nombre de provincia en función de un código. Acepto por adelantado que habría otras formas más adecuadas, pero es sólo un ejemplo.

Ejemplo:

```
Public Function Provincia(ByVal Codigo As Long) As String
    If Codigo < 1 Or Codigo > 52 Then
        Provincia = "Código de provincia incorrecto"
    ElseIf Codigo = 1 Then
        Provincia = "Álava"
    ElseIf Codigo = 8 Then
        Provincia = "Barcelona"
    ElseIf Codigo = 20 Then
        Provincia = "Guipuzcoa"
    ElseIf Codigo = 28 Then
        Provincia = "Madrid"
    ElseIf Codigo = 31 Then
        Provincia = "Navarra"
    ElseIf Codigo = 31 Then
        Provincia = "Navarra"
    ElseIf Codigo = 26 Then
        Provincia = "La Rioja"
    ElseIf Codigo = 48 Then
        Provincia = "Vizcaya"
    ElseIf Codigo = 50 Then
        Provincia = "Zaragoza"
    Else
        Provincia = "Otra Provincia"
    End If
End Function
```

Con este código `Provincia(31)` devolvería `"Navarra"`

Las instrucciones `If` se pueden **anidar**, (poner unas dentro de otras).

```
If comparación1 Then
    [Instrucciones de 1]
    If comparación2 Then
        [Instrucciones de 2]
    End If
```

```
End If
```

La Función **IIf**

Es una función similar a la estructura **If . . Then . . Else**

Devuelve uno de entre dos valores, en función del resultado de una expresión:

Su sintaxis es

```
IIf(expresión, ValorTrue , ValorFalse)
```

Se evalúa la expresión y si es **True**, devuelve **ValorTrue**; caso contrario devuelve **ValorFalse**.

Por ejemplo

```
Public Function EsPar (ByVal Numero As Long) As Boolean
    EsPar = IIf (Numero Mod 2 = 0, True, False)
End Function
```

La función **IIf**, en este caso, sería igual a hacer

```
Public Function EsPar2 (ByVal Numero As Long) As Boolean
    If Numero Mod 2 = 0 Then
        EsPar2 = True
    Else
        EsPar2 = False
    End If
End Function
```

Nota:

El operador **Mod** devuelve el resto de dividir `Numero` entre 2.

La Instrucción **Select Case**

Con **If . . Then** es posible crear estructuras de decisión complejas como hemos visto en la función **Provincia** de un ejemplo anterior.

El que sea posible ni impide que resulte un tanto “Farragoso”.

Para este tipo de casos existe en VBA la instrucción **Select Case** que simplifica esas operaciones, creando un código más potente, ordenado y claro.

Si vemos la ayuda de Acces podemos leer que la sintaxis de **Select Case** es:

```
Select Case expresión_prueba
[Case lista_expresion-1
[instrucciones-1]] ...
[Case lista_expresion-2
[instrucciones-2]] ...
- - - - -
[Case lista_expresion-n
[instrucciones-n]] ...
```

```

[Case Else
[instrucciones_else]]
End Select

```

expresión_prueba debe ser una variable, o expresión que devuelva una cadena ó un número.

lista_expresion son una serie de valores, del tipo que da *expresión_prueba*.

Si *expresión_prueba* coincide con alguno de los valores de *lista_expresion*, se ejecutarán las instrucciones que existen a continuación, hasta llegar al siguiente **Case**, ó **End Select**.

A partir de este punto se saldría de la estructura y se seguiría con el siguiente código.

Si no se cumpliera la condición de ninguno de los **Case** *lista_expresion*, y hubiera un **Case Else**, se ejecutarían las líneas de código contenido a partir de **Case Else**.

Ejemplo de las expresiones que pueden estar contenidas en las *lista_expresion*:

```

Case 3
Case 3, 5, 6, 7
Case 1 To 8, 0, -5
Case Is < 8
Case Is > 3
Case Is >= lngDias
Case "A", "B", "C", "Z"

```

Voy a poner un ejemplo para clarificarlo:

Supongamos que queremos crear una función que nos cualifique el tipo de los pagarés de los clientes en función del tiempo que queda hasta su cobro.

La función recibirá como parámetro la fecha del vencimiento. Si la fecha es anterior al día de hoy, deberá devolver la cadena **"Pago vencido"**. Si es del día de hoy **"Vence hoy"**, si quedan entre 1 y 3 días **"Cobro inmediato"**, si menos de 31 días **"Corto Plazo"** si son menos de 181 días **"Medio Plazo"** y si es mayor **"Largo Plazo"**

La función podría ser ésta:

```

Public Function TipoVencimiento( _
    Vencimiento As Date _
) As String

Dim lngDias As Long
' Date devuelve la fecha de hoy
lngDias = Vencimiento - Date

Select Case lngDias
    Case Is < 0 ' Si lngDias es menor que cero
        TipoVencimiento = "Pago vencido"
    Case 0 ' Si es cero
        TipoVencimiento = "Vence hoy"
    Case 1, 2, 3 ' De 1 a 3
        TipoVencimiento = "Cobro inmediato"

```

```

    Case 4 To 30 ' De 4 a 30
        TipoVencimiento = "Corto Plazo"
    Case 31 To 180 ' De 31 a 180
        TipoVencimiento = "Medio Plazo"
    Case Else ' Si ninguno de los anteriores
        TipoVencimiento = "Largo Plazo"
    End Select
End Function

```

Aquí mostramos algunas de las posibilidades de elaboración de la *lista_expresion*.

```
Case Is < 0
```

Is se puede utilizar junto con operadores de comparación.

Estos operadores son

```

=      Igual a
<      Menor que
<=     Menor ó igual que
>      Mayor que
>=     Mayor ó igual que
<>     Diferente que

```

Se pueden poner diferentes expresiones separadas por comas

Esta línea sería válida:

```
Case Is < 0, 4, 8, is > 10
```

Se ejecutarían las líneas correspondientes al *Case* para cualquier valor que sea menor que 0, mayor que 10 ó si su valor es 4 u 8.

Este sistema también puede aplicarse a cadenas de texto.

Supongamos que queremos clasificar a los alumnos de un colegio en 4 grupos, en función de su apellido.

Aquéllos cuyo apellido empiece por una letra comprendida entre la A y la D pertenecerán al grupo 1, entre la E y la L al grupo 2, entre la M y la P al 3 y entre la Q y la Z al 4.

La función sería

```

Public Function Grupo( _
    ByVal Apellido As String _
) As Long

    Apellido = Trim(UCase(Apellido))
    Select Case Apellido
        Case Is < "E"
            Grupo = 1
        Case "E" To "LZZZZ"
            Grupo = 2
        Case "M" To "PZZZZ"

```



```

        Grupo = 3
    Case "Q" To "TZZZZ"
        Grupo = 3
    Case Is >= "U"
        Grupo = 4
End Select
End Function

```

Nota:

Hemos utilizado, como auxiliares dos funciones de VBA. En concreto en la línea

```
Apellido = Trim(UCase(Apellido))
```

Primero se aplica la función **Ucase** sobre el parámetro **Apellido** y después la función **Trim**.

Ucase convierte las minúsculas que pueda haber en **Apellido** a Mayúsculas

Trim elimina los posibles espacios en blanco que pudiera haber a la izquierda y a la derecha de **Apellido**.

En concreto, si **Apellido** contuviera el valor " Olaz ", lo convertiría a "OLAZ".

La función **Grupo** (" Olaz "), devolvería el valor 3.

Al ser "OLAZ" mayor que "M" y menor que "PZZZZ" ejecutaría la línea

```
Grupo = 3
```

Nota:

Para que dos cadenas sean iguales, deben tener los mismos caracteres.

Una cadena A es menor que otra B si aplicando los criterios de ordenación, A estaría antes que B. En este caso podemos decir que B es mayor que A porque si estuvieran en una lista ordenada alfabéticamente, B estaría después que A.

El definir si "OLAZ" es menor que "Olaz" ó es igual, se especifica en la primera línea que aparece en el módulo.

```
Option Compare Database|Text|Binary
```

A continuación de Compare podemos poner **Database**, **Text** ó **Binary**

Si aparece **Text**, Olaz sería igual a OLAZ

Si aparece **Binary** Olaz sería mayor que OLAZ

Si aparece **Database** utilizaría el criterio de ordenación por defecto de la base de datos.

Option Compare Database es específico de Access.

Por ejemplo **Visual Basic** no lo contempla.

La expresión **CadenaInferior To CadenaSuperior** se utiliza de forma similar a

```
ValorNumericoInferior To ValorNumericoSuperior
```

En la función **TipoVencimiento** teníamos la siguiente línea

```
Case 1, 2, 3 ' De 1 a 3
```

Esto hacía que si la diferencia de días fuese de 1, 2 ó 3 se ejecutara el código de ese Case.

Esta forma de generar una lista de comparaciones también se puede realizar con caracteres de texto. Sería válido, por ejemplo `Case "A", "B", "C"`

Estructuras de Bucle.

Las Instrucciones For - - - Next

Supongamos que tenemos que construir una función que nos devuelva el Factorial de un número.

Os recuerdo que Factorial de n es igual a $1*2*3* \dots *(n-1)*n$, para n entero y mayor que cero.

Adicionalmente se define que Factorial de Cero tiene el valor 1.

Cómo se haría esta función:

```
Public Function Factorial(ByVal n As Integer) As Long
    Dim i As Integer
    Factorial = 1
    For i = 1 To n
        Factorial = Factorial * i
    Next i
End Function
```

Efectivamente funciona, ya que **Factorial** devuelve resultados correctos para valores de n entre 0 y 12.

Pero esta función no sería operativa para un uso profesional ya que tiene una serie de fallos. Por ejemplo, si hacemos **Factorial(-4)** devuelve el valor 1, lo que no es correcto, ya que no existe el factorial de un número negativo. Igualmente podemos pasarle valores superiores a 12, que nos darían un error de desbordamiento, ya que 13! supera el alcance de los números **Long**. Probad haciendo en la ventana inmediato

```
? Factorial(13).
```

Observad este código:

```
Public Function Factorial(ByVal n As Integer) As Long
    Dim i As Integer
    Select Case n
    Case Is < 0
        MsgBox "No existe el factorial de un número negativo", _
            vbCritical + vbOKOnly, _
            " Error en la función Factorial"
        Exit Function
    Case 0
        Factorial = 1
        Exit Function
    Case 1 To 12
        Factorial = 1
        For i = 1 To n
            Factorial = Factorial * i
```

```
        Next i
    Case Else
        MsgBox "Número demasiado grande", _
            vbCritical + vbOKOnly, _
            " Error en la función Factorial"
        Exit Function
    End Select
End Function
```

He puesto una serie de sentencias `Case` para filtrar los valores que darían resultados incorrectos, o producirían error, avisándole al usuario de que ha tratado de utilizar la función `Factorial` con unos valores fuera de su rango válido.

Así el mayor valor lo obtenemos de

12! = 479.001.600

El tener como rango válido del 0 a 12 ¿no resulta un poco corto ?.

Dependiendo para qué, sí.

Supongamos que estamos programando un sistema estadístico que hace uso de cálculos combinatorios grandes. Probablemente esta función no serviría, aunque se podrían usar trucos para saltarse sus limitaciones.

El problema surge porque el resultado supera el rango de los números `Long`, pero en el Capítulo 5º, y también en el Apéndice 01, vemos que existen dos tipos de números que superan esa limitación. Uno es el de los `Currency` y el otro el de los `Decimal`.

Vamos a cambiar el código para trabajar con `Currency`:

```
Public Function FactorialCurrency(ByVal n As Integer) As Currency
    Dim i As Integer
    Select Case n
    Case Is < 0
        MsgBox "No existe el Factorial de un número negativo", _
            vbCritical + vbOKOnly, _
            " Error en la función FactorialCurrency"
        Exit Function
    Case 0
        FactorialCurrency = 1
        Exit Function
    Case 1 To 17
        FactorialCurrency = 1
        For i = 1 To n
            FactorialCurrency = FactorialCurrency * i
        Next i
    Case Else
        MsgBox "Número demasiado grande", _
            vbCritical + vbOKOnly, _
```

```
        " Error en la función FactorialCurrency"  
        Exit Function  
    End Select  
End Function
```

Ahora nos permite trabajar desde 0 a 17.

17! = 355.687.428.096.000

Esta ya es una cifra importante. Pero supongamos que nos contrata el Fondo Monetario Internacional, para hacer unos estudios estadísticos a nivel mundial.

Es posible la capacidad de calcular hasta el factorial de 17, se nos quede corta.

Para ello echamos mano de un tipo numérico de rango aún más alto. El tipo **Decimal**.

El tipo **Decimal** tiene la particularidad de que VBA no puede trabajar directamente con él.

La variable que lo vaya a contener debe ser primero declarada como **Variant**, y luego convertida a **Decimal**.

```
Public Function FactorialDecimal( _  
    ByVal n As Integer _  
    ) As Variant  
    Dim i As Integer  
    Dim Resultado As Variant  
    ' Aquí hacemos la conversión de Variant a Decimal  
    Resultado = CDec(Resultado)  
  
    Select Case n  
    Case Is < 0  
        MsgBox "No existe el factorial de un número negativo", _  
            vbCritical + vbOKOnly, _  
            " Error en la función Resultado"  
        Exit Function  
    Case 0  
        FactorialDecimal = 1  
        Exit Function  
    Case 1 To 27  
        Resultado = 1  
        For i = 1 To n  
            Resultado = Resultado * CDec(i)  
        Next i  
    Case Else  
        MsgBox "Número demasiado grande", _  
            vbCritical + vbOKOnly, _  
            " Error en la función FactorialDecimal"  
        Exit Function  
    End Select  
End Function
```

```

        FactorialDecimal = Resultado
    End Function

```

La función `FactorialDecimal`, trabaja en el rango de **0!** a **27!**.

27! = 10888869450418352160768000000

Es decir con 29 cifras exactas.

Una puntualización respecto al tipo `Decimal`. El tipo decimal no es el que permite un rango mayor de valores; es el que permite un rango de valores con más cifras exactas.

Los bucles del tipo `For` - - `Next` se pueden anidar (Poner unos dentro de otros).

Por ejemplo, supongamos que tenemos que generar un procedimiento que nos imprima las tablas de multiplicar que van del 1 al 10.

```

Public Sub TablasDeMultiplicar()
    Dim n As Integer, m As Integer

    For n = 1 To 10
        Debug.Print "-----"
        For m = 1 To 10
            Debug.Print n & " x " & m & " = " & n * m
        Next m
    Next n
End Sub

```

Para cada valor que tomara `n`, ejecutaría el bucle completo de `For m --- Next m`, imprimiendo en la ventana Inmediato los resultados de las tablas

```

- - -
- - -
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
-----
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
10 x 4 = 40
10 x 5 = 50
10 x 6 = 60
10 x 7 = 70
10 x 8 = 80
10 x 9 = 90
10 x 10 = 100

```

En los ejemplos anteriores hemos utilizado la estructura más típica de VBA para la creación de Bucles.

La instrucción `For m --- Next m`, repite el código contenido entre la línea que contiene la palabra `For` y la línea que contiene a su correspondiente `Next`.

Su sintaxis es

```
For contador = principio To fin [Step incremento]
[instrucciones]
[Exit For]
[instrucciones]

Next [contador]
```

Contador es una variable numérica que irá tomando sucesivos valores, con incrementos ó decrementos iguales al valor de **incremento**.

Si no se pusiera el valor **incremento**, **contador** iría creciendo en una unidad cada vuelta.

El código se irá repitiendo hasta que contador tome el valor de **fin**, ó se encuentre con la instrucción **Exit For**.

En el siguiente ejemplo, el bucle **For Next** se ejecutará hasta que **lngSuma** sea mayor que **100**, momento en que saldrá del bucle o se imprima el número de impares especificado en el parámetro **Numero**. Si el parámetro **Numero** fuese cero ó menor, se sale directamente del procedimiento sin ejecutarse el bucle.

```
Public Sub ImprimeImpares (Numero As Long)
    Dim i As Long
    Dim lngImpar As Long
    Dim lngSuma As Long
    If Numero < 1 Then
        Exit Sub
    End If
    For i = 1 To Numero
        lngImpar = 2 * i - 1
        lngSuma = lngSuma + lngImpar
        If lngSuma > 100 Then
            Exit For
        End If
        Debug.Print i & " - " & lngImpar & " - " & lngSuma
    Next i
End Sub
```

La llamada al procedimiento se haría, por ejemplo para 4 impares

```
ImprimeImpares 4
```

Después de la palabra `Next`, no es imprescindible escribir el nombre de la variable que sirve como contador. Por ejemplo este bucle es válido a pesar de no escribir `Next i`:

```
For i = 1 To 10
    Debug.Print i
Next
```

La Instrucción For Each - - - Next

Esta estructura de bucle ya la hemos visto en los capítulos anteriores, por ejemplo cuando vimos las colecciones.

Es similar a la sentencia `For`, sólo que esta sentencia repite un grupo de instrucciones para cada elemento de una colección ó una matriz, siempre que ésta última no contenga una estructura tipo Registro, definida por el usuario.

La sintaxis es:

```
For Each elemento In grupo
[instrucciones]
Exit For
[instrucciones]
Next [elemento]
```

Como en el caso de `For - - - Next`, es posible salir del bucle utilizando la instrucción `Exit For`.

En las entregas anteriores, hemos puesto ejemplos de uso con **Colecciones**. El siguiente ejemplo extrae elementos de una **Matriz**.

```
Public Sub PruebaForEachConMatrices()
    Dim Datos() As String
    Dim Dato As Variant
    ' Llamamos al procedimiento _
    ' Que rellena la matriz con datos
    RellenaMatriz Datos
    ' Leemos los elementos de la matriz
    For Each Dato In Datos
        Debug.Print Dato
    Next Dato
End Sub

Public Sub RellenaMatriz(ByRef Matriz As Variant)
    Dim i As Long
    ReDim Matriz(1 To 20)
    For i = 1 To 20
        Matriz(i) = "Dato " & Format(i, "00")
    Next i
End Sub
```

De este código lo único que no hemos visto es :

```
Matriz(i) = "Dato " & Format(i, "00")
```

La función Format la veremos detenidamente más adelante. Aquí lo que hace es añadir "01", "02", "03", ..., "10", de forma sucesiva a la cadena "Dato ".

El resultado de este procedimiento es

```
Dato 01  
Dato 02  
Dato 03  
Dato 04  
Dato 05  
Dato 06  
Dato 07  
Dato 08  
Dato 09  
Dato 10  
Dato 11  
Dato 12  
Dato 13  
Dato 14  
Dato 15  
Dato 16  
Dato 17  
Dato 18  
Dato 19  
Dato 20
```


Comencemos a programar con
VBA - Access

Entrega **10**

Estructuras de Control II

Estructuras de Control, segunda parte

Las Instrucciones While - - - Wend

La estructura de bucle

```
For Contador = ValorInicial To ValorFinal Step Salto
- -
Next Contador
```

que hemos analizado en la entrega anterior, realiza una iteración del código un número de veces que resulta previsible en función de los valores **ValorInicial**, **ValorFinal** y **Salto**.

En las sucesivas iteraciones, la variable **Contador** va tomando valores que varían de forma constante entre un ciclo y otro.

El código incluido en el bucle se ejecutará al menos una vez, aunque fuera de forma incompleta si en su camino se tropezara con una sentencia **Exit For**.

Supongamos que necesitamos una estructura que se vaya ejecutando mientras el valor que va tomando una variable cumpla determinadas características, y además que esa variable pueda cambiar en forma no lineal.

Para realizar esta tarea podemos contar con la clásica estructura **While - - Wend**.

Digo lo de clásica porque es un tipo de estructura que ha existido desde las primeras versiones de Basic.

Esta estructura tiene la siguiente sintaxis

```
While condición
[instrucciones]
Wend
```

Condición es una expresión numérica o de tipo texto, que puede devolver **True**, **False** ó **Null**. Si devolviera **Null**, **While** lo consideraría como **False**.

Las **instrucciones** de código se ejecutarán mientras **condición** de cómo resultado **True**.

Supongamos que queremos crear un procedimiento que nos muestre los sucesivos valores que va tomando una variable, mientras esta variable sea menor que 100.

Los valores que irá tomando la variable serán cada vez el doble que la anterior.

Podríamos realizarlo de esta forma

```
Public Sub PruebaWhile()
    Dim lngControl As Long
    lngControl = 1
    While lngControl < 100
        Debug.Print lngControl
        lngControl = lngControl * 2
    Wend
End Sub
```

Este código nos mostrará en la ventana inmediato:

```
1
2
4
8
16
32
64
```

Tras efectuar el 7º ciclo, la variable `lngControl` tomará el valor **128**, por lo que la expresión `lngControl < 100` devolverá **False**.

Esto hará que el código pase a la línea siguiente a **Wend**, con lo que el procedimiento de prueba finalizará.

Una utilización tradicional para **While - - Wend** ha sido la lectura de ficheros secuenciales de texto, utilizando la función **Eof**, ficheros de los que de entrada no se conoce el número de líneas,.

Esta función, mientras no se llega al final del fichero devuelve el valor **False**.

Cuando llega al final devuelve el valor **True**.

Por ello el valor **Not Eof**, mientras no se haya llegado al final del fichero, devolverá lo contrario, es decir **True**.

Veamos el siguiente código:

```
Public Sub MuestraFichero( _
    ByVal Fichero As String)
    Dim intFichero As Integer
    Dim strLinea As String

    intFichero = FreeFile
    Open Fichero For Input As #intFichero
    While Not EOF(intFichero)
        Line Input #intFichero, strLinea
        Debug.Print strLinea
    Wend
End Sub
```

Éste es el clásico código para leer el contenido de un fichero secuencial.

Vamos a fijarnos en la estructura **While - - Wend**.

Traducido a lenguaje humano quiere decir:

Mientras no llegues al final del fichero **#intFichero**

Lee la línea del fichero, hasta que encuentres un retorno de carro y asígnaselo a la variable **strLinea**.

Imprime el contenido de la variable en la ventana inmediato

Vuelve a la línea de **While** para repetir el proceso

Las Instrucciones Do --- Loop

El conjunto de instrucciones **While** - - **Wend** nos permite crear bucles que se ejecuten sólo si una variable, o expresión toma determinados parámetros.

While - - **Wend** no posee ninguna expresión que permita salir desde dentro del bucle en un momento dado, sin antes haberlo completado.

VBA posee una instrucción más potente, es la instrucción **Do** - - - **Loop**.

Su sintaxis posee dos formas distintas de utilización

```
Do [{While | Until} condición]
    [instrucciones]
[Exit Do]
[instrucciones]
```

Loop

O con esta otra sintaxis:

```
Do
    [instrucciones]
[Exit Do]
[instrucciones]
Loop [{While | Until} condición]
```

Veamos la primera forma:

Después de **Do** nos permite seleccionar **While condición**, ó **Until condición**.

Si ponemos **While**, después de **Do** el bucle se ejecutaría mientras la condición sea cierta.

Si escribimos **Until**, el bucle se ejecutaría hasta que la condición sea cierta.

Si la condición no fuese cierta no se ejecutaría el bucle tanto si hemos puesto **While**, como si hubiéramos escrito **Until** después de **Do**.

Por lo tanto podría ocurrir, tanto con **While** como con **Until** en función del resultado de **Condición**, que no se llegara a ejecutar el bucle ni una sola vez.

Si deseáramos que siempre se ejecutara al menos una vez el bucle, deberíamos usar **While** ó **Until** después de **Loop**.

Supongamos que queremos escribir una función a la que pasándole un número entero positivo, nos indique si ese número es ó no primo.

Supongo que no hará falta recordaros que un número primo es aquél que sólo es divisible por 1 ó por sí mismo.

Este es el método que voy a emplear. – Sí. Ya se que no es el óptimo:

- Dividir el número entre valores enteros, empezando por el dos, y a continuación por los sucesivos valores impares, hasta que encontremos un valor que divida de forma exacta al número a probar (su resto = 0).
Si el resto de la división da cero indica que el número es divisible por ese valor, por lo que el número no será primo y deberemos salir del bucle.
- Seguir con el ciclo mientras el valor por el que se va a dividir el número no sea mayor que la raíz cuadrada del número.

Necesitáis saber que en **VBA**, el operador que devuelve el resto de una división es **Mod**.

Si dividimos 17 entre 3 da de resto 2 17 **Mod** 3 → 2

Ya sé que este código es manifiestamente mejorable, pero funciona y me viene bien para el ejemplo con **Do Loop**.

Funciona si el número que probamos es menor ó igual que **2.147.483.647**

Este es el máximo número Long positivo. Este número también es primo.

```
Public Function EsPrimo( _
                        ByVal Numero As Long _
                        ) As Boolean

    Dim lngValor As Long
    Dim dblRaiz As Double

    Select Case Numero
    Case Is < 1
        MsgBox (Numero & " está fuera de rango")
        EsPrimo = False
        Exit Function
    Case 1, 2
        EsPrimo = True
        Exit Function
    Case Else
        dblRaiz = Numero ^ 0.5
        lngValor = 2
        ' Comprobamos si Numero es divisible por lngValor
        If Numero Mod lngValor = 0 Then
            EsPrimo = False
            Exit Function
        End If
        lngValor = 3
        EsPrimo = True
        Do While lngValor <= dblRaiz
            If Numero Mod lngValor = 0 Then
                EsPrimo = False
                Exit Function
            End If
            lngValor = lngValor + 2
        Loop
    End Select
End Function
```

Nota:

En este código he usado para calcular la raíz cuadrada de un número, elevar éste a 0,5.

En VBA hay una función que calcula la raíz cuadrada directamente: **Sqr (Número)**.

Es equivalente a **Número^{0.5}**

Habiendo escrito la función **EsPrimo**, en un módulo estándar, vamos a crear un formulario en el que introduciendo un número en un cuadro de texto, tras pulsar un botón, nos diga si es primo ó no.

Cerramos el editor de código y creamos un nuevo formulario y lo ponemos en Vista Diseño.

Añadimos al formulario una etiqueta, un cuadro de texto y un botón.

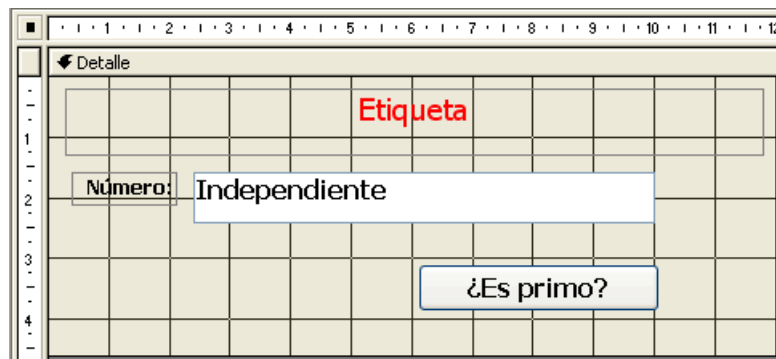
Nombres aplicados a los controles:

Etiqueta	lblMensaje
Cuadro de texto	txtNumero
Etiqueta del cuadro de texto	lblNumero
Botón	cmdPrimo

Ajustamos algunas de las propiedades del formulario, por ejemplo para quitar los separadores de registro, botones, etc....

Ya que va a ser un formulario con muy pocos controles, ponemos los textos algo mayores que lo normal, e incluso podemos jugar con los colores.

A mí me ha quedado así

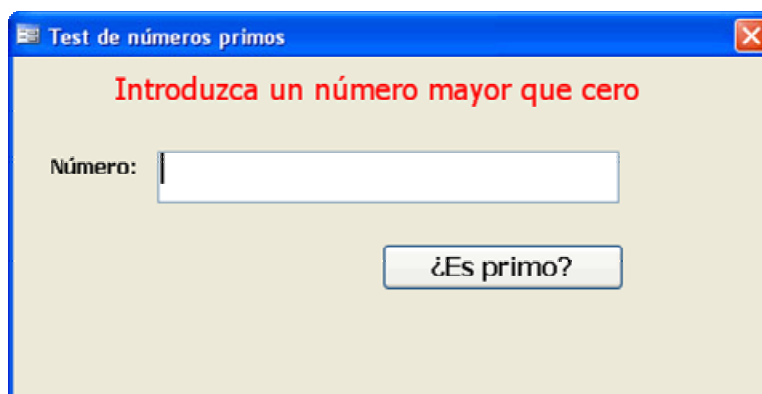


Abrimos la ventana de propiedades y teniendo seleccionado el formulario, vamos a la página de Eventos.

Hacemos que al abrir el formulario ponga como título del mismo "**Test de números primos**", y como texto de la etiqueta **lblMensaje**, "**Introduzca un número entero**".

```
Private Sub Form_Open(Cancel As Integer)
    Caption = "Test de números primos"
    lblmensaje.Caption = _
        "Introduzca un número mayor que cero"
End Sub
```

Al abrir el formulario quedará así:



Para que el formulario tenga este aspecto, he modificado algunas de sus propiedades:

Propiedad	Valor
Selectores de registro	No
Botones de desplazamiento	No
Separadores de registro	No
Estilo de los bordes	Diálogo

Vamos a hacer ahora que tras introducir un número en el cuadro de texto, y presionar el botón, nos diga en la etiqueta si el número es primo.

Volvemos a la hoja de propiedades y seleccionamos **Eventos**.

Teniendo seleccionado el botón, activamos el evento **Al hacer clic**, pulsamos en el botoncito que aparece con los tres puntos y seleccionamos **Generador de código**, y a continuación **Aceptar**.

Vamos a escribir el código:

Os recuerdo que detrás de la comilla simple lo que se escriba es un comentario (líneas en verde). Estas líneas VBA las ignora, sirviendo sólo como ayuda al usuario.

También os recuerdo que el espacio en blanco seguido de la barra inferior, al final de una línea, hace que la línea siguiente se considere como la misma línea.

El dividir así las líneas lo hago como ayuda para la composición de este texto y para ordenar el código.

```
Private Sub cmdPrimo_Click()
    Dim strNumero As String
    Dim lngNumero As Long

    ' Pasamos a la variable el contenido _
    ' de txtNumero, sin blancos en las esquinas
    ' Nz(txtNumero, "") devuelve una cadena vacía _
    ' si txtNumero contuviera Null
    ' Trim (Cadena) quita los "Espacios en blanco" _
    ' de las esquinas de la Cadena
    strNumero = Trim(Nz(txtNumero, ""))
```

```
' IsNumeric(strNumero) devuelve True _
  si strNumero representa a un número
If IsNumeric(strNumero) Then

    ' La función EsPrimo() _
      funciona con números long positivos _
      entre 1 y 2147483647
If Val(strNumero) > 2147483647# _
    Or Val(strNumero) < 1 Then
    lblmensaje.Caption = _
    "El número está fuera de rango"
    txtNumero.SetFocus
    Exit Sub
End If
lngNumero = Val(strNumero)
' Format(lngNumero, "#,##0") _
  devuelve una cadena con separadores de miles
strNumero = Format(lngNumero, "#,##0")
If EsPrimo(lngNumero) Then
    lblmensaje.Caption = _
    "El número " _
    & strNumero _
    & " es primo"
Else
    lblmensaje.Caption = _
    "El número " _
    & strNumero _
    & " no es primo"
End If
Else
    lblmensaje.Caption = _
    "No ha introducido un número"
End If
' El método SetFocus _
  hace que el control txtNumero tome el foco
txtNumero.SetFocus
End Sub
```

Tras presionar el botón **cmdPrimo** se produce el evento **click**, por lo que se ejecuta el procedimiento **cmdPrimo_Click()** que maneja ese evento. Este procedimiento lo primero que hace es declarar dos variables, **strNumero** de tipo **string** y **lngNumero** de tipo **Long**.

A continuación asigna el contenido del cuadro de texto `txtNumero`, procesado primero con la función `Nz`, que devuelve una cadena vacía si tiene el valor `Null`, y a continuación le quita los posibles espacios en blanco de los extremos mediante la función `Trim`.

Seguidamente pasa por la primera estructura de decisión `If`, controlando si la cadena `strNumero` es de tipo numérico.

Si no lo fuera muestra en la etiqueta el mensaje "No ha introducido un número".

Si lo fuera, primero comprueba si la expresión numérica de `strNumero` está entre 1 y 214748364, rango de valores válidos en el rango de los `Long`, para la función `EsPrimo`.

Si no fuera así, muestra el mensaje "El número está fuera de rango", lleva el cursor al control `txtNumero` y sale del procedimiento.

Supongamos que el contenido de `strNumero` ha logrado pasar todos estos controles.

Mediante la función `Val(strNumero)` asigna el valor a la variable `lngNumero`.

Como ya no vamos a utilizar la cadena `strNumero` para más cálculos, para mostrar el número, le asignamos el resultado de la función `Format(lngNumero, "#,##0")`.

Con esta utilización, la función `Format` devuelve una cadena formada por el número con los separadores de miles.

La función `Format` tiene un amplio abanico de posibilidades en la conversión de números y fechas a cadenas de texto.

Merece por sí misma un tratamiento más extenso.

Se lo daremos en una próxima entrega.

El siguiente paso es comprobar si el número `lngNumero` es primo, utilizando la función `EsPrimo` que escribimos anteriormente.

Si lo fuera, escribiríamos en la etiqueta "El número " seguido del contenido de la cadena `strNumero`, y el texto " es primo".

Si no lo fuera, escribiríamos lo mismo, pero indicando " no es primo".

Terminado todo esto llevamos el cursor al cuadro de texto `txtNumero` mediante su método `SetFocus`.

Todo muy bien.

El cliente está contento y el programa responde a lo que nos pedía, pero...

Casi siempre hay un pero...

Viendo lo efectivos y rápidos que hemos sido, al cliente se le ocurre que sería muy interesante poner dos botoncitos que al presionarlos, dado un número cualquiera, nos muestre el número primo inmediatamente mayor ó menor al número que hemos mostrado.

-Tiene que ser fácil, total ya has hecho lo más importante y éste es un pequeño detalle adicional, que no te costará prácticamente nada de tiempo y supongo que no tendrás problemas para hacérmelo sin aumentar el importe presupuestado...

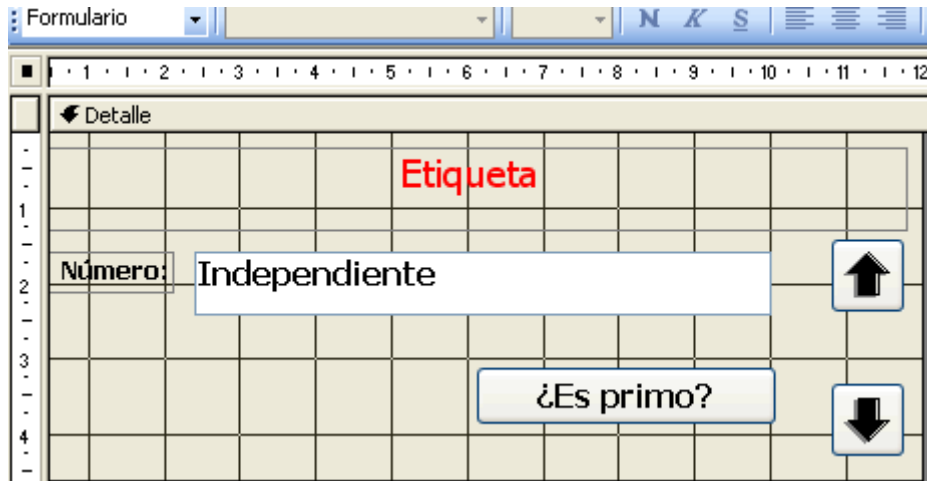
¿A alguno le suena esta conversación?

Y además, aunque ya has terminado lo que te pedían, como hay que añadirle este "pequeño detalle..." no te pagan hasta que no lo termines...

Decido añadir dos botones con unas flechas en su interior.

Al primero, con una flecha hacia arriba lo llamo `cmdPrimoSiguiente`, y al segundo, con una flecha hacia abajo, `cmdPrimoAnterior`.

Este es el diseño que le doy al formulario:



Los eventos clic de los dos botones los escribo así:

```
Private Sub cmdPrimoSiguiente_Click()
    ' La siguiente línea hace que se ignoren _
    ' los posibles errores en la ejecución.
    On Error Resume Next
    Dim strNumero As String
    Dim lngNumero As Long
    Dim blnPrimo As Boolean
    strNumero = Trim(Nz(txtNumero, ""))
    If IsNumeric(strNumero) Then
        lngNumero = Val(strNumero)
        ' Si lngNumero está entre 0 y 2147483646
        If lngNumero < 2147483647# And lngNumero >= 0 Then

            ' Mientras blnPrimo no sea Cierto _
            ' Es decir Mientras lngNumero no sea primo.
            Do While Not blnPrimo
                lngNumero = lngNumero + 1
                blnPrimo = EsPrimo(lngNumero)
            Loop
            txtNumero = CStr(lngNumero)
            cmdPrimo_Click
        Else
            txtNumero = "1"
            cmdPrimo_Click
        End If
    Else
        txtNumero = "1"
        cmdPrimo_Click
    End If
End Sub
```

```
End Sub
```

En el código anterior podemos ver algunas cosas interesantes.

Lo primero que nos puede llamar la atención es la sentencia:

```
On Error Resume Next
```

Esta es la forma más básica de efectuar un control de los errores que se puedan originar durante la ejecución de un programa en VBA.

Simplemente se le está indicando a VBA que si se produjera un error en algún punto del procedimiento lo ignore y vaya a la siguiente sentencia del código.

El ignorar los errores no es una verdadera forma de control.

Aprenderemos en otra entrega diferentes formas de manejar los posibles errores, ya sean generados por el código, por datos inadecuados de los usuarios, etc.

Más adelante nos encontramos con una sentencia If que evalúa una expresión doble

```
If lngNumero < 2147483647# And lngNumero >= 0 Then
```

Para que esta expresión sea cierta, lo tienen que ser a la vez las dos expresiones unidas por **And**; es decir **lngNumero** tiene que ser menor que **2147483647** y simultáneamente tiene que ser mayor ó igual que **0**.

Cuando varias expresiones estén unidas por el Operador Lógico **And**, para que la expresión total sea cierta, es necesario que lo sean cada una de esas expresiones. Con que haya una falsa, la expresión total será falsa.

Por el contrario, cuando varias expresiones estén unidas por el Operador Lógico **Or**, para que la expresión total sea cierta, es suficiente con que lo sea una cualquiera de las expresiones que la forman.

A continuación nos encontramos con otro Operador, es el operador negación **Not**.

```
Do While Not blnPrimo
```

Not hace que la expresión lógica que le sigue cambie su valor.

Así si **blnPrimo** contiene el valor **True**

```
Not blnPrimo
```

devolverá el valor **False**.

La expresión equivale a:

```
Mientras blnPrimo no sea cierto
```

Que es equivalente a

```
Mientras blnPrimo sea falso.
```

Con ello se ejecutará el código contenido entre la línea de **Do** y la línea del **Loop**.

Cuando **lngNumero** sea primo, la función **EsPrimo** asignará **True** a **blnPrimo**, con lo que se saldrá del bucle, pondrá la cadena de texto del número **txtNumero** en el cuadro de texto y ejecutará el procedimiento **cmdPrimo_Click**, como si se hubiera presionado en el botón [**cmdPrimo**].

Si el valor de **lngNumero** no hubiera cumplido con el rango de valores, pone un 1 en el cuadro de texto **txtNumero**, y ejecuta el procedimiento **cmdPrimo_Click**.

En el procedimiento que maneja la pulsación de la tecla [**cmdPrimoAnterior**] aunque tiene una estructura semejante, se introducen unos cambios que considero interesante remarcar.

```

Private Sub cmdPrimoAnterior_Click()
    ' Ignorar el error
    On Error Resume Next

    Dim strNumero As String
    Dim lngNumero As Long

    strNumero = Trim(Nz(txtNumero, ""))
    If IsNumeric(strNumero) Then
        lngNumero = Val(strNumero)
        If lngNumero < 2147483648# And lngNumero > 1 Then
            lngNumero = lngNumero - 1

            Do Until EsPrimo(lngNumero)
                lngNumero = lngNumero - 1
            Loop
            txtNumero = CStr(lngNumero)
            cmdPrimo_Click
        Else
            txtNumero = "2147483647"
            cmdPrimo_Click
        End If
    Else
        txtNumero = "2147483647"
        cmdPrimo_Click
    End If
End Sub

```

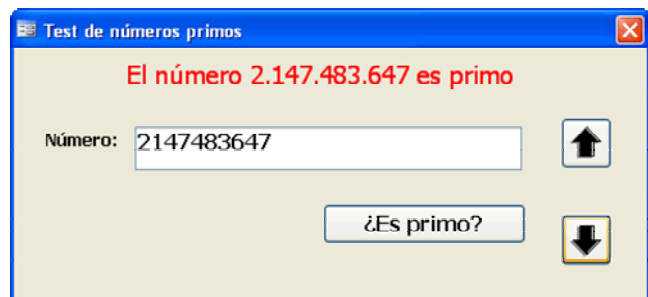
En primer lugar utilizamos una estructura del tipo **Do Until**, en vez de **Do While**.

Además, como condición no utiliza una variable como en el caso anterior, sino que lo compara directamente con el valor devuelto por la función `EsPrimo`, que devuelve **True** ó **False** según sea el caso:

```
Do Until EsPrimo(lngNumero)
```

Con esto nos evitamos utilizar una variable y una sentencia adicional. Además el código resulta algo más claro..

En este caso, si la variable no supera los filtros, pone el valor "2147483647" en el cuadro de texto.



Comencemos a programar con VBA - Access

Entrega 11

Gestión de errores

Errores

La gestión de los errores en procedimientos

A la hora de utilizar el código de un módulo, hay dos tiempos

- Tiempo de **Diseño**.
- Tiempo de **Ejecución**.

El **tiempo de diseño** transcurre mientras estamos modificando el contenido del código de un módulo, sea del tipo que sea, o cuando estamos cambiando las propiedades de controles, formularios o informes, en la llamada Vista Diseño, ya sea directamente ó mediante ejecución de código.

Esto a más de uno le sorprenderá:

*Access permite crear mediante código, formularios por ejemplo utilizando el método **CreateForm**, que devuelve una referencia a un nuevo formulario, también permite añadirle controles mediante el método **CreateControl**, e incluso asociarle un módulo, escribiendo dinámicamente todo su contenido. Para esto último tendríamos que crear una referencia al objeto **Module** del formulario, y para insertarle el código utilizar su método **InsertText**.*

*De forma semejante existe el método **CreateReport** para la creación dinámica de informes.*

*Si queremos usarlos posteriormente deberemos guardarlos, por ejemplo con el método **Save** del objeto **DoCmd**.*

El **tiempo de ejecución** transcurre cuando hemos creado una instancia de un objeto, formulario, informe, clase o hemos llamado a un procedimiento de un módulo estándar.

En el lenguaje “normal” podríamos decir que estamos en tiempo de ejecución cuando estamos “ejecutando” los objetos o el código de Access.

Errores en Tiempo de Diseño

En Tiempo de Diseño podemos cometer una serie de errores, a la hora de escribir el código.

Muchos de estos errores serán detectados inmediatamente por el editor de Access.

Cuando escribimos una línea de código, Access realiza un análisis del texto que estamos escribiendo. En este proceso se realiza fundamentalmente su análisis sintáctico.

También comprueba si hay sentencias incompletas, por ejemplo **If** sin **Then**.

Si encuentra una expresión errónea lanza un mensaje de **Error de compilación** e incluso aporta una cierta información que nos puede orientar sobre el origen del error.

```
Public Function PruebaDeError ()
    Dim i As Long
    If i < 4
```

```
End Func
```



La línea de código incorrecta queda marcada en color rojo.

Cuando ejecutamos el código, la primera vez que lo hace, no sólo realiza un análisis sintáctico, además va comprobando que todas las constantes y variables, ya sean de tipos estándar, referencias de objetos ó tipos definidos por el usuario, estén perfectamente declaradas, y los tipos de objeto existan y sean correctos.

```
Public Function PruebaDeError ()  
    Dim i As Long  
    n = 4  
End Function
```



Si se detecta algún error se interrumpe la ejecución del código y se lanza un aviso, marcando la zona del código donde el error se ha producido. Esta depuración del código se va realizando conforme se efectúan llamadas a los diferentes procedimientos.

Podría ocurrir que tuviéramos un procedimiento que sólo se usara en determinadas condiciones y que contuviera por ejemplo una variable mal declarada.

Si al ejecutar el código no se llega a utilizar ese procedimiento, no se detectaría el error que contiene.

Para evitar “sorpresas” posteriores, es aconsejable realizar una pre-compilación del código.

Para realizarla podemos utilizar la opción de menú **[Compilar NombreDelFicheroAccess]** de la opción de menú **[Depuración]**.

A esta opción de menú se puede llegar también mediante el botón **[Compilar]** .

Esta pre-compilación revisa todo el código, e incluso posibles procedimientos que no serían utilizados durante la ejecución del programa. Esto nos da más garantía sobre la calidad del código y nos protege frente a ciertos tipos de error que de otra forma no podríamos detectar.

Errores en Tiempo de Ejecución

Hay una serie de errores que se pueden producir durante la ejecución del código, que no son de sintaxis ni originados por código incompleto o declaraciones inadecuadas.

Son, por ejemplo los errores provenientes de valores no previstos por el código pasados ya sea por el propio usuario extraídos de tablas, ficheros u otras fuentes de origen.

Son los típicos errores de Tiempo de Ejecución.

Supongamos que tenemos que dividir entre sí dos cantidades; si el denominador vale cero, nos dará un error de división entre cero.

Podría ocurrir que no hayamos previsto que un cuadro de texto contuviera el valor **Null**. Esto nos podría generar error al intentar asignar este valor a una cadena de texto.

También podría ocurrir que en una expresión por la que queremos asignar el resultado de una operación a una variable, ese resultado superara el rango admisible por el tipo de la variable, con lo que tendríamos un error de **Desbordamiento**.

Un programa profesional debe adelantarse a todas estas posibilidades.

Por ejemplo, si tenemos que dividir dos números, se debería comprobar que el denominador no contuviese el valor Cero.

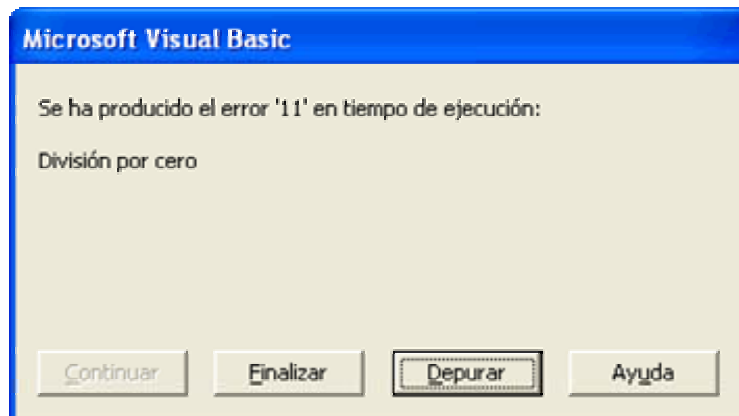
Si tuviéramos que obtener, un elemento de una matriz, ó colección, podríamos evitar un error de subíndice fuera de intervalo.

Hay muchas circunstancias en las que se pueden producir errores en tiempo de ejecución. VBA nos provee de una herramienta para poder controlarlos una vez que se han producido. Supongamos que tenemos el siguiente procedimiento.

```
Public Sub ErrorDivisionPorCero()  
    Dim n As Byte  
    n = 0  
    Debug.Print 4 / n  
End Sub
```

- Sí, ya se que es un código sin mucho sentido; lo pongo sólo como ejemplo didáctico.

Si se ejecutara este procedimiento, ya sea llamado desde la ventana **Inmediato** o desde cualquier punto de la aplicación se producirá el preceptivo error:



Inmediatamente se interrumpirá la aplicación.

Si esto nos ocurriera a nosotros mientras estamos haciendo pruebas no tendría mayor importancia. Pero si le ocurriera a nuestro cliente, no creo que nos llamara para felicitarnos.

Por lo tanto deberemos anticiparnos a cualquier error que se pueda originar durante la ejecución de nuestros programas.

Tan importante como la redacción previa de los Casos de Uso que nos ayudarán a definir nuestro programa, es la redacción de las pruebas a las que deberemos someter nuestro programa antes de entregar al cliente la nueva versión. Y no sólo redactarlas, sino también ponerlas en práctica.

Frente a la posibilidad de un error, hay dos principales caminos

- Realizar todo tipo de filtros en el código para evitar que llegue a darse una situación de error
- En caso de que el error se produzca, capturarlo y darle una respuesta “civilizada”.

Y sobre todo acordarse de la ley fundamental de la programación:

- Si en un segmento de código existe la más pequeña posibilidad de que se produzca un error, éste acabará produciéndose irremediamente, y además en el momento más inoportuno y de peores consecuencias.

Para capturar un error, VBA utiliza la más denostada, criticada y repudiada de las sentencias de código. Es la sentencia **Goto**.

Es la instrucción de salto que nos faltaba por ver en las dos entregas anteriores.

Instrucciones de salto.

La Instrucción Goto

Cuando el código se encuentra con esta sentencia realiza un salto incondicional.

Su sintaxis tiene esta estructura:

```
GoTo línea
```

Línea puede ser una Etiqueta de línea ó un número de línea.

Observa este código

```
Public Sub PruebaGoto()  
    GoTo Etiqueta_01  
    Debug.Print "No he saltado a Etiqueta_01"  
Etiqueta_01:  
    Debug.Print "*** Salto a la Etiqueta_01 ***"  
    GoTo 10  
    Debug.Print "No he saltado a 10"  
10    Debug.Print "*** Salto a la línea 10 ***"  
End Sub
```

Si ejecutamos el código nos imprime en la ventana Inmediato

```
*** Salto a la Etiqueta_01 ***  
*** Salto a la línea 10 ***
```

Una etiqueta es una palabra que comience con una letra y termine con los dos puntos.

Un número de línea es un número situado al principio de la línea.

La utilización de números de línea proviene de la época de Basic, cuando las líneas de código debían ir precedidas de un número que las identificara.

```
GoTo Etiqueta_01
```

Salta a la etiqueta sin ejecutar la línea intermedia.

```
GoTo 10
```

Salta a la línea precedida por el número 10.

Gosub - - Return

Es otra instrucción de salto, de la cual sólo voy a comentar que existe por compatibilidad con las antiguas versiones de Basic.

Si alguien quiere más información puede acudir a la ayuda de VBA.

Personalmente desaconsejo completamente su uso, ya que en VBA existen alternativas más eficientes y claras.

La utilización de **Goto** y **Gosub** se desaconseja ya que pueden convertir el código en una sucesión de saltos de canguro imposible de seguir de una forma coherente.

Capturar Errores

Ya hemos comentado que durante la ejecución de una aplicación pueden producirse diversos tipos de errores, como rangos de valores no válidos, división por cero, manipulación de un elemento de una matriz, colección, o un fichero que no existan, etc.

Si prevemos que en un procedimiento pudiera producirse un error, para poder gestionarlo, pondremos en su cabecera o en un punto anterior al lugar donde el error se pudiera generar, la sentencia:

```
On error Goto Etiqueta
```

Si un procedimiento incluye esta línea, y en tiempo de ejecución, en una línea posterior a ésta, se produjera un error, la ejecución del código saltará a la línea que incluye esa etiqueta, y mediante el objeto **Err** podremos controlar el error.

El objeto Err

Este objeto contiene la información de los errores que se producen en tiempo de ejecución.

Cuando se produce un error, el objeto o el procedimiento que lo ha generado puede asignar datos a sus propiedades.

Las propiedades más importantes, o las que en este nivel nos interesan más, son:

- Number
- Description
- Source

Hay otras propiedades que, de momento no vamos a analizar, como son

```
HelpContext      HelpFile      LastDLLError
```

La propiedad **Number** contiene un número que sirve como identificador del error.

Description incluye una cadena de texto que nos sirve para interpretar las características del error.

Source contiene información sobre el proyecto u objeto que ha generado el error.

Tomando como modelo el código que generaba un error de División por cero, vamos a hacer otro procedimiento que lo controle:

```
Public Sub ErrorControlado()  
    On Error GoTo HayError  
    Dim n As Byte  
    n = 0  
    Debug.Print 4 / n  
Salir:  
    Exit Sub  
HayError:  
    Debug.Print "Error nº " & Err.Number  
    Debug.Print Err.Description  
    Debug.Print Err.Source  
    Resume Salir  
End Sub
```

El resultado de la ejecución de este código es:

```
Error n° 11
División por cero
Entrega11
```

Para usar las propiedades del objeto **Err**, se utiliza la misma sintaxis que para otros objetos `NombreObjeto.Propiedad`

La instrucción Resume

Hace que la ejecución del código continúe en una línea determinada, tras gestionar un error.

En el ejemplo anterior Resume Salir hace que el código vaya a la línea marcada con la etiqueta **Salir**:

La ejecución de

```
Resume Salir
```

hará que se salga del procedimiento.

Después de la palabra Resume, puede ponerse una etiqueta ó un número de línea.

En ambos casos se producirá un salto hasta la línea especificada.

Si se pusiera

```
Resume           ó           Resume 0
```

Si el error se hubiera producido en el procedimiento que contiene el controlador de errores, la ejecución continúa en la instrucción que lo causó.

Si el error se produjera en un procedimiento llamado, la ejecución continuará en la instrucción para el control de errores desde la cual se llamó al procedimiento que contiene la rutina de gestión de errores..

Si se hubiera escrito

```
Resume Next
```

Se ejecutará la línea siguiente a aquella en la que se produjo el error ó se llamó al procedimiento para la gestión de errores.

Resume sólo se puede usar en una rutina de gestión de errores.

Vamos a ver cómo manejan las excepciones los asistentes de Access.

Creamos un nuevo formulario mediante **[Nuevo] - [Vista Diseño]**. Con ello se abre un nuevo formulario en Vista Diseño, sin estar ligado a ningún origen de datos.

Si no tuviéramos visible la barra de herramientas, vamos a activarla, mediante la opción de menú **[Ver] – [Cuadro de herramientas]**.

En esta barra, si no estuviese activado, activaremos el botón **[Asistentes para controles]**. En este gráfico ejemplo el botón está desactivado →



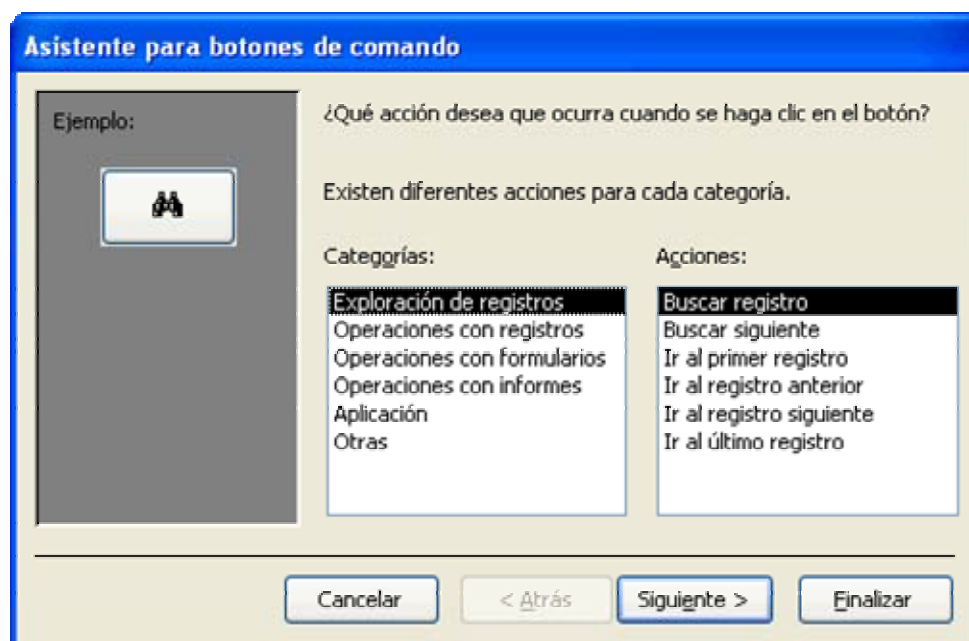
Una vez activado el botón de los asistentes, al seleccionar un control, Access generará código por nosotros, código que se ejecutará con alguno de los eventos del control.

Ahora vamos a poner un botón que servirá para cerrar el formulario cuando se presione sobre él.

Seleccionamos en la barra de herramientas el **[Botón de comando]** y lo colocamos en el formulario, pinchando sobre él y marcando, sin soltar el botón izquierdo, el rectángulo que va a ocupar el botón.

Tras hacer esto, se nos abre la ventana del Asistente para **[Botones de comando]**.

En la parte de la izquierda nos muestra un ejemplo de los iconos que se podrían poner en el botón.



A continuación hay una lista de categorías de acciones.

La lista de la derecha muestra las acciones concretas que tiene el asistente para cada categoría.

Como lo que queremos es que el botón **Cierre el Formulario**, en la lista de la izquierda seleccionaremos Operaciones con formularios.

Automáticamente nos mostrará las acciones disponibles con los formularios.

Seleccionaremos en la lista de la derecha la opción **Cerrar formulario**.

Pulsamos en el botón **[Siguiete]**.

Ahora nos aparecen nuevas opciones. En concreto nos permite seleccionar si queremos poner un texto ó una imagen en el botón.

Si seleccionáramos **[Texto]** podríamos cambiar el texto que nos sugiere Access.

Esta vez vamos a seleccionar una Imagen. Al seleccionar el botón **[Imagen]** se activan el botón **[Examinar]** y la Casilla de verificación **[Mostrar todas las imágenes]**.

El botón [Examinar] permite seleccionar cualquier imagen compatible que tengamos en el ordenador. La casilla [Mostrar todas las imágenes] nos muestra los nombres de todas las imágenes prediseñadas por Access.

No vamos a hacer caso de estas opciones y seleccionaremos sin más la imagen Salir, que nos muestra una Puerta entreabierta señalada con una flecha.

Presionamos el botón [Siguiete] y nos pedirá un nombre para ese botón.

Le vamos a poner como nombre **cmdsalir**.

Veamos qué nos ha hecho el asistente.

Seleccionamos el botón, [Código] ó la opción de menú [Ver] - [Código].

Vemos que nos ha colocado, en el módulo de clase del formulario, el procedimiento

```
Private Sub cmdSalir_Click()
```

Este procedimiento es el Manejador del evento **Click** para el botón **cmdSalir**.

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

Lo primero que hace es utilizar `On Error GoTo Err_cmdSalir_Click`; con ello si se produjera algún error saltará a la línea marcada con `Err_cmdSalir_Click`:

Esta es la forma como los asistentes de Access suelen nombrar las etiquetas que señalan el comienzo del código de Gestión de Errores **Err_NombreDelProcedimiento**:

A continuación nos encontramos con que se ejecuta el método **Close** del objeto **DoCmd**.

El objeto **DoCmd** es un objeto especial de Access. Contiene un gran número de métodos para ejecutar muchas de las tareas habituales con los objetos de Access.

A gran parte de estos métodos se les puede pasar una serie de argumentos.

Por sus amplias posibilidades, **DoCmd** merece un capítulo aparte. Lo desarrollaremos en próximas entregas.

Basta decir que en este caso, al llamar al método **Close** sin pasarle ningún argumento, **DoCmd** cierra la **ventana activa** y como ésta es el formulario del botón, al pulsarlo cierra el formulario, que es lo que queríamos conseguir.

Si no ha habido ningún problema, la ejecución del código se encuentra con la etiqueta que marca el punto a partir del cual se sale del procedimiento. Como aclaración, una etiqueta no ejecuta ninguna acción, sólo indica dónde comienza un segmento de código. En nuestro caso, a partir de este punto nos encontramos con la línea `Exit Sub` que nos hace salir del procedimiento `cmdSalir_Click`.

Después de esta última línea comienza el segmento de código que controla cualquier error en tiempo de ejecución que se pudiera originar dentro del procedimiento.

Primero, mediante un cuadro de mensaje, nos muestra el texto contenido en la propiedad **Descripción** del objeto **Err**.

A continuación, mediante `Resume Exit_cmdSalir_Click` hace que salte el código a la etiqueta que marca la salida del procedimiento.

Una puntualización: **Resume**, además de dar la orden de que el código se siga ejecutando desde un determinado punto del procedimiento, pone a "cero" las propiedades del objeto **Err**, lo que equivale a hacer que desaparezca el error.

Como hemos visto, cuando usamos un asistente para controles de Access, en el código generado se suele colocar un segmento para la gestión de errores.

Este código escrito por Access lo podemos cambiar; por ejemplo podríamos sustituir `Exit_cmdSalir_Click` por **Salir**, ó `Err_NombreDelProcedimiento` por **hayError**, e incluso lo podríamos quitar, si estuviéramos seguros de que nunca se podría llegar a producir un error o ya tuviéramos desarrollado nuestro propio sistema para la gestión de errores.

Gestionando errores

Supongamos que nos han encargado un programa para visualizar en un formulario el contenido, en formato texto, de los ficheros que seleccionemos.

Para especificar el fichero que se va a visualizar nos piden que su nombre, incluida su ruta, se escriba en un cuadro de texto.

Tras esto, y presionar un botón, su contenido se mostrará en un segundo cuadro de texto.

Se decide seleccionar un cuadro de texto como soporte, con el fin de que se puedan seleccionar y copiar segmentos del texto del contenido de los ficheros.

Debo aclarar que no todo el contenido de un fichero cualquiera se podrá mostrar en un cuadro de texto. Si el fichero es de tipo Binario, gráficos, ficheros exe ó multimedia, habrá caracteres que no se muestren ó que tengan secuencias ininteligibles.

Como ya habíamos escrito previamente un procedimiento para mostrar el contenido de ficheros (**MuestraFichero** del capítulo anterior) vamos a aprovecharlo, con algunas modificaciones para adaptarlo a los nuevos requerimientos.

El nuevo procedimiento exigirá que se le pase como parámetro un cuadro de texto; en este caso con nombre **Pizarra**, que será donde se irá escribiendo el contenido del fichero mientras se vaya leyendo.

Además le añadiremos un control de errores, ya que podría ocurrir que no se pudiera abrir el fichero por no existir, o porque estuviera abierto en modo exclusivo por algún usuario.

```
Public Sub MuestraFichero( _
    ByVal Fichero As String, _
    ByRef Pizarra As TextBox)
    On Error GoTo ProducidoError
    Dim intFichero As Integer
    Dim strLinea As String
    intFichero = FreeFile
    Open Fichero For Input As #intFichero
    While Not EOF(intFichero)
        Line Input #intFichero, strLinea
        Pizarra.Value = Pizarra.Value & strLinea
    Wend
Salir:
    Exit Sub
ProducidoError:
    MsgBox "Se ha producido el error " & Err.Number, _
        vbCritical + vbOKOnly, _
        "Error en el procedimiento MuestraFichero()"
```

```
Pizarra.Value = ""
Resume Salir
End Sub
```

Creamos un nuevo formulario y le añadimos los siguientes elementos

Control	Nombre
Cuadro de texto	txtFichero
Cuadro de texto	txtContenido
Botón	cmdVerFichero

El cuadro de texto **txtFichero** servirá para introducir el nombre completo del fichero a visualizar.

En **txtContenido** mostraremos el contenido del fichero.

El botón **cmdVerFichero** activará el procedimiento que sirve para mostrar el contenido del fichero. La llamada al procedimiento lo colocaremos en el evento Al hacer clic del botón **cmdVerFichero**.

Usaremos el procedimiento `MuestraFichero` al que le pasaremos como parámetro el control **txtContenido**.

En el cuadro de texto **txtContenido** activamos la Barra Vertical en la propiedad **Barras de Desplazamiento**.

En el evento Al hacer clic del botón escribiremos lo siguiente:

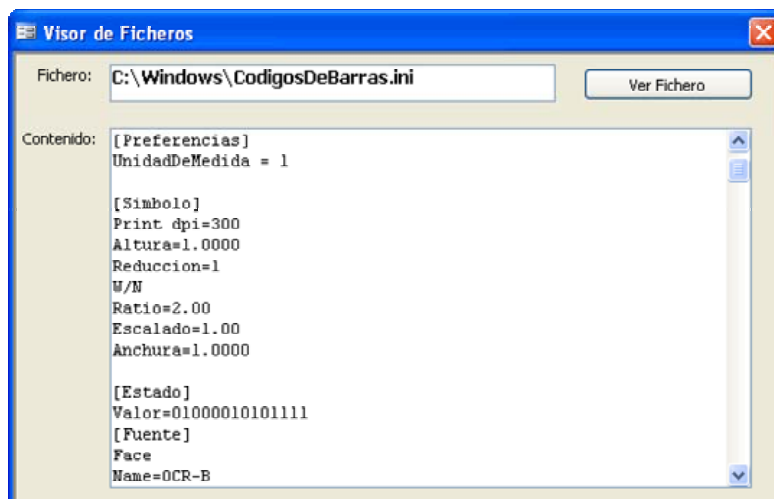
```
Private Sub cmdVerFichero_Click()
    txtContenido = ""
    MuestraFichero txtFichero, txtContenido
End Sub
```

Lo que hace el procedimiento es:

Limpia el posible contenido de `txtContenido` asignándole una cadena vacía.

Llama al procedimiento `MuestraFichero` Pasándole como parámetros el nombre del fichero que queremos abrir, contenido en el control `txtFichero`, y el control en el que queremos mostrar los datos `txtContenido`.

En el ejemplo se muestra el contenido del fichero `CodigosDeBarras.ini` ubicado en la carpeta `Windows`.



Generación directa de errores (Err.Raise)

Con el método **Raise** del objeto **Err**, podemos forzar la generación de un error en tiempo de ejecución

La sintaxis del método **Raise** es

Err.Raise NúmeroError, OrigenError, Descripción, FicheroAyuda, ContextoDeAyuda

NúmeroError es el número de error que queremos pasar al objeto Err.

OrigenError Objeto, Aplicación, Procedimiento que genera el error.

Descripción Texto que identifica al propio error.

FicheroAyuda Fichero de ayuda con información sobre el error.

ContextoAyuda Identificador de contexto que especifica el tema del archivo indicado en FicheroAyuda, que contiene la información de ayuda del error.

De los cinco parámetros, el único obligatorio es el Número que identifica al error.

A título de información, adelanto que con Access se pueden utilizar ficheros de ayuda creados al efecto por nosotros.

La creación y utilización de ficheros de ayuda es uno de los temas considerados como "avanzados".

Los números de error reservados para ser definidos por el usuario son los que están entre el número **513** y el **65535**.

```
Public Sub GenerarError()  
    On Error GoTo HayError  
    ' Aquí generamos el error  
    Err.Raise 666, "GenerarError", _  
        "¡Houston! Tenemos un problema"  
Salir:  
    Exit Sub  
HayError:  
    Debug.Print "Error nº " & Err.Number  
    Debug.Print "Se ha producido el error: " _  
        & Err.Description  
    Debug.Print "en el procedimiento: " _  
        ; Err.Source  
    Resume Salir  
End Sub
```

Si ejecutamos el procedimiento **GenerarError** nos mostrará en la ventana Inmediato:

```
Error nº 666  
Se ha producido el error: ¡Houston! Tenemos un problema  
en el procedimiento: GenerarError
```


Comencemos a programar con
VBA - Access

Entrega **12**

**Trabajando con
procedimientos**

Procedimientos

Hemos hablado sobre diversos aspectos de los procedimientos **Function** y **Sub**.

Relacionados con ellos, he dejado en el tintero completar la explicación de una serie de temas que vamos a tratar en esta entrega.

Entre ellos cabe destacar

- Declaración de variables como **Static**, a nivel de procedimiento
- Paso de parámetros **Por Valor** y **Por Referencia**, **ByVal** y **ByRef**
- Parámetros **Opcionales** **Optional**
- Resolución de problemas mediante Procedimientos **Recursivos** e **Iterativos**.
- Parámetros **Tipo Matriz** mediante **ParamArray**.
- Paso de parámetros **Con Nombre** mediante el operador de asignación :=
- Constantes **Enumeradas** **Enum** . . . **End Enum**
- Cuadros de diálogo predefinidos para intercambiar información entre VBA y el usuario **MsgBox** e **InputBox**

Variables Static

A nivel de procedimiento hemos comentado que una variable puede declararse con **Dim**.

No puede declararse ni como **Public** ni como **Private**.

Declarada con **Dim**, cuando acaba el procedimiento, el contenido de la variable desaparece, y la siguiente vez que se llama al procedimiento, tras su declaración, la variable se reinicia con el valor por defecto correspondiente a su tipo.

Como aclaración de esto último, por ejemplo, si tenemos el procedimiento:

```
Public Sub ValorConDim()  
    Dim n As Long  
    n = n + 1  
    Debug.Print "Valor de n: " & n  
End Sub
```

Cada vez que llamemos al procedimiento `ValorConDim` imprimirá en la ventana Inmediato

```
Valor de n: 1
```

Vamos ahora a escribir un nuevo procedimiento muy parecido al anterior

```
Public Sub ValorConStatic()  
    Static n As Long  
    n = n + 1  
    Debug.Print "Valor de n: " & n  
End Sub
```

Sólo cambia el nombre y la declaración de `n` que está declarada como **Static** en vez de con **Dim**.

Ahora la primera vez que se ejecuta muestra también `Valor de n: 1`

Pero las siguientes veces imprimirá:

```
Valor de n: 2
Valor de n: 3
Valor de n: 4
. . .
```

Comprobamos que cuando una variable es declarada como **Static**, se mantiene el valor que va tomando después de ejecutarse el procedimiento.

Vamos a usar ahora el primer procedimiento, cambiando sólo su cabecera

```
Public Static Sub ValorConStatic2()
    Dim n As Long
    n = n + 1
    Debug.Print "Valor de n: " & n
End Sub
```

Si ejecutamos ahora varias veces seguidas el procedimiento `ValorConStatic2` veremos que se comporta igual que en `ValorConStatic`, a pesar de haber declarado la variable `n` con **Dim**.

Esto ocurre porque al poner delante de **Sub** ó **Function**, la palabra **Static**, hace que las variables declaradas dentro del procedimiento, sean declaradas como **Static**, y por tanto mantengan su valor entre diferentes llamadas al mismo.

Paso de parámetros Por Valor y Por Referencia

A lo largo de los ejemplos de las entregas anteriores, hemos utilizado los argumentos **ByVal** y **ByRef**, en la declaración de los parámetros en la cabecera de los procedimientos.

Para averiguar lo que implica declararlas de una u otra manera vamos a escribir el siguiente código.

```
Public Sub DemoByValByRef()
    Dim lngN As Long, lngM As Long, lngO As Long
    lngM = 1
    lngN = 1
    lngO = 1
    PorValor lngM
    PorReferencia lngN
    PorDefecto lngO
    Debug.Print
    Debug.Print "lngM = " & lngM
    Debug.Print "lngN = " & lngN
    Debug.Print "lngO = " & lngO
End Sub

Public Sub PorValor(ByVal VariableByVal As Long)
    VariableByVal = 2 * VariableByVal
    Debug.Print "VariableByVal = " & VariableByVal
End Sub
```

```
Public Sub PorReferencia(ByRef VariableByRef As Long)
    VariableByRef = 2 * VariableByRef
    Debug.Print "VariableByRef = " & VariableByRef
End Sub
```

```
Public Sub PorDefecto(Variable As Long)
    Variable = 2 * Variable
    Debug.Print "Variable = " & Variable
End Sub
```

Si ejecutamos el procedimiento `DemoByValByRef`, nos imprimirá:

```
VariableByVal = 2
VariableByRef = 2
Variable = 2
```

```
lngM = 1
lngN = 2
lngO = 2
```

Vemos que `lngM` no ha cambiado su valor. Mantiene el valor 1 que se ha pasado al procedimiento `PorValor`.

En cambio tanto `lngM` como `lngO` han pasado de tener el valor 1 a tomar el valor 2.

Esto es así porque cuando se pasa una variable **Por Valor** a un procedimiento, se está haciendo que el procedimiento trabaje con una copia de la variable pasada al procedimiento, no con la variable misma. Por ello, los cambios que hagamos al valor del parámetro en el procedimiento, no afectarán a la variable original.

En cambio, si pasamos una variable a un procedimiento, **Por Referencia**, los cambios que hagamos en el parámetro se verán reflejados en la variable original.

Tanto el **VBA** de **Access** como el **VBA** de **Visual Basic**, o de otras aplicaciones, pasan por defecto los parámetros **Por Referencia**.

Esto no ocurre con otros programas, como **Pascal**, **C** o la plataforma de desarrollo **Net**, con programas como **VB.Net**, y **C#**, en las que la forma por defecto es **Por Valor**.

El hecho de que por defecto se pasen los parámetros por referencia puede dar lugar a errores de código difícilmente detectables. Por ello, y como norma general, recomiendo que cuando se pase un parámetro se indique siempre si es **Por Valor** o **Por Referencia** usando **ByVal** o **ByRef**.

No todo son inconvenientes, el paso de parámetros **Por Referencia**, permite que un procedimiento **Sub** o **Function** modifiquen varios valores simultáneamente.

Supongamos que necesitamos un procedimiento para obtener la potencia n de tres valores de tipo `Long` que se pasan como parámetros.

```
Public Sub ProbarByRef()
    Dim lng1 As Long, lng2 As Long, lng3 As Long
    Dim lngExponente As Long
    lng1 = 2
    lng2 = 3
```

```

lng3 = 4
lngExponente = 4
' Pasamos por referencia las tres variables
TresPotencias lng1, lng2, lng3, lngExponente
' Las variables lng1, lng2 y lng3 _
  han sido elevadas a la potencia 4
Debug.Print "Variable 1 " & lng1
Debug.Print "Variable 2 " & lng2
Debug.Print "Variable 3 " & lng3
End Sub

Public Sub TresPotencias( _
    ByRef Valor1 As Long, _
    ByRef Valor2 As Long, _
    ByRef Valor3 As Long, _
    ByVal Exponente As Long)
    Valor1 = Valor1 ^ Exponente
    Valor2 = Valor2 ^ Exponente
    Valor3 = Valor3 ^ Exponente
End Sub

```

Al ejecutar el procedimiento `ProbarByRef` nos mostrará

```

Variable 1 = 16
Variable 2 = 81
Variable 3 = 256

```

Que son los valores que han tomado `lng1`, `lng2` y `lng3` al elevarlos a la 4ª potencia.

Si en el procedimiento `TresPotencias` hubiéramos definido que los tres primeros parámetros iban a ser pasados Por Valor, `ProbarByRef` nos hubiera mostrado

```

Variable 1 = 2
Variable 2 = 3
Variable 3 = 4

```

Parámetros Opcionales

Tanto en los procedimientos **Function**, como en los **Sub** podemos definir algunos, e incluso todos los parámetros como Opcionales.

¿Qué quiere decir esto?.

- Tan simple como que podemos hacer que la introducción de su valor en un parámetro opcional no sea estrictamente necesaria.

Puntualizaciones sobre parámetros opcionales.

- Para definir que un parámetro es opcional ponemos delante de cualquier otro posible modificador, como **ByVal** ó **ByRef**, la palabra reservada **Optional**.
- Si en un procedimiento hay varios parámetros, y uno de ellos está definido como opcional, los parámetros que le sigan también deberán estar definidos como **Optional**.

- Si en un procedimiento, tenemos varios parámetros opcionales, que no vamos a introducir y varios de ellos son los últimos, y consecutivos, no será necesario introducir ninguna coma que actúe como separador.
- Si alguno de los parámetros que no vamos a introducir está entre dos que sí introduciremos, el espacio ocupado por el/los parámetros no introducidos los pondremos como un espacio en blanco. Por ejemplo:

```

MiProcedimiento Parámetro1, , , Parámetro4

```
- Si el que no vamos a introducir es el primero, pondremos una coma.

```

MiProcedimiento , Parámetro2 , Parámetro3

```
- Si no definimos un valor por defecto, para un parámetro opcional, y en la llamada al procedimiento tampoco se le asigna valor alguno, tomará el valor por defecto del tipo de dato. En los tipos numéricos ese valor será el Cero, en las cadenas de texto la cadena vacía y en los booleanos el valor False.
- Si necesitáramos saber si se ha pasado un valor a un parámetro opcional de tipo **Variant**, utilizaremos la función **IsMissing**.

```

Public Sub VariantOpcional( _
    Optional ByVal Valor As Variant)
    If IsMissing(Valor) Then
        Debug.Print "No se ha asignado Valor"
    Else
        Debug.Print "Se ha asignado un Valor"
    End If
End Sub

```

Si llamamos a este procedimiento sin asignar ningún contenido al parámetro Valor, por ejemplo llamando directamente

```

VariantOpcional

```

nos imprimirá

```

No se ha asignado Valor

```

En cambio si la llamamos

```

VariantOpcional "Parámetro String"

```

Nos mostrará

```

Se ha asignado un Valor

```

Supongamos ahora que tenemos que desarrollar una función que nos devuelva la edad en años de una persona; función que luego queremos utilizar, por ejemplo en consultas.

Para ello creamos un nuevo módulo y en él escribimos lo siguiente.

```

Public Const Pi As Double = 3.14159265358979

Public Function Edad( _
    ByVal Nacimiento As Date, _
    Optional ByVal FechaEdad As Date = Pi)
    ' Esta no es una función absolutamente exacta, _
    pero sí razonablemente operativa.

```

```

' 365.25 es por los años bisiestos
Const DiasAño As Double = 365.25

' Si no se introduce FechaEdad (FechaEdad = Pi) _
  le asignamos Date (la fecha de Hoy).
If FechaEdad = Pi Then
    FechaEdad = Date
End If
' Fix quita la parte decimal y devuelve _
  la parte entera de un número de coma flotante
Edad = Fix((FechaEdad - Nacimiento) / DiasAño)
End Function

```

Supongamos que hoy es el 14 de febrero de 2005

Llamar a la función `Edad(#9/3/53#, #2/14/05#)` sería lo mismo que llamarla con `Edad(#9/3/53#)`; en ambos casos nos devolvería **51**,

Primero la variable `FechaEdad` tomaría el valor **Pi**, por haber sido llamada la función `edad` por no haber pasado ningún valor al parámetro.

A continuación, tras comprobar que su valor es **Pi**, le asigna la **fecha de hoy** mediante la función de VBA `Date()`.

Puede resultar sorprendente la asignación del valor definido para **Pi** como valor por defecto del parámetro `FechaEdad`.

Ésta es una decisión puramente personal, basada en la práctica imposibilidad de que esa sea la fecha/hora en la que se quiera averiguar la edad de una persona.

Además **Pi** es un número por el que tengo “cierta debilidad”.

Por cierto, para el que sienta curiosidad, el valor usado como **Pi** está entre el segundo 53 y el segundo 54 de las 3 horas, 23 minutos del día 2 de enero del año 1900.

Resumiendo lo anterior, podemos definir en los procedimientos parámetros que serán opcionales, e incluso asignarles valores que tomarán si al llamar al procedimiento no se les asignara ningún valor.

Procedimientos Recursivos frente a Iterativos

Procedimientos Iterativos son aquéllos en los que se producen una serie de repeticiones.

Procedimientos recursivos son aquéllos que se van llamando a sí mismos, hasta que encuentran una condición base que permite deshacer todas las llamadas hacia atrás.

Ya sé que ésta es una definición un tanto Farragosa, pero lo entenderemos mejor con un ejemplo.

Vamos a definir la función `Factorial` que nos devolverá el factorial de un número.

Os recuerdo que factorial de un número entero positivo n es:

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

Se define el 1 como valor del factorial de cero $0! = 1$.

Una forma tradicional (iterativa) de resolver este problema sería el que mostramos en la entrega 09:

```
Public Function Factorial(ByVal n As Integer) As Long
    Dim i As Integer
    If n < 0 Then Exit Function
    Factorial = 1
    For i = 1 To n
        Factorial = Factorial * i
    Next i
End Function
```

En esta función no hay puesto un control de errores; de hecho para un valor de n superior a 12 el resultado supera el rango de los Long y dará error de desbordamiento.

Obvio estos detalles de cara a la simplificación del código, aunque en un desarrollo profesional esto sería imperdonable...

Este código es un claro caso de resolución mediante un procedimiento iterativo.

Por si sirve para aclarar conceptos, en el diccionario se define la iteración como el acto de repetir.

Vemos que en este procedimiento se repite una operación hasta obtener el valor que deseamos, utilizando para ello un bucle que va **iterando** ó **repitiendo** la operación producto, hasta que la variable i obtiene el valor de n .

La definición del factorial de un número n se puede hacer de otra manera:

$$n! = n * (n-1)! \text{ y el valor de } 0! \text{ es } 1$$

Nos dice que el factorial de un número n es igual a ese número multiplicado por el factorial de $n-1$.

Podemos realizar una función que actúe tal cual está definido el factorial de n :

```
Public Function Factorial( _
    ByVal n As Integer _
) As Long
    Dim i As Integer
    If n < 0 Then Exit Function
    If n = 1 Then
        Factorial = 1
        Exit Function
    End If
    Factorial = n * Factorial(n - 1)
End Function
```

Vemos que la función factorial se va llamando a sí misma:

$$\text{Factorial} = n * \text{Factorial}(n - 1)$$

Cada resultado parcial de `Factorial` lo va guardando en una posición de memoria.

Esto ocurre hasta que en alguna de las llamadas se encuentra con que el parámetro $n-1$ pasado a la función `Factorial` toma el valor 1.

En general un método recursivo se va llamando a sí mismo hasta que encuentra una solución elemental ó básica que le permite construir el proceso inverso y salir del mismo.

`Factorial(n - 1)` devolverá el valor 1 tras encontrarse con la línea en la que compara el valor de `n`. A partir de ese momento va realizando de forma inversa las operaciones con los resultados parciales anteriores de `factorial`.

Es decir calcula $1 * 2 * 3 * \dots * (n-1) * n$.

En este caso el método recursivo, para calcular el factorial de un número, necesita efectuar `n` llamadas a la función, almacenando los correspondientes resultados parciales en memoria.

Podemos deducir en este caso, que el método recursivo requiere más recursos de memoria y procesador que el método iterativo.

Hay una definición que afirma que cualquier método recursivo se puede transformar en iterativo, y viceversa.

Generalmente los métodos recursivos requieren más memoria y procesador que sus equivalentes iterativos. Entonces ¿para qué usarlos?

Hay problemas en los que una resolución iterativa es muchísimo más compleja que su correspondiente recursiva.

Apunto aquí un problema clásico como es el de “Las torres de Hanoi”, un juego de origen oriental que consiste en tres barras verticales y un conjunto de discos de diferentes diámetros con un orificio central, apilados en una de ellas y que hay que ir pasando a otra de las barras.

No voy a explicar la mecánica del juego ni la resolución del mismo. Si alguno tiene interés en conocerlo, Google, MSN, Altavista o cualquiera de los buscadores os indicarán una inmensa cantidad de páginas con información sobre el juego, el problema matemático que representa y sus algoritmos para resolverlo.

La solución Recursiva es casi “elemental” en cambio la iterativa es mucho más compleja.

Esto mismo es aplicable a ciertos algoritmos, por ejemplo de ordenación o de trazado gráfico de figuras englobadas en la “geometría fractal”.

Pasar un parámetro, tipo matriz, mediante ParamArray

En una entrega anterior, vimos que se podía asignar a un parámetro de tipo **Variant**, una matriz, y que incluso una función podía devolver una matriz mediante un **Variant**.

Supongamos que queremos definir una función que nos calcule la media de una serie de valores. Podemos utilizar una alternativa frente a las variables **Variant**.

Por definición, y de entrada, no sabemos cuántos valores vamos a manejar.

VBA nos permite resolverlo mediante la siguiente función utilizando **ParamArray**:

```
Public Function Media( _
    ParamArray Sumandos() As Variant _
) As Double
    Dim dblSuma As Double
    Dim lngElementos As Long
    Dim n As Variant
    For Each n In Sumandos
        dblSuma = dblSuma + n
    Next n
    lngElementos = UBound(Sumandos) + 1
```

```

    Media = dblSuma / lngElementos
End Function

```

Si desde la ventana inmediato escribimos

```
? media(1,2.3,5,6,9,8)
```

Nos mostrará 5,21666666666667, que efectivamente es la media de esos 6 valores.

Puntualizaciones:

- **Ubound** nos devuelve el valor más alto del índice de una matriz; como en este caso el índice más bajo de Sumandos es 0, el número de elementos de la matriz es el devuelto por **Ubound** más 1.
- Si en un procedimiento vamos a utilizar **ParamArray**, la matriz correspondiente será el último parámetro del procedimiento.
- **ParamArray** no puede usarse con **ByVal**, **ByRef** u **Optional**.

Uso de parámetros Con Nombre

Supongamos que hemos definido un procedimiento **Function** con la siguiente cabecera:

```

Public Sub ConNombre( _
    Optional ByVal Nombre As String, _
    Optional ByVal Apellido1 As String, _
    Optional ByVal Apellido2 As String, _
    Optional ByVal Sexo As String)

```

En un momento dado debemos pasar el dato "Manolo" y "Masculino".

La forma "habitual" de hacerlo sería

```
ConNombre "Manolo", , , "Masculino"
```

Esto genera un código "confuso".

Para facilitar la introducción de los parámetros ó argumentos, tenemos la posibilidad de efectuar la introducción **Con ó Por Nombre**.

En la introducción por nombre se indica el Nombre del parámetro, seguido del operador de asignación (dos puntos seguido del signo igual) En el caso anterior se haría así:

```
ConNombre Nombre := "Manolo", Sexo:= "Masculino"
```

Como curiosidad indicaré, que este operador es el mismo que el de asignación del lenguaje Pascal.

- El uso de parámetros con nombre se puede usar tanto en procedimientos **Function** como **Sub**
- Habrá que poner todos los argumentos no opcionales
- El orden de colocación de los parámetros es libre
- Algunos de los objetos de VBA no admiten parámetros con nombre
- El uso de parámetros con nombre está especialmente indicado en procedimientos que tengan múltiples parámetros opcionales.

Constantes Enumeradas

En algunos tipos de procedimientos o funciones, vemos que alguno de los parámetros sólo debe tomar un número limitado de valores. Sería muy interesante que a la hora de ir escribiendo un procedimiento, VBA nos fuera sugiriendo los valores válidos.

Para ayudar a este fin tenemos las **Constantes Enumeradas**.

Las Constantes Enumeradas son un conjunto de constantes agrupadas en un tipo enumerador. Los valores que pueden admitir son sólo del tipo **Long**

La forma de construirlas es

```
[Public | Private] Enum nombre
    nombre_miembro [= expresión_constante]
    nombre_miembro [= expresión_constante]
    . . .
End Enum
```

Se declaran a nivel del **encabezado de un módulo**, pudiendo ser declaradas como **Public** o **Private**. En los módulos de clase sólo pueden declararse como **Public**

La declaración comienza opcionalmente indicando primero su alcance a continuación la instrucción Enum seguida del nombre del tipo.

En las sucesivas líneas se van añadiendo los nombres de los sucesivos valores, opcionalmente con su valor correspondiente.

```
Public Enum Booleano
    blFalso
    blCierto
End Enum
```

En este caso blFalso tendría el valor 0 y blCierto tomaría el valor 1.

Podríamos haber hecho

```
Public Enum Booleano
    blFalso
    blCierto = -1
End Enum
```

Con lo que blFalso tendría el valor 0 y blCierto tomaría el valor -1.

Las constantes enumeradas, si no se les asigna un valor, toma el de la constante anterior, incrementada en 1.

Ya hemos dicho que si a la primera no se le asigna un valor toma el valor 0.

```
Public Enum DiasSemana
    dsLunes = 1
    dsMartes
    dsMiercoles
    dsJueves
    dsViernes
    dsSabado
    dsDomingo
```

```
End Enum
```

En este caso, por ejemplo `dsJueves` toma el valor 4.

He dicho que a las constantes enumeradas sólo se les puede asignar valores Long.

Esto no es del todo cierto. Esto por ejemplo funcionaría

```
Public Enum TipoSexo
    tsFemenino = "1"
    tsMasculino
End Enum
```

En esta declaración `tsFemenino` tendría el valor numérico 1 y `tsMasculino` el valor 2.

- ¿Y cómo es posible esto?

- Forma parte de las “pequeñas incoherencias” que tiene Visual Basic heredadas de las “viejas versiones”; incoherencias que en teoría son para ayudar al usuario, pero que pueden acabar despistándolo.

Uso de las Constantes Enumeradas

Supongamos que tenemos que crear una función, para una empresa de “Relaciones Interpersonales” que nos escriba el tratamiento que debemos escribir en la cabecera de las cartas dirigidas a sus “socios”.

La empresa define que a todos los hombres se les debe tratar como

Estimado Sr. Apellido

A las mujeres,

Si son solteras ó divorciadas

Estimada Sta. Apellido

En el resto de los casos

Estimada Sra. Apellido

Resultaría interesante que al escribir, por ejemplo este procedimiento, nos mostrara las posibilidades que podemos utilizar para cada parámetro.

Vamos a definir las constantes enumeradas correspondientes a cada caso.

Podrían ser algo así como:

```
Public Enum TipoSexo
    tsFemenino = 1
    tsMasculino = 2
End Enum
```

```
Public Enum EstadoCivil
    ecSoltero
    ecCasado
    ecSeparado
    ecDivorciado
    ecViudo
```

End Enum

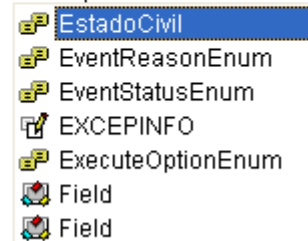
Como hemos comentado asignar el valor 2 a `tsMasculino` no sería necesario.

Una vez definidas estas constantes, están inmediatamente disponibles en la Ayuda Contextual, como podemos ver en la siguiente imagen, pudiéndose utilizar como un tipo de datos más.

```
Public Enum TipoSexo
    tsFemenino = 1
    tsMasculino = 2
End Enum
```

```
Public Enum EstadoCivil
    ecSoltero
    ecCasado
    ecSeparado
    ecDivorciado
    ecViudo
End Enum
```

```
Public Function Tratamiento( _
    ByVal Apellido As String, _
    Sexo As TipoSexo, _
    Optional ByVal Estado As Es|
End Function
```



Además, conforme vayamos desarrollando el código nos mostrará los posibles valores concretos que podrán tomar los parámetros

```
Public Function Tratamiento( _
    ByVal Apellido As String, _
    Sexo As TipoSexo, _
    Optional ByVal Estado As EstadoCivil _
) As String

    If Sexo = tsMasculino Then
    ElseIf Sexo = |
        tsFemenino
        tsMasculino
    End If
End Function
```

Completemos la función:

```
Public Function Tratamiento( _
    ByVal Apellido As String, _
    Sexo As TipoSexo, _
    Optional ByVal Estado As EstadoCivil _
) As String
```

```

If Sexo = tsMasculino Then
    Tratamiento = "Estimado Sr. " & Apellido
ElseIf Sexo = tsFemenino Then
    Select Case Estado
        Case ecSoltero, ecDivorciado
            Tratamiento = "Estimada Sta. " & Apellido
        Case Else
            Tratamiento = "Estimada Sra. " & Apellido
    End Select
Else
    Tratamiento = "Estimado/a Sr./Sra. " & Apellido
End If
End Function

```

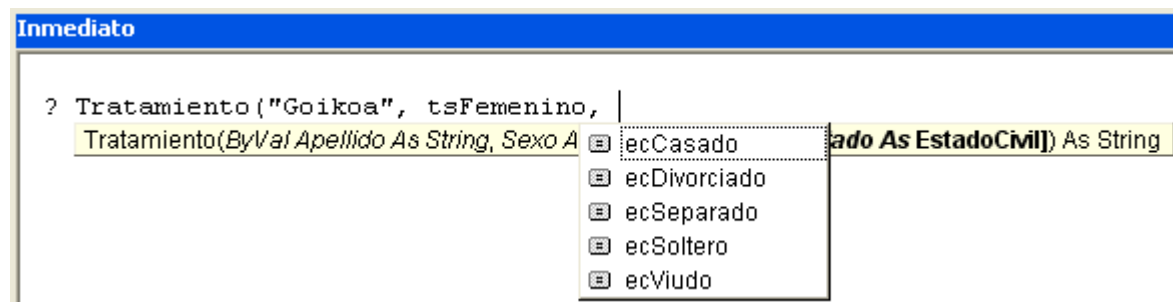
Con esta función si desde la ventana inmediato escribimos

```
? Tratamiento("Goikoa", tsFemenino, ecCasado)
```

Nos escribirá

```
Estimada Sra. Goikoa
```

Conforme escribimos la función nos irá apareciendo la ayuda



Si hacemos las pruebas con las diferentes posibilidades veremos que responde a lo solicitado por la empresa contratante.

Visto el éxito, nos piden que también creamos una función para la línea de la dirección de forma que nos ponga:

```
A la Att. de Don Nombre Apellido1
```

```
A la Att. de Doña Nombre Apellido1
```

Si no existiera el Nombre que pusiera

```
A la Att. del Sr. Apellido1
```

```
A la Att. de la Sta. Apellido1
```

```
A la Att. de la Sra. Apellido1
```

El desarrollo de esta función la dejo para el lector.

Para intercambiar información entre el usuario y VBA tenemos 2 procedimientos básicos que se corresponden a los cuadros de diálogo predefinidos en VBA.

- **MsgBox**
- **InputBox.**

Función MsgBox

La función **MsgBox**, muestra un mensaje en un Cuadro de diálogo con determinados botones, y puede devolver un número que identifica el botón pulsado por el usuario.

En este cuadro de diálogo se puede definir el texto de la barra de título, el mensaje mostrado, un icono que indica el tipo de mensaje y seleccionar botones de los predefinidos previamente por VBA.

La sintaxis es la siguiente

```
MsgBox(prompt[, buttons][, title][, helpfile, context])
```

Es decir

```
MsgBox(mensaje[, botones][, título][, ficheroayuda, contextoayuda])
```

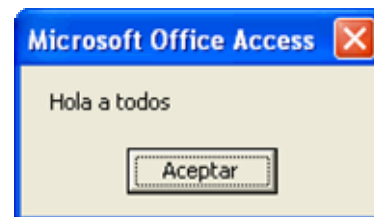
La forma más básica posible sería

```
MsgBox mensaje
```

Por ejemplo

```
MsgBox "Hola a todos"
```

Esta instrucción mostraría el siguiente mensaje



He hecho una llamada a **MsgBox** como si fuera un procedimiento de tipo **Sub**, no como una función.

En este caso tendría sentido ya que al haber sólo un botón, MsgBox sólo va a devolver un valor.

MsgBox admite los parámetros, ó argumentos **Con Nombre**. Así el código anterior podría haberse escrito

```
MsgBox Prompt:= "Hola a todos"
```

El valor devuelto dependerá del botón que se haya pulsado.

El parámetro **Buttons** indica por una parte el tipo de icono que debe mostrar el Cuadro de diálogo ó cuadro de Mensaje y los botones que va a contener.

Supongamos que en un formulario hemos puesto un botón que al presionarlo hará que se formatee el disco C:

No hace falta ser muy listo para pensar que existe la posibilidad de una pulsación accidental.

Para prevenir este caso. antes de lanzar el proceso de formateo, podríamos hacer lo siguiente.

```
Public Sub Formateo()  
Dim strMensaje As String
```

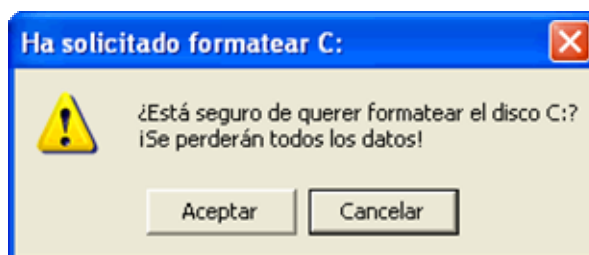
```

Dim lngRespuesta As Long
strMensaje = _
"¿Está seguro de querer formatear el disco C:?" _
    & vbCrLf _
    & "¡Se perderán todos los datos!"
lngRespuesta = MsgBox(strMensaje, _
    vbExclamation + vbOKCancel + vbDefaultButton2, _
    "Ha solicitado formatear C:")
Select Case lngRespuesta
    Case vbOK
        Formatear "C"
    Case vbCancel
        MsgBox Prompt:= _
            "Ha interrumpido el Formateo del disco", _
            Buttons:=vbInformation, _
            Title:="Ha salvado sus datos"
    Case Else
        MsgBox lngRespuesta
End Select
End Sub

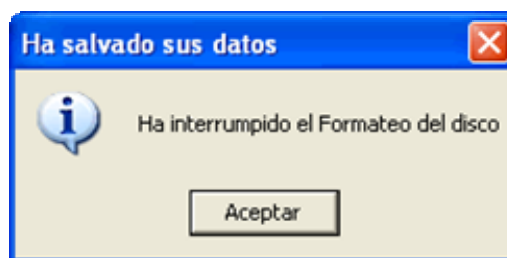
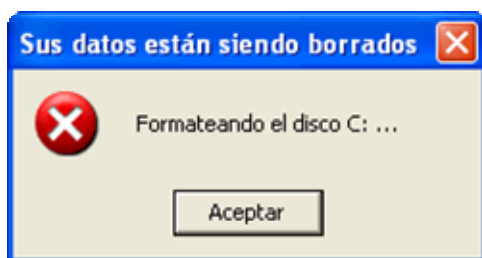
Public Sub Formatear(ByVal Disco As String)
    MsgBox Prompt:="Formateando el disco " _
        & Disco & ": ...", _
        Title:="Sus datos están siendo borrados"
End Sub

```





Este código mostrará el siguiente mensaje:



Dependiendo del botón pulsado mostrará



El tipo de icono que se va a mostrar lo dan las siguientes constantes de VBA.

Constante	Valor	Icono
VbInformation	64	
VbExclamation	48	
VbQuestion	32	
VbCritical	16	

La segunda parte del parámetro **Buttons** hace referencia a los botones que va a mostrar el cuadro de mensaje.

Sus posibilidades son

Constante	Valor	Botones
VbOKOnly	0	[Aceptar]
VbOKCancel	1	[Aceptar] [Cancelar]
VbAbortRetryIgnore	2	[Anular] [Reintentar] [Ignorar]
VbYesNoCancel	3	[Sí] [No] [Cancelar]
VbYesNo	4	[Sí] [No]
VbRetryCancel	5	[Reintentar] [Cancelar]
VbMsgBoxHelpButton	16384	Añade un botón [Ayuda] a los anteriores

El tercer grupo de constantes es el que define el botón seleccionado como predeterminado.

Será el que se active si directamente presionamos la tecla [**Intro**]

Constante	Valor	Botón predeterminado
VbDefaultButton1	0	Primer botón
VbDefaultButton2	256	Segundo botón
VbDefaultButton3	512	Tercer botón
VbDefaultButton4	768	Cuarto botón

El cuarto grupo de constantes define el tipo de presentación Modal del cuadro.

La presentación modal hace que tome prioridad el cuadro frente al resto de la aplicación, visualizándose en primer plano.

Constante	Valor	Tipo de visualización modal
VbApplicationModal	0	El cuadro es modal frente a la aplicación
VbSystemModal	4096	El cuadro es modal frente al resto de las aplicaciones

Hay una constante adicional **VbMsgBoxRight** que sirve para hacer que el texto se alinee a la derecha del mensaje. Su valor es 524288.

Las dos últimas constantes que permite usar MsgBox son **VbMsgBoxSetForeground** y **VbMsgBoxRtlReading**.

La primera define la ventana del cuadro como de primer plano, y la segunda afecta a la visualización de los textos en hebreo y árabe que se escriben de Derecha a Izquierda.

Usando las constantes:

Supongamos que queremos mostrar un mensaje con los botones

[Sí] [No] [Cancelar]

Como icono deberá tener el de **Exclamación**



Además queremos que su título sea **Proceso de Votación**

El mensaje que aparecerá será **Seleccione su voto**

El botón preseleccionado deberá ser el de [**Cancelar**].

Tras seleccionar una opción nos mostrará un mensaje informándonos del voto realizado.

Vamos a escribir el código que responde a estos requisitos:

```
Public Sub Votacion()
    Dim lngBotonesIcono As Long
    Dim strTitulo As String
    Dim strMensaje As String
    Dim strVoto As String
    Dim lngRespuesta As Long

    lngBotonesIcono = vbExclamation _
        + vbYesNoCancel _
        + vbDefaultButton3
    strTitulo = "Proceso de Votación"
    strMensaje = "Seleccione su voto"

    lngRespuesta = MsgBox( _
        strMensaje, _
        lngBotonesIcono, _
        strTitulo)

    ' Tras pasar el cuadro el botón seleccionado _
    ' a la variable lngRespuesta _
    ' mostramos el voto
    ' Para ello usamos la función TeclaPulsada
    strVoto = TeclaPulsada(lngRespuesta)
    If lngRespuesta = vbCancel Then
        strMensaje = "Has cancelado la votación"
    Else
        strMensaje = "Has votado: " _
            & TeclaPulsada(lngRespuesta)
    End If
End Sub
```

```
MsgBox strMensaje, _  
        vbInformation, _  
        "Resultado de la votación"  
End Sub
```

La función **NombreTecla**, llamada desde el procedimiento **Votacion**, sería la siguiente:

```
Public Function NombreTecla( _  
        ByVal Tecla As Long _  
        ) As String  
    Select Case Tecla  
        Case vbOK  
            NombreTecla = "Aceptar"  
        Case vbCancel  
            NombreTecla = "Cancelar"  
        Case vbAbort  
            NombreTecla = "Anular"  
        Case vbRetry  
            NombreTecla = "Reintentar"  
        Case vbIgnore  
            NombreTecla = "Ignorar"  
        Case vbYes  
            NombreTecla = "Sí"  
        Case vbNo  
            NombreTecla = "No"  
        Case Else  
            NombreTecla = "Desconocida"  
    End Select  
End Function
```

Las constantes aquí manejadas no son un caso particular del cuadro de mensaje.

Este tipo de constantes son muy habituales en VBA, y funcionan igual que las constantes enumeradas que hemos visto en esta misma entrega.

En concreto las últimas que hemos utilizado están definidas internamente como de los tipos enumerados **VbMsgBoxStyle** y **VbMsgBoxResult**.

Función InputBox

La función **InputBox**, al igual que **MsgBox**, puede mostrar un mensaje, pero se diferencia en que en vez de enviar un número Long en función de la tecla pulsada, nos pide que escribamos un texto, en un cuadro de texto. Su contenido será el dato que devuelva.

La sintaxis de **InputBox** es:

```
InputBox(prompt[, title][, default][, xpos][, ypos][,  
helpfile, context])
```

Parámetro	Descripción
<i>prompt</i>	Mensaje del cuadro de diálogo (obligatorio)
<i>title</i>	Texto a visualizar en la barra de título
<i>default</i>	Valor que devolverá, si no introducimos ningún texto
<i>xpos</i>	Coordenada X del cuadro (distancia desde el lado izquierdo)
<i>ypos</i>	Coordenada Y del cuadro (distancia desde arriba)
<i>helpfile</i>	Fichero de ayuda auxiliar
<i>context</i>	Contexto del fichero de ayuda

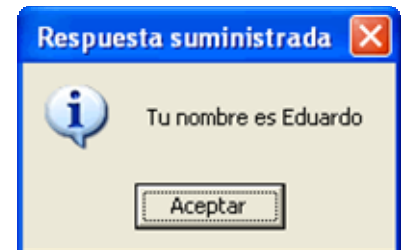
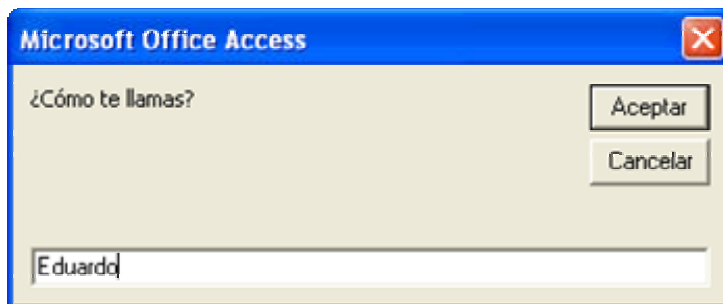
La forma más básica sería **inputbox ("Mensaje")**

Vamos a escribir el siguiente código

```
Public Sub Pregunta()
    Dim strRespuesta As String

    strRespuesta = InputBox("¿Cómo te llamas?")
    MsgBox "Tu nombre es " & strRespuesta, _
        vbInformation, " Respuesta suministrada"
End Sub
```

Si ejecutamos el procediendo nos muestra



Tras presionar el botón [Aceptar] el **MsgBox** nos mostrará el texto introducido en el **InputBox**.

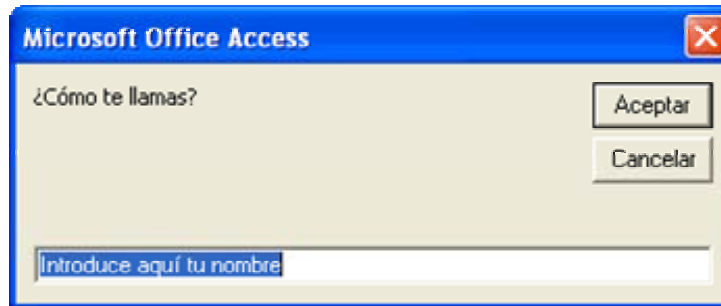
Si hubiéramos introducido en el InputBox el texto por defecto, ya fuera mediante

```
strRespuesta = InputBox( _
    "¿Cómo te llamas?", , _
    "Introduce aquí tu nombre")
```

O mediante

```
strRespuesta = InputBox( _
    Prompt:="¿Cómo te llamas?", _
    Default:="Introduce aquí tu nombre")
```

El InputBox, nos mostraría



Si no borramos ese texto, por ejemplo al introducir algún texto, éste será el texto que devolverá **InputBox**.

Al ejecutar ese código podemos comprobar que el cuadro de entrada nos aparece en la práctica centrado en la pantalla.

InputBox tiene dos parámetros que le permiten ubicar el cuadro en una posición concreta. Estos parámetros son Xpos e Ypos.

He modificado el código del procedimiento para que muestre un título y ubicarlo cerca de la posición superior izquierda

```
Public Sub Pregunta()
    Dim strRespuesta As String

    strRespuesta = InputBox( _
        Prompt:="¿Cómo te llamas?", _
        Default:"Introduce aquí tu nombre", _
        Title:"¿Escriba su nombre?", _
        XPos:=500, YPos:=500)
    MsgBox "Tu nombre es " & strRespuesta, _
        vbInformation, " Respuesta suministrada"
End Sub
```

Las medidas de posición se dan en **Twips**.

Un **Twip**, **Twentieth of a Point** equivale a la Vigésima parte de un punto.

El Twip equivale a **1/1440 de pulgada**, es decir, **567 twips** equivalen a un centímetro.

Esto es válido para la teoría.

En la vida real, las pruebas con diferentes valores son las que nos indicarán qué valores serán los adecuados a cada caso.

Nota:

Los parámetros *helpfile* y *context* hacen referencia a ficheros de ayuda desarrollados al efecto para la aplicación.

En este nivel de estudio (básico), no vamos a abordar el análisis del desarrollo de ficheros de ayuda.

Esto no es óbice para indicar que, tanto con **MsgBox** como con **InputBox**, se puede hacer referencia a determinados contenidos de ficheros de ayuda desarrollados ex profeso para la aplicación, y no sólo al fichero en sí, sino a una sección concreta del mismo.

Hay abundante documentación disponible en Internet al respecto.

Sólo quiero reseñar el magnífico trabajo desarrollado por “mi amigo Guille” en su página:

http://www.elquille.info/vb/VB_HLP.HTM

En la próxima entrega examinan algunos de los **operadores** disponibles con VBA.

Trataremos también funciones y procedimientos implementados en VBA.

Entre ellas incluiremos

- Funciones para el tratamientos de cadenas
 - Left
 - Mid
 - Right
 - InStr
 -
- Funciones de conversión de datos
- Funciones para el tratamientos de fechas
-

Comencemos a programar con
VBA - Access

Entrega **13**

Funciones de VBA

Funciones propias de VBA

VBA incluye un gran número de Funciones y Procedimientos propios que nos simplificarán las tareas de programación.

Funciones para la toma de decisiones

Ya hemos visto en la entrega 9 la estructura de decisión **If - Then - Else - EndIf**, la función **IIF**. Y la estructura **Select - Case - End Select**.

VBA posee funciones adicionales que nos servirán para la toma de decisiones en el código.

Función Choose

La función **Choose**, selecciona y devuelve un valor de entre una lista de argumentos.

Sintaxis

```
Choose(índice, opción-1 [, opción-2, ... [, opción-n]])
```

Choose busca el argumento situado en la posición definida por el índice.

Si índice fuese menor que 1, ó mayor que el total de argumentos, devolvería el valor **Null**.

Si *índice* fuese un número no entero, lo redondearía al entero más próximo.

En este ejemplo hemos definido, como constantes enumeradas los posibles puestos de una hipotética empresa.

Las constantes enumeradas contienen un valor long, por lo que vamos a desarrollar una función que nos devuelva, en base a su valor, la descripción literal del cargo.

```
Public Enum Puesto
    eoEmpleado = 1
    eoTecnico
    eoAdministrativo
    eoMandoIntermedio
    eoComercial
    eoDirectivo
    eoGerente
End Enum

Public Function PuestoDeTrabajo( _
    ByVal Cargo As Puesto _
) As String
    ' Comprobamos si Cargo está en un rango válido
    If Cargo < eoEmpleado Or Cargo > eoGerente Then
        PuestoDeTrabajo = ""
    Else
        PuestoDeTrabajo = Choose(Cargo, _
            "Empleado", _
            "Técnico", _
            "Administrativo", _
            "Mando Intermedio", _
```



```

        "Comercial", _
        "Directivo", _
        "Gerente")

    End If
End Function

```

Si escribimos

```
PuestoDeTrabajo (eoMandoIntermedio)
```

Nos devolverá

```
Mando Intermedio
```

Los argumentos de la función **Choose**, en realidad son un **ParamArray**, como el que vimos en la entrega anterior.

Función Switch

La función **Switch**, evalúa una lista de expresiones y devuelve un valor, o una expresión asociada, a la primera expresión de la lista que sea cierta.

Su sintaxis es

```
Switch(expresión-1, valor-1[, expresión-2, valor-2 ... [,
expresión-n,valor-n]])
```

Supongamos que queremos hacer una función que recibiendo como parámetro el nombre de un país nos devuelva su capital.

```

Public Function Capital(ByVal Pais As String) As String
    Dim varPais As Variant
    ' Uso un variant porque si Switch no encuentra _
    ' una expresión cierta, devuelve Null
    Pais = Trim(Pais)
    varPais = Switch(Pais = "España", "Madrid", _
        Pais = "Francia", "París", _
        Pais = "Portugal", "Lisboa", _
        Pais = "Italia", "Roma", _
        Len(Pais) = 0, "Tienes que introducir algo", _
        IsNumeric(Pais), "Los números no valen")
    If IsNull(varPais) Then
        Capital = "No conozco la capital de " & Pais
    Else
        Capital = varPais
    End If
End Function

```

En cada caso devolverá:

```

Capital("Italia") → "Roma"
Capital(3.1416) → "Los números no valen"
Capital("Japón ") → "No conozco la capital de Japón"

```

```
Capital(" ") → "Tienes que introducir algo"
```

Como puede comprobarse en la función anterior, **Switch** permite un tipo de código simple que posibilita evaluar expresiones de diferentes tipos .

No obstante, para las funciones **Choose**, **Switch** y la función **IIF** que vimos en la entrega 09 hay que tener en cuenta que su velocidad de ejecución es inferior a la de **If - Then** o **Select - Case**. Por ello, en aquellos procedimientos en los que el tiempo de ejecución puede ser crítico, es mejor no usarlas.

Función Format

La función format recibe una expresión que contiene una cadena, un valor numérico, una fecha, un valor booleano ó un valor nulo, y devuelve una cadena formateada de acuerdo a las instrucciones contenidas en la cadena formato.

Su sintaxis es

```
Format(expresión[, formato[, PrimerDíaDeLaSemana[,  
PrimeraSemanaDelAño]])
```

Utilización con cadenas String.

Aunque en VBA existen otras funciones más potentes para el manejo de cadenas, funciones que veremos próximamente, **Format** permite algunos tipos de formateo.

```
Format("Cadena a mayúsculas", ">") "CADENA A MAYÚSCULAS"
```

Si usamos como cadena de formato "<" devuelve la cadena convertida a minúsculas.

```
Format("Cadena a MINÚSCULAS", "<") " cadena a minúsculas "
```

Si usamos @ **Format**("Cadena", "@@@@@@@@@@@@@@")

Si en la posición donde está el signo @ hay un carácter, se presentará éste, en caso contrario pondrá un espacio en blanco. La cadena se mostrará alineada a la derecha

```
Format("Eduardo", "@@@@@@@@@@@@@@") Devolverá
```

 Eduardo He puesto el subrayado para mostrar los espacios en blanco.

Si ponemos delante de la cadena de formato el carácter ! hará que se alinee a la izquierda.

Otro carácter que podemos utilizar es &. Se diferencia de @ en que si en la posición donde está el signo & hay un carácter, se presentará éste, en caso contrario no pondrá nada.

Si la cadena con & es precedida del carácter "!" recortará la parte izquierda de la cadena.

Si ejecutamos el procedimiento PruebaFormat

```
Public Sub PruebaFormat()  
    Debug.Print Format("Esta cadena a mayúsculas", ">")  
    Debug.Print Format ("ESTA CADENA A minúsculas", "<")  
    Debug.Print "#" & Format("Derecha", "@@@@@@@@@@@@@@") & "#"  
    Debug.Print "#" & Format("Izquierda", "!@@@@@@@@@@@@@") & "#"  
    Debug.Print "#" & Format("Cadena Sin Cortar", "&&&&&") & "#"  
    Debug.Print "#" & Format("Cadena Sin Cortar", "@@@@@@") & "#"  
    Debug.Print "#" & Format("Cadena Cortada", "! " & "&&&&&&&") & "#"  
    Debug.Print "#" & Format("Eduardo", "@ @ @ @ @ @ @ @ @") & "#"  
    Debug.Print "#" & Format("Eduardo", "!@ @ @ @ @ @ @ @ @") & "#"
```

End Sub

Nos mostrará en la ventana Inmediato

```

ESTA CADENA A MAYÚSCULAS
esta cadena a minúsculas
# Derecha#
#Izquierda #
#Cadena Sin Cortar#
#Cadena Sin Cortar#
#Cortada#
# E d u a r d o#
#E d u a r d o #

```

Utilización con fechas.

Format está especialmente preparada para manejar valores numéricos y en concreto el formato de fechas.

Como cadena de formato admite tanto una cadena de caracteres como la llamadas cadenas con nombre.

Es precisamente para el formateo de fechas donde adquieren sentido los dos últimos parámetros opcionales de la función **Format**.

Recordemos su sintaxis:

```

Format(expresión[, formato[, PrimerDíaDeLaSemana[,
PrimeraSemanaDelAño]])

```

El parámetro *PrimerDíaDeLaSemana* sirve para indicar a la función **Format** qué día de la semana vamos a considerar como el primero.

A diferencia de España, en Estados Unidos se considera el Domingo como el primer día de la semana, y ésta es la opción por defecto.

En España se considera como primer día de la semana el lunes.

Se puede introducir, como primer día de la semana, cualquiera de estos valores

Constante	Valor	Significado
vbUseSystem	0	Primer día definido por el sistema
vbSunday	1	Domingo (Valor por defecto)
vbMonday	2	Lunes
vbTuesday	3	Martes
vbWednesday	4	Miércoles
vbThursday	5	Jueves
vbFriday	6	Viernes
vbSaturday	7	Sábado

El definir cuál es el primer día de la semana, puede afectar al último parámetro *PrimeraSemanaDelAño*.

VBA tiene varias formas de considerar cuál es la primera semana del año.

En concreto puede tomar 4 criterios diferentes, configurables mediante 4 constantes.

Constante	Valor	Significado
vbUseSystem	0	Primer semana definida por el sistema
vbFirstJan1	1	Será la que contenga al día 1 de enero
vbFirstFourDays	2	La primera que contenga al menos 4 días.
vbFirstFullWeek	3	La primera que contenga los 7 días

La configuración de estos dos parámetros afecta al resultado de **Format**, si queremos mostrar el número de semana de una fecha, mediante la cadena de formato **"ww"**.

```
format(#12/30/05#, "ww", vbSunday, vbFirstJan1) → 53
```

```
format(#12/30/05#, "ww", vbMonday, vbFirstFullWeek) → 52
```

```
format(#12/30/05#, "ww", vbSunday, vbFirstFourDays) → 52
```

Podemos combinar cadenas de formato. Por ejemplo si quisiéramos obtener el número de semana de una fecha poniéndole delante el año con cuatro cifras.

```
format(#12/30/05#, "yyyyww", vbSunday, vbFirstFourDays) → 200552
```

```
format(#12/30/05#, "yyyy-ww", vbSunday, vbFirstFourDays) → 2005-52
```

Los caracteres que podemos poner para formatear una fecha son

Resultados de aplicar Format **Format** (#2/8/05 9:5:3#, Cadena)

Fecha/Hora 9 horas 5 minutos 3 segundos del 8 de febrero de 2005

Cadena	Resultado	Descripción
"d"	"8"	Día del mes con 1 ó 2 dígitos
"dd"	"08"	Día del mes con 2 dígitos
"w"	vbMonday:"2", por defecto:"3"	Número de día de la semana
"ddd"	"mar"	Nombre del día con 3 caracteres
"dddd"	"martes"	Nombre del día de la semana
"dddddd"	"08/02/2005"	Fecha corta
"dddddd"	"martes 8 de febrero de 2005"	Fecha larga
"m" ó "mm"	"2" ó "02"	Número del mes con 1 ó 2 dígitos
"mmm"	"feb"	Nombre del mes con 3 caracteres
"mmmm"	"febrero"	Nombre del mes
"q"	"1"	Trimestre del año
"y"	"39"	Día del año
"yy"	"05"	Año con 2 dígitos

"yyyy"	"2005"	Año con 4 dígitos
"h" ó "hh"	"9" ó "09"	Hora con 1 ó 2 dígitos
"n" ó "nn"	"5" ó "05"	Minutos con 1 ó 2 dígitos
"s" ó "ss"	"3" ó "03"	Segundos con 1 ó 2 dígitos
"ww"	vbMonday vbFirstFourDays "6" (Por defecto da "7")	Número de semana del año
"h/n/s"	"9/5/3"	
"h:nn:ss"	"9:05:03"	

Si quisiéramos mostrar la hora en formato de 12 horas, seguido de **AM** ó **PM**, **A** ó **P**

`Format(#2/8/05 19:5:3#, "h:nn:ss AM/PM")` → 7:05:03 PM

`Format(#2/8/05 19:5:3#, "h:nn:ss A/P")` → 7:05:03 P

Podemos poner en la cadena de formato texto que queramos aparezca en el resultado final.

Para que no nos den problemas los caracteres que se pueden usar como formato y los interprete como texto se les pone la barra invertida delante.

`Format(Now, "Pa\mplo\na, a d \de mmmm \de yyyy")`

Nos devolverá algo así como:

Pamplona, a 18 de febrero de 2005

En cambio, si hubiéramos escrito

`Format(Now, "Pamplona, a d de mmmm de yyyy")`

Nos hubiera devuelto algo tan incoherente como

Pa2plo57a, a 18 18e febrero 18e 2005

Esto es así porque interpreta que las "m" son para poner el mes, las "d" para poner el día y las "n" están para indicar dónde poner los segundos.

Cadenas con nombre

Las cadenas **Con Nombre** son unas cadenas predefinidas en VBA, que nos ahorran el construir la secuencia de caracteres en la cadena de formato.

Tienen la ventaja de que son genéricas y se adaptan a la configuración regional de Windows que tenga el usuario, con lo que las podemos usar para formatos internacionales.

Existen tanto para el manejo de fechas como de números.

Los resultados siguientes son propios de la configuración de mi ordenador.

En otros ordenadores el resultado podría cambiar.

Resultado para diversas cadenas de: `Format(#2/8/05 9:5:3#, CadenaConNombre)`

Cadena con Nombre	Resultado
"General Number"	"38391,3785069444"
"Standard"	"38.391,38"
"General Date"	"08/02/2005 9:05:03"

"Long Date"	"martes 8 de febrero de 2005"
"Medium Date"	"08-feb-05"
"Short Date"	"08/02/2005"
"Long Time"	"9:05:03"
"Medium Time"	"09:05 AM"
"Short Time"	"09:05"

El procedimiento **PruebaFormatoFechas**, nos muestra varias maneras de dar formato a una fecha, utilizando las cadenas de formato propuestas en los puntos anteriores.

Así mismo muestra diversas Cadenas con Nombre y sus equivalentes en cadenas de formato con texto.

```
Public Sub PruebaFormatoFechas ()
    Dim datFecha As Date
    Dim strFecha As String

    datFecha = #9/3/1953 3:34:20 PM#
    Debug.Print "***** Modo 1 *****"
    strFecha = Format(datFecha, "Medium Date")
    Debug.Print strFecha
    strFecha = Format(datFecha, "dd-mmm-yy")
    Debug.Print strFecha

    Debug.Print "***** Modo 2 *****"
    strFecha = Format(datFecha, "Short Date")
    Debug.Print strFecha
    strFecha = Format(datFecha, "dd/mm/yyyy")
    Debug.Print strFecha

    Debug.Print "***** Modo 3 *****"
    strFecha = Format(datFecha, "Long Date")
    Debug.Print strFecha
    strFecha = Format(datFecha, "dddd d ") _
        & "de " _
        & Format(datFecha, "mmmm ") _
        & "de " _
        & Format(datFecha, "yyyy")
    Debug.Print strFecha
    strFecha = Format(datFecha, "dddd")
    Debug.Print strFecha

    Debug.Print "***** Modo 4 *****"
```

```

strFecha = Format(datFecha, "General Date")
Debug.Print strFecha
strFecha = Format(datFecha, "dddd hh:nn:ss")
Debug.Print strFecha
strFecha = Format(datFecha, "dd/mm/yyyy hh:nn:ss")
Debug.Print strFecha
End Sub

```

El resultado de este código es

```

***** Modo 1 *****
03-sep-53
03-sep-53
***** Modo 2 *****
03/09/1953
03/09/1953
***** Modo 3 *****
jueves 3 de septiembre de 1953
jueves 3 de septiembre de 1953
03/09/1953
***** Modo 4 *****
03/09/1953 15:34:20
03/09/1953 15:34:20
03/09/1953 15:34:20

```

Utilización de Format con números.

La utilización de Format con números, nos concede una gran flexibilidad a la hora de presentar los datos numéricos con el formato más adecuado a cada caso.

Tanto si usamos números, como en el caso de las cadenas de texto, no necesitaremos los parámetros para definir el primer día de la semana ni la primera semana del año.

Por ello su sintaxis quedará así:

```
Format(expresión[, formato])
```

Si ejecutamos directamente la orden Format sobre un número, nos lo convertirá simplemente a una cadena cambiando, el punto decimal por el carácter definido en la configuración regional.

```
Format(3.141592) → "3,141592"
```

Con los números podemos definir si queremos utilizar separadores de miles.

Al contrario que en el formato español, el separador de miles, en la cadena de formato, es la coma y el separador de los enteros con los decimales es el punto.

```

Format(1234567890, "#,###") → "1.234.567.890"
Format(3.141592, "#,###") → "3"
Format(3.141592, "#,###") → "3"
Format(3.141592, "#,###.##") → "3.14"

```

```

Format(1234567890, "#,###") → "1.234.567.890,"
Format(1234567890, "#,###.##") → "1.234.567.890,"
Format(1234567890, "#,###.00") → "1.234.567.890,00"
Format(3.141592654, "#,###.0000") → "3,1416"
Format(0.008, "#,###.0000") → ",0080"
Format(0.008, "#,##0.0000") → "0,0080"
Format(023.00800, "#,###.##") → "23,01"

```

El símbolo # representa un dígito cualquiera, salvo los ceros no significativos (los que están a la izquierda de la parte entera y a la derecha de la parte decimal).

El símbolo 0 muestra un dígito o un cero. Si el número a formatear tiene en la posición del cero de la cadena de formato alguna cifra, se mostrará ésta. Si no hubiera ninguna mostrará un cero.

```

Format(23.1, "0,000.00") → "0.023,10"
Format(3.14159, "#,##0.0000") → "3,1416"

```

Si el número tuviera más cifras significativas a la izquierda del punto decimal que caracteres de formato la cadena de formato, se mostrarán todos los caracteres del número.

Si el número tuviera más cifras significativas a la derecha del separador de decimales, que caracteres la cadena de formato, redondeará los decimales al mismo número que los caracteres de formato.

Se pueden además incluir caracteres como el signo menos, el signo más, el signo del dólar, el signo de tanto por ciento, los paréntesis y el espacio en blanco.

El signo de % nos devuelve el Tanto por Ciento de la cantidad formateada.

```
Format(0.08120, "0.00 %") → "8,12 %"
```

Para dar formateo tipo científico se puede usar la cadena de formato "0.00E+00".

```
Format(0.08120, "0.00E+00") → "8,12E-02"
```

Como en el caso de las cadenas podemos utilizar @ con ó sin el signo de exclamación

```

Format(1234567890, "@ @@@ @@@ @@@") → "1 234 567 890"
Format(12345, "@ @@@ @@@ @@@") → " 12 345"
Format(12345, "!@ @@@ @@@ @@@") → "1 234 5 "

```

Cadenas con nombre para números

Al igual que con las fechas, podemos aplicar a los números formatos con nombre.

Resultado para diversas cadenas de: **Format(1234567.8901, CadenaConNombre)**

Cadena con Nombre	Resultado
"General Number"	"1234567,89"
"Currency"	"1.234.567,89 € "
"Fixed"	" 1234567,89"
"Standard"	"1.234.567,89 "

"Percent"	"123456789,01%"		
"Scientific"	"1,23E+06"		
"Yes/No"	"Sí"	Si Cero	"No"
"True/False"	"Verdadero"	Si Cero	"Falso"
"On/Off"	"Activado"	Si Cero	"Desactivado"

Formato compuesto

Como los cuadros de texto de los formularios de Access, Format admite cadenas de formato compuesto, es decir formato diferente para distintos tipos de valores.

Estas cadenas pueden constar de 4 secciones, separadas cada una de ellas con un punto y coma.

En el caso de los números, la primera define cómo se mostrarán los valores Positivos, la segunda los negativos, la tercera para el cero y la cuarta para el valor nulo.

```
Format(3.14, "+ #,##0.00; (#,##0.00);Cero pelotero;Valor Nulo")
```

Devuelve "+ 3,14"

```
Format(-28, "+ #,##0.00; (#,##0.00);Cero pelotero;Valor Nulo")
```

Devuelve "(28,00)"

```
Format(0, "+ #,##0.00; (#,##0.00);Cero pelotero;Valor Nulo")
```

Devuelve "Cero pelotero"

```
Format(Null, "+ #,##0.00; (#,##0.00);Cero pelotero;Valor Nulo")
```

Devuelve "Valor Nulo"

Si alguna de las secciones no estuviera definida, se utilizaría para ella el formato positivo.

```
Format(-28, "#,##0.00 €;;Grati\s;Valor Nulo")
```

Devuelve "-28,00 €"

```
Format(0, "#,##0.00 €;;Grati\s;Valor Nulo")
```

Devuelve "Gratis"

Configuración Regional

La función **Format** nos permite averiguar cuál es la Configuración Regional en algunas de sus propiedades sin tener, por ejemplo, que acceder al registro de Windows.

Si ejecutamos el procedimiento **Configuración** nos mostrará algunos elementos de la Configuración Regional.

```
Public Sub Configuracion()
    Dim datPrueba As Date
    Dim sngDato As Single
    Dim strFormato As String
    Dim strSeparador As String
    Dim strDato As String
    Dim strSeparadorMiles As String
    Dim strSeparadorDecimal As String
```

```
datPrueba = #12/31/2000#
strDato = Format(datPrueba, "Short Date")
strSeparador = Mid(strDato, 3, 1)
If Left(strDato, 2) = "31" Then
    strFormato = "dd" & strSeparador _
                & "mm" & strSeparador & "yy"
Else
    strFormato = "mm" & strSeparador _
                & "dd" & strSeparador & "yy"
End If
Debug.Print "Separador fecha " & strSeparador
Debug.Print "Formato fecha " & strDato
sngDato = 1234.5
strDato = Format(sngDato, "Standard")
strSeparadorMiles = Mid(strDato, 2, 1)
strSeparadorDecimal = Mid(strDato, 6, 1)
Debug.Print "Separador de millares " _
            & strSeparadorMiles
Debug.Print "Separador de decimales " _
            & strSeparadorDecimal
Debug.Print "Formato numérico " & strDato
strDato = Format(sngDato, "Currency")
Debug.Print "Formato moneda " & strDato
End Sub
```

Este procedimiento en mi ordenador muestra:

```
Separador fecha /
Formato fecha 31/12/2000
Separador de millares .
Separador de decimales ,
Formato numérico 1.234,50
Formato moneda 1.234,50 €
```

El resultado será diferente en función de la Configuración Regional propia de cada equipo.

Comencemos a programar con
VBA - Access

Entrega **14**

Funciones de VBA II

Funciones adicionales para formato

Funciones derivadas de VBScript

Las últimas versiones de VBA han incorporado nuevas funciones para el formato de números y fechas.

Estas funciones son

- [FormatNumber](#)
- [FormatCurrency](#)
- [FormatPercent](#)
- [FormatDateTime](#)
- [MonthName](#)
- [WeekDayName](#)

Estas funciones están derivadas de sus correspondientes en **VBScript**; una versión de Visual Basic interpretado desarrollado inicialmente para su implementación en páginas Web.

Función FormatNumber

Devuelve una cadena obtenida al formatear un número.

Su sintaxis es

```
FormatNumber (expresión [, numDígitosDespuésDeDecimal [,  
incluirDígitoInicial [,  
utilizarParéntesisParaNúmerosNegativos [,  
agruparDígitos]]]])
```

La *expresión* puede ser cualquiera que devuelva un valor numérico.

numDígitosDespuésDeDecimal define el número de decimales del formato de salida. Si se pusiera el valor **-1**, usaría la configuración por defecto. Si necesita recortar los decimales de *expresión*, realizará en su caso, un redondeo de cifras.

incluirDígitoInicial define si se pone un cero inicial, antes del separador de decimales, si el valor de la parte entera fuera cero.

utilizarParéntesisParaNúmerosNegativos define si se utilizan los paréntesis para indicar un valor negativo.

agruparDígitos define si se agrupan los números mediante el delimitador de grupos (miles) especificado en la configuración regional.

Estos tres últimos parámetros toman valores long.

Sus valores válidos son [vbTrue](#), [vbFalse](#), [vbUseDefault](#).

[vbUseDefault](#) fuerza a usar la configuración Regional.

El único parámetro obligatorio es la expresión que devuelve el número a formatear.

Si no se pasa otro parámetro el formato se ajusta a lo definido en la Configuración Regional.

[FormatNumber](#) admite también la terminación con **\$**. Ver el párrafo "Consideraciones previas sobre el uso del signo Dólar" en esta misma entrega.

En mi ordenador **FormatNumber** devuelve las siguientes cadenas.

```
FormatNumber (1234567890.123) → " 1.234.567.890,12"
```

```
FormatNumber (-123456.1234, 3, , vbTrue, vbFalse) → "(123456,123)"
```

También admite parámetros, ó argumentos, por nombre

```
FormatNumber$(Expression:=3*45, NumDigitsAfterDecimal:=2) → "135,00"
```

Considerando las posibilidades que tiene la función **Format**, la función **FormatNumber** es útil sobre todo para el formato directo de números, sin usar parámetros:

```
FormatNumber (1234567890.123) → " 1.234.567.890,12"
```

Función FormatCurrency

FormatCurrency devuelve una cadena tras formatear un número al modo moneda.

Su sintaxis es

```
FormatCurrency (expresión [, numDígitosDespuésDeDecimal [,  
incluirDígitoInicial [,  
utilizarParéntesisParaNúmerosNegativos [,  
agruparDígitos]]]])
```

Todo lo comentado sobre los parámetros de la función **FormatNumber**, es aplicable para **FormatCurrency**.

```
FormatCurrency (1234567890.123) → " 1.234.567.890,12 €"
```

```
FormatCurrency (1234567890.123, 0) → " 1.234.567.890 €"
```

Función FormatPercent

FormatPercent devuelve una cadena tras formatear un número al modo porcentaje.

Multiplica el valor de *expresión* por **100** y le añade el signo de porcentaje %.

Su sintaxis es

```
FormatPercent (expresión [, numDígitosDespuésDeDecimal [,  
incluirDígitoInicial [,  
utilizarParéntesisParaNúmerosNegativos [,  
agruparDígitos]]]])
```

Todo lo comentado sobre los parámetros de la función **FormatNumber**, es aplicable con la función **FormatPercent**.

```
FormatPercent (0.123) → "12,3%"
```

```
FormatPercent (3.1415926, 4) → " 314,1593%"
```

Función FormatDateTime

FormatDateTime Devuelve una cadena tras formatear un número al modo fecha hora, ó fecha.

Su sintaxis es

FormatDateTime (*fecha* [, *formatoConNombre*])

formatoConNombre puede tomar cualquiera de estos valores

vbGeneralDate	Muestra la fecha en Fecha Corta , la hora en Hora Larga ó ambas en formato si el parámetro es una Fecha con una parte de Hora
vbLongDate	Muestra la fecha en el formato Fecha Larga de la configuración.
vbShortDate	Muestra la fecha en el formato Fecha Corta .
vbLongTime	Muestra la fecha en el formato especificado en la configuración.
vbShortTime	Muestra la fecha en el formato de 24 Horas (hh:mm) .

El siguiente procedimiento hace uso de las distintas posibilidades de **FormatDateTime**.

```
Public Sub PruebaFormatDateTime ()
    Dim datFechaHora As Date
    Dim datFecha As Date
    Dim datHora As Date

    datFecha = Date
    datHora = Time
    datFechaHora = datFecha + datHora

    Debug.Print FormatDateTime (datFecha)
    Debug.Print FormatDateTime (datHora)
    Debug.Print FormatDateTime (datFechaHora)
    Debug.Print
    Debug.Print FormatDateTime (datFechaHora, vbGeneralDate)
    Debug.Print FormatDateTime (datFechaHora, vbLongDate)
    Debug.Print FormatDateTime (datFechaHora, vbLongTime)
    Debug.Print FormatDateTime (datFechaHora, vbShortDate)
    Debug.Print FormatDateTime (datFechaHora, vbShortTime)
End Sub
```

Tras ejecutarlo, en mi ordenador, muestra los siguientes valores:

```
20/02/2005
21:15:35
20/02/2005 21:15:35

20/02/2005 21:15:35
domingo 20 de febrero de 2005
21:15:35
20/02/2005
21:15
```

El resultado puede cambiar, respecto a otros ordenadores, según su Configuración regional.

Función MonthName

MonthName Devuelve el nombre del mes en el idioma especificado en la configuración regional. Su sintaxis es

```
MonthName (mes [, abreviar])
```

El parámetro que se le pasa es un número que representa al mes. Su valor debe estar entre 1 y 12; caso contrario dará error de "Argumento o llamada a procedimiento no válida"

El parámetro *abreviar*, de tipo booleano, indica si queremos el nombre abreviado.

En mi ordenador devuelve los siguientes valores

```
MonthName (1) → "enero"  
MonthName (2) → "febrero"  
- - - - -  
MonthName (11) → "noviembre"  
MonthName (12) → "diciembre"  
  
MonthName (9, True) → "sep"  
MonthName (10, True) → "oct"
```

Función WeekdayName

WeekdayName Devuelve el nombre del día de la semana que se le pasa como parámetro, en el idioma especificado en la configuración regional. Su sintaxis es

```
WeekdayName (díaDeLaSemana, abreviar, primerDíaDeLaSemana)
```

El parámetro que se le pasa es un número que representa al día de la semana.

Su valor debe estar entre 1 y 7.

El parámetro *abreviar*, de tipo booleano, indica si queremos el nombre abreviado.

El parámetro *primerDíaDeLaSemana*, es una de las constantes que vimos en la función Format, indica el día que queremos adoptar como el primero de la semana.

```
vbUseSystem  
vbSunday  
vbMonday  
vbTuesday  
vbWednesday  
vbThursday  
vbFriday  
vbSaturday  
vbUseSystemDayOfWeek
```

En mi equipo esta función devuelve los siguientes valores para distintas llamadas.

`WeekdayName(1) → "lunes"`

`WeekdayName(1, True, vbUseSystemDayOfWeek) → "lun"`

`WeekdayName(1, False, vbSunday) → "domingo"`

`WeekdayName(1, , vbMonday) → "lunes"`

`WeekdayName(1, , vbMonday) → "lunes"`

Se pueden obtener resultados "exóticos" cambiando el parámetro *primerDíaDeLaSemana*.

Todas estas funciones tienen algo en común:

Facilitan la presentación de datos acorde con la configuración del ordenador de cada usuario, evitando tener que acceder a la configuración regional para efectuar el formateo adecuado del dato.

Manipulación de cadenas

Consideraciones previas sobre el uso del signo Dólar en funciones que devuelven cadenas de texto

Basic, desde sus principios, ya incluía todo un conjunto de procedimientos para el tratamiento de cadenas, entre ellos podemos mencionar **Left\$**, **Right\$** y **Mid\$**.

Estos mismos procedimientos están disponibles con VBA, pero con una salvedad; pueden usarse sin poner el signo del Dólar al final del nombre del procedimiento.

A pesar de que la ayuda de VBA no lo utiliza, podemos afirmar que su uso es muy recomendable.

La ayuda de VBA indica que estas funciones devuelven un tipo **Variant (String)**.

Esto supone que internamente VBA tiene que hacer una conversión previa de **Variant** a **String**.

La utilización del signo del Dólar fuerza a la función a devolver directamente un tipo **String**, con lo que la velocidad de proceso se incrementa.

Como veremos después la función **Left** devuelve la parte izquierda de una cadena.

Incluyo aquí el código para hacer una pequeña comprobación de velocidad.

La función **Timer** devuelve el número de segundos que han transcurrido desde las cero horas del día de hoy.

```
Public Sub Prueba ()
    Const Bucle As Long = 10000000
    Const conCadena As String = "ABCDEFGHJKLMN"
    Dim i As Long
    Dim strCadena As String
    Dim tim0 As Single
    tim0 = Timer
    For i = 1 To Bucle
        strCadena = Left(conCadena, 7)
    Next i
    Debug.Print Timer - tim0 & " Segundos"
    tim0 = Timer
    For i = 1 To Bucle
        strCadena = Left$(conCadena, 7)
    Next i
    Debug.Print Timer - tim0 & " Segundos"
    tim0 = Timer
    For i = 1 To Bucle
        strCadena = Left$(String:=conCadena, Length:=7)
    Next i
    Debug.Print Timer - tim0 & " Segundos"
End Sub
```

El resultado, en mi PC, tras ejecutar este código es el siguiente

4,0156 Segundos

1,4531 Segundos

1,4375 Segundos

Lo que indica que **Left\$** es casi **3 veces más rápido** que **Left**.

Se puede comprobar que si llamamos a la función utilizando parámetros por nombre, no se penaliza su velocidad; incluso es ligeramente más rápido.

Pruebas realizadas con el resto de funciones **Right\$**, **Mid\$**, etc. dan resultados semejantes.

La función **Format** que analizamos en la entrega anterior, también admite la utilización del signo Dólar: **Format\$(123456.123, "#,##0.00")** .

Tras efectuar una prueba similar pero con un bucle de 1.000.000 de iteraciones.

El resultado ha sido **2,3594** frente a **2,2812** Segundos con **Format\$**.

Con lo que se comprueba que aunque el código también es más rápido, no existe la diferencia de velocidad que se consigue por ejemplo con **Left\$**. Esto es lógico ya que la asignación de un **String** frente a un **Variant**, es un proceso menor si lo comparamos con las tareas que realiza internamente **Format**. Esa es la razón por la que no he utilizado el signo **\$** en la entrega anterior; lo que no quita para que yo recomiende su uso, incluso con la función **Format**.

Función Left

Devuelve los n **caracteres** situados a la izquierda de una cadena dada.

Sintaxis

Left(*string*, *length*)

Left\$(*string*, *length*)

Left\$("Eduardo Olaz", 7) → "Eduardo"

Left(string:= "Eduardo Olaz", length := 7) → "Eduardo"

length debe ser mayor ó igual a cero. Si es cero devolverá la cadena vacía "".

Si fuese menor a cero generará el error 5 "Argumento o llamada a procedimiento no válida".

Función LeftB

Devuelve los n **Bytes** situados a la izquierda de una cadena dada.

Sintaxis

LeftB(*string*, *length*)

LeftB\$(*string*, *length*)

LeftB\$("Eduardo Olaz", 14) → "Eduardo"

¿Cómo es posible esto?

Las cadenas que maneja VBA son del tipo **UNICODE** . En este formato cada carácter está formado por **2 Bytes**.

LeftB extrae **Bytes**. **Left** extrae **Caracteres**. Por ello **LeftB** es adecuada para el manejo directo de cadenas de Bytes y **Left** para el manejo de caracteres.

Lo dicho para **LeftB** será igualmente aplicable para las funciones **RightB** y **MidB** que se corresponden con **Right** y **Mid**, funciones que veremos a continuación.

Function Right

Devuelve los **n caracteres** situados a la derecha de una cadena dada.

Sintaxis

Right (*string*, *length*)

Right\$ (*string*, *length*)

Right ("Eduardo Olaz", 4) → " Olaz"

Right\$ (string:= "Eduardo Olaz", length := 4) → "Olaz"

length debe ser mayor ó igual a cero. Si es cero devolverá la cadena vacía "".

Tanto para **Left** como para **Right** si *length* es mayor que el número de caracteres contenidos en la cadena, devolverá la cadena completa sin generar un error.

Left\$ ("Eduardo Olaz", 50) → "Eduardo Olaz"

Right ("Eduardo Olaz", 50) → "Eduardo Olaz"

Function Mid

Devuelve los **n caracteres** de una cadena dada, situados a partir de una posición.

Sintaxis

Mid (*string*, *start* [, *length*])

Mid\$ (*string*, *start* [, *length*])

Mid\$ ("Eduardo Olaz", 9, 3) → "Ola"

Mid (string:="Eduardo Olaz", start:= 9, length:= 3) → "Ola"

String es la cadena de la que vamos a extraer caracteres.

start es la posición a partir de la cual vamos a extraerlos. Es de tipo **Long** y mayor que 0.

length es el número de caracteres que queremos extraer. Su tipo es **Long**.

Este parámetro es opcional. Si no se incluyera, o su valor fuera superior al número de caracteres que hay desde la posición de partida, incluyendo su carácter, devolvería todos los caracteres que hay desde la posición de *start*.

Mid ("Eduardo Olaz", 9) → "Olaz"

Mid\$ ("Eduardo Olaz", 9, 30) → "Olaz"

Instrucción Mid

La instrucción **Mid**, Reemplaza determinado número de caracteres de una cadena con los caracteres de otra.

Sintaxis

Mid (*VariableString*, *start* [, *length*]) = *Cadena*

Mid\$(VariableString, start[, length]) = Cadena

La cadena de la que queremos hacer el cambio se debe pasar mediante una variable de tipo **String** ó **Variant** (**VariableString**).

La cadena de la que queremos extraer los caracteres puede ser pasada de forma directa, o mediante una variable.

El número de caracteres reemplazados es menor o como mucho igual al número de caracteres de **VariableString**.

Tomemos como ejemplo el siguiente código

```
Public Sub PruebaInstruccionMid()
    Dim strAlumno As String

    strAlumno = "Maitane Gaztelu"
    Debug.Print "Cadena inicial      " & strAlumno
    Mid(strAlumno, 1, 7) = "Enrique"
    Debug.Print "Cadena cambiada    " & strAlumno
    Mid(strAlumno, 9, 7) = "Garayoa"
    Debug.Print "Segundo cambio      " & strAlumno
    ' se puede cambiar por una parte de la 2ª cadena
    Mid(strAlumno, 13, 3) = "llámame"
    Debug.Print "Cadena parcial      " & strAlumno
    ' Mid sólo puede cambiar caracteres que ya existan _
    ' Esto no va a funcionar bien
    Mid(strAlumno, 9, 10) = "Martínez"
    Debug.Print "Cambio incompleto " & strAlumno
End Sub
```

Si lo ejecutamos nos mostrará

```
Cadena inicial      Maitane Gaztelu
Cadena cambiada    Enrique Gaztelu
Segundo cambio     Enrique Garayoa
Cadena parcial     Enrique Garallá
Cambio incompleto Enrique Martíne
```

Si no se indica la longitud (parámetro **length**) pasará todo el contenido de la segunda cadena, si su longitud es menor o igual al resto de **VariableString**. Si fuera menor cambiará un número de caracteres igual a los caracteres que restan de la primera, contando desde el que está en la posición **start**. Hasta el final de **VariableString**.

```
Public Sub PruebaMid()
    Dim strInicial As String
    strInicial = "Cadena Inicial"
    Debug.Print strInicial
    Mid(strInicial, 7) = "123"
```

```

    Debug.Print strInicial
    Mid(strInicial, 7) = "$$$$$$$$$$"
    Debug.Print strInicial
End Sub

```

Al ejecutar **PruebaMid** nos mostrará

```

Cadena Inicial
Cadena123icial
Cadena$$$$$$$

```

Como en el caso de Las funciones anteriores, también existe la versión **MidB** para efectuar cambios a nivel de **Bytes**.

Funciones LTrim, Rtrim y Trim

Devuelve un tipo **Variant (String)** que contiene la copia de una cadena determinada a la que se le han eliminado los espacios en blanco de los extremos.

LTrim quita los posibles espacios de la izquierda.

RTrim quita los posibles espacios de la derecha.

Trim quita tanto los de la izquierda como los de la derecha.

Sintaxis

LTrim(cadena)

RTrim(cadena)

Trim(cadena)

El parámetro *cadena* es obligatorio, y se puede pasar cualquier cadena o expresión que devuelva una cadena de texto.

Como valor para *cadena* admite **Null**, en este caso devolverá también **Null**.

LTrim, **RTrim** y **Trim** admiten también ser llamadas con el signo de dólar.

En este caso si se les pasa un **Null** como parámetro, generarán un error.

Dim Variable as String * Longitud

LTrim(" 34567890 ") → "34567890 "

RTrim(" 34567890 ") → " 34567890"

Trim(" 34567890 ") → "34567890"

Trim(Null) → Null

Trim\$(" 34567890 ") → "34567890"

Funciones Len y LenB

Devuelven un tipo **Long** que contiene el número de caracteres de una cadena (**Len**) o el número de Bytes (**LenB**). Si *cadenaOVariable* contiene **Null**, devuelve **Null**.

Sintaxis

Len(cadenaOVariable)

LenB(cadenaOVariable)

El parámetro *cadenaOVariable* puede ser una cadena o una variable.

A Len puede pasársele una variable definida por el usuario.

Si a Len se le pasa un valor **Null**, devuelve **Null**.

Este código devuelve la longitud de diferentes tipos de variables.

Incluso da la longitud del Tipo **Linea** definido por el usuario y a su vez compuesto por dos tipos **Punto**, también definido por el usuario.

```
Public Type Punto
    X As Single
    Y As Single
End Type

Public Type Linea
    ptol As Punto
    pto2 As Punto
End Type

Public Sub PruebaLen()
    Dim intEntero As Integer
    Dim lngLong As Long
    Dim curMoneda As Currency
    Dim strCadena As String * 6
    Dim ptoCoordenadas As Punto
    Dim linSegmento As Linea
    Dim a(1 To 10) As String

    a(2) = "1234567890"

    Debug.Print "Longitud del tipo Integer " & _
        Len(intEntero)
    Debug.Print "Longitud del tipo Long " & _
        Len(lngLong)
    Debug.Print "Longitud del tipo Currency " & _
        Len(curMoneda)
    Debug.Print "Longitud del tipo Punto " & _
        Len(ptoCoordenadas)
    Debug.Print "Longitud del tipo Linea " & _
        Len(linSegmento)
    Debug.Print "Longitud caracteres de strCadena " & _
        Len(strCadena)
    Debug.Print "Longitud (Bytes) de strCadena " & _
        LenB(strCadena)
    Debug.Print "Longitud de a(1) " & _
        Len(a(1))
```

```

    Debug.Print "Longitud Caracteres de a(2) " & _
        Len(a(2))
    Debug.Print "Longitud (Bytes) de a(2) " & _
        LenB(a(2))

End Sub

```

El resultado de este código es:

```

Longitud del tipo Integer 2
Longitud del tipo Long 4
Longitud del tipo Currency 8
Longitud del tipo Punto 8
Longitud del tipo Linea 16
Longitud caracteres de strCadena 6
Longitud (Bytes) de strCadena 12
Longitud de a(1) 0
Longitud Caracteres de a(2) 10
Longitud (Bytes) de a(2) 20

```

Buscar y sustituir cadenas

Entre el bagaje de funciones que posee VBA existen varias para la búsqueda y sustitución de elementos en cadenas.

Ya hemos visto la Instrucción **Mid** que permite realizar sustituciones en una cadena, aunque con ciertas limitaciones.

Igualmente las funciones **Left**, **Mid** y **Right** podrían usarse para buscar subcadenas dentro de una cadena de texto.

Función InStr

Devuelve un tipo **Variant**, convertido a **Long** que indica la posición en la que podemos encontrar una Subcadena en otra cadena.

Si como cadena buscada, o a buscar, se pasa un **Null**, devolverá un **Null**.

Sintaxis

```
InStr([start, ]string1, string2[, comparar])
```

Parámetros

start: opcional, posición donde empezar a buscar.

string1: Cadena ó expresión de cadena en la que se busca

string2: Cadena buscada

comparar: opcional, constante que indica qué se consideran cadenas iguales. Si no se escribe toma el valor por defecto.

Las opciones para **comparar** son:

vbUseCompareOption	-1	Sigue el criterio definido en Option Compare
vbBinaryCompare	0	Hace una comparación a nivel de Bytes.
vbTextCompare	1	Compara los textos
vbDatabaseCompare	2	Sigue el criterio definido en la base de datos Access

Como en el caso de las funciones anteriores, también existe la función **InStrB** que busca valores Byte en una cadena de Bytes.

Función InStrReverse

Es similar a **InStr**, salvo que la búsqueda comienza desde el final de la cadena

Como puede comprobarse, su sintaxis difiere ligeramente de la de **InStr**.

Sintaxis

```
InStrRev(cadena1, cadena2[, inicio[, comparar]])
```

inicio define la posición final de la búsqueda. Si se omite empieza por el último carácter.

Al igual que **InStr** si se introdujera un nulo como cadena a buscar o en la que buscar, devolvería un **Null**.

Función StrComp

Devuelve un entero como resultado de la comparación de dos cadenas. Si como parámetro de alguna de las dos cadenas se pasara un nulo, devolvería **Null**.

Sintaxis:

```
StrComp(string1, string2[, comparar])
```

Si *string1* es menor que *string2* devuelve -1

Si *string1* es igual a *string2* devuelve 0

Si *string1* es mayor que *string2* devuelve 1

Si *string1* o *string2* es **Null** devuelve **Null**

La forma de comparar dependerá del valor(opcional) de *comparar*.

Como referencia ver *comparar* en **InStr**.

```
StrComp ("curso", "VBA") devuelve -1
```

```
StrComp ("curso", "VBA", , vbBinaryCompare) devuelve 1
```

Función Replace

Devuelve una cadena en la que se han cambiado una subcadena por otra dada.

Sintaxis:

Sintaxis

```
Replace(expresión, encontrar, reemplazarCon [, inicio[,  
contar[, comparar]])
```

expresión: Cadena o expresión de cadena en la que buscar

encontrar: Subcadena buscada

reemplazarCon: Subcadena que reemplazará a la anterior

inicio: Cadena o expresión

comparar: Forma de comparar. Como referencia ver *comparar* en **InStr**.

Este código sirve para ver distintas formas de utilizar la función, y sus resultados:


```

Public Sub PruebaReplace()
    Dim strCadena As String
    strCadena = "aaabbbcccdddeeeeEEEfff"
    Debug.Print strCadena
    Debug.Print Replace(strCadena, "a", "B")
    Debug.Print Replace(strCadena, "f", "C", 8)
    Debug.Print Replace(strCadena, "e", "1", , 2)
    Debug.Print Replace(strCadena, _
        "e", "1", , , vbBinaryCompare)
    Debug.Print Replace(strCadena, "e", "1")
    Debug.Print Replace(strCadena, "cccddd", "$$")
End Sub

```

El resultado de este código es

```

aaabbbcccdddeeeeEEEfff
BBBbbbcccdddeeeeEEEfff
cccdddeeeeEEECCE
aaabbbcccddd11eEEEfff
aaabbbcccddd111EEEfff
aaabbbcccddd111111fff
aaabbb$$eeeEEEfff

```

Función StrReverse

La función **StrReverse** devuelve una cadena con los caracteres invertidos respecto a la cadena pasada como parámetro.

Sintaxis

StrReverse (*cadena*)

StrReverse ("ABCDEFGHIJK") → "KJIHGFEDCBA"

Función Filter

La función **Filter** devuelve un array con los elementos de otro array que contienen (o no contienen) un determinado valor. La cadena devuelta está basada en el índice 0.

Sintaxis

Filter(*sourcesrray*, *match* [, *include* [, *comparar*]])

Sourcesrray es la matriz en donde se va a buscar.

match es el valor con el que queremos comparar.

include Si es **True** hace que se busquen los valores que contengan a **match**.

Si es **False**, los que no lo contengan.

comparar Forma de comparar. Como referencia ver **comparar** en **InStr**.

El siguiente procedimiento primero busca en una matriz llamada **aMatriz** los datos que contengan la palabra "Jesús" y se los asigna a la matriz **aFiltrada**. Muestra los datos en la ventana **Inmediato** y seguidamente busca aquellos elementos que no contengan la palabra "Eduardo" y los muestra.

```
Public Sub PruebaFilter()  
    Dim aFiltrada() As String  
    Dim aMatriz(1 To 6) As String  
    Dim i As Long  
    aMatriz(1) = "Antonio Martínez"  
    aMatriz(2) = "Jesús Martínez"  
    aMatriz(3) = "Eduardo Olaz"  
    aMatriz(4) = "Eduardo Pérez"  
    aMatriz(5) = "Jesús Pérez"  
    aMatriz(6) = "Juan Pérez"  
  
    ' Mostramos los elementos _  
    ' que contienen "Jesús"  
    aFiltrada = Filter(aMatriz, "Jesús")  
    For i = LBound(aFiltrada) To UBound(aFiltrada)  
        Debug.Print aFiltrada(i)  
    Next i  
    Debug.Print  
  
    ' Mostramos los elementos _  
    ' que no contienen "Eduardo"  
    aFiltrada = Filter(aMatriz, "Eduardo", False)  
    For i = LBound(aFiltrada) To UBound(aFiltrada)  
        Debug.Print aFiltrada(i)  
    Next i  
End Sub
```

El resultado de todo esto es

```
Jesús Martínez  
Jesús Pérez
```

```
Antonio Martínez  
Jesús Martínez  
Jesús Pérez  
Juan Pérez
```

Función Split

Devuelve una matriz, basada en el índice 0, que contiene un número especificado de subcadenas extraídas de una cadena de texto.

Sintaxis

Split(*expresión*[, *delimitador*[, *contar*[, *comparar*]])

expresión Una cadena o una expresión que devuelva una cadena de caracteres. La cadena puede contener caracteres especificadores de separación.

delimitador Es un carácter que sirve para definir los límites de las subcadenas dentro de la cadena pasada. Si no se indica, toma como carácter de separación el espacio en blanco.

contar Indica el número de subcadenas que queremos extraer. Si se pasa el valor -1, se especifica que queremos extraer todas las subcadenas. Es el modo por defecto

comparar Al igual que en las funciones anteriores, forma de comparar entre sí las cadenas. Como referencia ver **comparar** en **InStr**.

La función Split es útil para obtener datos desde un fichero de texto con los campos separados por Delimitadores.

El siguiente código hace primero un

```
Public Sub PruebaSplit()  
    Dim strDatos As String  
    Dim aDatos() As String  
    Dim i As Long  
  
    strDatos = "Andrés Tomás Iker"  
    aDatos = Split(strDatos)  
    For i = LBound(aDatos) To UBound(aDatos)  
        Debug.Print aDatos(i)  
    Next i  
    Debug.Print  
    strDatos = "Martínez Pérez;Gómez Iturri;García Martín"  
    aDatos = Split(strDatos, ";")  
    For i = LBound(aDatos) To UBound(aDatos)  
        Debug.Print aDatos(i)  
    Next i  
End Sub
```

Este código mostrará lo siguiente

Andrés
Tomás
Iker

Martínez Pérez
Gómez Iturri
García Martín

Función Join

La función **Join** se podría decir que es la inversa de la función **Split**.

Toma una matriz de caracteres y los junta en una única cadena.

El siguiente código hace lo siguiente

- Asigna las estaciones del año a una matriz mediante la función **Split**.
- A continuación vuelve a juntar los datos en pero usando como separador un espacio en blanco " ".

```
Public Sub PruebaJoin()  
    Dim strDatos As String  
    Dim aDatos() As String  
    Dim strResultado As String  
  
    strDatos = "Primavera;Verano;Otoño;Invierno"  
    aDatos = Split(strDatos, ";")  
    strResultado = Join(aDatos, " ")  
    Debug.Print strResultado  
End Sub
```

Tras ejecutar el código se muestra

```
Primavera Verano Otoño Invierno
```

Operador Like

Este operador se utiliza para comparar dos cadena de caracteres.

```
resultado = cadena Like patrón
```

Realiza una comparación entre **cadena** y **patrón** basándose en la configuración **Option Compare** ubicada en el comienzo del módulo. Los valores asignables a Option Compare son:

- Option Compare **Text**
- Option Compare **Binary**
- Option Compare **Database**

Si la cadena responde a la configuración definida en patrón devolverá **True**, en caso contrario devolverá **False**.

Si alguno de los operadores fuese **Null**, devolverá **Null**.

Como **cadena** puede usarse una cadena o una expresión que devuelva una cadena.

Para una comparación con Option Compare **Text**, las siguientes expresiones devolverán:

```
left("Eduardo Olaz", 7) like "EDUARDO" → True  
left(" Eduardo Olaz", 7) like "EDUARDO" → False
```

"Eduardo Olaz" like **Null** → **Null**

El operador **Like** admite "comodines"

Estos comodines son

- ? Representa a cualquier carácter ó número (uno por cada ?).
- * Representa a cualquier número de caracteres (incluso ninguno)
- # Representa un dígito (del 0 al 9)

[*listacaracteres*] Un carácter cualquiera incluido en *listacaracteres*

[!*listacaracteres*] Ninguno de los caracteres incluido en *listacaracteres*

"BMP1956" like "???1956" → **True**

"BMP1956" like "???####" → **True**

"BMP1956" like "???????" → **True**

"BMP1956" like "*####" → **True**

"BMP1956" like "*##57" → **False**

"Cualquier cadena" like "*" → **True**

"Referencia de producto BMP1956" like "*1956" → **True**

"BMP1956" like "???1956" → **True**

"BMP1956" like "*####" → **True**

left("Eduardo Olaz", 7) like "EDUARDO" → **True**

"Eduardo Olaz" like **Null** → **Null**

Null like **Null** → **Null**

Funciones Asc y AscB

La función **Asc** devuelve un entero con el código del primer carácter de la cadena pasada como parámetro.

AscB devuelve un entero con el código del primer Byte de la cadena.

Sintaxis

Asc(cadena)

AscB(cadena)

Asc("ABCDEFGHIJK") → 65

Funciones Chr y Chr\$

La función Chr y su equivalente Chr\$, actúan de forma inversa a la función Asc.

Devuelven un carácter que tiene como código de carácter el valor que se le pasa como parámetro.

Sintaxis

Chr(códigocar)

Así como **Asc**("A") devuelve 65, **Chr**(65) → "A"

El procedimiento **PruebaChrAsc**, muestra los códigos de carácter correspondientes a las letras que van de la "A" a la "Z".

```
Public Sub PruebaChrAsc()  
    Dim i As Long  
    For i = Asc("A") To Asc("Z")  
        Debug.Print Format(i, "00 - ") & Chr$(i)  
    Next i  
End Sub
```

El resultado será

```
65 - A  
66 - B  
67 - C  
68 - D  
69 - E  
70 - F  
.  
.  
.  
.  
84 - T  
85 - U  
86 - V  
87 - W  
88 - X  
89 - Y  
90 - Z
```

Diferencia entre funciones que trabajan en modo Carácter y en modo Byte.

Ya he comentado que existen funciones intrínsecas de VBA, como **Asc**, **Left**, **Mid** y **Right** que trabajan sobre caracteres y que tienen sus equivalentes **AscB**, **LeftB**, **MidB** y **RightB** que trabajan sobre Bytes.

Con ellas podemos ver que las cadenas de VBA están en realidad formadas por caracteres que utilizan 2 Bytes para ser representados.

Tras ejecutar este código podemos ver la diferencia del resultado utilizando **Asc** y **Mid**, respecto a **AscB** y **MidB**.

```
Public Sub ComparaFuncionB(ByVal Cadena As String)  
    Dim i As Long  
    Dim intCodigo As Integer  
    Dim strCaracter  
    Debug.Print "Códigos de Caracteres"  
    For i = 1 To Len(Cadena)  
        intCodigo = Asc(Mid(Cadena, i, 1))  
        strCaracter = Mid(Cadena, i, 1)  
        Debug.Print Format(intCodigo, "000 ") _
```

```
        & strCaracter & " ";  
    Next i  
    Debug.Print  
    Debug.Print "Códigos Bytes"  
    For i = 1 To 2 * Len(Cadena)  
        intCodigo = AscB(MidB(Cadena, i, 1))  
        Debug.Print Format(intCodigo, "000 ");  
    Next i  
    Debug.Print  
End Sub
```

Recordemos que

MidB va extrayendo **Byte** a **Byte**

Mid extrae **carácter** a **carácter**.

Ejecutemos este procedimiento mediante

```
ComparaFuncionB " VBA €"
```

Imprimirá lo siguiente en la ventana inmediato:

```
Códigos de Caracteres
```

```
086 V 066 B 065 A 032      128 €
```

```
Códigos Bytes
```

```
086 000 066 000 065 000 032 000 172 032
```

A primera vista vemos que por cada carácter que extrae la función **Mid**, la función **MidB** extrae 2.

Esto se produce porque las cadenas **String** de VBA manejan caracteres **Unicode**, y un carácter **Unicode** está compuesto de 2 **Bytes**.

Voy a hacer una reflexión especial sobre el carácter €, correspondiente al Euro.

La función **Asc** ("€") devuelve 128, que es el número de carácter que maneja Windows.

En cambio, de forma similar a lo que hemos visto en el procedimiento anterior

```
AscB(LeftB("€",1)) → 172
```

```
AscB(RightB("€",1)) → 32
```

Lo que nos indica que internamente VBA utiliza dos bytes, siendo el primero 172 y el segundo 32.

Comencemos a programar con
VBA - Access

Entrega **15**

Operadores

Operadores

A la hora de construir instrucciones en VBA, que contengan operaciones, no sólo manejamos constantes, variable y expresiones, sino que utilizamos unos elementos llamados Operadores, que aplicados a uno ó varios operandos, generan el resultado de la operación.

Tipos de Operadores

Aritméticos	Se usan para efectuar cálculos matemáticos
De Comparación	Permiten efectuar comparaciones
De Concatenación	Combinan cadenas de caracteres
Lógicos	Realizan operaciones lógicas

Operadores aritméticos

VBA maneja la mayor parte de los operadores aritméticos habituales en los lenguajes de programación.

Estos operadores son

- + Suma
- - Resta
- * Producto
- / División
- ^ Elevar a potencia
- \ División entera
- **Mod** Módulo ó Resto

En general, el tipo devuelto por el resultado de una operación, es el del tipo del más preciso de los operadores, salvo que el resultado supere su rango; en ese caso devolverá el siguiente tipo de mayor precisión. Esta regla tiene abundantes excepciones.

Para más información consulte la ayuda de Access en la que se relata toda la casuística pormenorizada para cada uno de los operadores.

Si el resultado de una operación fuese un dato de coma flotante y se asignara a un tipo entero, se efectuaría un redondeo.

Si se trata de asignar un resultado fuera del rango de valores de la variable que va a recibir el resultado de la operación, se generará un error de "Desbordamiento" y se interrumpirá la ejecución del código, salvo que el error fuera capturado y tratado.

Operador Suma

El operador **suma** (+), sirve para asignar el resultado de la suma de dos números, ó en el caso de cadenas, dar como resultado una cadena compuesta por las dos anteriores.

La forma de usarlo es

$$\text{resultado} = \text{expresión1} + \text{expresión2}$$

Si *expresión1* ó *expresión2* tuvieran el valor **Null**, el resultado sería también el valor **Null**.

expresión1 y **expresión2** son los operandos, pudiendo ser cualquier valor numérico ó expresiones que los generen.

Al contrario de otros lenguajes, VBA permite utilizar como operadores, tipos numéricos distintos. Por ejemplo podemos sumar un tipo **Byte** con un tipo **Long**, ó con un tipo **Date**.

Igualmente la variable **resultado** no tiene por qué ser del mismo tipo que los operandos.

Una de las limitaciones es que el resultado de la operación no debe sobrepasar la capacidad del tipo correspondiente a la variable que va a recibir el resultado de la misma.

Esto es aplicable al resto de los operadores.

Por ejemplo

```
Public Sub SumaConError()  
  
    Dim bytResultado As Byte  
  
    bytResultado = 10 + 23  
    Debug.Print bytResultado  
    bytResultado = 150 + 150  
    Debug.Print bytResultado  
End Sub
```

Nos imprimirá el resultado 33 y a continuación nos generará el error nº 6 “Desbordamiento”, ya que un tipo **Byte** sólo admite valores que van de 0 a 255.

Nos daría ese mismo tipo de error si tratáramos de hacer

```
bytResultado = 15 + (-16)
```

Ya que a un **Byte** no se le pueden asignar valores negativos

Lo mismo ocurriría en el siguiente código

```
Dim intResultado As Integer  
  
intResultado = 30000 + 30000
```

ya que un tipo **Integer** maneja valores entre **-32.768** y **32.767**.

Como hemos indicado en un punto anterior, si a una variable que maneja números enteros le asignamos el resultado de una suma de números de coma flotante, efectuará un redondeo del resultado al número entero más próximo.

```
Public Sub PruebaSumaComaFlotante()  
    Dim intResultado As Integer  
    intResultado = 3.1416 + 2.5468  
    Debug.Print intResultado  
    intResultado = -3.1416 + (-2.5468)  
    Debug.Print intResultado  
End Sub
```

Nos mostrará como resultado los valores **6** y **-6**.

Cuando se utilizan como operandos dos valores de tipos distintos, VBA cambia el menos preciso al tipo del más preciso. Por ejemplo si vamos a sumar un **Integer** con un **Long**, VBA realiza un “moldeado de tipo” con el **Integer**, convirtiéndolo a **Long** antes de realizar la operación.

Si uno de los operadores fuera del tipo **Date**, el resultado también lo será.

Si desde la ventana Inmediato hacemos

```
? #3/14/5# + 1
```

nos mostrará

```
15/03/2005
```

(El comando **?** es equivalente a **Print**)

En este caso vemos que al sumar **1** a la fecha devuelve la fecha del día siguiente.

Como ya hemos comentado el operador Suma permite sumar o concatenar cadenas.

Esta facilidad puede en algún momento crearnos más problemas que ventajas sobre todo cuando manejamos cadenas que contienen posibles valores numéricos.

```
Public Sub PruebaSumaCadenas()
    Dim strResultado As String

    strResultado = 3 + 4
    Debug.Print strResultado

    strResultado = 3 + "4"
    Debug.Print strResultado

    strResultado = "3" + "4"
    Debug.Print strResultado
End Sub
```

Este código nos devolverá

```
7
7
34
```

Curiosamente `3 + "4"` devuelve 7.

Además si tratamos de hacer

```
101 + " dálmatas"
```

Nos dará el “bonito error” nº 13; “No coinciden los tipos”.

Por ello recomendamos usar el operador **&** como operador para sumar, ó unir cadenas, en vez del operador Suma **+**.

```
101 & " dálmatas" → "101 dálmatas"
3 & "4" → "34"
```

Hay otro sumando permitido por VBA que no deja de ser sorprendente. Es el valor **Empty** (vacío) que si lo usamos con un número, ó consigo mismo, se asimila al **Cero**. Y con una cadena a la cadena vacía "".

Empty + "A" → "A"

Empty + 3.1416 → 3.1416

Empty + **Empty** → 0

Esta "promiscuidad" en el manejo y la asignación entre tipos diferentes de datos que permite Visual Basic para Aplicaciones, es algo que personalmente no me termina de convencer, pero como decía el castizo, - "es lo que hay...".

VBA no permite los operadores del tipo Pre ó Post Incremental como serían:

$Y = ++X$ ó $Y = X++$

Operador Resta

El operador **resta** (-), sirve para asignar el resultado de la sustracción entre dos números.

Tiene dos formas sintácticas

resultado = expresión1-expresión2
- *expresión*

En la primera, la variable **resultado** recibe el valor resultante de restar **expresión2** a **expresión1**.

En la segunda se cambia el signo al valor numérico de **expresión**.

Si **expresión1** ó **expresión2** tuvieran el valor **Null**, **resultado** recibirá también el valor **Null**.

Empty lo considera como valor **Cero**.

Como en el caso de la suma, si uno de los operadores fuera del tipo **Date**, el resultado también lo será. Si desde la ventana Inmediato hacemos:

Print #3/14/5# - 100

nos mostrará

04/12/2004

Para obtener información sobre los distintos tipos devueltos, en función del de los tipos de los operadores, consulte la ayuda de VBA.

Operador Producto

El operador **producto** (*), sirve para asignar el resultado del producto de dos números.

La forma de usarlo es

*resultado = expresión1*expresión2*

resultado es una variable de tipo numérico y tanto **expresión1** como **expresión2** pueden ser cualquier expresión que de como resultado un valor numérico.

El tipo numérico que se suministra a **resultado** dependerá de los tipos de **expresión1** y **expresión2**.

Si uno de los operandos es del tipo **Single** y el otro del tipo **Long**, **resultado** recibirá un tipo **Double**. Para más información consultar la ayuda de VBA.

Si alguno de los operadores es **Null**, **resultado** tomará también el valor **Null**.

Si alguno de los operandos es **Empty**, **resultado** será **Cero**. Ya que considerará que ese operando contiene el valor **Cero**.

Operador División

El operador **división** (**/**), asigna el resultado de la división de dos números.

La forma de usarlo es

$$\text{resultado} = \text{expresión1} / \text{expresión2}$$

resultado es una variable de tipo numérico y tanto **expresión1** como **expresión2** pueden ser cualquier expresión que de como resultado un valor numérico.

Si **expresión2** fuera el valor **Cero**, daría el error **11** de "División por cero".

El tipo numérico que recibe **resultado** normalmente será del tipo **Double**.

Esta regla tiene varias excepciones

Si los operandos son del tipo **Byte**, **Integer** ó **Single**, **resultado** recibirá un **Single**, a menos que supere el rango de **Single** en cuyo caso será del tipo **Double**.

Si uno de los operandos fuera del tipo **Decimal**, resultado también lo será.

Es aplicable lo dicho en los anteriores operadores matemáticos respecto a **Null** y **Empty**.

Para más información consultar la ayuda de VBA.

Operador Elevar a potencia

El operador para **elevar a potencia** (**^**), asigna el resultado de elevar la base a la potencia del exponente.

La forma de usarlo es

$$\text{resultado} = \text{expresión1}^{\text{exponente}}$$

resultado es una variable de tipo numérico y tanto **expresión1** como **exponente** pueden ser cualquier expresión que de como resultado un valor numérico.

Si **exponente** fuera el valor **Cero**, daría como resultado **1**, incluso en el caso de que la base tuviera el valor **Cero**. ¿???

$$45^0 \rightarrow 1$$

$$0^0 \rightarrow 1 \quad (\text{algo no muy correcto matemáticamente hablando})$$

$$3.1416 \wedge \text{Empty} \rightarrow 1$$

$$\text{Null} \wedge 0 \rightarrow \text{Null}$$

$$3.1416 \wedge \text{Null} \rightarrow \text{Null}$$

Exponente puede ser un valor fraccionario.

Así, si quisiéramos obtener la raíz cúbica de 343 podríamos hacer

$$343 \wedge (1/3) \rightarrow 7$$

Actuaríamos igual para la raíz cuadrada

$$16 \wedge 0.5 \rightarrow 4$$

Adicionalmente, para obtener la raíz cuadrada en VBA existe la función **Sqr**.

$$\text{Sqr}(16) \rightarrow 4$$

Cuando estudiábamos matemáticas nos contaron que en el campo de los números reales, la raíz cuadrada de un número negativo no tiene solución. Por ello se crearon los números imaginarios.

Si tratamos de hacer **Sqr**(-16) nos generará un error 5 en tiempo de ejecución, indicando que el valor -16 no es correcto para la función **Sqr**.

Puede sorprender que esto no ocurra si calculamos la raíz cuadrada elevando a 0.5

$$-16 \wedge 0.5 \rightarrow -4$$

En cambio si hacemos $(-16) \wedge 0.5$, sí dará el error.

La razón la veremos más adelante en la prioridad de operadores.

El cálculo del operador potencia se realiza antes que el del operador de cambio de signo.

Tanto en la base como en el exponente admite números de coma flotante.

Por ejemplo podríamos obtener algo tan exótico como el resultado de elevar el número **e**, base de los logaritmos Neperianos, al valor de **Pi**

$$2.7182818 \wedge 3.1415926 \rightarrow 23,1406906315563$$

Operador División entera

El operador **división entera** (****), realiza dos procesos.

Si no tuvieran valores enteros, efectuaría el redondeo del numerador y del denominador.

A continuación realiza la división de los valores resultantes, devolviendo la parte entera del resultado.

La forma de usarlo es

$$\text{resultado} = \text{expresión1} \backslash \text{expresión2}$$

resultado recibirá un valor entero Byte, Integer ó Long.

$$8 \backslash 3 \rightarrow 2$$

$$-8 \backslash 3 \rightarrow -2$$

$$12 \backslash 3 \rightarrow 4$$

Es aplicable lo comentado con anterioridad respecto a **Empty** y **Null**.

$$\text{Null} \backslash 3 \rightarrow \text{Null}$$

$$\text{Empty} \backslash 3 \rightarrow 0$$

Si al redondear **expresión2** diera como resultado el valor Cero, se produciría el error **11** de "División por cero".

$$4 \backslash 0.1 \rightarrow \text{Error 11}$$

$$4 \backslash 0.5 \rightarrow \text{Error 11}$$

$$4 \backslash 0.51 \rightarrow 4$$

$$4 \backslash \text{Empty} \rightarrow \text{Error 11}$$

4 \ 0 → **Error 11**

Hay que tener un especial cuidado con este operador, y no olvidar que previamente realiza un redondeo a cero decimales, tanto del numerador como del denominador.

Este **redondeo** utiliza el método de redondeo vigente en el **sistema bancario americano**, que es ligeramente diferente al europeo.

De este tema hablaremos cuando veamos la función **Round**.

Operador Módulo o Resto

El operador **módulo** (**Mod**), Asigna el resto de una división entre dos números.

Como en el caso de la división entera, previamente se realiza un redondeo a cero decimales, tanto del dividendo como del divisor, si éstos tuvieran un valor que no fuera entero.

Su sintaxis es

resultado = expresión1 Mod expresión2

Para el operador **Mod**, es aplicable lo comentado en el operador División entera sobre el redondeo del numerador y denominador.

8 **Mod** 7 → 1
 8.6 **Mod** 7 → 2
 8.9 **Mod** 7.51 → 1
Null Mod 7.51 → **Null**
Empty Mod 7.51 → 0
 27 **Mod** 12 → 3

Operadores de Comparación

En algunos procesos podemos necesitar establecer si un valor es menor, igual ó mayor que otro, por ejemplo para la toma de una decisión mediante una sentencia **If . . . Then**.

En VBA tenemos 6 tipos de operadores para esta tarea.

Estos operadores son

- < Menor que
- <= =< Menor ó igual que
- > Mayor que
- >= => Mayor ó igual que
- = Igual a
- <> >< Distinto de

Vemos que para los operadores **menor o igual**, **mayor o igual** y **distinto de** hay dos posibles formas de escritura. La forma más usual de uso es la escrita en primer lugar.

Su sintaxis es

resultado = expresión1 Operador expresión2

resultado es una variable de tipo **booleano** y tanto **expresión1** como **expresión2** pueden ser cualquier expresión que de como resultado un valor numérico o de cadena.

Si **expresión1** o **expresión2** contuvieran el valor **Null** devolvería **Null**.

Ejemplos de resultados

8 < 7 → **False**

7 < 8 → **True**

8 <= 8 → **True**

8 >= 8 → **True**

8 > 7 → **True**

8 = 8 → **True**

8 <> 8 → **False**

"Avila" < "Barcelona" → **True**

A la hora de comparar cadenas, hay que tener en cuenta que el resultado puede cambiar en función de cómo hayamos establecido la forma de comparación en

Option Compare

Por ejemplo, si no hemos establecido, a nivel del módulo, valor para **Option Compare**, o hemos puesto **Option Compare Binary**, las siguiente expresiones darán **False**.

"A"="a" → **False**

"Ávila" < "Barcelona" → **False**

En cambio si tenemos **Option Compare Text**, las siguientes expresiones darán **True**.

"A"="a" → **True**

"Ávila" < "Barcelona" → **True**

Si utilizamos **Option Compare Database**, (sólo con bases de datos Access) dependerá de cómo hemos definido la comparación a nivel local en la base de datos.

Todo lo anterior es aplicable para el resto de los operadores de comparación con cadenas de texto.

Operador Like

En la entrega anterior vimos la función **InStr**, que permite buscar una cadena dentro de otra y nos devuelve la posición en la que se encuentra. También vimos la función **StrComp** que compara dos cadenas diciéndonos cuál es "menor".

Para la comparación entre cadenas tenemos también un operador adicional

- **Like** Permite comparar una expresión patrón con una cadena, indicándonos si esa cadena cumple con lo indicado en la patrón.

Su sintaxis es

resultado = *cadena* **Like** *patrón*

Para el operador **Like**, también es aplicable lo comentado anteriormente sobre **Option Compare**. Para obtener más información al respecto puede consultar la información sobre el operador **Like** en la ayuda de VBA.

Si el contenido de *cadena* coincide con el contenido de *patrón*, **resultado** recibirá el valor **True**.

En caso contrario se le asignará **False**, salvo que alguno de los operandos *cadena* o *patrón*, tenga como valor **Null**, en cuyo caso se le asignará **Null**.

El parámetro **patrón** se construye con una serie de caracteres, algunos de los cuales pueden funcionar como "comodines".

Supongamos que queremos ver si una cadena comienza por "MAR"

```
"MARTINEZ ELIZONDO" Like "MAR*" → True
```

En este caso "MAR*" devuelve **True**, porque "MARTINEZ ELIZONDO" comienza por "MAR". El carácter "*" sustituye a cualquier conjunto de caracteres.

Incluso "MAR" Like "MAR*" ó "MAR-2" Like "MAR*" también devolverían **True**.

Hay otros dos caracteres comodines.

"?" sustituye a un carácter cualquiera

"#" sustituye a un dígito cualquiera

```
"MARTA" Like "MART?" → True
"MART2" Like "MART?" → True
"MART24" Like "MART?" → False
"MARTA" Like "MART#" → False
"MART2" Like "MART#" → True
"MART24" Like "MART#" → False
```

Podemos ver si un carácter está dentro de un intervalo de caracteres, mediante **Listas de caracteres**. Las listas de caracteres se especifican escribiendo los caracteres con los que queremos comparar el carácter, entre corchetes, empezando el patrón con comillas.

Hay varias formas de construir una lista de caracteres.

Caracteres sueltos entre comas

```
"a" Like "[a,b,c,d]" → True
"n" Like "[a,b,c,d]" → False
```

Una cadena de caracteres

```
"a" Like "[abcd]" → True
"n" Like "[abcd]" → False
```

Podemos definir rangos de caracteres indicando el carácter inferior y el superior separados por un guión.

```
"a" Like "[a-d]" → True
"n" Like "[a-d]" → False
```

Podemos combinar caracteres sueltos y rangos separándolos por comas.

Supongamos que queremos comprobar si un carácter está entre la "a" y la "d" o entre la "m" y la "q", considerando también como carácter válido la "ñ".

```
"a" Like "[a-d,m-q,ñ]" → True
"o" Like "[a-d,m-q,ñ]" → True
"ñ" Like "[a-d,m-q,ñ]" → True
```

```
"e" Like "[a-d,m-q,ñ]" → False
```

```
"1" Like "[a-d,m-q,ñ]" → False
```

Podemos combinar en las cadenas patrón, caracteres, comodines y listas lo que nos permite definir patrones complejos.

Supongamos que queremos que nos de **True** si la cadena chequeada cumple con estas condiciones:

1. Empieza por una letra entre la "C" y la "H", o entre la "Q" y la "S".
2. El segundo carácter debe ser un número
3. El tercer carácter puede ser cualquiera
4. Los cuatro siguientes caracteres deben ser "00BB"
5. El 8º carácter no debe ser la letra V. (Si utilizamos el carácter Exclamación "!" equivale a la negación de lo que le sigue).

Vayamos por partes y construyamos los componentes de la cadena patrón.

1. "[Q-R]".
2. "#".
3. "?".
4. "00BB"
5. "[!V]".

Juntándolo nos quedará "[C-H,Q-R]#?00BB[!V]*".

```
"R5Ñ00BBS" Like "[C-H,Q-R]#?00BB[!V]*" → True
```

```
"R5Z00BBS-12345" Like "[C-H,Q-R]#?00BB[!V]*" → True
```

```
"Q8W00BBAACC" Like "[C-H,Q-R]#?00BB[!V]*" → True
```

De lo aquí expresado podemos ver que el operador **Like** permite efectuar comprobaciones ó validaciones de cadenas verdaderamente complejas y con una sola línea de código.

Nota:

A la hora de definir una lista de rangos del tipo [A-H], el primer carácter debe ser menor que el segundo. Por ejemplo [H-A] no sería un patrón válido.

En la ayuda de VBA se realiza una descripción pormenorizada de toda la casuística con la que nos podemos encontrar.

Operador Is

Para la comparación entre objetos tenemos un operador adicional, es el operador **Is**.

- **Is** devuelve **True** o **False** en función de que dos variable objeto hagan referencia al mismo objeto.

Este tema lo veremos cuando veamos más a fondo los Objetos.

Hay otras 2 formas de utilizar el operador **Is**.

La primera ya la vimos con la estructura **Select . . . Case**

Lo usamos, por ejemplo en las expresiones del tipo

```
Select Case N
Case is < 10
- - -
```

```
Case 10
```

```
- - -
```

```
Case is > 10
```

En ellas comprobamos si la variable pasada como testigo es menor ó mayor que 10.

Hay otro entorno en el que se usa **Is** y es en estructuras del tipo **If . . . Then**; en concreto acompañando a la expresión **TypeOf**.

TypeOf devuelve el tipo de un objeto.

Se usa de la siguiente forma

```
TypeOf nombre_objeto Is tipo_objeto
```

Esto nos puede servir, por ejemplo para cambiar las propiedades de objetos en función del tipo al que pertenecen.

Para ilustrar esto vamos a crear un formulario nuevo.

Al cargar el formulario, examinará los controles que hay en él.

Asignará a todos los controles la misma altura (propiedad **Height**) y ordenará las etiquetas, cuadros de texto y botones que contenga.

Ajustará la distancia que haya entre la parte izquierda del control y la parte izquierda del formulario (propiedad **Left**).

Ajustará también la distancia respecto a la parte superior de la sección donde están ubicados (propiedad **Top**).

Las otras dos propiedades que ajustará serán

Texto del control (propiedades **Caption** o el valor por defecto de los cuadros de texto)

Anchura del control (propiedad **Width**).

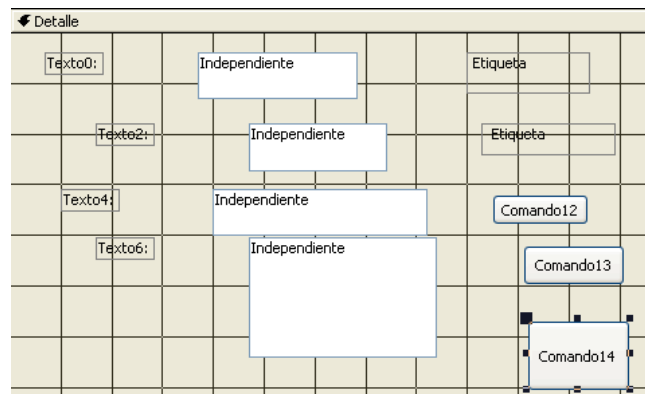
En la barra del menú del Cuadro de Herramientas, desactivamos el botón **[Asistentes para controles]** (la de la varita mágica).

Ponemos en el formulario

- 4 Cuadros de texto
- 2 Etiquetas adicionales (ponles un texto para ver el efecto)
- 3 botones

Procuraremos poner los controles de una manera desordenada, y variando sus tamaños.

Nuestro formulario podría tener un aspecto tan patético como éste:



Vemos que no es probable que obtenga el premio al diseño del formulario del año.

Para tratar de poner orden en este caos, vamos a hacer que sea Access el que lo haga utilizando código. Aprovecharemos el evento **Al cargar** del formulario

Teniendo seleccionado el formulario, en la ventana de propiedades seleccionamos el evento Al cargar y en el editor de código escribimos lo siguiente:

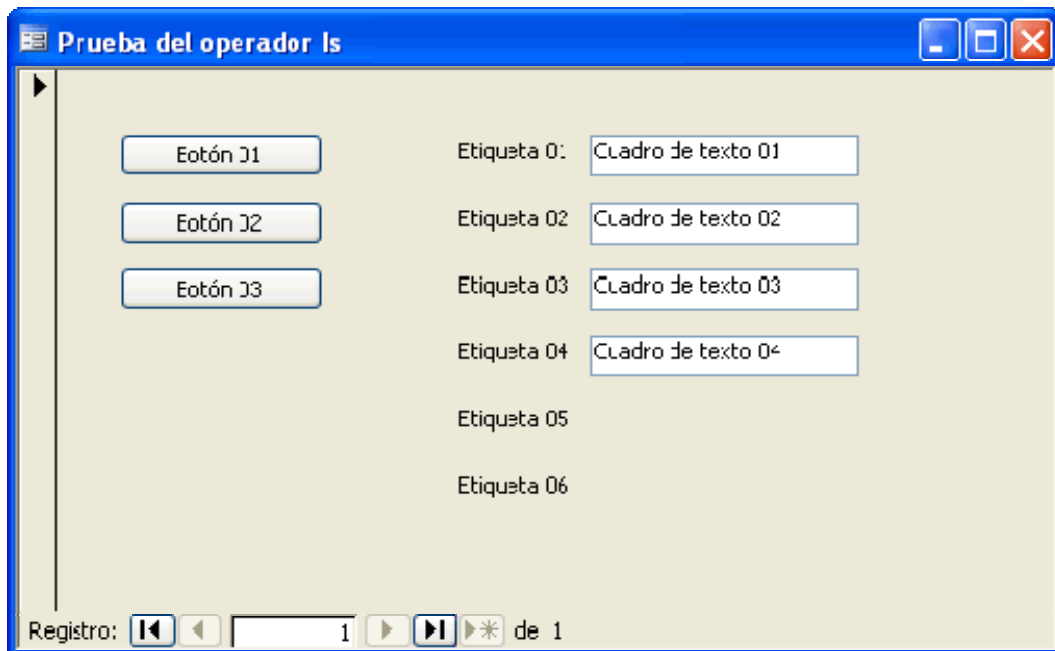
```
Private Sub Form_Load()  
    Const conlngAltura As Long = 300  
    Dim ctr As Control  
    Dim lngEtiquetas As Long  
    Dim lngCuadrosTexto As Long  
    Dim lngBotones As Long  
  
    Caption = "Prueba del operador Is"  
    For Each ctr In Me.Controls  
        ctr.Height = conlngAltura  
  
        If TypeEnum ctr Is CommandButton Then  
            ' Si el control es un botón  
            lngBotones = lngBotones + 1  
            ctr.Left = 500  
            ctr.Top = lngBotones * 500  
            ctr.Width = 1500  
            ctr.Caption = "Botón " _  
                & Format(lngBotones, "00")  
        End If  
        If TypeEnum ctr Is Label Then  
            ' Si el control es una etiqueta  
            lngEtiquetas = lngEtiquetas + 1  
            ctr.Left = 3000  
            ctr.Top = lngEtiquetas * 500  
            ctr.Width = 2500  
            ctr.Caption = "Etiqueta " _  
                & Format(lngEtiquetas, "00")  
        End If  
        If TypeEnum ctr Is TextBox Then  
            ' Si el control es un cuadro de texto  
            lngCuadrosTexto = lngCuadrosTexto + 1  
            ctr.Left = 4000  
            ctr.Top = lngCuadrosTexto * 500  
            ctr.Width = 2000  
            ctr = "Cuadro de texto " _  
                & Format(lngCuadrosTexto, "00")  
        End If  
    End For  
End Sub
```

```

        End If
    Next ctr
End Sub

```

Este código hace que el formulario, tras cargarse, tome este otro aspecto



Podemos decir que esto ya es otra cosa.

En vez de utilizar una sentencia `If TypeOf ctr Is Control` podríamos haber utilizado una de las propiedades que tienen los controles que es `ControlType`.

```

Ctr.ControlType

```

nos hubiera devuelto una constante enumerada del tipo `acControlType`. En concreto, y en este caso, `acLabel`, `acTextBox` o `acCommandButton`.

Hay una función adicional que tiene relación con `TypeOf`. Es la función `TypeName`.

`TypeName` devuelve una cadena con información acerca de la variable que se le pase como parámetro.

Su sintaxis es `TypeName (nombrevariable)`

La variable *nombrevariable* puede ser de prácticamente cualquier tipo.

En nuestro caso, vamos a añadir la línea escrita en negrita al código anterior

```

For Each ctr In Me.Controls
    ctr.Height = conlngAltura
    Debug.Print TypeName (ctr)

```

Este código nos irá imprimiendo los tipos de controles que va examinando; en concreto las cadenas `Textbox`, `Label` y `CommandButton`.

Dejo al lector el desarrollo de un código equivalente al visto, pero utilizando la propiedad `ControlType` o la función `TypeName`.

Tanto **ControlType** como **TypeOf** `ctr` **Is** `Control` o **TypeName** tienen utilidad, por ejemplo para asignar dinámicamente propiedades a controles y evitar aquellas propiedades que no pertenecen al control con el que se esté trabajando.

Esto sucede por ejemplo, en aplicaciones multi idioma, en las que queremos adaptar los textos de las diferentes Interfaces al idioma seleccionado por el usuario, de forma dinámica.

Otra ejemplo de aplicación sería diseñar formularios ajustables a la resolución que utilice el usuario en su pantalla.

Operadores de concatenación

Los operadores de **concatenación** sirven para unir varias cadenas en una sola.

Estos operadores son

- **+** Suma
- **&** Ampersand

Ya he comentado la posibilidad que tiene el operador suma para unir cadenas, al igual que el operador **&**.

También he comentado mi preferencia por el operador **&** para efectuar estas tareas.

La sintaxis para estos operadores es

```
resultado = expresión1 + expresión2
```

```
resultado = expresión1 & expresión2
```

Operadores lógicos

Los operadores **lógicos** sirven para realizar operaciones lógicas con los operandos.

Estos operadores son

- **And** Conjunción Lógica
- **Or** Disyunción lógica
- **Xor** Exclusión lógica
- **Not** Negación lógica
- **Eqv** Equivalencia lógica
- **Imp** Implicación lógica

Los más usados son los cuatro primeros, **And**, **Or**, **Xor** y **Not**.

Los seis operadores, además de trabajar con expresiones de tipo lógico

```
Resultado = ExpresionBooleana Operador ExpresionBooleana
```

permiten realizar comparaciones a nivel de bit con operandos numéricos.

Los operandos puede tener el valor **Null**, en cuyo caso el resultado dependerá de si es el primero, el segundo ó los dos.

Operador And

El operador **And**, realiza una conjunción lógica entre dos expresiones operandos.

Su sintaxis es

resultado = *expresión1* **And** *expresión2*

Resultado puede ser cualquier variable numérica ó booleana.

resultado tomará el valor **True**, sólo y sólo si las dos expresiones son ciertas.

Las combinaciones de **Null** con **True** dan **Null**.

Null con **Null** da **Null**

Las combinaciones que contengan **False** dan **False**.

Cuadro de resultados del operador **And**

Expresión 1 ^a	Expresión 2 ^a	Resultado
True	True	True
True	False	False
True	Null	Null
False	True	False
False	False	False
False	Null	False
Null	True	Null
Null	False	False
Null	Null	Null

Veamos las siguientes expresiones

8 > 4 **And** 4 >= 3 → True

IsNumeric("1234") **And** Len("1234") = 4 → True

IsNumeric("Pepe") **And** Len("1234") = 4 → False

Las operaciones de **And** con números se realizan a nivel de Bits.

7 And 13 → 5

¿Cómo puede ser esto?

7, en binario es igual a 0111

13 es igual a 1101

El resultado será 0101

El binario 101 equivale en notación decimal al número 5

And devuelve el bit 1 sólo si los dos bits correspondientes de los operandos, valen 1.

Este operador es útil para poder averiguar si un determinado bit de una expresión está activado (tiene el valor 1).

Por ejemplo, si queremos averiguar directamente el valor del tercer bit de un número entero **M**, es suficiente ver el resultado de realizar **M And 4**. Si el resultado es **4** el tercer bit de **M** contiene el valor 1, si es 0 el tercer bit contendrá el valor 0.

En general si queremos averiguar el bit N° **N** de una expresión **M** comprobaremos si el resultado de **M And 2^(N-1)** es igual a **2^(N-1)**.

Hay determinados dispositivos en los que la configuración viene determinada por los bits de una determinada propiedad.

Operador Or

El operador **Or**, realiza una Disyunción lógica entre dos operandos.

Su sintaxis es

resultado = expresión1 Or expresión2

Resultado puede ser cualquier variable numérica ó boleana.

resultado tomará el valor **True**, si cualquiera de las dos expresiones es **True**.

Las combinaciones de **False** con **False** da **False**.

Null con **False** ó con **Null** da **Null**

Cuadro de resultados del operador **Or**

Expresión 1ª	Expresión 2ª	Resultado
True	True	True
True	False	True
True	Null	True
False	True	True
False	False	False
False	Null	Null
Null	True	True
Null	False	Null
Null	Null	Null

A nivel de bits, en operaciones con números, **Or** dará **1** salvo en el caso de que ambos bits valgan **0**.

5 **Or** 13 → 13
 0101
 1101
 1101

El binario **1101** es en notación decimal el número **13**

Con el operador **And**, hemos visto que podemos averiguar qué bit está activado en una expresión.

El operador **Or** nos permite definir el valor de un determinado bit de un valor.

Supongamos que tenemos un dispositivo que contiene la propiedad **Config** de tipo **Long**, que controla el funcionamiento del mismo.

Por circunstancias de la programación queremos hacer que el 4º bit, contando por la derecha, de dicha propiedad, tome el valor 1.

Para ello utilizaremos el operador **Or** de la siguiente manera:

```
Dispositivo.Config = Dispositivo.Config Or 2^(3)
```

En general para activar el bit número N de un valor M, haremos

```
M = M or 2^(N-1)
```

Operador Xor

El operador **Xor**, realiza una Exclusión lógica entre dos operandos.

Su sintaxis es

```
resultado = expresión1 Xor expresión2
```

Resultado puede ser cualquier variable numérica ó booleana.

Tomará el valor **True**, si una de las dos expresiones es **True** y la otra **False**.

Si cualquiera de ellas es **Null**, dará **Null**.

Si las dos son a la vez **False** o **True** dará **False**.

Cuadro de resultados del operador **Xor**

Expresión 1ª	Expresión 2ª	Resultado
True	True	False
True	False	True
False	True	True
False	False	False

En operaciones a nivel de bit, dará 1 si uno de los dos es 0 y el otro 1.

Si los dos bits fueran iguales dará 0.

```
5 Xor 13 → 8
0101
1101
1000
```

El binario 1000 es el número 8

Supongamos que queremos hacer que el 4º bit, contando por la derecha, de un valor **M** tome el valor 1.

Esto lo haremos en dos pasos

1. Comprobamos si el valor **M** ya tiene el bit a cero.
2. Si no fuera así lo cambiamos a cero usando **Xor**

```
If M And 8 = 0 then ' 8 es 2^3 ó lo que es lo mismo 2^(4-1)
M = M Xor 8
End if
```

Operador Not

El operador **Not**, realiza una Negación lógica sobre una expresión.

Su sintaxis es

$$\text{resultado} = \text{Not } \text{expresión}$$

Resultado puede ser cualquier variable numérica ó booleana.

Tomará el valor **True**, si *expresión* es **False** y a la inversa.

Si *expresión* fuese **Null**, devolverá **Null**.

Cuadro de resultados del operador **Not**

Expresión	Resultado
True	False
False	True
Null	Null

A nivel de bits también se puede utilizar el operador **Not**.

```

Not 13    →    -14
13          0000000000001101
Not 13     1111111111110010
  
```

El binario **1111111111110010** es el número **-14**

Operador Eqv

El operador **Eqv**, realiza una Equivalencia lógica entre dos expresiones.

Su sintaxis es

$$\text{resultado} = \text{expresión1 } \text{Eqv } \text{expresión2}$$

resultado puede ser cualquier variable numérica ó booleana.

Si alguna de las expresiones fuera **Null**, el resultado será **Null**.

Si los dos operandos fuesen **True** ó **False**, daría como resultado **True**.

En caso contrario daría **False**.

Cuadro de resultados del operador **Eqv**

Expresión 1ª	Expresión 2ª	Resultado
True	True	True
True	False	False
False	True	False
False	False	True

Podría considerarse como el operador inverso a **Xor**.

En operaciones a nivel de bit, dará 1 si los dos bits fueran iguales.

Si uno de los dos es 0 y el otro 1 dará 0.

```

5 Eqv 13 → -9
00000000000000000101
000000000000000001101
11111111111110111

```

El binario 11111111111110111 es el número -9

Operador Imp

El operador **Imp**, realiza una Implicación lógica entre dos expresiones.

Su sintaxis es

resultado = *expresión1* **Imp** *expresión2*

resultado puede ser cualquier variable numérica ó booleana.

Cuadro de resultados del operador **Imp**

Expresión 1ª	Expresión 2ª	Resultado
True	True	True
True	False	False
True	Null	Null
False	True	True
False	False	True
False	Null	True
Null	True	True
Null	False	Null
Null	Null	Null

De forma equivalente, a nivel de bits los resultados serían

Expresión 1ª	Expresión 2ª	Resultado
0	0	1
0	1	1
1	0	0
1	1	1

Prioridad de los operadores

Cuando tenemos una expresión con un operador que une dos operandos, no se plantea problema alguno de interpretación.

Si tenemos

```

intA = 2
intB = 4
intC = INTA + intB

```

La variable `intC` sabemos intuitivamente que tomará el valor 6.

En expresiones más complejas, puede que no lo tengamos tan claro.

Por ejemplo, si a continuación de ese código hiciéramos:

```
dblD = intA ^ 12 / intC
```

Suponemos que las variables están correctamente declaradas

¿Qué valor tomará `dblD`?

En concreto, `intC` a qué divide, al 12 o al resultado de `intA` elevado a 12.

La respuesta correcta es la segunda.

Si fuera la primera, `dblD` tomaría el valor 4.

En este caso toma el valor 682,666666666667, ya que primero efectúa la potencia, y posteriormente divide el resultado por 12.

Cada tipo de operador tiene una prioridad en su evaluación.

Si en una misma expresión se incluyen operadores de distintas categorías, primero se evalúan los de más prioridad, y en caso de haber de la misma, se empieza por los que están situados más a la izquierda.

La prioridad en los operadores sigue estos órdenes para cada tipo de operador

Aritméticos	De comparación	Lógicos
Exponenciación (^)	Igualdad (=)	Not
Negación (-)	Desigualdad (<>)	And
Multiplicación y división (*, /)	Menor que (<)	Or
División de enteros (\)	Mayor que (>)	Xor
Módulo aritmético (Mod)	Menor o igual que (<=)	Eqv
Adición y substracción (+, -)	Mayor o igual que (>=)	Imp
Concatenación de cadenas (&)	Like	
	Is	

Las multiplicaciones y divisiones se evalúan entre sí de izquierda a derecha, siguiendo el orden en el que están escritas.

Esto mismo ocurre con la suma y la resta.

Los operadores aritméticos se evalúan antes que los de comparación.

Todo esto lleva a la conveniencia de la utilización de paréntesis, para tener un perfecto control del orden de interpretación.

En una expresión lo primero que se evalúa son las subexpresiones contenidas dentro de paréntesis, si las hubiera.

En el caso del ejemplo, `dblD = intA ^ 12 / intC` si quisiéramos que primero se efectuara la división de 12 entre `intC` deberíamos haber escrito

```
dblD = intA ^ (12 / intC)
```

En el caso de la expresión:

```
4 Or 19 And 7 Or 13
```

se evalúa en este orden

1. 19 And 7 → 3
2. 4 Or 3 → 7
3. 7 Or 13 → 15

Si quisiéramos que evaluara primero 4 or 19, 7 or 13 y luego los respectivos resultados mediante And, deberíamos haber escrito

(4 Or 19) And (7 Or 13)

Que nos dará como resultado 7.

La utilización de paréntesis aclara el código y ayuda a evitar errores en el diseño del mismo, muchas veces con resultados inesperados, y de difícil detección.

Comencemos a programar con
VBA - Access

Entrega **16**

Código vs. Macros
Objeto DoCmd

Código frente a macros

Access posee un potente juego de macros que permiten un alto grado de automatización en una serie de tareas.

La mayor parte de los cursos de Access impartidos en los diversos centros de formación, incluyen en su última parte una introducción al manejo de las macros.

La cruda realidad es que la mayoría de las veces apenas si se llega a mostrar su manejo, si no es en un curso de los llamados “avanzados”.

En este momento no pretendo explicar cómo se manejan las Macros, pero ya que podemos vernos en la necesidad de transformar macros en código haré una pequeña introducción.

Si vamos a la ayuda de Access veremos que una macro se define como un conjunto de acciones creadas para automatizar algunas de las tareas comunes.

Veamos en la práctica cómo se manejan.

Vamos a crear un formulario y en él pondremos una etiqueta grande con un texto en rojo, por ejemplo **Formulario abierto**.

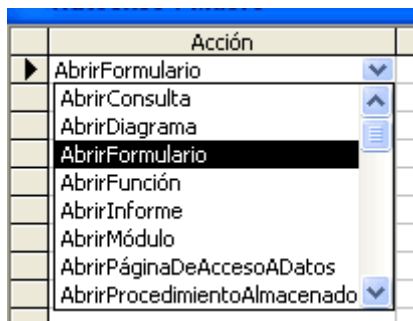
Guardamos el formulario con el nombre **PruebaMacro**.

Abrimos la ventana macros, pinchando en la pestaña macros y en el botón [**Nuevo**].

Una vez dentro podremos ver la columna **Acción**, y si activáramos la opción **Condiciones** del menú **Ver** nos aparecerá la columna **Condición**.

En la primera fila de columna Acción seleccionamos la acción [**AbrirFormulario**].

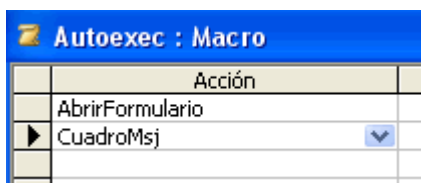
Inmediatamente después de hacerlo, nos aparece en la parte inferior unos cuadros en los que podemos introducir los **argumentos** de la macro.



Nombre del formulario	PruebaMacro
Vista	Formulario
Nombre del filtro	
Condición WHERE	
Modo de datos	
Modo de la ventana	Normal

Si nos ponemos en el cuadro **Nombre del formulario**, vemos que nos aparece una flecha hacia abajo que nos permite seleccionar un formulario.

Si la base de datos era nueva nos aparecerá únicamente el formulario **PruebaMacro** que acabamos de guardar. Seleccionaremos ese formulario.



Mensaje	Formulario abierto
Bip	Sí
Tipo	Información
Título	

Vamos a la fila siguiente de la columna Acción, y seleccionamos la acción [**CuadroMsj**], y en los cuadros de abajo ponemos:

En Mensaje: **Formulario abierto**, y en Tipo: **Información**.

A continuación presionamos en el botón guardar, asignándole como nombre **Autoexec**. Cerramos la ventana de macros, e incluso Access.

Volvemos a abrir el fichero mdb y veremos que ahora se abre el formulario PruebaMacro inmediatamente después de cargar el fichero. y nos muestra una cuadro de mensaje tipo a los que ya hemos visto con la función **MsgBox** en la Entrega 12.

Hemos comprobado lo siguiente: si a una macro, le ponemos como nombre **Autoexec**, se ejecutarán sus acciones en el momento de abrir el fichero Access.

Este es un comportamiento interesante al que en algún momento podremos sacar partido.

¿Y a cuento de qué viene ahora el hablar de las macros?. ¿No es ir hacia atrás?

Las macros ha sido una herramienta muy utilizada por “usuarios avanzados de Access” que no tenían conocimientos de programación y que querían dotar a sus aplicativos de cierto grado de automatismo.

Lo que quizás algunos de esos usuarios no sabían es que Access permite realizar una conversión directa de las macros a código VBA.

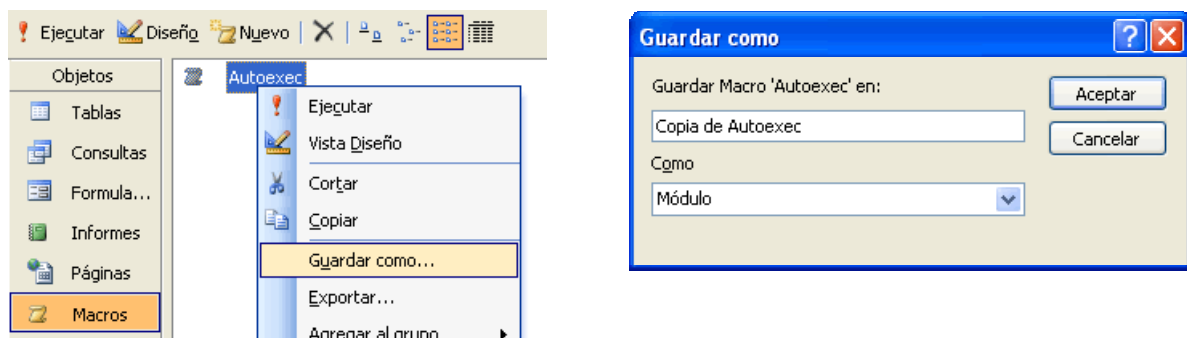
Las funciones y procedimientos de VBA tienen más flexibilidad y posibilidades que las macros. Además las macros no pueden realizar tareas como el control de errores, o el acceso a un único registro con la misma simplicidad que con VBA.

Vamos a la pestaña de Macros, colocamos el curso encima del nombre de la macro Autoexec, y presionamos el botón derecho del ratón.

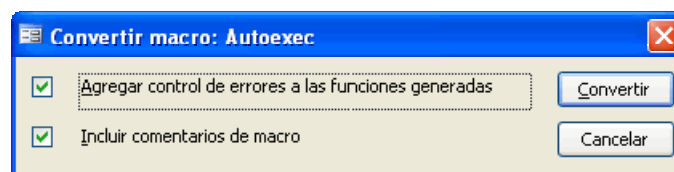
Nos aparecerá un menú contextual en el que seleccionaremos la opción [Guardar como...].

Tras esto nos aparece una nueva ventana en la que se nos propone guardar la macro con el nombre Copia de Autoexec.

En el segundo cuadro seleccionamos [Módulo] y presionamos el botón [Aceptar].



Tras ello nos aparece una nueva ventana en la que nos sugiere que la exportación a un módulo la va a efectuar con control de errores y comentarios.



No cambiamos esos valores y presionamos el botón [Convertir].

Si ahora nos vamos a la ventana de Módulos veremos que aparece un nuevo módulo llamado **Macro convertida - Autoexec**.

El proceso de conversión de macros se puede también realizar siguiendo estos pasos

1. Seleccione la macro a convertir en la ventana de macros.
2. Seleccionamos la opción de menú **[Herramientas]** y dentro de ella **[Macro]**
3. Del menú que se despliega seleccionamos **[Convertir macros a Visual Basic]**
4. Nos aparece el cuadro Convertir macro del punto anterior.

Se pueden convertir tanto las macros generales como las macros asociadas a formularios o informes.

Si abrimos el módulo generado, podremos ver que contiene el siguiente código.

```
Option Compare Database
```

```
Option Explicit
```

```
'-----
' Autoexec
'
'-----

Function Autoexec()
On Error GoTo Autoexec_Err

    DoCmd.OpenForm "PruebaMacro", acNormal, "", "", , acNormal
    Beep
    MsgBox "Formulario abierto", vbInformation, ""

Autoexec_Exit:
    Exit Function

Autoexec_Err:
    MsgBox Error$
    Resume Autoexec_Exit

End Function
```

Vemos que la macro ha sido cambiada por un módulo que contiene la función **Autoexec**.

*Quiero suponer que en vez de un procedimiento **sub**, genera una función que no devuelve explícitamente ningún valor, para que pueda ser utilizada directamente desde la pestaña eventos de la ventana de propiedades de un formulario, informe o control. Ya hablaremos más adelante de este tema...*

Esta función tiene dos partes que se corresponden con las dos acciones de la macro.

La acción **AbrirFormulario** se sustituye por el método **OpenForm** del objeto **DoCmd**.

La acción **CuadroMsj** se sustituye por la función **MsgBox** que ya hemos visto.

Haré mención aparte a la instrucción **Beep**.

Esta instrucción en teoría sirve para que el equipo emita un sonido de advertencia.

Dependiendo de la tarjeta de sonido que se tenga, y la configuración del software, se oirá un pitido, un sonido o incluso puede que no se oiga absolutamente nada.

Las mismas acciones que hemos puesto en la macro **Autoexec**, podríamos haberlas puesto en un mismo fichero de Macro, como dos macros independientes.

Para ello vamos a hacer lo siguiente:

Desde la ventana de Macros, ponemos el cursor del ratón encima de la macro Autoexec y presionamos el botón derecho del ratón (el izquierdo para los zurdos que así lo tengan configurado).

Nos aparecerá un menú contextual, una de cuyas opciones es **Cambiar nombre**.

Cambiamos el nombre de la macro **Autoexec**, por el de **MacroInicio**.

A continuación pulsamos el botón de Diseño (el la escuadra, la regla y el lápiz) y se nos abrirá la macro en vista diseño.

Nombre de macro	Acción
AbrirFormulario	
CuadroMsj	

Nombre de macro	Acción
	AbrirFormulario
	CuadroMsj

Vamos a activar, si no lo estuviera, una columna a la izquierda que contendrá el nombre de la macro.

Recordemos que hay que hacer visible, en el editor de macros, la columna correspondiente al nombre de macro. Para ello, desde el editor de Macros, en la opción de menú **[Ver]** activamos la opción **[Nombres de macro]**.

Tras esto nos aparecerá una columna vacía a la izquierda.

Por pura cuestión de orden vamos a separar las dos acciones, que se convertirán en macros independientes.

Para ello ponemos el cursor a la izquierda de la celda donde aparece CuadroMsj, y pulsamos dos veces en la opción de menú **[Insertar] [Filas]**.

Ahora nos ponemos a la izquierda de la primera Acción (AbrirFormulario) y escribimos como nombre para esa macro **Autoexec**.

En la celda que está a la izquierda de la acción CuadroMsj escribimos como nombre de la macro algo tan original como **Mensaje**.

Con esto ya tenemos dos macros diferentes escritas en el mismo fichero de macros.

En el caso anterior teníamos una única macro llamada Autoexec que ejecutaba 2 acciones diferentes.

Ahora, al arrancar el fichero de access se ejecutará la macro **Autoexec**, que nos abrirá el formulario, pero no se ejecutará la acción de la macro **Mensaje**. Puedes comprobarlo cerrando el fichero access y volviéndolo a abrir.

Para solventar este problema, podemos llamar a la macro Mensaje desde la macro Autoexec. Lo podemos conseguir mediante la Acción EjecutarMacro.

En la columna Acción, nos ponemos debajo de la celda en la que está escrita la acción AbrirFormulario y seleccionamos la acción EjecutarMacro.

A continuación seleccionamos como parámetro la macro **MacroInicio.Mensaje**.

Nombre de macro
Número de repeticiones
Expresión de repetición

MacroInicio.Mensaje

Nombre de macro	Acción
Autoexec	AbrirFormulario
	EjecutarMacro
Mensaje	CuadroMsj

Guardamos todo y vemos que ahora sí que se muestra el cuadro de mensaje para indicarnos la apertura del formulario.

¿Cómo afectaría este cambio en las macros al código que se generaría con el conversor de macros a código VBA?

La respuesta la podemos obtener de forma inmediata.

Guardamos la macro, si es que no lo habíamos hecho, y desde la ventana macro, seleccionamos nuestra flamante **MacroInicio** y activamos la opción de menú **[Herramientas] [Macro] [Convertir Macros a Visual Basic]**

Se nos genera un nuevo módulo, esta vez con el nombre

Macro convertida- MacroInicio.

El código que contiene ese módulo es el siguiente

```
Option Compare Database
Option Explicit

'-----
' MacroInicio_Autoexec
'
'-----

Function MacroInicio_Autoexec()
On Error GoTo MacroInicio_Autoexec_Err

    DoCmd.OpenForm "PruebaMacro", acNormal, "", "", , acNormal
    DoCmd.RunMacro "MacroInicio.Mensaje", , ""

MacroInicio_Autoexec_Exit:
    Exit Function

MacroInicio_Autoexec_Err:
    MsgBox Error$
    Resume MacroInicio_Autoexec_Exit

End Function

'-----
' MacroInicio_Mensaje
'
'-----

Function MacroInicio_Mensaje()
On Error GoTo MacroInicio_Mensaje_Err
```

```
Beep
MsgBox "Formulario abierto", vbOKOnly, ""
```

```
MacrosInicio_Mensaje_Exit:
```

```
Exit Function
```

```
MacrosInicio_Mensaje_Err:
```

```
MsgBox Error$
```

```
Resume MacrosInicio_Mensaje_Exit
```

```
End Function
```

Así como la primera vez nos creó un módulo con una única función, al tener ahora dos macros en un mismo fichero de macros, nos crea un módulo con dos funciones.

¿Cuándo usar Macros y cuándo código VBA?

Las macros permiten automatizar, de una forma simple, tareas como Abrir un formulario, cerrar un informe o desplazarnos entre los registros de una tabla.

Si acudimos a la ayuda de Access, en el apartado con título semejante vemos que nos recomienda el uso de macros fundamentalmente para realizar asignaciones globales de teclas. Incluso para la realización de determinadas acciones al arrancar la base de datos, comenta la posibilidad de utilizar la opción Inicio del menú Herramientas.

Para el resto de las acciones recomienda la utilización de VBA.

La mayor parte de las acciones ejecutables mediante macros, pueden ser realizadas ventajosamente usando VBA mediante los métodos del objeto **DoCmd**, que las implementa.

Como recordatorio indicaré que si por ejemplo quisiéramos que al abrir una base de datos se realice una determinada acción podemos usar una macro llamada Autoexec.

Como también hemos comentado, ésta no es la única opción, ya que podríamos hacer que aparezca un formulario concreto al arrancar la aplicación, mediante la opción **[Inicio]** del menú **[Herramientas]** y efectuar las llamadas a los procedimientos que nos interesen desde el código de ese formulario.

El objeto DoCmd

El objeto **DoCmd** es un objeto específico de Access, creado para sustituir a las acciones de las Macros. De hecho sustituye con ventaja a casi todas ellas.

Hasta Access 97, no existía ese objeto, pero sí existía el procedimiento **DoCmd**.

Por tanto el objeto **DoCmd** empezó a existir con Access 97.

Los argumentos de la acción serán ahora los argumentos del método de **DoCmd**.

En la acción **AbrirFormulario**, poníamos como Nombre del formulario **PruebaMacro**, como vista **Formulario** y como modo de la ventana **Normal**.

Al método **OpenForm**, que abre un formulario, le pasamos como nombre del formulario (**FormName**) **"PruebaMacro"**, como vista (**View**) **acNormal**, y como tipo de ventana (**WindowMode**) **acNormal**.

Las constante **acNormal** está definida como una constante enumerada miembro de Access.AcFormView. Su valor numérico es 0.

Ya comentamos las **Constantes Enumeradas** en la entrega 12. Este tipo de constantes van a ir apareciendo con mucha frecuencia conforme vayamos avanzando en VBA.

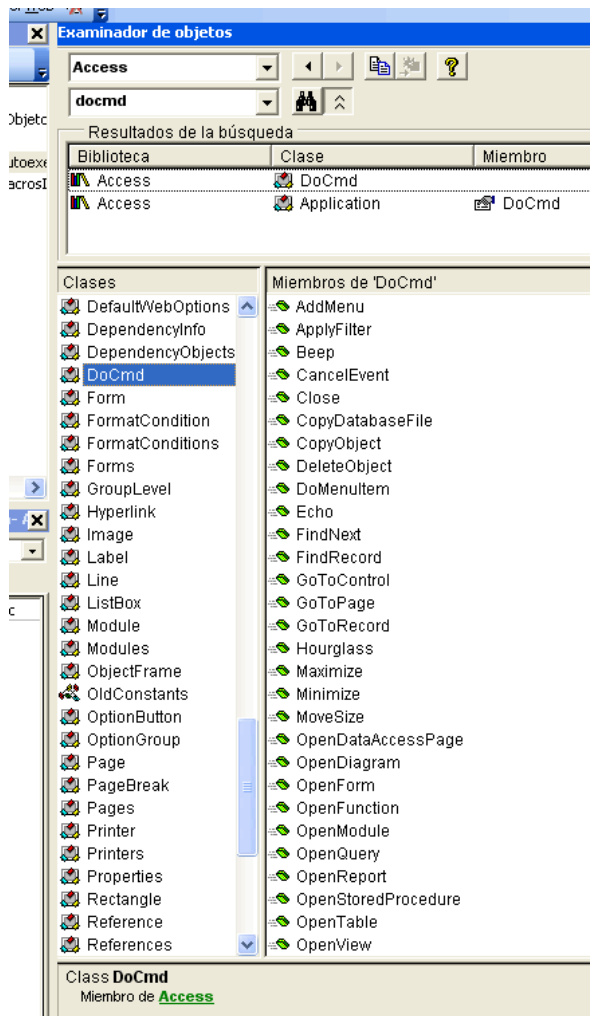
Como habrás podido ver hay un paralelismo total entre la acción de macro y el método correspondiente de **DoCmd**.

Pero no todas las Acciones de las macros están implementadas en los métodos de **DoCmd**.

Por ejemplo, en la conversión de macros a código VBA hemos visto que **CuadroMsj** se sustituye por la función **MsgBox**.

Otras acciones no implementadas en **DoCmd** son

Acción	Equivalencia en VBA
RunApp	Función Shell
RunCode	Llamada a la subrutina correspondiente
SendKeys	Instrucción SendKeys
SetValue	Operador = de asignación
StopAllMacros	Instrucciones Stop o End
StopMacro	Instrucciones Exit Sub o Exit Function



Podemos ver directamente los métodos implementados en la clase **DoCmd**, mediante la ventana del Examinador de objetos. Para activar esa ventana pulsaremos la tecla [F2] desde el editor de código.

Para localizar la clase **DoCmd** seleccionamos la biblioteca de **Access** y escribimos el nombre de la clase que queremos buscar.

Tras pulsar la tecla de búsqueda se nos posicionará en la clase **DoCmd**.

En la columna de la derecha podremos ver sus métodos.

Se puede comprobar que, al contrario de otras clases, no posee ni propiedades ni eventos.

Para la distinción entre Clase, Objeto, Propiedad, Evento y Método, os remito a las anteriores entregas en las que se daba una explicación a estos conceptos, así como a entregas posteriores en las que se profundizará sobre los mismos.

A continuación se describen los métodos implementados en **DoCmd**, junto con algunos pequeños ejemplos de código.

Si se requiriera una información más extensa se puede acudir a la ayuda de Access.

Método	Funcionalidad
AddMenu	<p>Permite crear barras de menú y menús contextuales.</p> <p>Se le pasa el <i>Nombre del menú</i>, el <i>Nombre de la macro de menú</i> y el <i>Texto de la barra de estado</i>.</p>
ApplyFilter	<p>Permite aplicar un filtro, una consulta o una cláusula "Tipo WHERE" de una instrucción SQL a una tabla, a un formulario o a un informe.</p> <p>Se le pasan como parámetros el <i>Nombre del filtro</i> y la <i>Condición</i> que aparecería después del Where.</p> <p>La siguiente línea de código hace que el formulario en donde está ubicada muestre sólo aquellos registros cuyo campo Marca comience por A, B ó C.</p> <pre>DoCmd.ApplyFilter , "Marca Like '[ABC]*'"</pre>
Beep	Se puede utilizar para emitir un sonido por los altavoces del PC
CancelEvent	Se utiliza para cancelar un evento. Sólo tiene efecto cuando se ejecuta como resultado de un evento. No usa parámetros.
Close	<p>Permite cerrar la ventana de un objeto.</p> <p>Hay que pasarle como parámetros el <i>Tipo</i> y el <i>Nombre del objeto</i>. También se le puede pasar un tercer argumento para indicar si se guardan o no los posibles cambios.</p> <p>Este código cierra el formulario actual</p> <pre>DoCmd.Close</pre> <p>Esta línea cierra el formulario PruebaMacro y guarda los posibles cambios sin preguntar si deben o no ser guardados.</p> <pre>DoCmd.Close acForm, " PruebaMacro", acSaveYes</pre>
CopyDatabaseFile	<p>Copia la base de datos conectada al proyecto actual en un archivo de base de datos de Microsoft SQL Server para la exportación.</p> <p>Los parámetros son <i>Nombre del Archivo Base De Datos</i>, <i>Sobrescribir un Archivo Existente</i> y <i>Desconectar Todos Los Usuarios</i>.</p> <p>Este ejemplo está sacado de la ayuda de Access.</p> <p>Copia la base de datos conectada al proyecto activo en un archivo de base de datos SQL Server.</p> <p>Si el archivo ya existiera, lo sobrescribe, y se desconectan todos los demás usuarios que estén conectados a la base de datos antes de realizar la copia.</p> <pre>DoCmd.CopySQLDatabaseFile _ DatabaseFileName:="C:\Export\Sales.mdf", _ OverwriteExistingFile:=True, _ DisconnectAllUsers:=True</pre> <p>Para pasar los parámetros utiliza parámetros "Con Nombre" mediante el operador de asignación :=.</p>
CopyObject	Permite copiar objetos de la base de datos, como Tablas,

Consultas, Formularios, Informes, Módulos, etc... en la base de datos actual o en otra que se especifique.

Permite incluso copiar un objeto cambiándole el nombre.

Como parámetros se pasa el *Nombre de la base de datos destino*, el *Nuevo nombre* que vaya a tener el objeto copiado, el *Tipo de objeto* a copiar y el *Nombre* del mismo.

Este código copia la tabla Marcas en la tabla CopiaMarcas

```
DoCmd.CopyObject, "CopiaMarcas", acTable, "Marcas"
```

Si se aplica sobre tablas vinculadas crea un nuevo vínculo.

DeleteObject

Elimina un objeto de una base de datos.

Hay que pasarle como parámetros el *Tipo* y el *Nombre* del objeto.

Este ejemplo borra la tabla CopiaMarcas.

```
DoCmd.DeleteObject acTable, "CopiaMarcas"
```

Si se aplica sobre tablas vinculadas elimina sólo el vínculo.

DoMenuItem

Este método es un método obsoleto, mantenido por compatibilidad con las versiones anteriores a Access 97.

Ha sido sustituido por [RunCommand](#).

Consulte la ayuda para mayor información

Echo

Se utiliza para mostrar u ocultar los resultados de la ejecución de una macro mientras se está ejecutando.

```
DoCmd.Echo True
```

Para desactivar **Eco** mostrando un mensaje

```
DoCmd.Echo False, "Ejecutándose código de VBA"
```

FindNext

Busca el siguiente registro que cumpla las condiciones definidas previamente mediante el método [FindRecord](#).

FindRecord

Busca un registro que cumpla determinados criterios.

Usa los mismos criterios que la opción de menú [Edición] [Buscar].

Los criterios son *Cadena a buscar*, *Coincidencia* de la cadena de búsqueda, *Distinguir mayúsculas*, *Sentido* a buscar, buscar en *Campo activo* o en *Todos*, *Buscar el primer registro* o en *Siguientes*, buscar en *Campos con formato*, en el *Campo activo* y buscar *Primero*

El parámetro *Coincidencia* toma una de las siguientes constantes del tipo AcFindMatch

```
acAnywhere
acEntire (valor predeterminado)
acStart
```

Como *Sentido* toma una de las constantes AcSearchDirection

```
acDown
acSearchAll (valor predeterminado)
acUp
```

Como *Campo activo* toma una de las constantes AcFindField

```
acAll
```

acCurrent (valor predeterminado)

La siguiente instrucción busca el primer registro que contenga el texto "Olaz" en cualquiera de sus campos, distinguiendo entre mayúsculas y minúsculas, considerando también los campos con formato.

```
DoCmd.FindRecord "Olaz", _
    acAnywhere, _
    True, _
    acSearchAll, _
    True
```

GoToControl

Desplaza el foco al campo o control especificado.

El siguiente código desplaza el foco al control txtFecha del formulario.

```
DoCmd.GoToControl "txtFecha"
```

GoToPage

Lleva el foco al primer control de la página especificada en un formulario en el que se han colocado saltos de página, o a una ficha determinada de un control ficha.

El parámetro *Número* de página es obligatorio.

Los parámetros opcionales posición *Horizontal* y posición *Vertical* son útiles si el formulario es mayor que el tamaño de la ventana de Windows. Este código traslada el foco al primer control de la página 3 del formulario.

```
DoCmd.GoToPage 3
```

GoToRecord

La acción GoToRecord (IrARegistro) convierte el registro especificado en el registro actual de un conjunto de resultados de una tabla, formulario o consulta.

Admite los siguientes parámetros:

Tipo de objeto, *Nombre*, *Registro* (Anterior, Siguiente, Primero, Último, Ir a o Nuevo), *Número* de registros a desplazar.

Como tipo de objeto se utiliza una constante AcDataObjectType

Esta constante puede tomar los valores

```
acActiveDataObject (Valor predeterminado)
acDataForm
acDataFunction
acDataQuery
acDataServerView
acDataStoredProcedure
acDataTable
```

El parámetro *Registro* puede tomar el valor de una constante del tipo AcRecord. Sus posibles valores son

```
acFirst
acGoTo
acLast
acNewRec
acNext (Valor predeterminado)
acPrevious
```

Este código activa el séptimo registro del formulario "Empleados".

```
DoCmd.GoToRecord acDataForm, "Empleados",
acGoTo, 7
```


Hourglass	<p>Hace aparecer un reloj de arena en vez del cursor normal del ratón.</p> <p>Si el parámetro es False, vuelve a colocar el cursor por defecto.</p> <pre>DoCmd.Hourglass True</pre>
Maximize	<p>Maximiza la ventana actual hasta ocupar totalmente la ventana de Windows. No tiene parámetros.</p> <pre>DoCmd.Maximize</pre>
Minimize	<p>Minimiza la ventana actual y la coloca en la parte inferior de la ventana de Microsoft Access.</p> <pre>DoCmd.Minimize</pre>
MoveSize	<p>Permite desplazar y cambiar el tamaño de la ventana activa. Se corresponde con la macro DesplazarTamaño.</p> <p>Sus parámetros son: <i>PosiciónHorizontal</i> y <i>Vertical</i> de la esquina superior izquierda, más la <i>Anchura</i> y <i>Altura</i> nuevos de la ventana. Los cuatro parámetros son opcionales.</p> <p>Este código cambia la posición izquierda de la ventana actual, así como su altura.</p> <pre>DoCmd.MoveSize 3000, , , 4000</pre>
OpenDataAccessPage	<p>Abre una página de acceso a datos.</p> <p>Sus parámetros son el <i>Nombre</i> de la página y el <i>Modo de apertura</i>.</p> <p>En el parámetro <i>Modo de apertura</i> se utiliza una de las constantes del tipo <code>AcDataAccessPageView</code>.</p> <p>Sus valores pueden ser</p> <p style="padding-left: 40px;"><code>acDataAccessPageBrowse</code> (predeterminado)</p> <p style="padding-left: 40px;"><code>acDataAccessPageDesign</code> (Vista Diseño)</p> <p>El siguiente código abre la página Alumnos en modo Examinar.</p> <pre>DoCmd.OpenDataAccessPage "Alumnos", _ acDataAccessPageBrowse</pre>
OpenDiagram	<p>Este método sirve, en un fichero de Proyecto (.adp) conectado a una base de datos SQLServer, para abrir en modo diseño un diagrama de relaciones definido en la base de datos.</p> <p>Como parámetro se introduce el <i>Nombre</i> del diagrama.</p> <p>En esta línea se abre el diagrama guardado previamente como <code>ModeloDeDatos</code>.</p> <pre>DoCmd.OpenDiagram "ModeloDedatos"</pre>
OpenForm	<p>Abre el formulario especificado</p> <p>Se pueden pasar hasta 7 parámetros que controlarán el <i>Nombre</i> del formulario a abrir, el <i>Tipo</i> de vista como se <i>Mostrará</i> el formulario, el <i>Filtro</i> que se aplicará a los datos, una <i>Cláusula Where</i> que define las características que deben cumplir los datos, el <i>Modo como se editarán</i> (sólo lectura, etc.) y el <i>Modo como se mostrará</i> la ventana. Adicionalmente se le puede incluir los <i>Argumentos de apertura</i>.</p> <p>Como <i>Tipo</i> de vista se puede usar una de las siguientes constantes <code>AcFormView</code></p>

acDesign
acFormDS
acFormPivotChart
acFormPivotTable
acNormal (predeterminado)
acPreview

El Modo de edición toma una constante AcFormOpenDataMode

acFormAdd Se puede agregar registros nuevos pero no se pueden modificar los existentes.

acFormEdit Se pueden modificar los registros existentes y agregar registros nuevos.

acFormPropertySettings (predeterminado)

acFormReadOnly Los registros sólo se pueden ver.

El *Modo como se mostrará* la ventana toma uno de los valores AcWindowMode

acDialog El formulario será Modal y Emergente.

acHidden El formulario estará oculto.

acIcon Se abre minimizado.

acWindowNormal valor predeterminado

La siguiente línea abre el formulario Alumnos, en modo de sólo lectura, con los datos de los alumnos residentes en Pamplona.

```
DoCmd.OpenForm "Alumnos", , , _
"Poblacion = 'Pamplona'", acFormReadOnly
```

OpenFunction

Abre una función definida por el usuario en una base de datos de Microsoft SQL Server para verla desde Microsoft Access.

Los parámetros son el nombre de la función, la *Vista* como se va a mostrar la función, y el *Modo* de visualización o edición de los datos.

La Vista puede ser una constante del tipo AcView.

acViewDesign Abre la función en la vista Diseño.

acViewNormal (predeterminado).

acViewPivotChart Vista Gráfico dinámico.

acViewPivotTable Vista Tabla dinámica.

acViewPreview Abre la función en la Vista preliminar.

El modo de visualización es del tipo AcOpenDataMode.

acAdd Abre la función para la inserción de datos.

acEdit (predeterminado). Abre la función para actualizar los datos existentes.

acReadOnly Abre la función en Sólo lectura.

La siguiente línea abre la función Ajuste en vista normal y en el modo edición de datos.

```
DoCmd.OpenFunction "Ajuste", , acEdit
```

OpenModule

Abre un procedimiento dentro de un módulo en el modo diseño.

Como parámetros se pueden pasar el *Nombre del módulo* y el del *Procedimiento* a abrir.

Si sólo se indica el nombre del módulo se abre en el primer procedimiento del mismo.

Si sólo se indica el nombre del procedimiento, lo busca entre

todos los módulos y si lo encuentra lo abre en el mismo.
 Si no existiera el módulo generará el error 2516.
 Si no existiera el procedimiento generará el error 2517.
 Esta línea abre el módulo Utilidades en la función Edad.

```
DoCmd.OpenModule "Utilidades", "Edad"
```

OpenQuery

Se puede utilizar para abrir una consulta de selección o una consulta de referencias cruzadas en la *Vista* Hoja de datos, diseño o en vista preliminar.

Esta acción ejecuta una consulta de acción. También se puede seleccionar un modo de entrada de datos para la consulta.

Los parámetros a introducir son nombre de la consulta, modo de presentación y *Modo de edición* de los datos.

El parámetro *Vista* es del tipo AcFormView ya visto en **OpenForm**.

El *Modo de edición* de los datos es del tipo AcOpenDataMode

```
acAdd
acEdit (predeterminado)
acReadOnly
```

El siguiente código ejecutará una consulta de acción llamada ctaActualizarPrecios.

```
DoCmd.OpenQuery "ctaActualizarPrecios"
```

Este código abrirá una consulta llamada ctaPrecios en modo Vista Previa.

```
DoCmd.OpenQuery "ctaPrecios ", acViewPreview
```

Este código abrirá la misma consulta pero en modo edición.

```
DoCmd.OpenQuery "ctaPrecios ", , acEdit
```

OpenReport

Abre el informe especificado

Los parámetros controlarán el nombre del informe, la vista preliminar o en modo diseño, el filtro que se aplicará a los datos, y la condición Where que deben cumplir los datos, el modo de presentación de la ventana del informe, pudiéndose incluir argumentos de apertura.

La siguiente línea abre el Informe infAlumnos, en vista previa, con los datos de los alumnos residentes en Pamplona.

```
DoCmd.OpenReport "infAlumnos", _
    acViewPreview, , _
    "Poblacion = 'Pamplona'"
```

OpenStoredProcedure

Abre un procedimiento almacenado de una base de datos SQLServer desde un archivo de proyecto .adp

Como argumentos se pasan el Nombre del procedimiento, el *Modo de la Vista* del tipo **AcView** visto en **OpenFunction** y el *Modo de edición* de los datos del tipo AcOpenDataMode visto en **OpenQuery**.

El siguiente código abre el procedimiento almacenado ControlAlumnos en el modo Normal pero de sólo lectura.

```
DoCmd.OpenStoredProcedure "ControlAlumnos", _
    acViewNormal", _
    acReadOnly
```

OpenTable

Abre una tabla en vista hoja de datos, diseño o vista previa, y permite, en su caso, especificar el *Modo de la Vista* (tipo AcView) y el del *Modo de edición* de los datos (tipo AcOpenDataMode).

El siguiente código abre la tabla Alumnos en el modo Normal pero de sólo lectura.

```
DoCmd.OpenTable "Alumnos", _
    acViewNormal", _
    acReadOnly
```

OpenView

Abre una vista de una base de datos SQLServer desde un archivo de proyecto .adp

La vista puede estar en modo hoja de datos, diseño o vista previa.

Como parámetros se pasan el *Nombre* de la vista, el *Modo como se va a ver* (AcView) y el *Modo de edición* de los datos (tipo AcOpenDataMode).

El siguiente código abre la vista AlumnosMatriculados en el modo tabla dinámica de sólo lectura.

```
DoCmd.OpenView "AlumnosMatriculados", _
    acViewPivotTable, _
    acReadOnly
```

OutputTo

Este método permite presentar los datos de un objeto (tabla, consulta, formulario, informe, módulo, página de acceso a datos, vista, procedimiento almacenado o función) en formato HTML (*.htm; *.html), texto (*.txt), Microsoft Active Server Pages (*.asp), Excel (*.xls), Microsoft Internet Information Server IIS (*.htx, *.idc), texto enriquecido (*.rtf), Páginas de acceso a datos (*.htm; *.html) o datos XML (*.xml).

Como página de acceso a datos sólo se pueden presentar los formularios e informes.

Las páginas de acceso a datos sólo se pueden presentar en formato HTML.

Los formatos de IIS y ASP sólo están disponibles para tablas, consultas y formularios.

Los argumentos del método son

Tipo de objeto, Nombre del objeto, Formato de salida, Archivo de salida, Autoinicio, Archivo de plantilla, Codificación.

Como *Tipo de objeto* se utiliza una constante AcOutputObjectType

```
acOutputDataAccessPage (No permitido)
acOutputForm
acOutputFunction
acOutputModule
acOutputQuery
acOutputReport
acOutputServerView
acOutputStoredProcedure
acOutputTable
```

El siguiente código guarda la tabla Alumnos en el fichero **Alumnos.rtf** ubicado en la carpeta **C:\Datos** e inmediatamente después lo abre con Word para su posible edición.

```
DoCmd.OutputTo acOutputTable, _
                "Alumnos", _
                acFormatRTF, _
                "C:\Datos\Alumnos.rtf", _
                True
```

Consulte la ayuda de Access para completar esta información.

PrintOut

Imprime el objeto activo (hoja de datos, informe, formulario, página de acceso a datos o módulo).

Se puede especificar el *Rango* a imprimir, páginas *Desde* y *Hasta* las que imprimir, Calidad de la impresión, *Número de copias*, e *Intercalar copias*.

Como *Rango* se utiliza una constante del tipo AcPrintRange.

```
acPages
acPrintAll (Valor predeterminado)
acSelection
```

Como *Calidad* se utiliza una constante del tipo AcPrintQuality.

```
acDraft
acHigh (Valor predeterminado)
acLow
acMedium
```

El siguiente código imprimirá 4 copias de todo el objeto actual en alta calidad, intercalando las páginas.

```
DoCmd.PrintOut acPrintAll, , , _
                acHigh , _
                4, _
                True
```

Quit

Cierra todas las ventanas de Access y sale del programa.

Se puede introducir como parámetro una *Opción de salida*, constante del tipo AcQuitOption para que pregunte si tiene que guardar los cambios, los guarde directamente o no tenga en cuenta los cambios.

```
acQuitPrompt Muestra un cuadro de diálogo que le pregunta si desea guardar cualquier objeto de base de datos que se haya modificado pero no se haya guardado.
acQuitSaveAll (Predeterminado) Guarda todos los objetos sin mostrar un cuadro de diálogo.
acQuitSaveNone Sale de Microsoft Access sin guardar ningún objeto
```

La siguiente línea hace que se salga sin guardar los posibles cambios

```
DoCmd.Quit acQuitSaveNone
```

Rename

Sirve para cambiar el nombre a un objeto de la base de datos

Sus argumentos son: el *Nuevo nombre* del objeto, el *Tipo* de objeto y el *Nombre anterior* del mismo

Como *Tipo* de objeto se usa una constante del tipo `AcObjectType`

<code>acDataAccessPage</code>	<code>acDefault</code> (predeterminado)
<code>acDiagram</code>	<code>acForm</code>
<code>acFunction</code>	<code>acMacro</code>
<code>acModule</code>	<code>acQuery</code>
<code>acReport</code>	<code>acServerView</code>
<code>acStoredProcedure</code>	<code>acTable</code>

Esta línea de código cambia el nombre del formulario

"tmpCuentas" a "Cuentas".

```
DoCmd.Rename "Cuentas" , acForm, "tmpCuentas"
```

RepaintObject

Se usa para finalizar las actualizaciones pendientes de pantalla para un objeto de base de datos o para el objeto de base de datos activo, si no se especifica ninguno.

Incluyen los recálculos pendientes de los controles del objeto.

Como argumentos opcionales se pasan el *Tipo* de objeto del tipo `AcObjectType` (ver el método anterior) y el *Nombre* del mismo.

```
DoCmd.RepaintObject acForm, "Cuentas"
```

Requery

Actualiza los datos de un objeto, al volver a ejecutar la consulta origen del mismo.

Como parámetro utiliza el nombre del objeto. Se puede aplicar a formularios, subformularios, cuadros de lista y combinados, objetos ole y controles con funciones agregadas de dominio.

```
DoCmd.Requery "lstClientes"
```

Restore

Vuelve a mostrar, a su tamaño normal, una ventana de Windows que pudiera estar minimizada ó maximizada.

No utiliza parámetros, y se ejecuta en el objeto que lo llama.

```
DoCmd.Restore
```

RunCommand

El método `RunCommand` ejecuta un comando de menú integrado de barra de herramientas.

Reemplaza al método `DoMenuItem` de `DoCmd`.

Como parámetro se pasa el comando a ejecutar, representado por una de las constantes `AcCommand`.

El método `RunCommand` no se puede usar para ejecutar un comando en un menú o barra de herramientas personalizada.

Se usa con menús y barras de herramientas integradas.

La siguiente línea muestra el cuadro de diálogo:

```
[Acerca de Microsoft Office Access]
```

```
DoCmd.RunCommand acCmdAboutMicrosoftAccess
```

Hay una gran cantidad de constantes para llamar a cada uno de los posibles comandos de menú o de barra de herramientas de Microsoft Access.

Las constantes `AcCommand` son:

acCmdAboutMicrosoftAccess	acCmdChangeToLabel	acCmdDelete
acCmdAddInManager	acCmdChangeToListBox	acCmdDeleteGroup
acCmdAddToNewGroup	acCmdChangeToOptionButton	acCmdDeletePage
acCmdAddWatch	acCmdChangeToTextBox	acCmdDeleteQueryColumn
acCmdAdvancedFilterSort	acCmdChangeToToggleButton	acCmdDeleteRecord
acCmdAlignBottom	acCmdChartSortAscByTotal	acCmdDeleteRows
acCmdAlignCenter	acCmdChartSortDescByTotal	acCmdDeleteTab
acCmdAlignLeft	acCmdClearAll	acCmdDeleteTable
acCmdAlignmentAndSizing	acCmdClearAllBreakPoints	acCmdDeleteTableColumn
acCmdAlignMiddle	acCmdClearGrid	acCmdDeleteWatch
acCmdAlignRight	acCmdClearHyperlink	acCmdDemote
acCmdAlignToGrid	acCmdClearItemDefaults	acCmdDesignView
acCmdAlignTop	acCmdClose	acCmdDiagramAddRelatedTables
acCmdAlignToShortest	acCmdCloseWindow	acCmdDiagramAutoSizeSelectedTables
acCmdAlignToTallest	acCmdColumnWidth	acCmdDiagramDeleteRelationship
acCmdAnalyzePerformance	acCmdCompactDatabase	acCmdDiagramLayoutDiagram
acCmdAnalyzeTable	acCmdCompileAllModules	acCmdDiagramLayoutSelection
acCmdAnswerWizard	acCmdCompileAndSaveAllModules	acCmdDiagramModifyUserDefinedView
acCmdApplyDefault	acCmdCompileLoadedModules	acCmdDiagramNewLabel
acCmdApplyFilterSort	acCmdCompleteWord	acCmdDiagramNewTable
acCmdAppMaximize	acCmdConditionalFormatting	acCmdDiagramRecalculatePageBreaks
acCmdAppMinimize	acCmdConnection	acCmdDiagramShowRelationshipLabels
acCmdAppMove	acCmdControlWizardsToggle	acCmdDiagramViewPageBreaks
acCmdAppRestore	acCmdConvertDatabase	acCmdDocMaximize
acCmdAppSize	acCmdConvertMacrosToVisualBasic	acCmdDocMinimize
acCmdArrangelconsAuto	acCmdCopy	acCmdDocMove
acCmdArrangelconsByCreated	acCmdCopyDatabaseFile	acCmdDocRestore
acCmdArrangelconsByModified	acCmdCopyHyperlink	acCmdDocSize
acCmdArrangelconsByName	acCmdCreateMenuFromMacro	acCmdDocumenter
acCmdArrangelconsByType	acCmdCreateRelationship	acCmdDropSQLDatabase
acCmdAutoCorrect	acCmdCreateReplica	acCmdDuplicate
acCmdAutoDial	acCmdCreateShortcut	acCmdEditHyperlink
acCmdAutoFormat	acCmdCreateShortcutMenuFromMacro	acCmdEditingAllowed
acCmdBackgroundPicture	acCmdCreateToolbarFromMacro	acCmdEditRelationship
acCmdBackgroundSound	acCmdCut	acCmdEditTriggers
acCmdBackup	acCmdDataAccessPageAddToPage	acCmdEditWatch
acCmdBookmarksClearAll	acCmdDataAccessPageBrowse	acCmdEncryptDecryptDatabase
acCmdBookmarksNext	acCmdDataAccessPageDesignView	acCmdEnd
acCmdBookmarksPrevious	acCmdDataAccessPageFieldListRefresh	acCmdExit
acCmdBookmarksToggle	acCmdDatabaseProperties	acCmdExport
acCmdBringToFront	acCmdDatabaseSplitter	acCmdFavoritesAddTo
acCmdCallStack	acCmdDataEntry	acCmdFavoritesOpen
acCmdChangeToCheckBox	acCmdDataOutline	acCmdFieldList
acCmdChangeToComboBox	acCmdDatashheetView	acCmdFilterByForm
acCmdChangeToCommandButton	acCmdDateAndTime	acCmdFilterBySelection
acCmdChangeToImage	acCmdDebugWindow	acCmdFilterExcludingSelection

acCmdFind	acCmdJoinProperties	acCmdOpenStartPage
acCmdFindNext	acCmdLastPosition	acCmdOpenTable
acCmdFindNextWordUnderCursor	acCmdLayoutPreview	acCmdOpenURL
acCmdFindPrevious	acCmdLineUpIcons	acCmdOptions
acCmdFindPrevWordUnderCursor	acCmdLinkedTableManager	acCmdOutdent
acCmdFitToWindow	acCmdLinkTables	acCmdOutputToExcel
acCmdFont	acCmdListConstants	acCmdOutputToRTF
acCmdFormatCells	acCmdLoadFromQuery	acCmdOutputToText
acCmdFormHdrFtr	acCmdMacroConditions	acCmdPageHdrFtr
acCmdFormView	acCmdMacroNames	acCmdPageNumber
acCmdFreezeColumn	acCmdMakeMDEFile	acCmdPageProperties
acCmdGoBack	acCmdMaximumRecords	acCmdPageSetup
acCmdGoContinue	acCmdMicrosoftAccessHelpTopics	acCmdParameterInfo
acCmdGoForward	acCmdMicrosoftOnTheWeb	acCmdPartialReplicaWizard
acCmdGroupByTable	acCmdMicrosoftScriptEditor	acCmdPaste
acCmdGroupControls	acCmdMoreWindows	acCmdPasteAppend
acCmdHideColumns	acCmdNewDatabase	acCmdPasteAsHyperlink
acCmdHidePane	acCmdNewGroup	acCmdPasteSpecial
acCmdHideTable	acCmdNewObjectAutoForm	acCmdPivotAutoAverage
acCmdHorizontalSpacingDecrease	acCmdNewObjectAutoReport	acCmdPivotAutoCount
acCmdHorizontalSpacingIncrease	acCmdNewObjectClassModule	acCmdPivotAutoFilter
acCmdHorizontalSpacingMakeEqual	acCmdNewObjectDataAccessPage	acCmdPivotAutoMax
acCmdHyperlinkDisplayText	acCmdNewObjectDiagram	acCmdPivotAutoMin
acCmdImport	acCmdNewObjectForm	acCmdPivotAutoStdDev
acCmdIndent	acCmdNewObjectFunction	acCmdPivotAutoStdDevP
acCmdIndexes	acCmdNewObjectMacro	acCmdPivotAutoSum
acCmdInsertActiveXControl	acCmdNewObjectModule	acCmdPivotAutoVar
acCmdInsertChart	acCmdNewObjectQuery	acCmdPivotAutoVarP
acCmdInsertFile	acCmdNewObjectReport	acCmdPivotChartByRowByColumn
acCmdInsertFileIntoModule	acCmdNewObjectStoredProcedure	acCmdPivotChartDrillInto
acCmdInsertHyperlink	acCmdNewObjectTable	acCmdPivotChartDrillOut
acCmdInsertLookupColumn	acCmdNewObjectView	acCmdPivotChartMultiplePlots
acCmdInsertLookupField	acCmdObjBrwFindWholeWordOnly	acCmdPivotChartMultiplePlotsUnifiedScale
acCmdInsertMovieFromFile	acCmdObjBrwGroupMembers	acCmdPivotChartShowLegend
acCmdInsertObject	acCmdObjBrwHelp	acCmdPivotChartType
acCmdInsertPage	acCmdObjBrwShowHiddenMembers	acCmdPivotChartUndo
acCmdInsertPicture	acCmdObjBrwViewDefinition	acCmdPivotChartView
acCmdInsertPivotTable	acCmdObjectBrowser	acCmdPivotCollapse
acCmdInsertProcedure	acCmdOfficeClipboard	acCmdPivotDelete
acCmdInsertQueryColumn	acCmdOLEDELinks	acCmdPivotDropAreas
acCmdInsertRows	acCmdOLEObjectConvert	acCmdPivotExpand
acCmdInsertSpreadsheet	acCmdOLEObjectDefaultVerb	acCmdPivotRefresh
acCmdInsertSubdatasheet	acCmdOpenDatabase	acCmdPivotShowAll
acCmdInsertTableColumn	acCmdOpenHyperlink	acCmdPivotShowBottom1
acCmdInsertUnboundSection	acCmdOpenNewHyperlink	acCmdPivotShowBottom10
acCmdInvokeBuilder	acCmdOpenSearchPage	acCmdPivotShowBottom10Percent

acCmdPivotShowBottom1Percent	acCmdPreviewOnePage	acCmdRepairDatabase
acCmdPivotShowBottom2	acCmdPreviewTwelvePages	acCmdReplace
acCmdPivotShowBottom25	acCmdPreviewTwoPages	acCmdReportHdrFtr
acCmdPivotShowBottom25Percent	acCmdPrimaryKey	acCmdReset
acCmdPivotShowBottom2Percent	acCmdPrint	acCmdResolveConflicts
acCmdPivotShowBottom5	acCmdPrintPreview	acCmdRestore
acCmdPivotShowBottom5Percent	acCmdPrintRelationships	acCmdRowHeight
acCmdPivotShowBottomOther	acCmdProcedureDefinition	acCmdRun
acCmdPivotShowTop1	acCmdPromote	acCmdRunMacro
acCmdPivotShowTop10	acCmdProperties	acCmdRunOpenMacro
acCmdPivotShowTop10Percent	acCmdPublish	acCmdSave
acCmdPivotShowTop1Percent	acCmdPublishDefaults	acCmdSaveAllModules
acCmdPivotShowTop2	acCmdQueryAddToOutput	acCmdSaveAllRecords
acCmdPivotShowTop25	acCmdQueryGroupBy	acCmdSaveAs
acCmdPivotShowTop25Percent	acCmdQueryParameters	acCmdSaveAsASP
acCmdPivotShowTop2Percent	acCmdQueryTotals	acCmdSaveAsDataAccessPage
acCmdPivotShowTop5	acCmdQueryTypeAppend	acCmdSaveAsHTML
acCmdPivotShowTop5Percent	acCmdQueryTypeCrosstab	acCmdSaveAsIDC
acCmdPivotShowTopOther	acCmdQueryTypeDelete	acCmdSaveAsQuery
acCmdPivotTableClearCustomOrdering	acCmdQueryTypeMakeTable	acCmdSaveAsReport
acCmdPivotTableCreateCalcField	acCmdQueryTypeSelect	acCmdSaveLayout
acCmdPivotTableCreateCalcTotal	acCmdQueryTypeSQLDataDefinition	acCmdSaveModuleAsText
acCmdPivotTableDemote	acCmdQueryTypeSQLPassThrough	acCmdSaveRecord
acCmdPivotTableExpandIndicators	acCmdQueryTypeSQLUnion	acCmdSelectAll
acCmdPivotTableExportToExcel	acCmdQueryTypeUpdate	acCmdSelectAllRecords
acCmdPivotTableFilterBySelection	acCmdQuickInfo	acCmdSelectDataAccessPage
acCmdPivotTableGroupItems	acCmdQuickPrint	acCmdSelectForm
acCmdPivotTableHideDetails	acCmdQuickWatch	acCmdSelectRecord
acCmdPivotTableMoveToColumnArea	acCmdRecordsGoToFirst	acCmdSelectReport
acCmdPivotTableMoveToDetailArea	acCmdRecordsGoToLast	acCmdSend
acCmdPivotTableMoveToFilterArea	acCmdRecordsGoToNew	acCmdSendToBack
acCmdPivotTableMoveToRowArea	acCmdRecordsGoToNext	acCmdServerFilterByForm
acCmdPivotTablePercentColumnTotal	acCmdRecordsGoToPrevious	acCmdServerProperties
acCmdPivotTablePercentGrandTotal	acCmdRecoverDesignMaster	acCmdSetControlDefaults
acCmdPivotTablePercentParentColumnItem	acCmdRedo	acCmdSetDatabasePassword
acCmdPivotTablePercentParentRowItem	acCmdReferences	acCmdSetNextStatement
acCmdPivotTablePercentRowTotal	acCmdRefresh	acCmdShowAllRelationships
acCmdPivotTablePromote	acCmdRefreshPage	acCmdShowDirectRelationships
acCmdPivotTableRemove	acCmdRegisterActiveXControls	acCmdShowEnvelope
acCmdPivotTableShowAsNormal	acCmdRelationships	acCmdShowMembers
acCmdPivotTableShowDetails	acCmdRemove	acCmdShowNextStatement
acCmdPivotTableSubtotal	acCmdRemoveFilterSort	acCmdShowOnlyWebToolbar
acCmdPivotTableUngroupItems	acCmdRemoveTable	acCmdShowTable
acCmdPivotTableView	acCmdRename	acCmdSingleStep
acCmdPreviewEightPages	acCmdRenameColumn	acCmdSizeToFit
acCmdPreviewFourPages	acCmdRenameGroup	acCmdSizeToFitForm

acCmdSizeToGrid	acCmdToggleBreakPoint	acCmdViewShowPaneGrid
acCmdSizeToNarrowest	acCmdToggleFilter	acCmdViewShowPaneSQL
acCmdSizeToWidest	acCmdToolbarControlProperties	acCmdViewSmallIcons
acCmdSnapToGrid	acCmdToolbarsCustomize	acCmdViewStoredProcedures
acCmdSortAscending	acCmdTransferSQLDatabase	acCmdViewTableColumnNames
acCmdSortDescending	acCmdTransparentBackground	acCmdViewTableColumnProperties
acCmdSortingAndGrouping	acCmdTransparentBorder	acCmdViewTableKeys
acCmdSpeech	acCmdUndo	acCmdViewTableNameOnly
acCmdSpelling	acCmdUndoAllRecords	acCmdViewTables
acCmdSQLView	acCmdUnfreezeAllColumns	acCmdViewTableUserView
acCmdStartupProperties	acCmdUngroupControls	acCmdViewToolbox
acCmdStepInto	acCmdUnhideColumns	acCmdViewVerifySQL
acCmdStepOut	acCmdUpsizingWizard	acCmdViewViews
acCmdStepOver	acCmdUserAndGroupAccounts	acCmdVisualBasicEditor
acCmdStepToCursor	acCmdUserAndGroupPermissions	acCmdWebPagePreview
acCmdStopLoadingPage	acCmdUserLevelSecurityWizard	acCmdWebPageProperties
acCmdSubdatasheetCollapseAll	acCmdVerticalSpacingDecrease	acCmdWebTheme
acCmdSubdatasheetExpandAll	acCmdVerticalSpacingIncrease	acCmdWindowArrangelcons
acCmdSubdatasheetRemove	acCmdVerticalSpacingMakeEqual	acCmdWindowCascade
acCmdSubformDatasheet	acCmdViewCode	acCmdWindowHide
acCmdSubformDatasheetView	acCmdViewDataAccessPages	acCmdWindowSplit
acCmdSubformFormView	acCmdViewDetails	acCmdWindowUnhide
acCmdSubformInNewWindow	acCmdViewDiagrams	acCmdWordMailMerge
acCmdSubformPivotChartView	acCmdViewFieldList	acCmdWorkgroupAdministrator
acCmdSubformPivotTableView	acCmdViewForms	acCmdZoom10
acCmdSwitchboardManager	acCmdViewFunctions	acCmdZoom100
acCmdSynchronizeNow	acCmdViewGrid	acCmdZoom1000
acCmdTabControlPageOrder	acCmdViewLargelcons	acCmdZoom150
acCmdTableAddTable	acCmdViewList	acCmdZoom200
acCmdTableCustomView	acCmdViewMacros	acCmdZoom25
acCmdTableNames	acCmdViewModules	acCmdZoom50
acCmdTabOrder	acCmdViewQueries	acCmdZoom500
acCmdTestValidationRules	acCmdViewReports	acCmdZoom75
acCmdTileHorizontally	acCmdViewRuler	acCmdZoomBox
acCmdTileVertically	acCmdViewShowPaneDiagram	acCmdZoomSelection

RunMacro

Sirve para ejecutar una Macro grabada.

La Macro puede estar guardada de forma individual, o estar integrada dentro de un grupo de macros.

Como parámetros se pasa el *Nombre de la macro* y opcionalmente el *Número de veces* que se va a repetir la macro y una *Expresión numérica* que se evalúa cada vez que se ejecuta la macro. Si esta expresión diera False (0), se detendría la ejecución de la macro.

La siguiente línea de código ejecuta 3 veces la macro **Mensaje** guardada en el grupo de macros **MacrosInicio**. Esta macro es la que habíamos creado en un punto anterior.

```
DoCmd.RunMacro "MacrosInicio.Mensaje", 3
```

RunSQL

La acción RunSQL (EjecutarSQL) ejecuta una consulta de acción utilizando una instrucción SQL correspondiente. También puede utilizarse para ejecutar una consulta de definición de datos.

Como consultas de acción se puede anexar, eliminar y actualizar registros e incluso guardar un conjunto de datos resultado de una consulta dentro de una nueva tabla.

Como consulta de definición de datos se pueden usar las siguientes instrucciones SQL

Para crear una tabla	Create Table
Para modificar una tabla	Alter Table
Para borrar una tabla	Drop Table
Para crear un índice	Create Index
Para borrar un índice	Drop Index

El siguiente ejemplo crea la tabla Alumnos, con diferentes tipos de campo, utilizando el método RunSQL.

```
Public Sub CrearTablaAlumnos()
    Dim strSQL As String

    strSQL = "Create Table Alumnos " _
        & "(IDAlumno Integer, " _
        & "Nombre Varchar(20), " _
        & "Apellido2 Varchar(20), " _
        & "Apellido1 Varchar(20), " _
        & "FechaNacimiento Date, " _
        & "Sexo Varchar(1), " _
        & "Activo bit)"

    DoCmd.RunSQL strSQL
End Sub
```

Save

Guarda el objeto especificado o, si no se indica, el objeto activo.

Como parámetros opcionales se pueden pasar el *Tipo* de objeto y el *Nombre* del mismo.

El parámetro *Tipo de objeto* puede ser cualquiera de las constantes AcObjectType, vistas en **RenameObject**.

```
DoCmd.Save acForm, "AlumnosActivos"
```

SelectObject

Seleccionar el objeto especificado de la base de datos.

Como parámetros obligatorios hay que usar el Tipo de objeto (constante del tipo AcObjectType descrita en **RenameObject**) y el *Nombre* del mismo. Opcionalmente se puede indicar si se *Selecciona* el objeto en la ventana Base de datos, mediante **True** ó **False**.

El siguiente ejemplo selecciona el formulario Alumnos en la ventana Base de datos:

```
DoCmd.SelectObject acForm, "Alumnos", True
```

SendObject

Se utiliza para incluir una hoja de datos, formulario, informe,

módulo o página de acceso a datos de Access en un mensaje de correo electrónico. En las aplicaciones de correo electrónico que admitan la interfaz estándar Microsoft MAPI, se puede incluir objetos con los formatos de Microsoft Excel (*.xls), Texto MS-DOS (*.txt), Texto enriquecido (*.rtf), o HTML (*.html).

Como parámetros se utilizan

Tipo de objeto: Tabla, Consulta, Formulario, Informe, Módulo, Página de acceso a datos, Vista de servidor o Procedimientos almacenados. Parámetros adicionales son *Nombre del objeto*, *Formato de salida*, *Enviar a*, *Enviar copia a*, *Enviar copia oculta a*, *Asunto del mensaje*, *Texto del mensaje*, *Abrir (sí ó no) el programa de correo* para editar el mensaje antes de enviarlo, *Plantilla* para un archivo HTML.

El tipo de objeto a enviar viene dado por una constante del tipo `AcSendObjectType`

Puede ser una de las siguientes

```
AcSendObjectType.  
acSendDataAccessPage  
acSendForm  
acSendModule  
acSendNoObject (valor predeterminado)  
acSendQuery  
acSendReport  
acSendTable
```

El tipo de objeto a enviar viene dado por una de las siguientes constantes

```
acFormatXLS  
acFormatTXT  
acFormatRTF  
acFormatHTML
```

El siguiente código envía los datos de la tabla `DatosNumericos` como datos adjuntos en formato Excel, al correo correspondiente a Eduardo Olaz, una copia a Javier Itóiz y una copia oculta a Edurne Goizueta.

El mensaje tiene como encabezado `Datos Test maquinaria` y como texto `"Datos resultado de las pruebas"`.

El mensaje se envía sin necesidad de abrir el programa de correo.

```
DoCmd.SendObject acSendTable, _  
    "DatosNumericos", _  
    acFormatXLS, _  
    "Eduardo Olaz", _  
    "Javier Itoiz", _  
    "Edurne Goizueta", _  
    "Datos Test maquinaria", _  
    "Datos resultado de las pruebas", _  
    False
```

- SetMenuItem** Establece el estado de los elementos (habilitado o deshabilitado, activado o desactivado) de las barras de menú y barras de menú generales personalizadas creadas con macros de barras de menús.
- Los parámetros son *Índice de menú*, *Índice de comando*, *Índice de sub-comando* e *Indicador*.
- El *Indicador* es una constante del tipo `AcMenuItemType` y toma alguno de los siguientes valores
- `acMenuItemCheck`
 - `acMenuItemGray`
 - `acMenuItemUncheck`
 - `acMenuItemUngray` (valor predeterminado)
- Los índices empiezan con el valor 0.
- Esta línea de código desactiva el tercer comando del segundo menú de la barra de menús personalizada para la ventana activa:
- ```
DoCmd.SetMenuItem 1, 2, , acMenuItemGray
```
- SetWarnings** Activa o desactiva los mensajes de advertencia del sistema.
- Como parámetro *Activar advertencias* se pasa `True` ó `False`.
- Este código desactiva los mensajes de advertencia
- ```
DoCmd.SetMenuItem False
```
- ShowAllRecords** Quita los filtros aplicados a una tabla, conjunto de resultados de una consulta o del formulario activo y muestra todos los registros.
- ```
DoCmd.ShowAllRecords
```
- ShowToolbar** Muestra u oculta una barra de herramientas integrada o una barra de herramientas personalizada.
- Sus parámetros son el *Nombre de la barra* de herramientas, y una constante del tipo `AcShowToolbar` que indica cómo se quiere *Mostrar la barra* de herramientas.
- Los valores de `AcShowToolbar` pueden ser
- `acToolbarNo`
  - `acToolbarWhereApprop`
  - `acToolbarYes` (valor predeterminado)
- La siguiente línea de código muestra la barra de herramientas personalizada denominada **MiBarra** en todas las ventanas Microsoft Access que se vuelvan activas:
- ```
DoCmd.ShowToolbar "MiBarra" acToolbarYes
```
- Para más información vea, en la ayuda de Access la información sobre la Acción **ShowToolbar** (`MostrarBarraDeHerramientas`)
- TransferDatabase** Importa o exporta datos entre la base de datos de Microsoft Access actual o el proyecto de Microsoft Access (.adp) actual y otra base de datos.
- También puede vincular una tabla a la base de datos de Access actual desde otra base de datos.

Como parámetros podemos usar

Tipo de transferencia una constante del tipo `AcDataTransferType` que puede tomar los valores

```
acExport
acImport (valor predeterminado)
acLink
```

El segundo parámetro indica el *Tipo de base de datos*. El tipo viene indicado por una expresión de cadena que puede tomar los siguientes valores

```
Microsoft Access (predeterminada)
Jet 2.x
Jet 3.x
dBase III
dBase IV
dBase 5.0
Paradox 3.x
Paradox 4.x
Paradox 5.x
Paradox 7.x
ODBC Database
WSS
```

El siguiente parámetro es el *Nombre de la base de datos*.

A continuación se indica el *Tipo de objeto* que se va a importar ó exportar. Éste se define mediante una constante del tipo puede ser cualquiera de las constantes **AcObjectType**, vistas en **RenameObject**.

El parámetro *Origen* define el nombre de la tabla, consulta de selección u objeto de Access que se desea importar, exportar o vincular. Si la tabla es un fichero, como en el caso de las tablas DBF, se debe indicar la extensión de ésta.

Destino define el nombre que tendrá el objeto, en la base de datos destino, una vez importado, exportado o vinculado.

El parámetro *Estructura solamente* especifica si se va a importar o exportar la estructura de una tabla sin los datos.

El último parámetro *StoreLogin* especifica si, para una tabla vinculada desde la base de datos, se almacenan en la cadena de conexión la identificación de inicio de sesión y la contraseña de una base de datos ODBC.

Los siguientes ejemplos están basados en los que aparecen en la ayuda de Access, correspondientes al Método `TransferDatabase`

Este siguiente ejemplo importa el informe `rptVentasDeAbril` desde la base de datos Access `Ventas.mdb` al informe `VentasDeAbril` en la base de datos activa:

```
DoCmd.TransferDatabase acImport, _
    "Microsoft Access", _
    "C:\Mis Documentos\ Ventas.mdb", _
    acReport, _
    "rptVentasDeAbril", _
    "VentasDeAbril"
```

El siguiente ejemplo vincula la tabla de la base de datos ODBC Autores a la base de datos activa:

```
DoCmd.TransferDatabase acLink, _
"ODBC Database", _
"ODBC;DSN=DataSource1;" _
& "UID=User2;PWD=www;" _
& "LANGUAGE=us_english;" _
& "DATABASE=pubs", _
acTable, _
" Autores", _
"dbo Autores"
```

En el siguiente ejemplo se exporta el contenido de la tabla Clientes a una lista nueva denominada Lista de clientes en el sitio Windows SharePoint Services "http://example/WSSSite".

```
DoCmd.TransferDatabase transfertype:=acExport, _
databasetype:="WSS", _
databasename:="http://example/WSSSite", _
objecttype:=acTable, _
Source:="Clientes", _
Destination:=" Lista de clientes", _
structureonly:=False
```

TransferSpreadsheet

Importa o exporta datos entre la base de datos Access o el proyecto de Access (.adp) actual y un archivo de hoja de cálculo.

Se puede vincular los datos de una hoja de cálculo de Microsoft Excel a la base de datos de Access actual.

Puede establecerse vínculos, de sólo lectura, a los datos de un archivo de hoja de cálculo Lotus 1-2-3.

Los parámetros son.

Tipo de transferencia, una constante del tipo `AcDataTransferType` vista en el método **TransferDatabase**.

Tipo de hoja de cálculo del tipo `AcSpreadSheetType`

Puede tomar los valores

```
acSpreadsheetTypeExcel3
acSpreadsheetTypeExcel4
acSpreadsheetTypeExcel5
acSpreadsheetTypeExcel7
acSpreadsheetTypeExcel8 (valor predeterminado)
acSpreadsheetTypeExcel9 (valor predeterminado)
acSpreadsheetTypeLotusWJ2 - sólo versión Japonesa
acSpreadsheetTypeLotusWK1
acSpreadsheetTypeLotusWK3
acSpreadsheetTypeLotusWK4
```

Nombre de la tabla a la que se van a importar, de la que se van a exportar o con la que se van a vincular datos de hoja de cálculo.

Nombre de archivo desde el que se va a importar, al que se va a exportar o con el que se va a establecer un vínculo

Contiene nombres de campo. Especifica si la primera fila de la hoja de cálculo contiene los nombres de los campos.

Rango. Rango de celdas que se van a importar o vincular.

El siguiente ejemplo importa los datos ubicados en el rango (A1:G12), de la hoja de cálculo Lotus **DatosEmpleados.wk3** a la tabla **Empleados**.

Los nombres de los campos están contenidos en la primera fila.

```
DoCmd.TransferSpreadsheet acImport, _
    acSpreadsheetTypeLotusWK3, _
    "Empleados", _
    "C:\Lotus\DatosEmpleados.wk3", _
    True, _
    "A1:G12"
```

TransferSQLDatabase Transfiere toda la base de datos del tipo Microsoft SQL Server especificada a otra base de datos SQL Server.

La sintaxis completa de uso es la siguiente

```
DoCmd.TransferSQLDatabase ( _
    Servidor, _
    BaseDeDatos, _
    UsarConexiónDeConfianza, _
    InicioDeSesión, _
    Contraseña, _
    TransferirCopiaDatos)
```

Como parámetros se incluyen

Servidor (server): nombre del servidor al que se va a transferir la base de datos.

BaseDeDatos (DataBase): nombre de la nueva base de datos en el servidor especificado.

UsarConexiónDeConfianza (UseTrustedConnection): indicando con True que la conexión activa utiliza un inicio de sesión con privilegios de administrador del sistema. Si tomara otro valor, se deberán especificar los Parámetros *InicioDeSesión* y *Contraseña*.

InicioDeSesión (Login): Nombre de un inicio de sesión del servidor de destino con privilegios de administrador de sistema

Contraseña (Password): La contraseña para el inicio de sesión especificado en *InicioDeSesión*.

TransferirCopiaDatos (TransferCopyData): **True** si todos los datos de la base de datos se transfieren a la base de datos de destino. En caso contrario sólo se transfiere el esquema de la base de datos.

Este código transfiere la base de datos SQL Server actual a la nueva base de datos SQL Server denominada **CopiaGestion** en el servidor **Principal**. (El usuario debe disponer de privilegios de administrador del sistema en **Principal**.)

Se copian los datos y el esquema de la base de datos.

```
DoCmd.TransferCompleteSQLDatabase _
    Server:="Principal", _
    Database:="CopiaGestion", _
    UseTrustedConnection:=True, _
    TransferCopyData:=True
```

TransferText Importar o exporta texto entre la base de datos Access o el proyecto de Access (.adp) actual y un archivo de texto.

Puede vincular los datos de un archivo de texto a la base de datos de Access actual.

También puede importar, exportar y establecer vínculos con una tabla o lista de un archivo HTML.

La sintaxis completa de uso es la siguiente

```
DoCmd. TransferText ( _
    TipoDeTransferencia, _
    NombreDeEspecificación, _
    NombreDeLaTabla, _
    NombreDelFichero, _
    ContieneNombresDeCampo, _
    NombreDeLaTablaHTML, _
    PáginaDeCódigos)
```

Los parámetros son los siguientes

TipoDeTransferencia (**TransferType**): Importar, exportar datos o establecer un vínculo con archivos de texto con datos delimitados o de ancho fijo y archivos HTML. También se puede exportar datos a un archivo de datos de combinación de correspondencia de Microsoft Word. Su valor es una constante del tipo **AcTextTransferType** que puede tomar los siguientes valores

```
acExportDelim
acExportFixed
acExportHTML
acExportMerge
acImportDelim (valor predeterminado)
acImportFixed
acImportHTML
acLinkDelim
acLinkFixed
acLinkHTML
```

NombreDeEspecificación (**SpecificationName**): nombre de una especificación de importación o exportación creada y guardada en la base de datos activa. Con un archivo de texto de ancho fijo, se debe especificar un argumento o utilizar un archivo `schema.ini`, guardado en la misma carpeta que el archivo de texto importado, vinculado o exportado.

NombreDeLaTabla (**TableName**): Nombre de la tabla Access a la que desea importar, de la que desea exportar o a la que desea vincular datos de texto, o nombre de la consulta de Access cuyos resultados se desean exportar a un archivo de texto.

NombreDelFichero (**FileName**): Nombre completo, incluso ruta de acceso, del archivo de texto del que desea importar, al que desea exportar o con el que desea crear un vínculo.

ContieneNombresDeCampo (**HasFieldNames**): Si el valor es **True** o **-1**, se indica que la primera fila del archivo de texto se va a usar como nombres de campos al importar, exportar o vincular.

NombreDeLaTablaHTML (**HTMLTableName**): Nombre de la tabla o lista del archivo HTML que desea importar o vincular. Este argumento se omite salvo que el argumento *TipoDeTransferencia* se establezca en **acImportHTML** o **acLinkHTML**.

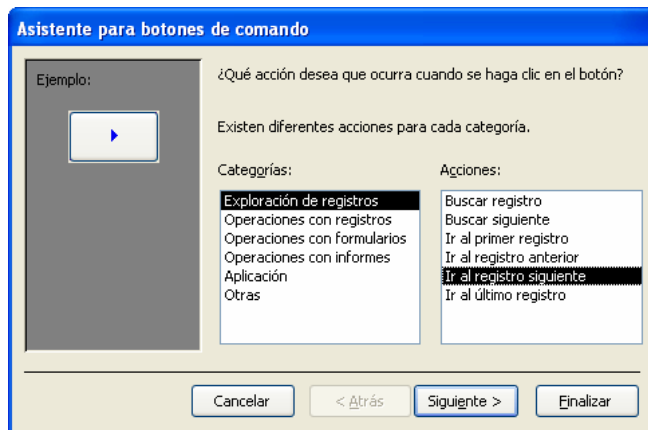
PáginaDeCódigos (CodePage): Valor Long que indica el conjunto de caracteres de la página de códigos

Uso de DoCmd en los Asistentes para controles

Los Asistentes para controles de Access, por ejemplo en los formularios, hacen un amplio uso del objeto DoCmd.



Si por ejemplo se añade un botón a un formulario, teniendo activado el asistente para controles, (botón señalado con la flecha roja) veremos que Access nos muestra un formulario en el que nos pide información acerca de la acción que queremos que ocurra cuando presionemos en el botón.



En nuestro caso vamos a crear un botón que haga que vayamos al registro siguiente respecto al registro actual.

Primero debemos decidir qué categoría vamos a utilizar, por lo que seleccionaremos en la izquierda la [Exploración de registros].

En la ventana de la derecha (acciones) seleccionaremos [Ir al registro siguiente].

Presionamos dos veces las sucesivas teclas [Siguiete] y ponemos como

nombre del botón cmdRegistroSiguiete, y presionamos la tecla [Finalizar].

Si hemos seguido estos pasos, nos mostrará en el formulario un botón con una flecha a la derecha.

Si abrimos el módulo de clase del formulario veremos que ha escrito el siguiente código

```
Private Sub cmdRegistroSiguiete_Click()
On Error GoTo Err_cmdRegistroSiguiete_Click

DoCmd.GoToRecord , , acNext

Exit_cmdRegistroSiguiete_Click:
Exit Sub

Err_cmdRegistroSiguiete_Click:
MsgBox Err.Description
Resume Exit_cmdRegistroSiguiete_Click

End Sub
```

Vemos que nos ha generado un procedimiento de evento que define qué debe pasar cuando se presione el botón [cmdRegistroSiguiete].

En concreto vemos que usa el objeto DoCmd con el método GotoRecord, y el parámetro acNext.

Esto hace que intente posicionarse en el registro siguiente. Incluso define un sistema para controlar posibles errores que se puedan generar.

Sugiero al lector que intente realizar un formulario, conectado a una tabla o consulta y que contenga botones para efectuar las siguientes operaciones:

1. Ir al primer registro
2. Ir al registro anterior
3. Ir al registro siguiente
4. Ir al último registro
5. Agregar nuevo registro
6. Guardar registro
7. Cerrar Formulario

Una vez realizadas estas operaciones analizar el código que se genera.

Así mismo sugiero que efectúe pruebas con diferentes métodos de DoCmd.

Comencemos a programar con
VBA - Access

Entrega **17**

Trabajar con ficheros

Trabajar con Ficheros

Desde VBA podemos acceder no sólo a las tablas de nuestras bases de datos, sino también leer y escribir en ficheros de texto e incluso otros tipos de formato, como ficheros de XML, XSL o HTML.

Tenemos a nuestra disposición una serie de procedimientos, e incluso objetos, que nos brindan esa posibilidad.

Además podemos buscar ficheros, borrarlos, crearlos, crear carpetas y otras operaciones estándar.

Trabajar con carpetas

Cuando desarrollamos una aplicación, debemos tener muy claro en qué carpeta estamos haciendo el desarrollo y dónde van a estar los ficheros de datos.

Por ejemplo, si tenemos una aplicación monopuesto, podríamos tener el fichero del programa en la carpeta C:\MiPrograma, y los datos en la carpeta C:\MiPrograma\Datos.

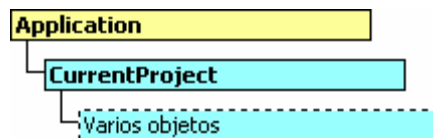
Si distribuimos la aplicación es muy probable que el cliente la instale en otra carpeta.

En ese caso los enlaces que tenemos previstos no funcionarían.

Esto nos lleva a la conclusión de que es clave el que podamos controlar en qué carpeta y unidad de disco está instalada la aplicación.

Este dato lo podemos obtener mediante el objeto **CurrentProject**.

Este objeto pertenece a su vez al objeto **Application** y tiene toda una serie de métodos, colecciones, objetos y métodos, que estudiaremos en una entrega posterior.



Los métodos que ahora nos interesan son **Path**, **Name** y **FullName**.

Veamos el siguiente código:

```
Public Function CarpetaActual() As String
    CarpetaActual = CurrentProject.Path
End Function
```

```
Public Function NombreBaseDatos() As String
    NombreBaseDatos = CurrentProject.Name
End Function
```

```
Public Function NombreCompletoBaseDatos() As String
    NombreCompletoBaseDatos = CurrentProject.FullName
End Function
```

Tras definir estas funciones podemos utilizarlas, por ejemplo para mostrar los datos en un cuadro de mensaje.

```
Public Sub PropiedadesBaseDatos()
```

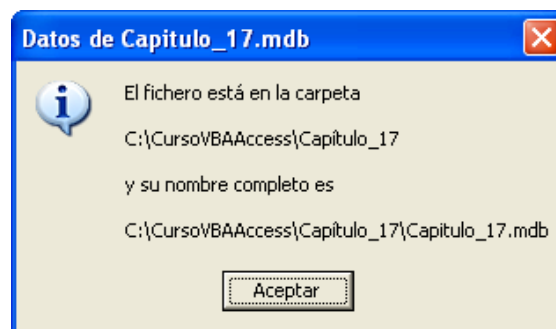
```

Dim strMensaje As String
strMensaje = "El fichero está en la carpeta " _
    & vbCrLf & vbCrLf _
    & CarpetaActual() _
    & vbCrLf & vbCrLf _
    & "y su nombre completo es" _
    & vbCrLf & vbCrLf _
    & NombreCompletoBaseDatos
MsgBox strMensaje, _
    vbInformation + vbOKOnly, _
    "Datos de " & NombreBaseDatos()

End Sub

```

Tras esto, si ejecuto el procedimiento **PropiedadesBaseDatos** en mi ordenador, me muestra el siguiente mensaje:



De esta manera, si los datos estuvieran en el fichero Datos.mdb ubicado en una carpeta llamada Datos, que cuelgue de la carpeta de la aplicación, podríamos definir una función llamada FicheroDatos que nos devuelva la ruta completa del fichero.

Podría ser algo así como:

```

Public Function FicheroDatos() As String
    Const conBaseDatos As String = "Datos.mdb"
    FicheroDatos = CarpetaActual() _
        & "\Datos\" & conBaseDatos
End Function

```

Si ejecuto en mi ordenador esta función, me devuelve la cadena

```
C:\CursoVBAAccess\Capítulo_17\Datos\Datos.mdb
```

Otro paso interesante sería comprobar si existen la carpeta y el fichero en nuestro disco.

Una de las formas de averiguarlo sería utilizando la función **Dir()**.

Función Dir

La función **Dir()**, nos devuelve un valor de tipo **String** que representa el nombre de un archivo, o carpeta que coincide con el patrón o atributo de archivo que se le ha pasado como parámetro. También nos puede devolver la etiqueta de volumen de una unidad de disco.

Su sintaxis es `Dir [(NombreDeRuta[, Atributos])]`

NombreDeRuta es una expresión de cadena que indica la ruta donde queremos buscar y el tipo de fichero que nos interesa.

Además de la ruta podemos especificar características que tenga el nombre del fichero, mediante el uso de los comodines "*" y "?". Por ejemplo para obtener los ficheros de cualquier tipo que empezaran por la letra A, en la carpeta "C:\Graficos\" podríamos escribir `Dir ("C:\Graficos\A*.*)`.

El símbolo "*" sustituye a cualquier número de caracteres.

El símbolo ? sustituye a 1 carácter, o ninguno.

Por ejemplo, `Dir ("C:\Graficos\????.*")` nos devolvería nombres de ficheros con hasta cuatro caracteres en el nombre, y con cualquier tipo de extensión.

`Dir ("C:\Graficos\?a??.*")` nos devolvería nombres con hasta cuatro caracteres en el nombre, que tuvieran la letra a como segundo carácter en el nombre, y con cualquier tipo de extensión. Estos nombres de fichero serían válidos para este último caso

```
gato.bmp
bart.jpg
```

Para obtener todos los nombres de fichero de una carpeta que cumplan determinada condición, la primera llamada se efectúa con el nombre de la ruta y sus atributos.

Las siguientes llamadas se hacen únicamente con `Dir ()` devolviendo el nombre de los sucesivos ficheros.

Mientras existan ficheros que cumplan la condición, `Dir ()` devolverá su nombre. Cuando deje de haberlos, devolverá la cadena vacía "".

Atributos es una constante. Los valores que puede tomar son:

<code>vbNormal</code>	(Predeterminado) Especifica archivos sin atributos.
<code>vbReadOnly</code>	Sólo lectura y sin atributos
<code>vbHidden</code>	Archivos ocultos y sin atributos
<code>VbSystem</code>	Del sistema y sin atributos
<code>vbVolume</code>	Etiqueta del volumen
<code>vbDirectory</code>	Carpetas y archivos sin atributos

Si por ejemplo quisiéramos obtener todos los ficheros que fueran del tipo `Gif` y que estuvieran en la carpeta `C:\Graficos\`, podríamos hacer:

```
Public Sub FicherosGif()
    Dim colFicheros As New Collection
    Dim strCarpeta As String
    Dim strFichero As String
    Dim i As Long

    strCarpeta = "C:\Graficos\"

    strFichero = Dir(strCarpeta & "*.gif")
    If strFichero = "" Then
```

```
Exit Sub
Else
  Do Until strFichero = ""
    colFicheros.Add strFichero
    strFichero = Dir()
  Loop
End If
For i = colFicheros.Count To 1 Step -1
  Debug.Print colFicheros(i)
  colFicheros.Remove (i)
Next i
Set colFicheros = Nothing
End Sub
```

En este ejemplo busca los ficheros del tipo gif, en la carpeta "C:\Graficos\" mediante la expresión

```
Dir(strCarpeta & "*.gif")
```

Si en la primera llamada encuentra alguno, `strFichero` será diferente a la cadena vacía, añade el resultado a la colección `colFicheros`.

Repite el bucle con `Dir()` y va añadiendo el resultado a la colección, hasta que devuelva la cadena vacía.

En ese momento hace un bucle que va recorriendo los diferentes elementos de la colección, desde el último hasta el primero, los muestra en la ventana Inmediato y los descarga de la colección.

Al final elimina el objeto `colFicheros` asignándole el valor `Nothing`.

Si en el primer `Dir` no hubiéramos indicado la carpeta donde queremos buscar, por ejemplo escribiendo

```
Dir("*.gif")
```

El procedimiento hubiera mostrado los posibles ficheros `gif` que existieran en la ruta de acceso actual. Una de las carpetas habituales suele ser la de **"Mis documentos"**.

El valor de esta carpeta la podemos obtener mediante la función `CurDir`.

Mediante la función `Dir`, podemos también obtener el nombre del volumen, usando la letra de la unidad y la constante `vbVolume`.

Por ejemplo:

```
Dir("C:", vbVolume)
Dir("E:", vbVolume)
```

Función CurDir

La función `CurDir()`, nos devuelve un valor de tipo `String` que representa el nombre de la ruta de acceso actual.

Su sintaxis es `CurDir[(Unidad)]`

Podemos usar el nombre de la unidad de la que queremos obtener la ruta de acceso.

`CurDir ("C")``CurDir ("D")``CurDir ("F")`

Si no se especifica la unidad de disco o el parámetro *unidad* es la cadena vacía (""), la función `CurDir` devuelve la ruta de acceso de la unidad de disco actual.

Podemos establecer desde VBA la ruta actual deseada.

Para ello se utiliza la Instrucción `ChDir`.

Instrucción ChDir

La instrucción `ChDir`, establece la carpeta actual. Para ello se le pasa como parámetro la carpeta deseada.

Su sintaxis es `ChDir Ruta`

Si la ruta no existiera, generaría el error 76

"No se ha encontrado la ruta de acceso"

Para cambiar la carpeta actual a "C:\Programa\Datos".

```
ChDir "C:\Programa\Datos"
```

`ChDir` no cambia la unidad de disco actual, sólo la carpeta del disco especificado

La siguiente instrucción cambia la carpeta de D:

```
ChDir "D:\Comercial\Datos"
```

Si la unidad seleccionada hasta ese momento era la C, seguirá siéndolo después de esta última instrucción.

Si en la ruta no se especifica la unidad, se cambia el directorio o carpeta predeterminado de la unidad actual.

Se puede hacer que la carpeta actual sea la de un nivel superior pasándole dos puntos.

```
ChDir "C:\Programa\Datos"
```

Tras esto la carpeta actual es "C:\Programa\Datos\".

```
ChDir ".."
```

Tras esto la carpeta actual pasa a ser "C:\Programa\".

Si quisiéramos cambiar la unidad de disco actual, debemos utilizar la instrucción `ChDrive`.

Instrucción ChDrive

La instrucción `ChDrive`, establece la unidad de disco actual.

Su sintaxis es `ChDrive Unidad`

```
ChDrive "D"
```

Si no existiera, o no estuviera disponible la unidad pasada como parámetro, generaría el error 68

"Dispositivo no disponible"

A veces podemos vernos en la necesidad de crear una carpeta nueva.

Para ello tenemos la instrucción `MkDir`.

Instrucción MkDir

La instrucción `MkDir`, crea una carpeta.

Su sintaxis es **MkDir Ruta**

```
Mkdir "C:\MiNuevaCarpeta"
```

```
Mkdir "C:\MiNuevaCarpeta\Subcarpeta_1"
```

```
Mkdir "C:\MiNuevaCarpeta\Subcarpeta_2"
```

Tras esto tendremos la carpeta `C:\MiNuevaCarpeta` con dos nuevas carpetas `Subcarpeta_1` y `Subcarpeta_2`.

Instrucción Rmdir

La instrucción **Rmdir**, elimina una carpeta existente.

Su sintaxis es **Rmdir Ruta**

Tras crear las carpetas del punto anterior podríamos eliminarlas utilizando

```
Rmdir "C:\MiNuevaCarpeta\Subcarpeta_1"
```

```
Rmdir "C:\MiNuevaCarpeta\Subcarpeta_2"
```

```
Rmdir "C:\MiNuevaCarpeta"
```

Para eliminar una carpeta con **Rmdir** es preciso que ésta esté vacía, es decir: no debe tener ningún archivo ni otra subcarpeta. Caso contrario generaría el error 75

```
"Error de acceso a la ruta o el archivo"
```

Instrucción Kill

La instrucción **Kill**, elimina un archivo existente.

Su sintaxis es **Kill Ruta**

El argumento requerido **Ruta** es una cadena que especifica el nombre o los nombres de los archivos que se van a eliminar.

Puede incluir la ruta completa del fichero.

Al igual que la función **Dir**, también admite utilizar los comodines asterisco "*" y "?".

Por ejemplo, si quisiéramos borrar todos los ficheros `gif` de la carpeta actual, usaríamos

```
Kill "*.gif"
```

Si quisiéramos borrar todos los ficheros de la carpeta actual, usaríamos

```
Kill " *.*"
```

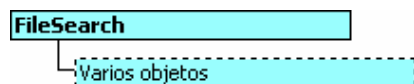
Si quisiéramos borrar todos los archivos que empezaran por **Temp** :

```
Kill "Temp*.*"
```

Si quisiéramos borrar todos los ficheros cuyo nombre empiece por **A** y tengan hasta 4 caracteres en el nombre.

```
Kill "A????.*"
```

El objeto FileSearch



Además de los procedimientos vistos con anterioridad, VBA incorpora un objeto que nos puede suministrar una información más completa que los procedimientos vistos en los puntos anteriores. Es el objeto **FileSearch**.

El objeto **FileSearch** es devuelto por el objeto **Application**, a través de su propiedad **FileSearch**.

Propiedades y métodos de FileSearch

Propiedad LookIn

La propiedad **LookIn** nos permite averiguar o establecer la carpeta en donde se va a efectuar la búsqueda de los ficheros. Es por tanto de lectura y escritura.

Propiedad Filename

La propiedad **Filename** devuelve o establece el nombre de los ficheros que se van a buscar. Como en el caso de la función Dir y del procedimiento Kill vistos en los puntos anteriores, se pueden utilizar caracteres comodín "*" y "?".

Propiedad SearchSubFolders

La propiedad **SearchSubFolders** (de tipo **Boolean**) devuelve o establece si se va a buscar, además de en la carpeta especificada por la propiedad **LookIn**, en las carpetas que “cuelguen” de ella.

Método Execute

El método **Execute** devuelve el valor 0 si no se ha encontrado ningún fichero con las características buscadas, y un **Long** positivo caso de encontrarse.

Su sintaxis es :

ObjetoFileSearch.Execute (SortBy, SortOrder, AlwaysAccurate)

El método **Execute** admite tres parámetros con los que se puede establecer la forma de ordenar los resultados:

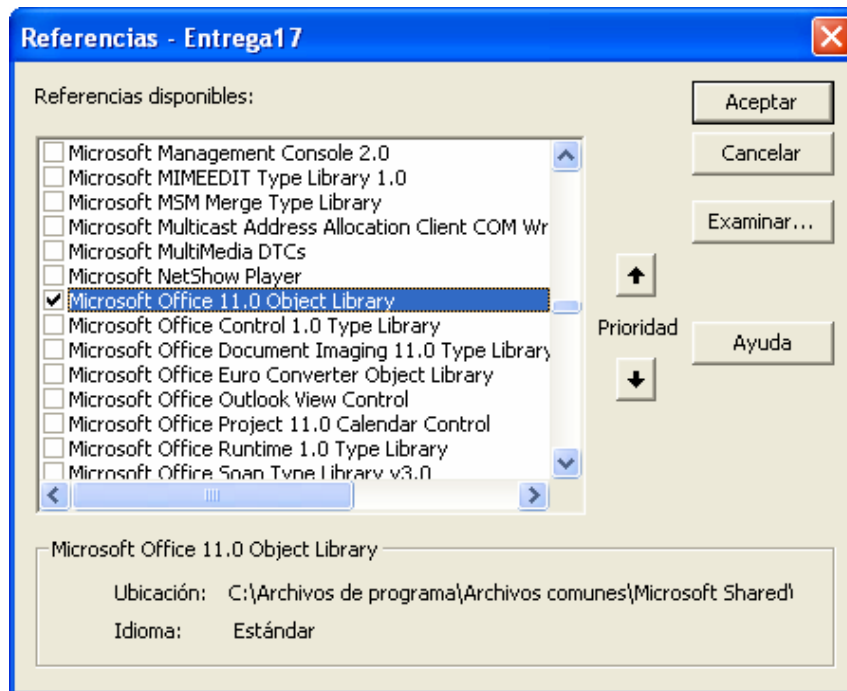
SortBy especifica el método utilizado para la ordenación de los datos obtenidos.

Nombre de la constante	Valor
msoSortByFileName <i>predeterminado</i>	1
msoSortByFileType	3
msoSortByLastModified	4
msoSortByNone	5
msoSortBySize	2

Nota:

Aunque FileSearch puede funcionar sin activar desde Access la referencia a la librería “Microsoft Office NN.N Object Library” si no se hace referencia explícita a ella, no se tiene acceso a las constantes propias de **FileSearch**, así como a la declaración de ciertas variables. Por ello es aconsejable activar la referencia a esa librería.

Para hacerlo, desde la opción de menú [Herramientas] se selecciona la opción [Referencias] y se activa la opción correspondiente a la Biblioteca. En mi ordenador queda así:



Para poder trabajar con **FileSearch**, aunque no se haya efectuado una referencia a la biblioteca mencionada, y que se identifique el valor de las constantes, en esta y sucesivas tablas pongo su nombre y su valor.

SortOrder especifica el orden de los datos obtenidos.

Nombre de la constante	Valor
msoSortOrderAscending <i>predeterminado</i>	1
msoSortOrderDescending	2

AlwaysAccurate Un valor **boolean** que especifica si se van a incluir los nombres de los ficheros que hayan sido agregados, modificados o eliminados desde que se actualizó por última vez el índice. Su valor predeterminado es True.

Este ejemplo busca los ficheros gráficos del tipo **jpg** en la carpeta "C:\Gráficos".

Los almacenará en la colección **FoundFiles** según su nombre y en orden ascendente.

```
Public Sub UsoDeExecute()
    Dim objFileSearch As Object
    Dim i As Long
    Set objFileSearch = Application.FileSearch
    With objFileSearch
        .LookIn = "C:\Gráficos"
        .FileName = "*.jpg"
        ' msoSortbyFileName tiene el valor 1 _
        msoSortOrderAscending tiene el valor 1
    If .Execute(SortBy:=msoSortByFileName, _
```

```

SortOrder:=msoSortOrderAscending) > 0 Then
MsgBox "Se han encontrado " & _
    .FoundFiles.Count & _
    " ficheros de tipo gráfico jpg"
For i = 1 To .FoundFiles.Count
    MsgBox .FoundFiles(i), _
        vbInformation, _
        "Fichero N° " & _
        Format(i, "0000")
Next i
Else
    MsgBox "No hay ficheros jpg"
End If
End With

End Sub

```

Propiedad LastModified

La propiedad **LastModified** (de tipo **Boolean**) devuelve o establece una constante que indica el tiempo transcurrido desde la última vez que se modificó y guardó el archivo.

El valor predeterminado para esta propiedad es **msoLastModifiedAnyTime**, con el valor 7.

Nombre de la constante	Valor
msoLastModifiedAnyTime <i>predeterminado</i>	7
msoLastModifiedLastMonth	5
msoLastModifiedLastWeek	3
msoLastModifiedThisMonth	6
msoLastModifiedThisWeek	4
msoLastModifiedToday	2
msoLastModifiedYesterday	1

Este ejemplo, extraído de la ayuda de Access, establece las opciones para la búsqueda de un archivo. Los archivos que devolverá esta búsqueda han sido previamente modificados y están ubicados en la carpeta **C:\Gráficos** o en una subcarpeta de ésta.

Objeto FoundFiles



El Objeto **FoundFiles**, es un objeto devuelto por la propiedad **FoundFiles** del objeto **FileSearch**, y que contiene una colección que almacena los datos de los ficheros encontrados mediante el método **Execute**.

Este objeto posee las propiedades de las colecciones **Count** e **Item**, además de las propiedades **Application** y **Creator**.

Count devuelve el número de ficheros encontrados al ejecutar el método **Execute** del objeto **FileSearch**.

Item devuelve los nombres de los ficheros encontrados. Para ello le pasamos el índice correspondiente basado en **cero**.

Application devuelve una referencia a la aplicación contenedora; en nuestro caso **Access**.

Creator devuelve un número **long** asociado a la aplicación contenedora.

Ejemplo

Supongamos que queremos mostrar todos los ficheros que están en la carpeta correspondiente al actual proyecto de Access. Es la carpeta de **CurrentProject.Path**

Para ello utilizaremos el siguiente código:

```
Public Sub UsoDeFileSearch()
    Dim i As Long
    Dim strFicheros As String
    With Application.FileSearch
        .LookIn = CurrentProject.Path
        .FileName = "*.*)"
        .SearchSubFolders = True
    If .Execute() > 0 Then
        For i = 1 To .FoundFiles.Count
            strFicheros = strFicheros _
                & .FoundFiles(i) _
                & vbCrLf
        Next i
        MsgBox strFicheros, _
            vbInformation + vbOKOnly, _
            "Se han encontrado " _
            & .FoundFiles.Count & _
            " ficheros"
    Else
        MsgBox "No se han encontrado ficheros", _
            vbInformation + vbOKOnly, _
            "Búsqueda en "
```

```

        & .LookIn
    End If
End With
End Sub

```

Método NewSearch

El método **NewSearch** restablece los valores por defecto de todos los criterios de búsqueda.

Su sintaxis es :

```
ObjetoFileSearch.NewSearch
```

Propiedad FileType

La propiedad **FileType** nos permite averiguar o establecer el tipo de archivo que debe buscarse. Es de lectura y escritura

Esta propiedad se debe corresponder a alguna de las constantes **MsoFileType**.

Sus valores posibles son:

Nombre de la constante	Valor
msoFileTypeAllFiles	1
msoFileTypeBinders	6
msoFileTypeCalendarItem	11
msoFileTypeContactItem	12
msoFileTypeCustom	
msoFileTypeDatabases	7
msoFileTypeDataConnectionFiles	17
msoFileTypeDesignerFiles	22
msoFileTypeDocumentImagingFiles	20
msoFileTypeExcelWorkbooks	4
msoFileTypeJournalItem	14
msoFileTypeMailItem	10
msoFileTypeNoteItem	13
msoFileTypeOfficeFiles	2
msoFileTypeOutlookItems	9
msoFileTypePhotoDrawFiles	16
msoFileTypePowerPointPresentations	5
msoFileTypeProjectFiles	19

msoFileTypePublisherFiles	18
msoFileTypeTaskItem	15
msoFileTypeTemplates	8
msoFileTypeVisioFiles	21
msoFileTypeWebPages	23
msoFileTypeWordDocuments	3

Nota:

La constante **msoFileTypeAllFiles**, de valor 1, hace que se busquen todos los archivos.

La constante **msoFileTypeOfficeFiles**, cuyo valor es 2, incluye todos los archivos que tienen una de las siguientes extensiones: *.doc, *.xls, *.ppt, *.pps, *.obd, *.mdb, *.mpd, *.dot, *.xlt, *.pot, *.obt, *.htm, o *.html.

Ejemplo:

El siguiente código busca en la carpeta "C:\Gráficos", sin tomar en cuenta las subcarpetas, los ficheros tipo **Página Web**.

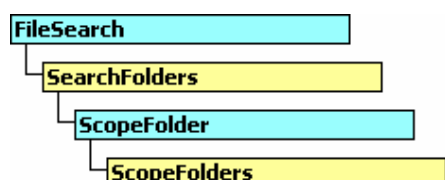
```
Public Sub UsoDeFileType()  
    Dim objFileSearch As Object  
    Dim i As Long  
    Set objFileSearch = Application.FileSearch  
  
    With objFileSearch  
        .NewSearch  
        .SearchSubFolders = False  
        .LookIn = "C:\Gráficos"  
        ' msoFileTypeWebPages tiene el valor 23  
        .FileType = msoFileTypeWebPages  
        If .Execute > 0 Then  
            MsgBox "Se han encontrado " & _  
                & .FoundFiles.Count & _  
                " ficheros de tipo página web"  
            For i = 1 To .FoundFiles.Count  
                MsgBox .FoundFiles(i), _  
                    vbInformation, _  
                    "Fichero N° " & Format(i, "0000")  
            Next i  
        Else  
            MsgBox "No hay ficheros web"  
        End If  
    End With  
End Sub
```


Otras propiedades y métodos

En el objeto **FileSearch** podemos encontrar una estructura muy rica de colecciones y objetos.

No pretendo en esta entrega cubrirlos por completo. Para ello remito al lector a la ayuda de Visual Basic para Aplicaciones.

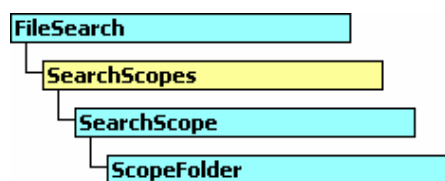
La colección **SearchFolders** contiene la colección de objetos **ScopeFolder** que determina en qué carpetas se realizará la búsqueda al activar el método **Execute** del objeto **FileSearch**.



El objeto **ScopeFolder** se corresponde a una carpeta en la que se pueden realizar búsquedas. Los objetos **ScopeFolder** pueden utilizarse con la colección **SearchFolders**.

La colección **ScopeFolders** contiene la colección de objetos **ScopeFolder** que determina en qué carpetas se realizará la búsqueda al activar el método **Execute** del objeto **FileSearch**.

La colección **SearchScopes** pertenece al objeto **FileSearch** y contiene los objetos **SearchScope**.



El objeto **SearchScope** se utiliza para proporcionar acceso a los objetos **ScopeFolder** que pueden agregarse a la colección **SearchFolders**.

Se corresponde a un tipo de árbol de carpetas en las que pueden efectuarse búsquedas utilizando el objeto **FileSearch**.

Como se indica en la ayuda de VBA, las unidades locales de su equipo representan un solo ámbito de búsqueda. Las carpetas de red y las de Microsoft Outlook son también dos ámbitos individuales de búsqueda disponibles. Cada objeto **SearchScope** incluye un solo objeto **ScopeFolder** que corresponde a la carpeta raíz del ámbito de búsqueda.

El ejemplo siguiente está adaptado de la ayuda de VBA y muestra todos los objetos **SearchScope** disponibles actualmente.

```
Public Sub MostrarLosAmbitosDisponibles()

    ' Declara una variable que hace referencia _
    ' a un objeto SearchScope
    Dim ss As SearchScope

    ' Utiliza un bloque With...End With para referenciar _
    ' el objeto FileSearch
```

```
With Application.FileSearch

    ' Recorre la colección SearchScopes
    For Each ss In .SearchScopes
        Select Case ss.Type
            Case msoSearchInMyComputer
                MsgBox "Mi PC" _
                & " es un ámbito de búsqueda disponible."
            Case msoSearchInMyNetworkPlaces
                MsgBox "Mis sitios de red" _
                & " son un ámbito de búsqueda disponible."
            Case msoSearchInOutlook
                MsgBox "Outlook" _
                & " es un ámbito de búsqueda disponible."
            Case msoSearchInCustom
                MsgBox "Hay disponible" _
                & " un ámbito personalizado de búsqueda."
            Case Else
                MsgBox "No puedo determinar" _
                & " el ámbito de búsqueda."
        End Select
    Next ss
End With
End Sub
```

El método **RefreshScopes** actualiza la lista de objetos **ScopeFolder** disponibles actualmente.

Su sintaxis es :

ObjetoFileSearch.RefreshScopes

El siguiente ejemplo, adaptado de la ayuda de VBA, muestra todos los objetos **ScopeFolder** disponibles actualmente en la unidad **C:** del ámbito de **Mi PC** y demuestra la necesidad de utilizar el método **RefreshScopes** cuando se producen cambios en la lista de carpetas.

```
Sub PruebaDelMetodoRefreshScopes ()
    ' Muestra lo que sucede antes o después
    ' de llamar al método RefreshScopes
    ' si previamente se ha añadido una nueva carpeta
    ' a la lista del ámbito de búsqueda.

    Dim strCarpeta As String
    strCarpeta = "C:\_BorrarDespuésDeSerUsado"
```

```
' Si ya existe la carpeta la borramos
If Len(Dir(strCarpeta)) > 1 Then
    Rmdir Path:=strCarpeta
End If

' Refrescamos la lista de carpetas.
Application.FileSearch.RefreshScopes

' Lista antes de crear la carpeta
Call ListarNombresDeCarpetas

' Creamos una nueva carpeta en el disco C:\
' Se producirá un error si la carpeta ya existiera

Mkdir Path:=strCarpeta

' Lista después de haber creado la carpeta
' La carpeta nueva no aparece en la lista.
Call ListarNombresDeCarpetas

' Refrescamos la lista de carpetas.
Application.FileSearch.RefreshScopes

' Ahora la carpeta nueva sí aparece en la lista.
Call ListarNombresDeCarpetas

' Borramos la carpeta
Rmdir Path:=strCarpeta
End Sub

Sub ListarNombresDeCarpetas()
    Dim i As Integer
    Dim strResultados As String

    ' Recorre todas las carpetas en el disco C:\
    ' en Mi Pc e informa de los resultados
    ' .SearchScopes.Item(1) = "Mi Pc"
    ' .ScopeFolders.Item(2) = "C:\"

    With Application.FileSearch.SearchScopes.Item(1). _
        ScopeFolder.ScopeFolders.Item(2)
```

```
For i = 1 To .ScopeFolders.Count
    strResultados = strResultados & .ScopeFolders. _
        Item(i).Name & vbCrLf
Next i

MsgBox "Nombres de carpetas en C:\...." _
    & vbCrLf _
    & vbCrLf _
    & strResultados

End With
End Sub
```

Nota:

*Los apartados anteriores, referentes al objeto **FileSearch**, tienen exclusivamente como objetivo, introducir al lector en el uso de este objeto y sus posibilidades.*

Profundizar en los mismos está fuera del alcance y de los objetivos de este texto, por lo que remito a los posibles interesados a la ayuda de VBA, en donde podrán encontrar una extensa reseña sobre ellos.

Comencemos a programar con
VBA - Access

Entrega **18**

Trabajar con ficheros II

Trabajando con Ficheros

En el capítulo anterior hemos visto la forma de buscar y renombrar ficheros, obtener datos sobre algunas de sus propiedades, crear carpetas, etc.

Un fichero informático no es más que una serie de datos Byte que están almacenados en un soporte magnético u óptico y que podemos manipular.

En esta entrega vamos a ver procedimientos básicos para manejar ficheros y los métodos para almacenar, exportar o recuperar datos.

Comenzaré con el acceso a ficheros de tipo **Secuencial**, es decir ficheros en los que para llegar a una posición concreta del mismo es preciso recorrerse todas las posiciones anteriores.

Instrucción Open

Para poder acceder a un fichero lo primero que hay que hacer es abrirlo.

La instrucción **Open** es la que se encarga de activar las funciones de Entrada/Salida en un fichero.

Su sintaxis es

```
Open RutaDeAcceso For Modo [Access TipodeDeAcceso]
[Bloquear] As [#]NúmeroArchivo [Len=LongitudRegistro]
```

RutaDeAcceso es el nombre del fichero, y opcionalmente su ruta completa

Por ejemplo **Datos.txt** o **C:\Comercial\Datos.txt**

En el primer caso buscará en la carpeta definida por defecto; carpeta que podemos establecer o averiguar con las funciones **CurDir** y **ChDir**, tal y como explicamos en la entrega anterior. En el segundo lo hará en la carpeta **C:\Comercial**.

Si no existiera esa carpeta generaría el **error 76: No se ha encontrado la ruta de acceso**.

Modo indica la forma como vamos a acceder al fichero. Este parámetro es obligatorio. Si no se indicara definiría **For Random**. Se utiliza una de estas palabras clave:

Append	Añadir datos secuencialmente a partir del final.
Binary	Acceso a ficheros binarios sin longitud fija.
Input	Acceso en modo lectura secuencial.
Output	Acceso en modo escritura secuencial.
Random	Acceso en modo aleatorio directo por número de registro.

TipodeDeAcceso. Parámetro opcional que indica los tipos de actuación permitidas. Se utiliza una de estas palabras clave:

Read	Permite efectuar operaciones de lectura.
Write	Operaciones de escritura.
Read Write	Lectura y Escritura.

Bloquear. Especifica las acciones permitidas para otros procesos concurrentes

Shared	Fichero compartido.
---------------	---------------------

Lock Read	Bloqueado para Lectura.
Lock Write	Bloqueado para Escritura.
Lock Read Write	Bloqueado para Lectura y Escritura.

NúmeroArchivo. Parámetro obligatorio que especifica un número entero como manejador de archivo. Cuando se efectúan operaciones de lectura y escritura hay que hacer mención a este número. Su valor debe estar entre **1** y **511**.

*La función **FreeFile**, que veremos a continuación, nos permite obtener un número libre para el manejador del archivo.*

LongitudRegistro. Opcional. Permite definir el número de caracteres almacenados en el buffer para el *Acceso Secuencial* a un fichero, y la longitud de un registro completo en el modo de *Acceso Aleatorio*. Estos dos modos de acceso los veremos en los apartados siguientes. Su valor debe ser inferior a **32.768**.

Función FreeFile

La Función **FreeFile**, nos devuelve un valor del tipo **Integer** que podremos utilizar con la instrucción **Open** para abrir, leer, escribir y cerrar ficheros.

Su sintaxis es

```
FreeFile [(NúmeroIntervalo)]
```

NúmeroIntervalo es un parámetro opcional que puede tomar el valor **0** por defecto y **1**.

Si se pasa como parámetro **0**, devuelve un número libre para el manejador de ficheros, en el rango de **1** a **255**. Si se pasa el parámetro **1**, el rango va de **256** a **511**.

El rango **256** a **511** se suele usar para ficheros que van a ser compartidos.

Instrucción Print

La Instrucción **Print #**, graba los datos tal y como se mostrarían, por ejemplo en la **Ventana Inmediato**, usando la sentencia **Print**.

Su sintaxis es

```
Print #NúmeroArchivo, [ListaAGrabar]
```

NúmeroArchivo es un entero entre **1** y **511**.

Se aconseja usar uno devuelto por la función **FreeFile**.

ListaAGrabar (Opcional) es una lista de datos que queremos incorporar al fichero abierto con **Open**.

Este procedimiento graba un párrafo con las primeras frases de *Don Quijote de la Mancha*, en el fichero `Texto.txt`.

Finalmente cierra el fichero mediante **Close #NúmeroArchivo**.

```
Public Sub UsoDePrint()
    Dim lngFichero As Long
    Dim strFichero As String

    strFichero = CurrentProject.Path
```

```

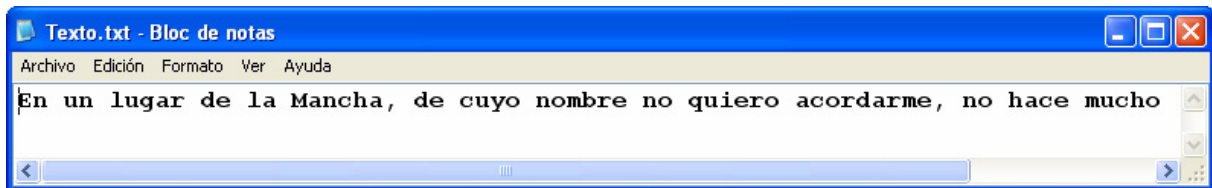
' Definimos como carpeta actual la del fichero mdb
ChDir (strFichero)
strFichero = strFichero & "\Texto.txt"

lngFichero = FreeFile
' Abrimos el fichero, como de Escritura _
  y si no existe lo crea
Open strFichero For Output As #lngFichero
' Escribimos en el fichero
Print #lngFichero, "En un lugar de la Mancha, ";
Print #lngFichero, "de cuyo nombre no quiero acordarme,";
Print #lngFichero, " no hace mucho que vivía . . ."

Close #lngFichero
End Sub

```

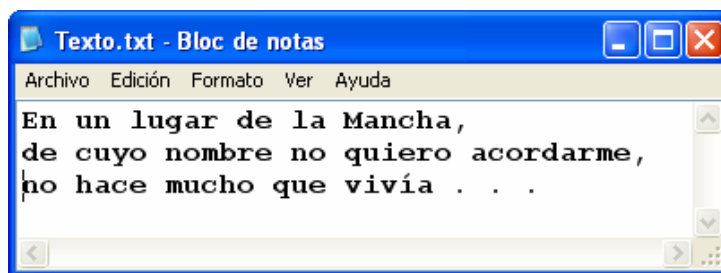
Ejecute este procedimiento y abra el archivo, por ejemplo con el **Bloc de Notas**.



Como puede apreciar, las diferentes partes del texto han sido colocadas una detrás de la otra. Esto ha sido así, porque he colocado un punto y coma después de cada texto.

```
Print #lngFichero, "de cuyo nombre no quiero acordarme,";
```

Si no se hubiera colocado el punto y coma, cada tramo de texto se hubiera grabado en un párrafo diferente.



Print# añade detrás de cada texto un *Retorno de Carro* **vbCr** y un *Salto de Línea* **vbLf**, salvo que encuentre un punto y coma, o una coma tras el texto a grabar.

El conjunto **vbCr** y **vbLf** equivale a la constante **vbCrLf**.

Si en lugar de poner un punto y coma, tras la expresión a grabar, pusiéramos una coma, **Print#** añadiría un **tabulador**.

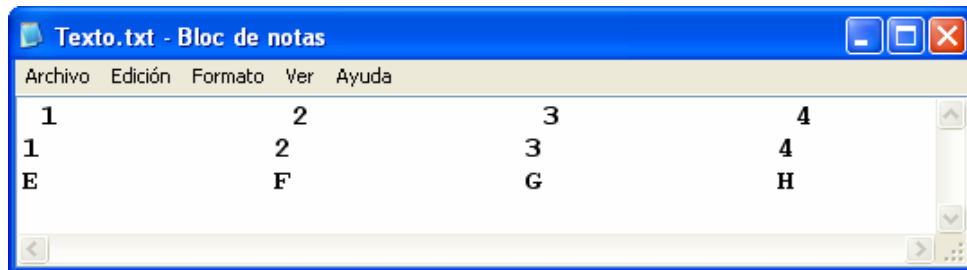
Por ejemplo, la expresión:

```
Print #lngFichero, 1, 2, 3, 4
Print #lngFichero, "1", "2", "3", "4"
```



```
Print #lngFichero, "E", "F", "G", "H"
```

Grabaría un fichero tal y como se puede apreciar en la siguiente imagen



Nótese que deja un espacio de tabulación entre cada carácter.

Podemos apreciar que la primera sentencia `Print #`, deja un espacio en blanco delante de cada uno de los números.

En cambio la segunda, que graba los números como caracteres, no lo hace.

Con `Print #` se pueden usar, no sólo tabuladores mediante comas, sino que podemos usar las funciones `Tab` y `Spc`.

Los datos grabados con `Print #`, normalmente se leerán con `Line Input #` ó `Input #`. Estas instrucciones las veremos más adelante en esta misma entrega.

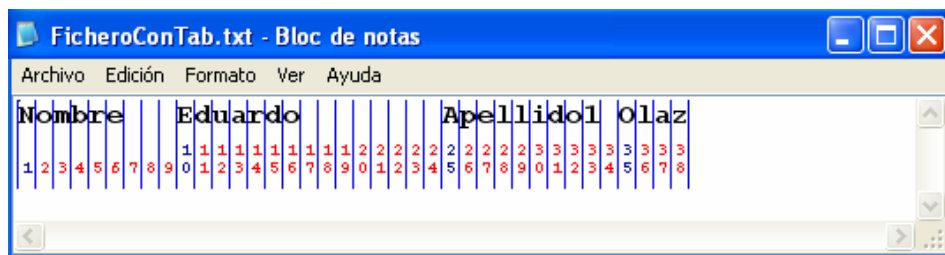
Función Tab

La Función `Tab`, marca el número de columna en el que se empezará a grabar el primer carácter de la lista de datos o la expresión a grabar mediante la instrucción `Print #` o el método `Print`.

Sintaxis: `Tab [(n)]`

`n` representa el número de columna.

Por ejemplo si ejecutamos este código obtendremos un fichero como el de la imagen:



```
Public Sub UsoDeTab ()
    Dim lngFichero As Long
    Dim strFichero As String

    strFichero = CurrentProject.Path
    ChDir (strFichero)
    strFichero = strFichero & "\FicheroConTab.txt"

    lngFichero = FreeFile(1)
```

```

Open strFichero For Output As #lngFichero
Print #lngFichero, Tab(1); "Nombre"; Tab(10); "Eduardo";
Print #lngFichero, Tab(25); "Apellido1"; Tab(35); "Olaz"
Close #lngFichero
End Sub

```

Como podemos apreciar los 4 datos se han grabado en las posiciones que hemos especificado: **1, 10, 25 y 35**.

Función Spc

La Función **Spc**, especifica el número de espacios en blanco antes de grabar el primer carácter de la lista de datos o expresión, mediante **Print #** o el método **Print**.

Sintaxis: **Spc (n)**

El parámetro n es obligatorio y marca el número de espacios.

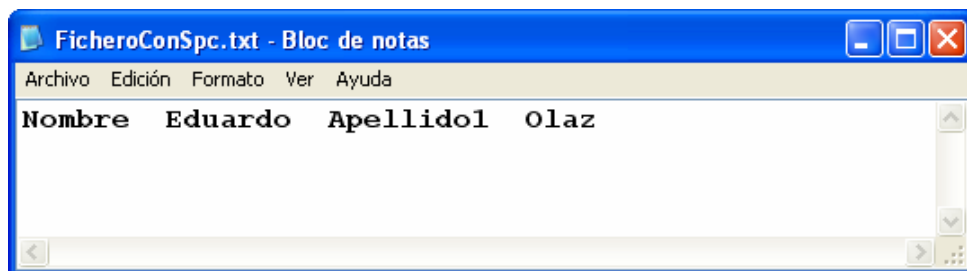
```

Public Sub UsoDeSpc ()
    Dim lngFichero As Long
    Dim strFichero As String

    strFichero = CurrentProject.Path
    ChDir (strFichero)
    strFichero = strFichero & "\FicheroConSpc.txt"

    lngFichero = FreeFile(1)
    Open strFichero For Output As #lngFichero
    Print #lngFichero, "Nombre"; Spc(3); "Eduardo";
    Print #lngFichero, Spc(3); "Apellido1"; Spc(3); "Olaz"
    Close #lngFichero
End Sub

```



Spc toma en cuenta la anchura de la línea, si ésta se ha definido mediante la instrucción **Width #**. Si no se ha definido esta anchura, **Spc** funciona de forma semejante a la función **Space**.

Para obtener más información sobre estas funciones, consulte la ayuda de Access.

Instrucción Width

La instrucción **Width #**, especifica el ancho de línea de salida a un archivo abierto mediante la instrucción Open.

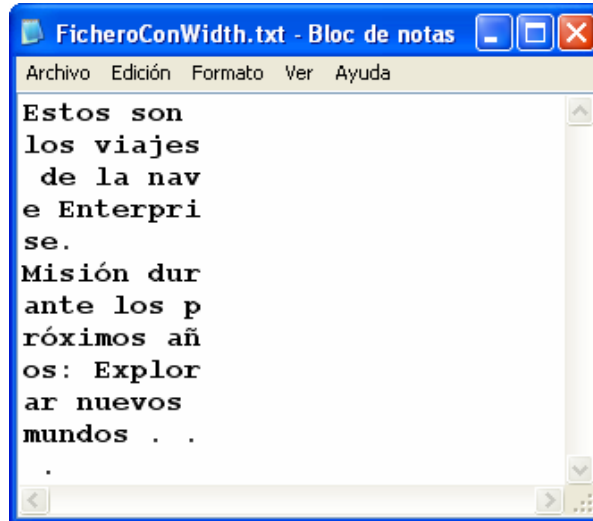
Sintaxis: **Width # NúmeroArchivo, Ancho**

El parámetro **Ancho** es obligatorio y puede ser un número ó una expresión numérica cuyo resultado esté entre 0 y 255.

Si ejecutamos el siguiente código:

```
Public Sub UsoDeWidth()  
    Dim lngFichero As Long  
    Dim i As Long  
    Dim strTexto As String  
    Dim strFichero As String  
    Dim strCaracter As String * 1  
  
    strFichero = CurrentProject.Path  
    ChDir (strFichero)  
    strFichero = strFichero & "\FicheroConWidth.txt"  
    strTexto = "Estos son los viajes de la nave Enterprise." _  
        & vbCrLf _  
        & "Misión durante los próximos años: " _  
        & "Explorar nuevos mundos . . ."  
    lngFichero = FreeFile(1)  
    Open strFichero For Output As #lngFichero  
    ' Establece el ancho de la línea a 10.  
    VBA.Width# lngFichero, 10  
  
    For i = 1 To Len(strTexto)  
        ' Graba el texto carácter a carácter  
        strCaracter = Mid$(strTexto, i, 1)  
        Print #lngFichero, strCaracter;  
    Next i  
    Close #lngFichero  
End Sub
```

Obtendremos el siguiente resultado:



La utilización combinada de estos procedimientos, nos permite la construcción de diversos formatos de ficheros de texto y datos.

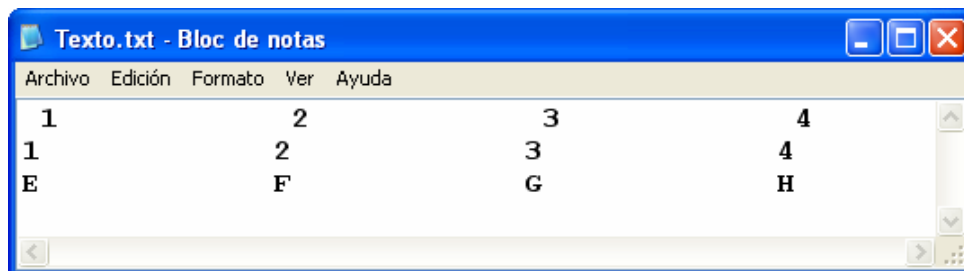
Especialmente las funciones **Tab**, **Spc** y la instrucción **Width**, posibilitan la elaboración de ficheros cuya característica sea la definición de campos de longitudes y posiciones establecidas, como ocurre por ejemplo con los ficheros para el intercambio de datos bancarios.

Instrucción Write

La instrucción **Write #**, es semejante a **Print#**, y sirve también para escribir datos en un fichero secuencial.

La diferencia fundamental con **Print #** es que **Write #**, escribe las cadenas marcándolas con el carácter Comilla doble como delimitador, además separándolos con una coma, y los números los escribe separándolos con comas.

En el ejemplo con **Print #**, nos generaba el siguiente fichero:



Vamos a cambiar ahora las instrucciones **Print #** por **Write #**

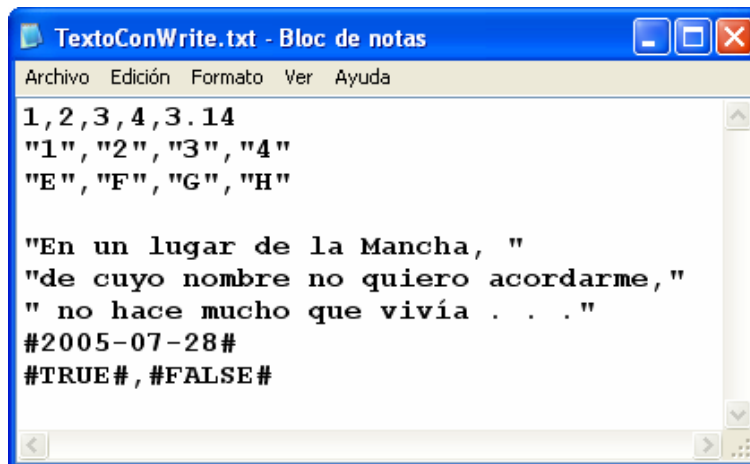
```
Public Sub UsoDeWrite()
    Dim lngFichero As Long
    Dim strFichero As String

    strFichero = CurrentProject.Path
    ' Definimos como carpeta actual la del fichero mdb
    ChDir (strFichero)
    strFichero = strFichero & "\TextoConWrite.txt"
```

```
lngFichero = FreeFile
' Abrimos el fichero, como de Escritura _
  y si no existe lo crea
Open strFichero For Output As #lngFichero
' Escribimos en el fichero

Write #lngFichero, 1, 2, 3, 4, 3.14
Write #lngFichero, "1", "2", "3", "4"
Write #lngFichero, "E", "F", "G", "H"
Write #lngFichero,
Write #lngFichero, "En un lugar de la Mancha, "
Write #lngFichero, "de cuyo nombre no quiero acordarme,"
Write #lngFichero, " no hace mucho que vivía . . ."
Write #lngFichero, " no hace mucho que vivía . . ."
Write #lngFichero, Date
Write #lngFichero, True, False
Close #lngFichero
End Sub
```

El resultado será:



Podemos observar que efectivamente nos crea un fichero con los elementos separados por comas. Otro aspecto interesante es que, tanto las fechas como los valores booleanos, los delimita con almohadillas #.

Si hubiera que grabar un dato cuyo valor fuera **Null**, grabaría **#Null#**.

Un dato de Error, lo grabaría con el siguiente formato **#ERROR códigoerror#**

Si hubiera que grabar un dato cuyo valor fuera **Empty**, no grabaría nada en el fichero.

Los datos grabados con **Print #**, normalmente se leerán con **Line Input #** ó **Input #**. Estas instrucciones las veremos a continuación.

Instrucciones Input # y Line Input

La instrucción **Input**, lee datos de un archivo secuencial abierto y asigna esos datos a sus correspondientes variables.

Su sintaxis es:

Input # NúmeroArchivo, ListaDeVariables

La lista de variables es una lista de ellas, separadas por comas.

La instrucción **Line Input #**, presenta una variación frente a **Input**.

Su sintaxis es:

Line Input # númeroarchivo, VariableCadena

VariableCadena es una variable de tipo **String** en la que se almacenarán los datos de una línea del fichero.

Veamos cómo funciona todo esto:

```
Public Sub LecturaFichero()  
    Dim lngFichero As Long  
    Dim strFichero As String  
    Dim strLinea As String  
    Dim strCliente As String  
    Dim datAperturaCuenta As Date  
    Dim curSaldoCuenta As Currency  
  
    strFichero = CurrentProject.Path  
    ' Definimos como carpeta actual la del fichero mdb  
    ChDir (strFichero)  
    strFichero = strFichero & "\ClientesBanco.dat"  
  
    lngFichero = FreeFile  
    ' Abrimos el fichero, como de Escritura _  
    ' y si no existe lo crea  
    Open strFichero For Output As #lngFichero  
    ' Primero creamos el fichero ClientesBanco.dat _  
    ' que luego leeremos  
    Write #lngFichero, _  
        "Pedro Rodríguez", #5/25/1983#, 100000.25  
    Write #lngFichero, _  
        "Juan Gorostiza", #12/30/2001#, 43235.37  
    Write #lngFichero, _  
        "Jon Estarriaga", #7/7/2004#, -105.85  
    Write #lngFichero, _  
        "Andrea Olazábal", #2/14/1995#, 128.36
```

```
Close #lngFichero

' Ahora ya tenemos el fichero creado
' Primero Vamos a leerlo con Line Input
lngFichero = FreeFile
' Abrimos el fichero, como de lectura
Open strFichero For Input As #lngFichero
' Leemos las líneas del mismo
' Realiza el bucle hasta el final del fichero.
' EOF indica que hemos llegado al final
Debug.Print
Debug.Print "Fichero " & strFichero
Debug.Print
Debug.Print "Datos leídos con Line Input #"
Do While Not EOF(lngFichero)
    ' Lee la línea y se la asigna a la variable String
    Line Input #lngFichero, strLinea
    Debug.Print strLinea
Loop
Close #lngFichero

lngFichero = FreeFile
' Ahora vamos a leer el fichero con Input
Open strFichero For Input As #lngFichero
Debug.Print
Debug.Print "Datos leídos con Input"
' Para ello deberemos asignárselos a sus variables
Do While Not EOF(lngFichero)
    ' Lee la línea y se la asigna a las variables
    Input #lngFichero, _
        strCliente, datAperturaCuenta, curSaldoCuenta
    Debug.Print strCliente, _
        datAperturaCuenta, _
        curSaldoCuenta

Loop
Close #lngFichero
End Sub
```

Al ejecutar este código, nos mostrará en la ventana **[Inmediato]** lo siguiente:

```

Fichero C:\Curso_VBA\Capítulo_18\ClientesBanco.dat

Datos leídos con Line Input #
"Pedro Rodríguez",#1983-05-25#,100000.25
"Juan Gorostiza",#2001-12-30#,43235.37
"Jon Estarriaga",#2004-07-07#,-105.85
"Andrea Olazábal",#1995-02-14#,128.36

Datos leídos con Input
Pedro Rodríguez           25/05/1983           100000,25
Juan Gorostiza             30/12/2001           43235,37
Jon Estarriaga             07/07/2004           -105,85
Andrea Olazábal           14/02/1995           128,36

```

En el código y en el resultado, podemos apreciar la diferencia fundamental entre **Line Input #** e **Input #** la lectura de una línea completa, en el primer caso, y de los datos individuales en el segundo.

Lo que hemos visto hasta ahora se refiere a los llamados **Ficheros de acceso Secuencial**, en los que la información se va leyendo desde el inicio hasta el final, de manera secuencial.

Ya desde los tiempos de **BASICA**, **GWBasic** ó **QBasic**, existían los llamados **Ficheros de acceso aleatorio**, en los que se podía grabar o leer una información en una posición concreta.

Aunque está considerado como un formato "obsoleto", vamos a analizarlo "por encima".

Ficheros de acceso Aleatorio

Para que VBA pueda leer un registro concreto de un fichero de acceso Aleatorio, primero debe conocer en qué posición del fichero debe empezar la lectura o escritura.

Para ello debe tener la siguiente información

- Qué tamaño tiene cada registro (los registros y los campos deben ser de una longitud determinada)
- Qué nº de registro debe leer o escribir

El tamaño del registro se especifica en el momento de la apertura del fichero.

Open "Fichero" For Random As #Numero Len = TamañoRegistro

Random especifica que se va a abrir el fichero en modo de acceso Aleatorio (modo por defecto de **Open** si no se especifica otro modo).

Len indica el tamaño del registro.

Para grabar los datos en un fichero de este tipo, debemos indicarle a VBA en qué posición vamos a grabarlos.

Si no indicamos una posición de registro, escribirá al final del fichero.

Para leer los datos de un fichero da acceso aleatorio se utiliza la instrucción **Get**; para escribir se usa la instrucción **Put**.

Instrucción Get

La instrucción **Get**, lee datos de un archivo abierto y asigna esos datos a una variable.

Su sintaxis es:

```
Get #NúmeroArchivo, [NumeroRegistro], Variable
```

La variable suele ser una variable **Type** de tipo registro.

```
Open "Clientes.dat" For Random As #lngFichero _  
    Len = Len(Datos)  
Get #lngFichero, 2, Datos  
Close #lngFichero
```

Tras esta instrucción, se habrán pasado los datos del registro N° 2 a la variable **Datos**.

El numero de registro es opcional

Instrucción Put

La instrucción **Put**, escribe datos en un archivo abierto , volcándolos desde una variable.

Su sintaxis es:

```
Put #NúmeroArchivo, [NumeroRegistro], Variable
```

La variable suele ser una variable **Type** de tipo registro.

Si quisiéramos grabar los datos contenidos en la variable **Datos**, en el registro N° 2 de la tabla **Clientes.dat**, podríamos hacerlo así:

```
Open "Clientes.dat" For Random As #lngFichero _  
    Len = Len(Datos)  
Put #lngFichero, 2, Datos  
Close #lngFichero
```

El procedimiento **PruebaFicheroRandom** graba 5 registros en **Clientes.dat** y luego los lee, mostrando los datos en la ventana de depuración:

```
Public Type ClienteBanco  
    Nombre As String * 15  
    Apellido1 As String * 15  
    Apellido2 As String * 15  
    Cuenta As Long  
    Saldo As Currency  
End Type  
  
Public Sub PruebaFicheroRandom()  
    GrabaFicheroRandom  
    LeeFicheroRandom  
End Sub
```

```
Public Sub GrabaFicheroRandom()  
    Dim Cliente As ClienteBanco  
    Dim i As Long  
    Dim strNumero As String  
    ' Rnd genera un número aleatorio entre 0 y 0.999999...  
    ' Randomize Timer inicializa cada vez una secuencia _  
    Diferente con Rnd  
    Randomize Timer  
    For i = 1 To 5  
        strNumero = "_" & Format(i, "000")  
        With Cliente  
            .Nombre = "Nombre" & strNumero  
            .Apellido1 = "Apellido1" & strNumero  
            .Apellido2 = "Apellido2" & strNumero  
            .Cuenta = Format(100000 * Rnd, "00000")  
            .Saldo = Int(1000000 * Rnd) / 100  
        End With  
        GrabaCliente Cliente, i  
    Next i  
End Sub  
  
Public Sub LeeFicheroRandom()  
    Dim Cliente As ClienteBanco  
    Dim i As Long  
    For i = 1 To 5  
        LeeCliente Cliente, i  
        Debug.Print  
        With Cliente  
            Debug.Print "Cliente " & _  
                & CStr(i) & ": " & _  
                & Trim(.Nombre) & " " & _  
                & Trim(.Apellido1) & " " & _  
                & Trim(.Apellido2)  
            Debug.Print "Cuenta " & .Cuenta  
            Debug.Print "Saldo " & Format(.Saldo, "#,##0.00 €")  
        End With  
    Next i  
End Sub  
  
Public Sub GrabaCliente( _
```

```
        Datos As ClienteBanco, _
        Optional ByVal Posicion As Long = -1)
On Error GoTo HayError
Dim lngFichero As Long
Dim strFichero As String
lngFichero = FreeFile
Open "Clientes.dat" For Random As #lngFichero _
    Len = Len(Datos)
'Grabamos los datos
If Posicion >= 0 Then
    Put #lngFichero, Posicion, Datos
Else
    Put #lngFichero, , Datos
End If
Close #lngFichero
Salir:
Exit Sub

HayError:
MsgBox "Error al grabar " & strFichero _
    & vbCrLf _
    & Err.Description
Resume Salir
End Sub

Public Sub LeeCliente( _
    Datos As ClienteBanco, _
    Optional ByVal Posicion As Long = -1)
On Error GoTo HayError
Dim lngFichero As Long
Dim strFichero As String

lngFichero = FreeFile
Open "Clientes.dat" For Random As #lngFichero Len =
Len(Datos)
'Leemos los datos
If Posicion >= 0 Then
    Get #lngFichero, Posicion, Datos
Else
    Get #lngFichero, , Datos
```

```
End If
Close #lngFichero
Salir:
Exit Sub

HayError:
MsgBox "Error al leer " & strFichero _
      & vbCrLf _
      & Err.Description
Resume Salir
End Sub
```

Tras ejecutarse este código, mostrará algo así como:

```
Cliente 1: Nombre_001 Apellido1_001 Apellido2_001
Cuenta 73549
Saldo 5.571,35 €

Cliente 2: Nombre_002 Apellido1_002 Apellido2_002
Cuenta 41781
Saldo 2.964,22 €

Cliente 3: Nombre_003 Apellido1_003 Apellido2_003
Cuenta 96618
Saldo 9.775,93 €

Cliente 4: Nombre_004 Apellido1_004 Apellido2_004
Cuenta 6879
Saldo 8.564,24 €

Cliente 5: Nombre_005 Apellido1_005 Apellido2_005
Cuenta 99736
Saldo 5,18 €
```

Función Seek

La instrucción **Seek**, devuelve un valor del tipo **Long** que nos indica el número de registro de la posición actual dentro de un fichero abierto.

Su sintaxis es:

Seek (#NúmeroArchivo)

Por ejemplo. Tras ejecutarse este código, **Seek** devolverá el valor 4.

En el caso de los ficheros de acceso **Random**, **Seek** devuelve la posición siguiente al último registro al que se ha accedido para lectura o escritura.

```
Public Sub PruebaSeek()  
    Dim lngFichero As Long  
    Dim Datos As ClienteBanco  
    lngFichero = FreeFile  
    Open "Clientes.dat" For Random As #lngFichero _  
        Len = Len(Datos)  
    Get #lngFichero, 2, Datos  
    Debug.Print "La posición actual es el registro " _  
        & CStr(Seek(lngFichero))  
    Close #lngFichero  
End Sub
```

El resultado en la ventana de depuración será:

La posición actual es el registro 3

Ejemplos de apertura de ficheros con OPEN

```
Open "Datos.txt" For Input As #1
```

Abre el fichero `Datos.txt` en modo secuencial para **Lectura**.

```
Open "Datos.app" For Output As #2
```

Abre el fichero `Datos.app` en modo secuencial para **Escritura**.

```
Open "DatosBinarios.dat" For Binary Access Write As #3
```

Abre el fichero `DatosBinarios.dat` en modo binario para sólo **Escritura**.

Para ver las posibilidades de manejar ficheros en modo Binario, consulte la ayuda de VBA.

```
Open "Datos.app" For Random As #4 Len = Len(Cliente)
```

Abre el fichero `Datos.app` en el modo de acceso **Aleatorio**.

Se supone que la estructura de los registros es idéntica a la estructura del tipo **Cliente**.

```
Open "Datos.txt" For Output Shared As #2
```

Abre el fichero `Datos.txt` en modo secuencial para **Escritura**.

Sin embargo, al no estar bloqueado, instrucción **Shared**, otro proceso puede manejarlo simultáneamente.

```
Open "Datos.txt" For Output Lock As #2
```

Abre el fichero `Datos.txt` en modo secuencial para **Escritura**.

El fichero quedará bloqueado, instrucción **Lock**, para el acceso simultáneo desde cualquier otro proceso.

```
Open "Datos.txt" For Output Lock Write As #2
```

Abre el fichero `Datos.txt` en modo secuencial para **Escritura**.

El fichero quedará bloqueado, instrucción **Lock Write**, para la escritura desde cualquier otro proceso.

Nota sobre esta entrega

En esta entrega se han explicado formas de acceso a ficheros en lectura y escritura, que han sido clásicas en las versiones Basic de Microsoft, prácticamente desde sus inicios.

Aunque puedan considerarse como “obsoletas” su conocimiento es muy interesante para enfrentarse a casos especiales en los que se requiera elaborar o leer ficheros de datos específicos. Sería el caso definido en los cuadernos bancarios, Ficheros de texto con apuestas de quinielas de fútbol, etc.

Además podemos encontrarnos con aplicaciones “antiguas” que nos obliguen a leer sus datos, escritos en formatos especiales.

No he pretendido realizar un repaso exhaustivo de estos procedimientos, por lo que para la ampliación de conceptos remito a la ayuda de VBA o al MSDN de Microsoft.

Afortunadamente el mundo cambia y Microsoft nos ha aportado herramientas mucho más potentes que las aquí expuestas, como la utilización de **Schema.ini** que veremos en la siguiente entrega.

Comencemos a programar con VBA - Access

Entrega 19

Trabajar con ficheros III

Exportar, importar y vincular ficheros de texto

Access posee una serie de herramientas muy potentes que posibilitan las operaciones de Exportación, Importación y Vinculación de ficheros de texto, suministrándonos además unos asistentes que nos facilitarán estas tareas.

No obstante, hay casos en los que somos nosotros los que deberemos tomar las riendas de todo el proceso.

En la entrega anterior vimos varias formas de leer y escribir en ficheros de texto.

En esta entrega vamos a ver la posibilidad que poseen tanto VBA como VB, de trabajar con ficheros de texto definidos con la ayuda de un fichero adicional que contiene un esquema de su estructura; estoy hablando del fichero **Schema.ini**.

También veremos cómo trabajar con ficheros especiales, tanto para datos de inicialización como de información; los ficheros **ini**, y su alternativa más actual, leer y escribir en el **Registro de Windows**, lo que de paso nos dará pie a introducirnos en el “apasionante mundo” del **API** de Windows.

Hay un tipo especial de ficheros de datos que aunque no vamos a abordarlos en esta entrega, serán objeto especial de un Apéndice: los ficheros de datos con formato **XML**.

Nota:

En las líneas de código se van a utilizar objetos que todavía no hemos visto, como las librerías DAO y ADO, y algunos procedimientos del API de Windows.

Ruego paciencia al lector si hay partes del código que no llega a entender.

Los temas aquí tratados y que resulten totalmente nuevos, se verán con más profundidad en posteriores entregas.

Fichero Schema.ini.

Este archivo es un archivo especial, que contiene una descripción de la estructura, de un fichero de texto organizado como una tabla de datos, proporcionando información sobre:

- El formato que tiene el archivo.
- Los nombres, la longitud y tipos los tipos de datos de los campos.
- El juego de caracteres que se ha utilizado.
- Información sobre las conversiones especiales de los tipos de datos.

El fichero **Schema.ini** debe encontrarse en la misma carpeta donde estará ubicado el fichero de datos.

La información contenida en un fichero **Schema.ini**, sobre los formatos de datos, tiene prioridad frente a la configuración predefinida en el registro de Windows.

Para que VBA pueda manejarlo de forma adecuada, el nombre de este fichero debe mantenerse, por lo que deberá usarse siempre como **Schema.ini**.

Como veremos, Schema.ini es un tipo especial de los ficheros **ini** de configuración.

El tipo genérico de ficheros **ini** será analizado en un punto más avanzado de esta misma entrega.

Estructura de un fichero Schema.ini.

Como todos los ficheros ini, los datos contenidos en Schema.ini están en las denominadas **Secciones**.

Una sección se define por un nombre delimitado por paréntesis cuadrados (corchetes).

En el caso del fichero Schema.ini, la primera sección contiene el nombre del fichero de datos. Por ejemplo **[Datos.txt]**.

En una sección de un fichero ini, se puede incluir nombres de variables o claves, seguidas del signo igual = y a continuación el valor que queremos que tome esa variable o clave.

Podemos definir, ficheros con campos de longitud fija, o con caracteres delimitadores.

A continuación incluyo el contenido de un fichero **Schema.ini** y pasaremos a describir el contenido del mismo. El siguiente paso será analizar las diferentes posibilidades que puede presentar cada clave.

Primero vamos a definir el fichero Schema.ini que contendrá la información del fichero **Alumnos.txt**, con una serie de campos de longitud fija.

Para crearlo utilizaremos un simple editor de texto, como el **Bloc de Notas**.

```
[Alumnos.txt]
ColNameHeader=False
Format=FixedLength
MaxScanRows=25
CharacterSet=Oem
Col1=idAlumno Integer Width 6
Col2=Nombre Char Width 15
Col3=Apellido1 Char Width 15
Col4=Apellido2 Char Width 15
Col5=FechaNacimiento Date Width 11
Col6=Poblacion Char Width 15
Col7=idCurso Integer Width 5
```

Grabamos el fichero con el nombre **Schema.ini** en la misma carpeta que la base de datos.

Antes que nada veamos cómo se puede, por código, exportar datos de una tabla a un fichero de texto. Voy a usar los métodos de la librería DAO (que veremos más adelante).

Se supone que la tabla a exportar está en una base de datos llamada **Datos.mdb**, en la tabla **Alumnos** y que contiene los campos descritos.

Para poder usar DAO, si no está activada, activamos la referencia a su Librería. Para ello usaremos, desde el editor de código, la opción **Referencias...** del menú **Herramientas**.

Y en concreto la librería **Microsoft DAO N.M Object Library**. Tras esto escribimos

```
Public Sub ExportacionConDAO()
    Dim db As DAO.Database
    Dim strSQL As String
    Dim strcarpeta As String
    Dim strBaseDatos As String
```

```
Dim strTabla As String
Dim strFichero_txt As String
Dim n As Long
' Nombres de los elementos _
  intervinientes en el proceso
strcarpeta = CurrentProject.Path
strBaseDatos = strcarpeta & "\Datos.mdb"
strTabla = "Alumnos"
strFichero_txt = strTabla & ".txt"

' Abrimos la base de datos
Set db = OpenDatabase(strBaseDatos)

' Si existe la tabla, la borramos
If Dir(strcarpeta & "\" & strFichero_txt) =
strFichero_txt Then
    Kill strcarpeta & "\" & strFichero_txt
End If

' definimos la consulta de acción
strSQL = "SELECT * " _
        & "INTO [TEXT;DATABASE=" _
        & strcarpeta _
        & "]." _
        & strFichero_txt _
        & " FROM " _
        & strTabla

' Hacemos que se ejecute la consulta de acción
db.Execute strSQL
' Cerramos la Base de Datos
db.Close
' Eliminamos la referencia a la Base de datos
Set db = Nothing

End Sub
```

No se preocupe demasiado si no entiende ese código.

Lo que hace es exportar el contenido de la tabla **Alumnos** de la base de datos **Datos.mdb** a la tabla **Alumnos.txt**.

Veamos ahora qué significa cada una de las líneas del fichero **Schema.ini**.

Ya hemos indicado que la primera línea, [**Alumnos.txt**] indica el nombre del fichero al que se va a exportar.

La segunda línea **ColNameHeader=False** indica que la tabla no va a contener los nombres de los campos.

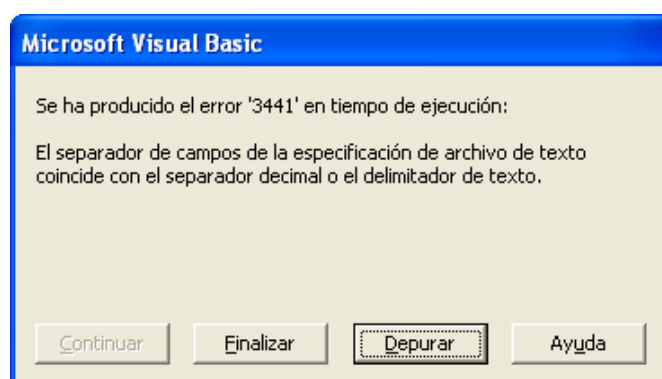
Si hubiéramos puesto: **ColNameHeader=True**, el resultado de exportar la tabla sería:

```
"idAlumno", "Nombre", "Apellido1", "Apellido2", "FechaNacimiento", "Poblacion", "idCurso"
1      Antonio      Martínez      Zúñiga      02/10/1984 Pamplona      35
2      Pedro        Iturralde    Etxegárate  03/01/1985 Tudela        14
3      Marta        Valduz      Goikoa      27/07/1985 Tafalla      10
4      Enrique      Viana       Vergara     03/09/1984 Pamplona     10
```

La tercera línea **Format= FixedLength** hace que los datos se exporten con un formato de longitud variable. Los posibles valores que puede tomar esta clave son:

TabDelimited	Los campos se delimitan mediante el carácter Tab
CSVDelimited	Utiliza comas como separador de campos
Delimited (*)	Hace que los campos se delimiten por asteriscos. Se puede utilizar cualquier carácter, excepto las comillas dobles ("). Por ejemplo (;) haría que el delimitador fuera un punto y coma.
FixedLength	Hace que los campos sean de longitud fija

En España, al usar en la configuración regional la coma como separador decimal, si también la usáramos como separador de campos, nos generaría el error 3441, por esa coincidencia entre el separador de campos y el separador decimal.



Por ello sería aconsejable, en su lugar, la utilización del punto y coma **Format= Delimited (;)** como delimitador de campos.

La cuarta línea **MaxScanRows=25** le indica al motor de la base de datos, el número de filas que debe examinar en la tabla, para determinar el tipo de dato que contiene cada campo. Si ponemos **MaxScanRows=0** le estamos pidiendo que examine toda la tabla.

La quinta línea **CharacterSet=OEM** define el conjunto de caracteres que se va a utilizar. Esto es importante, dependiendo de sobre qué plataforma se va a exportar.

Los posibles valores para **CharacterSet** son **Ansi**, **Oem** y **Unicode**.

En lugar de las palabras **Ansi** / **Oem** / **Unicode**, se puede usar unos valores numéricos que las sustituyen. Estos valores son **1252** para el juego de caracteres **Ansi**, **1200** para **Unicode**, y **850** para el **Oem**.

El valor predeterminado existente en el registro de Windows para una configuración regional de Español es el valor **1252 (Ansi)**.

El usar la opción **Oem** nos permite exportar la tabla sin problemas a un fichero que vaya a ser trabajado con herramientas del tipo **MSDos**.

Por ejemplo, si abrimos el fichero desde el emulador de Dos, con el antiguo editor de textos **Edit** nos mostrará lo siguiente:

```

C:\ Símbolo del sistema - edit
Archivo Edición Buscar Ver Opciones Ayuda
C:\Access_Curso_UBA_News\Capítulo_19\Alumnos.txt
1 Antonio Martínez Zúñiga 02/10/1984 Pamplona 35
2 Pedro Iturralde Etxegárate 03/01/1985 Tudela 14
3 Marta Valduz Goikoa 27/07/1985 Tafalla 10
4 Enrique Viana Vergara 03/09/1984 Pamplona 10
  
```

Si abriéramos el fichero con un editor Windows, por ejemplo con el Bloc de Notas, veríamos el fichero de forma incorrecta.

```

Alumnos.txt - Bloc de notas
Archivo Edición Formato Ver Ayuda
1 Antonio Martınez Zẽñiga 02/10/1984 Pamplona 35
2 Pedro Iturralde Etxegárate 03/01/1985 Tudela 14
3 Marta Valduz Goikoa 27/07/1985 Tafalla 10
4 Enrique Viana Vergara 03/09/1984 Pamplona 10
  
```

En cambio, si hubiéramos puesto **CharacterSet=Ansi**, el resultado con Edit sería el siguiente:

```

C:\ Símbolo del sistema - edit
Archivo Edición Buscar Ver Opciones Ayuda
C:\Access_Curso_UBA_News\Capítulo_19\Alumnos.txt
1 Antonio Martınez Z-ñiga 02/10/1984 Pamplona 35
2 Pedro Iturralde Etxegárate 03/01/1985 Tudela 14
3 Marta Valduz Goikoa 27/07/1985 Tafalla 10
4 Enrique Viana Vergara 03/09/1984 Pamplona 10
  
```

Vamos que las vocales acentuadas se ven de forma incorrecta, lo mismo que el carácter **ñ**.

Si abrimos ahora el mismo fichero con el Bloc de Notas, lo veríamos de forma correcta.

1	Antonio	Martínez	Zúñiga	02/10/1984	Pamplona	35
2	Pedro	Iturralde	Etxegárate	03/01/1985	Tudela	14
3	Marta	Valduz	Goikoa	27/07/1985	Tafalla	10
4	Enrique	Viana	Vergara	03/09/1984	Pamplona	10

La opción para **CharacterSet Unicode**, permitiría exportar tablas con caracteres internacionales, ubicados por encima del carácter ASCII N° 255, por ejemplo los caracteres chinos.

Desde la versión 2000 Access, puede trabajar con caracteres **Unicode**, que necesitan 2 Bytes para el almacenamiento de cada carácter. Esta es la razón por la que, si no se usa la opción Compresión Unicode en el diseño de las tablas, éstas llegan a ocupar casi el doble de espacio que con las tablas Access de la versión 97 ó anteriores.

Las siguientes líneas empiezan con la sílaba **Col** seguida de un número.

```
Col1 Col2 . . . Col7
```

En ellas se especifica qué campos van a ser exportados de la tabla, y en qué orden.

Además se indica el tipo de dato y la anchura de la columna en que se va a exportar.

Es importante que la anchura de la columna sea la misma que la del campo en la tabla, ya que en caso contrario se podrían cortar su contenido.

Indicar los campos y el orden en que se van a exportar, es opcional para los ficheros con separador de campos y obligatorio para los ficheros con campos de longitud fija.

El siguiente fichero **Schema.ini** exportará los campos de la tabla a **Alumnos.txt**

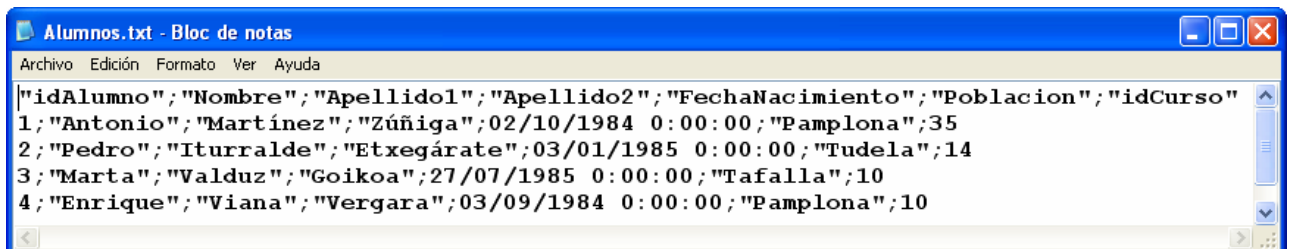
Incluirá una cabecera con los nombres de los campos

Como delimitador de campos usará el carácter punto y coma (;).

Hará una exploración previa de toda la tabla para averiguar los tipos de los campos.

En la exportación usará el juego de caracteres **ANSI**.

```
[Alumnos.txt]
ColumnNameHeader=True
Format=Delimited(;)
MaxScanRows=0
CharacterSet=Ansi
```



```
Alumnos.txt - Bloc de notas
Archivo Edición Formato Ver Ayuda
|"idAlumno"; "Nombre"; "Apellido1"; "Apellido2"; "FechaNacimiento"; "Poblacion"; "idCurso"
1; "Antonio"; "Martínez"; "Zúñiga"; 02/10/1984 0:00:00; "Pamplona"; 35
2; "Pedro"; "Iturralde"; "Etxegárate"; 03/01/1985 0:00:00; "Tudela"; 14
3; "Marta"; "Valdúz"; "Goikoa"; 27/07/1985 0:00:00; "Tafalla"; 10
4; "Enrique"; "Viana"; "Vergara"; 03/09/1984 0:00:00; "Pamplona"; 10
```

Vemos que, además del delimitador de campos, los campos tipo texto están contenidos entre comillas.

En cambio ni los valores numéricos ni los del tipo Fecha/Hora, lo hacen.

Las posibilidades de los ficheros Eschema.ini, llegan mucho más lejos que lo expuesto hasta ahora.

Por ejemplo hay todo un juego de posibilidades para dar formato a los campos, por ejemplo para mostrar los números con decimales ó las fechas con un formato específico.

En la exportación anterior vemos que la fecha de nacimiento la ha exportado con el formato " **dd/mm/yyyy hh:nn:ss** "

Probablemente nos interesará que sólo aparezcan los datos de fecha.

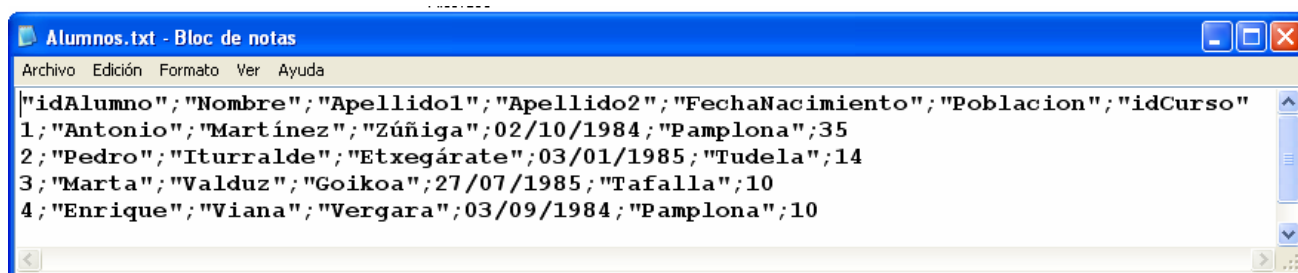
Esto se podría hacer con la siguiente opción de formato:

```
DateTimeFormat = dd/mm/yyyy
```

Si cambiamos el fichero anterior por

```
[Alumnos.txt]
ColNameHeader=True
Format=Delimited(;)
MaxScanRows=0
CharacterSet=Ansi
DateTimeFormat = dd/mm/yyyy
```

El resultado sería



Vemos que ahora la fecha de nacimiento se muestra sin la hora.

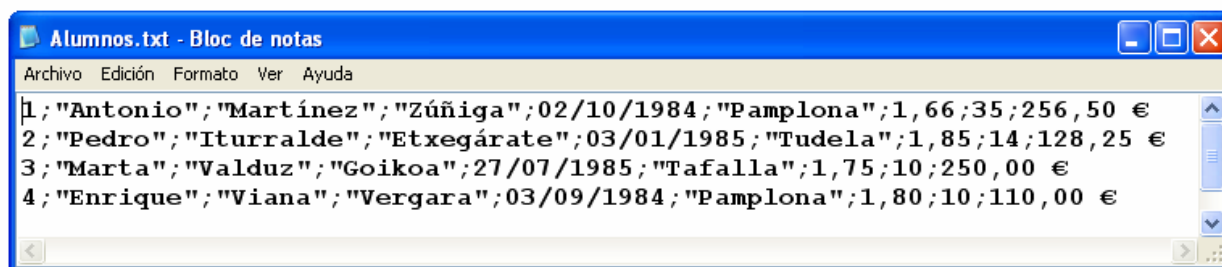
Para ver las opciones de formato que afectan al símbolo monetario y al formato de números decimales, voy a añadir a la tabla original dos nuevos campos:

Altura del tipo **Single** (Simple) e **ImportePagado** del tipo **Currency** (Moneda).

El nuevo fichero **Schema.ini** que vamos a usar será el siguiente:

```
[Alumnos.txt]
ColNameHeader=False
Format=Delimited(;)
MaxScanRows=0
CharacterSet=Ansi
DateTimeFormat=dd/mm/yyyy
CurrencySymbol=€
NumberDigits=2
```

El resultado es



A la hora de definir las columnas, un fichero con campos de anchura fija, se debe especificar el tipo de campo y la anchura que debe mostrar. Los tipos de campo que podemos usar son:

Byte, Bit
Currency
DateTime, Date
Double, Float
Long
Memo, LongChar
Short, Integer
Single
Text, Char

Otros tipos de formato en ficheros Schema.ini

Además de los visto hasta ahora, podemos definir cómo se representarán las cantidades monetarias negativas.

Por ejemplo si tenemos el número **-128.60**, podemos representarlo de diferentes formas mediante los valores asignados a la clave **CurrencyNegFormat**, suponiendo que hemos definido como símbolo monetario el del **Euro**, mediante **CurrencySymbol=€**

Valor	Resultado	Valor	Resultado
0	-€128,60	8	-128,60 € (Predeterminado)
1	-€128,60	9	-€ 128,60
2	€-128,60	10	128,60 €-
3	€128,60-	11	€ 128,60-
4	(128,60€)	12	€ -128,60
5	-128,60€	13	128,60- €
6	128,60-€	14	(€ 128,6)
7	128,60€-	15	(128,60 €)

CurrencyNegFormat=8 devolvería el formato **-128,60 €**

También podemos definir el formato de las cantidades monetarias positivas, mediante la clave **CurrencyNegFormat**, que puede tomar uno de estos valores:

Valor	Resultado	Valor	Resultado
0	€128,60	2	€ 128,60
1	€128,60	3	128,60 €

CurrencyNegFormat=3 devolvería el formato 53,55 €

Adicionalmente podríamos definir el separador de decimales, en las cantidades monetarias, con el carácter que nos interese, mediante la clave **CurrencyDecimalSymbol**.

CurrencyDecimalSymbol=. devolvería el formato 53.55 €

Podemos definir el número de dígitos decimales en un valor monetario, mediante la clave **CurrencyDigits**.

CurrencyDigits=0 devolvería el formato 53 €

Podemos definir el símbolo que va a usar como separador de miles en los valores monetarios mediante la clave **CurrencyThousandSymbol**.

CurrencyThousandSymbol=, devolvería el formato 1,253.35 €

DecimalSymbol define el carácter para el separador de decimales.

NumberLeadingZeros es un valor Boleano que define si los valores numéricos menores que 1 y mayores que -1, deben llevar un cero a la izquierda.

NumberLeadingZeros=False haría que 0,123 se representara como ,123

Si definiéramos el fichero **Schema.ini** como

```
[Alumnos.txt]
ColNameHeader=False
Format=FixedLength
MaxScanRows=25
CharacterSet=1252
CurrencySymbol=€
CurrencyDecimalSymbol=.
CurrencyNegFormat=15
NumberDigits=2
Col1= idAlumno Integer Width 6
Col2= FechaNacimiento Date Width 11
Col3= Poblacion Char Width 15
Col4= idCurso Integer Width 5
Col5= ImportePagado Currency Width 11
Col6= Altura Single Width 5
```

Obtendremos este resultado

idAlumno	FechaNacimiento	Poblacion	idCurso	ImportePagado	Altura
6	25/05/1983	Huarte	5	1110.00 €	0,97
1	02/10/1984	Pamplona	35	256.50 €	1,66
2	03/01/1985	Tudela	14	128.25 €	1,85
3	27/07/1985	Tafalla	10	250.00 €	1,75
4	03/09/1984	Pamplona	10	110.00 €	1,80
5	24/12/1983	Villava	1	(1253.35 €)	1,55

He añadido nuevos datos a la tabla original para comprobar los efectos del formato.

Varios esquemas en un único archivo Schema.ini

Podemos incluir en un archivo de texto varias configuraciones distintas para la Importación / Exportación de diversos archivos.

Lógicamente se deberán separar por secciones, en las que el nombre de la sección, especificará el fichero al que exportar o del que importar.

```
[Alumnos.txt]
ColNameHeader=True
Format=Delimited(;)
MaxScanRows=0
CharacterSet=Ansi
```

```
[Profesores.txt]
ColNameHeader=True
Format=Delimited(;)
MaxScanRows=0
CharacterSet=Ansi
```

Abrir ficheros de texto como si fuera un Recordset

Una vez escrito un fichero de texto podemos hacer que lo cargue como un **Recordset** y pueda trabajar con él.

Para entender de forma simple qué es un **Recordset** diremos que es un conjunto de datos, por ejemplo leídos de una tabla o proveniente de una consulta.

Si existe un fichero **Schema.ini** tratará de aprovecharlo para facilitar la lectura de datos.

```
Sub AbrirFicheroDeTextoDAO()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strcarpeta As String
    Dim strBaseDatos As String
    Dim strTabla As String
    Dim m As Long, n As Long

    strcarpeta = CurrentProject.Path
    strBaseDatos = strcarpeta & "\Datos.mdb"
    strTabla = "Alumnos.txt"

    ' Hacemos que la carpeta donde está el fichero _
    ' la trate como una base de datos (DAO)
    Set db = OpenDatabase(strcarpeta, _
```

```

                False, _
                False, _
                "Text;")
' Carga el recordset desde la tabla de texto
Set rst = db.OpenRecordset(strTabla)
With rst
    ' Mostramos los registros
    Do While Not .EOF
        For m = 0 To .Fields.Count - 1
            ' Mostramos los campos
            Debug.Print .Fields(m),
        Next m
        Debug.Print
        ' Vamos al siguiente registro
        .MoveNext
    Loop
End With
rst.Close
Set rst = Nothing
db.Close
Set db = Nothing
End Sub

```

En mi ordenador el código anterior imprime lo siguiente:

6	25/05/1983	Huarte	5	1110	0,97
1	02/10/1984	Pamplona	35	256,5	1,66
2	03/01/1985	Tudela	14	128,25	1,85
3	27/07/1985	Tafalla	10	250	1,75
4	03/09/1984	Pamplona	10	110	1,8
5	24/12/1983	Villava	1	-1253,35	1,55

Este código utiliza la librería de **DAO**. Os recuerdo lo comentado en la página 3 sobre activar la referencia a la librería correspondiente.

Podemos hacerlo de forma semejante usando la librería de **ADO** en vez de la de **DAO**.

```

Sub AbrirFicheroDeTextoADO ()
    Dim cnn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strcarpeta As String
    Dim strTabla As String
    Dim m As Long, n As Long

    strcarpeta = CurrentProject.Path
    strTabla = "Alumnos.txt"

```

```
' Creamos una nueva conexión (ADO)
Set cnn = New ADODB.Connection
' Asignamos los parámetros de la conexión _
  Incluida la carpeta donde está el fichero.
With cnn
  .Provider = "Microsoft.Jet.OLEDB.4.0"
  .ConnectionString = "Data Source=" & strcarpeta
  .Properties("Extended Properties") = "TEXT;HDR=Yes"
  .Open
End With

' Creamos un nuevo objeto Recordset
Set rst = New ADODB.Recordset

' Abrimos el Recordset, _
  asignándole la tabla de la carpeta
rst.Open strTabla, cnn, , , adCmdTable
With rst
  ' Mostramos los registros
  Do While Not .EOF
  ' Recorremos la colección de campos
    For m = 0 To .Fields.Count - 1
      ' Mostramos los campos
      Debug.Print .Fields(m),
    Next m
  Debug.Print
  ' Vamos al siguiente registro
  .MoveNext
  Loop
End With
rst.Close
Set rst = Nothing
cnn.Close
Set cnn = Nothing
End Sub
```

El resultado es exactamente el mismo que con el procedimiento **DAO**.

Nota:

La utilización de la librería **DAO** y la más moderna **ADO**, con sus métodos y propiedades más usuales las veremos en sus correspondientes entregas posteriores.

No se preocupe si no termina de entender este código. En entregas posteriores lo veremos.

Tome el código como referencia cuando ya conozca estos objetos, o como plantilla si necesita cargar datos, por código, desde un fichero de texto.

Ficheros ini

En la época del Windows de 16 bits, (**Windows 3.x**) los programadores, para establecer una serie de valores de inicialización, utilizábamos ficheros **ini**.

Un fichero ini, del que el fichero **Schema.ini** es un caso particular, consta de una serie de secciones encabezadas por su nombre, entre paréntesis cuadrados (corchetes), y seguidas de una serie de claves, con sus valores asignados. Recordemos un fichero .

```

                [Alumnos.txt]                ← Sección
Clave → ColNameHeader=True ← Valor
Clave → Format=Delimited(;) ← Valor

```

Otro ejemplo:

```

[Inicio]
UltimoUsuario=Eduardo
FechaDesconexion=03/09/2005
[Datos]
idAlumno=123
idClase=25

```

Nos indica una serie de variables y los valores que han ido tomando.

Leer y escribir un fichero ini

En un fichero ini podría leerse o escribirse de forma directa, como en cualquier otro fichero de texto, por ejemplo utilizando las sentencias que vimos en la entrega anterior.

Realizar esta tarea "a mano" y por código, puede resultar una tarea ingrata, por decirlo de una forma suave...

Aunque VBA no posee unas herramientas específicas para grabar o leer los ficheros ini, es el propio Windows, con las llamadas **Funciones API**, el que nos brinda esta posibilidad.

Las funciones API las veremos en un capítulo específicamente dedicado a ellas.

En este punto utilizaremos algunas específicas.

No obstante podemos decir que no es algo excesivamente difícil de utilizar.

Las funciones API que vamos a utilizar son:

```

GetPrivateProfileString
GetPrivateProfileInt
GetPrivateProfileSection
WritePrivateProfileString
WritePrivateProfileSection

```

Introducción (necesaria) a la utilización de las funciones API.

Las funciones API son funciones definidas en el interior de unas librerías dll de Windows.

Su nombre API viene de **Application Programming Interface**, lo que “en cristiano” viene a significar algo así como **Interfaz para la Programación de Aplicaciones**.

Estas funciones son externas a los lenguajes de programación, como **VBA, VB.net, C++, C#, Pascal**, etc... Devuelven un valor y admiten parámetros, que al pasarlos, permiten obtener una serie de valores.

Además pueden ejecutar diversas acciones, en nuestro caso leer o escribir ficheros ini.

Para poder usar una función API, primero debemos declararla

Como ejemplo, vamos a usar la clásica función API que nos devuelve el nombre de la carpeta donde está instalado Windows. Es la función **GetWindowsDirectory**

Para poder usar esta función API, desde VBA primero debemos declararla con la instrucción **Declare**.

En esa instrucción se debe poner

- El nombre de la función
- La librería de Windows donde está definida
- El nombre de su Alias, o nombre nativo de la función en la librería.
- Los nombres de los parámetros y su tipo
- El tipo de valor devuelto por ella

En nuestro caso la declaración de la función **GetWindowsDirectory** es la siguiente

```
Declare Function GetWindowsDirectory _  
    Lib "kernel32.dll" _  
    Alias "GetWindowsDirectoryA" ( _  
    ByVal lpBuffer As String, _  
    ByVal nSize As Long) _  
    As Long
```

Podemos cambiar a nuestra discreción tanto el nombre de la función como el de los parámetros que utiliza, siempre que respetemos el orden y el tipo de de los mismos y el nombre del Alias de la función. Por ejemplo, la siguiente declaración sería equivalente a la primera:

```
Declare Function DameNombreCarpetaWindows _  
    Lib "kernel32.dll" _  
    Alias "GetWindowsDirectoryA" ( _  
    ByVal ContenedorNombre As String, _  
    ByVal TamañoContenedor As Long) _  
    As Long
```

Aunque puede hacerse, y de hecho parece una declaración más clara, es algo que no se suele hacer. Microsoft tiene ya definidas una serie de declaraciones estándar para las funciones API, que tratan de respetar la mayoría de los programadores. Ya hablaremos de este tema en una entrega posterior.

En este caso, el valor devuelto por la función, será la longitud de la cadena que contiene el nombre de la ruta completa de la carpeta de Windows.

Una vez declarada la función podemos utilizarla sin problemas como si fuera una función desarrollada por nosotros mismos, o propia de VBA.

El nombre que utilizaremos para llamar a la función es el que aparece detrás de **Declare**

Veamos el código que utilizará la función.

He creado la función **DameCarpetaWindows** que devuelve una cadena con el nombre de la carpeta donde está instalado **Windows**

Este es su código:

```
Public Declare Function GetWindowsDirectory _
    Lib "kernel32.dll" _
    Alias "GetWindowsDirectoryA" ( _
    ByVal lpBuffer As String, _
    ByVal nSize As Long) _
    As Long

Public Function DameCarpetaWindows() As String

    'Devuelve la carpeta de Windows

    Const conlngCadena As Long = 255

    Dim strDirectorio As String * conlngCadena
    Dim lngDirectorio As Long
    Dim lngPath As Long

    lngDirectorio = conlngCadena
    lngPath = GetWindowsDirectory( _
        strDirectorio, _
        lngDirectorio)
    DameCarpetaWindows = Left$( _
        strDirectorio, _
        lngPath)

End Function
```

A la función **GetWindowsDirectory** le pasamos la cadena **strDirectorio** y se almacenará en ella la ruta de la carpeta de **Windows**.

La variable **lngPath** recoge la longitud de la cadena obtenida, por lo que si hacemos

```
Left$(strDirectorio, lngPath)
```

Obtendremos la ruta completa de Windows.

Si en mi ordenador escribo `Debug.Print DameCarpetaWindows()`

Me mostrará **C:\WINDOWS**, que es donde efectivamente he instalado Windows en mi PC.

Información en Internet sobre las APIs de Windows

Además del MSDN de Microsoft, hay un gran número de páginas de Internet en donde podemos obtener información sobre la utilización de las APIs de Windows.

A continuación pongo los enlaces a algunas de esas páginas:

<http://www.silared.com/usuarios/vbasic/api/index.htm>

<http://www.ciberteca.net/articulos/programacion/win32apis/ejemplo.asp>

<http://www.mentalis.org/index2.shtml> La muy recomendable página de AllApi.net.

<http://mech.math.msu.su/~vfnik/WinApi/index.html> Otra página clásica muy interesante.

<http://custom.programming-in.net/>

<http://www.mvps.org/access/api/>

<http://vbnet.mvps.org/index.html?code/fileapi/>

Declaración de las funciones API para el manejo de un fichero ini.

Creemos un módulo estándar con un nombre descriptivo, como **DeclaracionesAPI**

Lógicamente en él pondremos las declaraciones de las funciones que vamos a utilizar.

Los parámetros de cada función están descritos en los propios comentarios.

Option Explicit

```

' GetPrivateProfileString _
  Recupera una cadena en un fichero de inicialización
Declare Function GetPrivateProfileString _
    Lib "kernel32.dll" _
    Alias "GetPrivateProfileStringA" ( _
    ByVal lpApplicationName As String, _
    ByVal lpKeyName As String, _
    ByVal lpDefault As String, _
    ByVal lpReturnedString As String, _
    ByVal nSize As Long, _
    ByVal lpFileName As String _
    ) As Long

' lpApplicationName _
  Sección en la que se va a buscar la Clave.
' lpKeyName _
  Nombre de la clave a buscar.
' lpDefault _
  Valor por defecto a devolver _
  si no se encuentra la clave.
' lpReturnedString _
  Cadena en la que escribirá el resultado. _
  Debe tener como mínimo la longitud de nSize.
' lpFileName _
  Nombre del archivo de inicialización.

```

```

' GetPrivateProfileInt _
  Recupera un entero de un fichero de inicialización
Declare Function GetPrivateProfileInt _
  Lib "kernel32.dll" _
  Alias "GetPrivateProfileIntA" ( _
  ByVal lpApplicationName As String, _
  ByVal lpKeyName As String, _
  ByVal nDefault As Long, _
  ByVal lpFileName As String _
  ) As Long

' nDefault _
  Valor por defecto a devolver _
  si no se encuentra la clave.

' GetPrivateProfileSection _
  Recupera una lista con los nombres de todas las claves _
  de una Sección y sus valores.
Declare Function GetPrivateProfileSection _
  Lib "kernel32" _
  Alias "GetPrivateProfileSectionA" ( _
  ByVal lpApplicationName As String, _
  ByVal lpReturnedString As String, _
  ByVal nSize As Long, _
  ByVal lpFileName As String _
  ) As Long

' lpReturnedString _
  Lista en la que se cargan las claves y sus valores. _
  Cada cadena está separada por un valor Nulo _
  Al final de la lista se encontrarán dos valores nulos.

' nSize _
  Tamaño asignado a lpReturnedString

Declare Function WritePrivateProfileString _
  Lib "kernel32.dll" _
  Alias "WritePrivateProfileStringA" ( _
  ByVal lpApplicationName As String, _
  ByVal lpKeyName As String, _
  ByVal lpString As String, _
  ByVal lpFileName As String _
  ) As Long

' lpString _

```


Valor a escribir en una clave. _
 Si lo que se quiere es borrar la clave _
 Pasar el valor **vbNullString**

```
Declare Function WritePrivateProfileSection _
    Lib "kernel32" _
    Alias "WritePrivateProfileSectionA" ( _
    ByVal lpApplicationName As String, _
    ByVal lpString As String, _
    ByVal lpFileName As String _
    ) As Long
' lpApplicationName _
  Sección en la que vamos a escribir las Claves y Valores
' lpString _
  Lista en la que se pasan las claves y sus Valores. _
  Como en el caso de lpReturnedString _
  cada cadena está separada por un valor Nulo _
  Al final de la lista se pondrán dos valores nulos.
Declare Function GetPrivateProfileSectionNames _
    Lib "kernel32" _
    Alias "GetPrivateProfileSectionNamesA" ( _
    ByVal lpReturnedString As String, _
    ByVal nSize As Long, _
    ByVal lpFileName As String _
    ) As Long
' lpReturnedString contiene la lista de las Secciones _
  cada cadena está separada por un valor Nulo _
  Al final de la lista se pondrán dos valores nulos.
```

Escritura y lectura en un fichero ini

Para escribir los procedimientos, de escritura y lectura en un fichero ini vamos a abrir un nuevo módulo estándar al que vamos a llamar, por ejemplo, **ProcedimientosINI**.

Ya hemos dicho que un fichero ini posee un **Nombre**, con la **extensión ini**, una serie de **Secciones** con su nombre escrito entre corchetes (paréntesis cuadrados) y una serie de **Claves** con un **Valor**.

Vamos a crear un procedimiento que escriba un valor de una clave en una sección de un fichero ini. A este procedimiento lo vamos a llamar **EscribeClaveINI**.

Para ello utilizará la función API: **WritePrivateProfileString**.

```
Public Sub EscribeClaveINI( _
    ByVal FicheroINI As String, _
    ByVal Seccion As String, _
```

```
        ByVal Clave As String, _
        ByVal Valor As String, _
        Optional ByVal Carpeta As String)

Dim lngRetornado As Long

' Se hace un pequeño control _
  La Carpeta pasada
Carpeta = CarpetaValida(Carpeta)
' Si la carpeta no existiera se generaría un error
' Validamos también el fichero
' Si el fichero no existiera generaría un error
Fichero = FicheroValido(Fichero, Carpeta)
' Llamamos ahora a la función API
'      WritePrivateProfileString _
que hemos declarado anteriormente. _
El valor de retorno de la función _
se lo asignamos a lngRetornado
lngRetornado = WritePrivateProfileString( _
        Seccion, _
        Clave, _
        Valor, _
        Carpeta & FicheroINI)

End Sub
```

Este es el código de la función **CarpetaValida**, que vamos a usar en varios procedimientos, como una comprobación **muy primaria** de la carpeta pasada:

```
Public Function CarpetaValida( _
        Optional ByVal Carpeta As String = "" _
        ) As String

On Error GoTo HayError
' Esta función comprueba la carpeta pasada como _
  Parámetro.
Carpeta = Trim(Carpeta) ' Quita blancos de los extremos
' Si no se pasa nada, devuelve la de la aplicación.
If Carpeta = "" Then
    Carpeta = CurrentProject.Path
End If
' Si el nombre de la carpeta no acaba con "\" lo pone
```

```
If Right(Carpeta, 1) <> "\" Then
    Carpeta = Carpeta & "\"
End If
' Si no existe la carpeta genera un error'
If Len(Dir(Carpeta, vbDirectory)) = 0 Then
    Err.Raise 1000, _
        "CarpetaValida", _
        "No existe la Carpeta " _
        & Carpeta
End If
Salida:
    CarpetaValida = Carpeta
Exit Function
HayError:
    If Err.Number = 1000 Then
        MsgBox "No existe la carpeta " _
            & Carpeta, _
            vbCritical + vbOKOnly, _
            " Error en la función " & Err.Source
    Else
        MsgBox "Se ha producido el error " _
            & Format(Err.Number, "#,##0") _
            & vbCrLf _
            & Err.Description, _
            vbCritical + vbOKOnly, _
            " Error no controlado en la función " _
            & "CarpetaValida()"
    End If
Resume Salida
End Function
```

De una forma similar a como hemos creado la función **CarpetaValida**, vamos a crear la función **FicheroValido**, que comprobará si existe el fichero.

```
Public Function FicheroValido( _
    ByVal Fichero As String, _
```

```
        Optional ByVal Carpeta As String = "" _
    ) As String
On Error GoTo HayError
' Esta función comprueba la existencia del fichero _
  pasado como Parámetro.
Carpeta = Trim(Carpeta) ' Quita blancos de los extremos
' Si no se pasa nada, devuelve la de la aplicación.
If Carpeta = "" Then
    Carpeta = CurrentProject.Path
End If
' Si el nombre de la carpeta no acaba con "\" lo pone
If Right(Carpeta, 1) <> "\" Then
    Carpeta = Carpeta & "\"
End If
' Si no existe el fichero genera un error'
If Len(Dir(Carpeta & Fichero)) = 0 Then
    Err.Raise 1100, _
        "FicheroValido", _
        "No existe el fichero " _
        & Fichero

    End If
Salida:
    FicheroValido = Fichero
    Exit Function
HayError:
    If Err.Number = 1100 Then
        MsgBox "No existe el fichero " _
            & Carpeta & Fichero, _
            vbCritical + vbOKOnly, _
            " Error en la función " & Err.Source
    Else
        MsgBox "Se ha producido el error " _
            & Format(Err.Number, "#,##0") _
            & vbCrLf _
```

```
& Err.Description, _  
vbCritical + vbOKOnly, _  
" Error no controlado en la función " _  
& "FicheroValido() "  
  
End If  
Resume Salida  
End Function
```

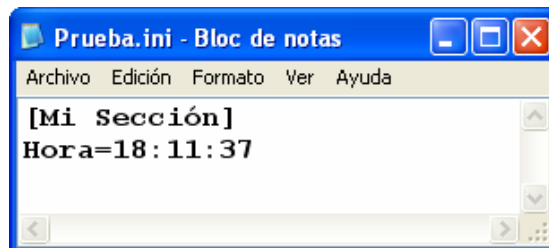
Vamos a probar el procedimiento, para lo que en la ventana de depuración escribimos:

```
EscribeClaveINI "Prueba.ini", "Mi Sección", "Hora", CStr(Time)
```

Si abrimos con el explorador de Windows la carpeta donde está ubicado el programa, veremos que tenemos un bonito fichero Prueba.ini



Si abrimos el fichero con el Bloc de notas veremos que contiene lo que en principio esperábamos:



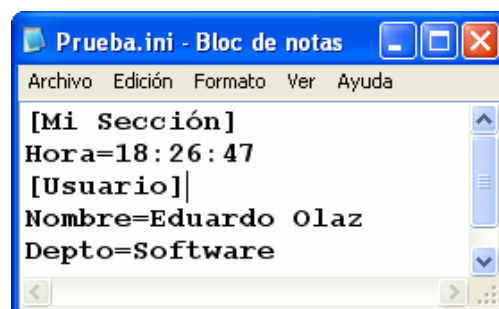
Si volvemos a efectuar la misma llamada, podremos comprobar que se actualiza la hora.

Vamos a llamarlo ahora con estos parámetros:

```
EscribeClaveINI "Prueba.ini", "Usuario", "Nombre", "Eduardo"  
EscribeClaveINI "Prueba.ini", "Usuario", "Depto", "Software"
```

Si lo editamos con el bloc de notas, veremos que Prueba.ini ha cambiado.

De hecho se ha actualizado la hora, se ha añadido la sección Usuario que contiene las claves Nombre, con mi nombre, y Depto con el valor Software.



¿Cómo podríamos **borrar una clave por código**?

Muy sencillo, pasándole la cadena nula representada por la constante `vbNullString`

Si hacemos

```
EscribeClaveINI "Prueba.ini", "Usuarios", "Depto", vbNullString
```

Podremos comprobar que ha desaparecido la clave **Depto** en **Prueba.ini**

Conclusiones respecto al código anterior:

- Si el fichero **ini** no existiera, lo crearía
- Si existiera una clave en la sección correspondiente, la actualizaría con el nuevo valor pasado como parámetro.
- Si no existe la clave, la crea en su sección
- Si no existiera la sección la crearía.
- Para borrar una clave le pasamos como valor `vbNullString`

Vamos a ver cómo podríamos leer La clave **Nombre** en la Sección **Usuario**.

Para ello vamos a utilizar la función API `GetPrivateProfileString`, que nos devolverá una cadena con el **valor** de la **clave**.

```
Public Function LeeCadenaClaveINI( _  
    ByVal FicheroINI As String, _  
    ByVal Seccion As String, _  
    ByVal Clave As String, _  
    Optional ByVal Carpeta As String, _  
    Optional ByVal ValorDefecto As String _  
    ) As String  
  
    Const conLongitud As Long = 255  
    Dim lngRetornado As Long  
    Dim strDevuelto As String  
  
    Carpeta = CarpetaValida(Carpeta)  
    FicheroINI= FicheroValido(FicheroINI, Carpeta)  
  
    ' Creamos la cadena contenedora con 255 caracteres * _  
    Esta cadena la pasaremos como cadena que contendrá _  
    el valor leído.  
    strDevuelto = String(conLongitud, "*")  
    ' Llamamos ahora a la función API  
    lngRetornado = GetPrivateProfileString( _
```

```

        Seccion, _
        Clave, _
        ValorDefecto, _
        strDevuelto, _
        Len(strDevuelto), _
        Carpeta & FicheroINI)
    ' Devolvemos el dato
    ' El número de caracteres leídos estará en lngRetornado
    ' La cadena leída en strDevuelto
    LeeCadenaClaveINI = Left(strDevuelto, lngRetornado)
End Function

```

Con el código anterior, si escribimos la sentencia

```
LeeCadenaClaveINI("Prueba.ini", "Usuario", "Nombre")
```

nos devolverá la cadena "Eduardo". Con lo que habremos leído una cadena de tipo String.

Si supiéramos que determinada clave contiene un valor **entero**, podríamos utilizar la función API , **GetPrivateProfileInt** que es más sencilla de manejar, ya que la propia función devolvería el entero solicitado en formato Long.

Antes que nada, vamos a grabar en nuestro fichero ini, un valor entero **Long**.

Para ello utilizaremos el procedimiento creado con anterioridad **EscribeClaveINI**.

Supongamos que queremos grabar el valor **123456.**, como valor de la Clave **idUsuario** en la sección **Usuario** de nuestro fichero ini.

Llamaríamos a nuestro procedimiento **EscribeClaveINI** de la siguiente forma:

```

EscribeClaveINI "Prueba.ini", _
                "Usuario", _
                "idUsuario", _
                CStr(12345)

```

Tras esto, el fichero **Prueba.ini** contendrá lo siguiente:

```

[Mi Sección]
Hora=18:26:47
[Usuario]
Nombre=Eduardo
Depto=Software
idUsuario=12345

```

Vamos ahora a crearnos una función llamada **LeeEnteroClaveINI**, que nos devolverá un entero de tipo **Long**, llamando a la función API **GetPrivateProfileInt**.

```

Public Function LeeEnteroClaveINI( _
    ByVal FicheroINI As String, _
    ByVal Seccion As String, _

```

```
        ByVal Clave As String, _
        Optional ByVal Carpeta As String, _
        Optional ByVal ValorDefecto As Long _
    ) As Long

    Carpeta = CarpetaValida(Carpeta)
    FicheroINI= FicheroValido(FicheroINI, Carpeta)

    ' Invocamos directamente la función API _
      GetPrivateProfileInt
    LeeEnteroClaveINI = GetPrivateProfileInt( _
        Seccion, _
        Clave, _
        ValorDefecto, _
        Carpeta & FicheroINI)

End Function
```

Si llamamos a la función mediante:

```
LeeEnteroClaveINI("Prueba.ini", "Usuario", "idUsuario", , 0)
```

nos devolverá la última clave introducida, es decir **12345**

También podríamos haber extraído ese valor usando la función **LeeCadenaClaveINI**

```
LeeCadenaClaveINI("Prueba.ini", "Usuario", "idUsuario", , 0)
```

Pero en ese caso nos lo hubiera devuelto en forma de cadena, es decir **"12345"**.

Lista las Secciones de un fichero ini

La lista de las secciones de un fichero ini, se puede obtener mediante la función API **GetPrivateProfileSectionNames**.

Esta función devuelve una cadena con las secciones separadas por el valor ASCII 0, y a continuación del nombre de la última sección, poniendo dos caracteres Nulos de ASCII 0.

Vamos a crear una función a la que llamaremos **SeccionesINI**, que devolverá un **array** con los nombres de las secciones.

Veamos su código:

```
Public Function SeccionesINI( _
    ByVal FicheroINI As String, _
    Optional ByVal Carpeta As String _
) As Variant

    On Error GoTo HayError
    Dim lngLongitud As Long
    Dim lngDevuelto As Long
```



```
Dim strContenedor As String
Dim strSecciones As String
Dim aSecciones As Variant
On Error GoTo HayError

' lngLongitud será la longitud de strContenedor _
  que recogerá los nombres de las secciones
lngLongitud = 2 ^ 12
strContenedor = String(lngLongitud, "*")

Carpeta = CarpetaValida(Carpeta)
FicheroINI = FicheroValido(FicheroINI, Carpeta)

lngDevuelto = GetPrivateProfileSectionNames( _
    strContenedor, _
    Len(strContenedor), _
    Carpeta & FicheroINI)
' Si el valor devuelto fuera 0 sería porque _
  no ha encontrado secciones en el fichero _
  o no ha encontrado el fichero
If Not CBool(lngDevuelto) Then
    ' Preparamos un array sin datos
    ReDim aSecciones(0 To 0)
    aSecciones(0) = ""
Else
    strSecciones = Left(strContenedor, lngDevuelto - 1)
    ' El separador es el carácter Chr$(0)
    ' Split devuelve un Array _
      desde una cadena con separadores
    aSecciones = Split(strSecciones, Chr$(0))
End If
' Devuelve el array
SeccionesINI = aSecciones
Salida:
SeccionesINI = aSecciones
Exit Function
HayError:
    MsgBox "Se ha producido el error " _
        & Format(Err.Number, "#,##0") _
        & vbCrLf _
```

```
        & Err.Description, _  
        vbCritical + vbOKOnly, _  
        " Error no controlado en la función " _  
        & "SeccionesINI() "  
    ReDim aSecciones(0 To 0)  
        aSecciones(0) = ""  
    Resume Salida  
End Function
```

Si efectuamos una llamada a la función mediante **SeccionesINI("Prueba.ini")(0)** en mi ordenador, considerando que antes hemos creado esas secciones, devuelve el nombre de la sección **"Usuario"**

Si Hacemos **SeccionesINI("Prueba.ini")(1)**, devuelve **"Mi Sección"**

Fijémonos en que la función **SeccionesINI** devuelve un **Array**, y por tanto se le puede llamar pasándole, además de los parámetros propios de la misma, el **índice** del Array devuelto.

Podemos generar un procedimiento que muestre los nombres de las secciones de un fichero del tipo **ini**.

```
Public Sub MuestraSeccionesIni( _  
    ByVal FicheroINI As String, _  
    Optional ByVal Carpeta As String)  
    Dim aSecciones As Variant  
    Dim i As Long  
  
    Carpeta = CarpetaValida(Carpeta)  
    aSecciones = SeccionesINI(FicheroINI, Carpeta)  
  
    Debug.Print "Secciones de " _  
        & Carpeta & FicheroINI  
    For i = 0 To UBound(aSecciones)  
        Debug.Print aSecciones(i)  
    Next i  
End Sub
```

Si ejecuto este procedimiento en mi ordenador me muestra

```
Secciones de C:\Access_Curso_VBA_News\Capítulo_19\Prueba.ini  
Mi Sección  
Usuario
```

Así como tenemos posibilidad de leer, en un solo paso, todas las secciones de un fichero **ini**, podemos también leer ó escribir todas las **claves**, y sus **valores** correspondientes, dentro de una **Sección**.

Lectura y escritura en bloque de toda una sección en un fichero ini

Como he comentado, además de leer y escribir cada clave, en un procedimiento ini, podemos también leer o escribir todos los elementos de una sección de un solo paso.

Para poder leer una sección completa, usaremos la función API

GetPrivateProfileSection.

Recordemos que la declaración de esta función es

```
Declare Function GetPrivateProfileSection _
    Lib "kernel32" _
    Alias "GetPrivateProfileSectionA" ( _
    ByVal lpApplicationName As String, _
    ByVal lpReturnedString As String, _
    ByVal nSize As Long, _
    ByVal lpFileName As String _
    ) As Long
```

lpApplicationName representa el nombre de la sección.

lpReturnedString es la cadena que recogerá los datos de la sección, con las claves y valores separados por el carácter ASCII 0.

nSize es el tamaño con el que se ha pasado la cadena.

El valor Long devuelto por la función es el número de caracteres cargados en la variable lpReturnedString.

El nombre que vamos a poner a la función que va a devolvernos los datos de una sección va a ser **LeeSeccionINI**.

```
Public Function LeeSeccionINI( _
    ByVal FicheroINI As String, _
    ByVal Seccion As String, _
    Optional ByVal Carpeta As String _
    ) As Variant
    ' Devuelve los datos de una sección en un array _
    ' de 2 dimensiones, de tipo String
    ' aDatos(1 to N, 0 to 1) As String
    ' Para ello utilizará la función API _
    GetPrivateProfileSection

    Dim lngLongitud As Long
    Dim lngDevuelto As Long
    Dim strContenedor As String
    Dim strSecciones As String
```

```

' hacemos que el tamaño de la cadena contenedora _
  sea "razonablemente grande"
lngLongitud = 2 ^ 15 - 1
strContenedor = String(lngLongitud, "*")

Carpeta = CarpetaValida(Carpeta)
FicheroINI = FicheroValido(FicheroINI)

lngDevuelto = GetPrivateProfileSection( _
    Seccion, _
    strContenedor, _
    lngLongitud, _
    Carpeta & FicheroINI)
' La cadena con los datos separados por Chr$(0) _
  está en los primeros lngDevuelto - 1 Caracteres _
  de strContenedor
strContenedor = Left(strContenedor, lngDevuelto - 1)
' Llamamos a la función que extrae los datos _
  desde la cadena
LeeSeccionINI = DescomponerCadenaConNulos( _
    strContenedor)

End Function

```

La función auxiliar **DescomponerCadenaConNulos** descompone los datos de una cadena del tipo **Clave1=Dato1 Clave2=Dato2 Clave3=Dato3 . . . ClaveN=DatoN**, en la que los sucesivos pares de **Clave_i=Dato_i**, están separados por el carácter **ASCII** de índice = 0 y los pasa a un array de 2 dimensiones.

Veamos cómo podemos hacer que esta función devuelva un **array** más manejable.

```

Public Function DescomponerCadenaConNulos( _
    ByVal Cadena As String _
    ) As Variant
' Esta función recibe una cadena _
  con los datos separados por nulos _
  y devuelve un array String del tipo _
  aDatos(0 to n-1, 0 to 1)
Dim lngLongitud As Long
Dim lngDatos As Long
Dim strNulo As String
Dim aCadenas As Variant
Dim aResultado() As String
Dim aLinea As Variant

```

```
Dim i As Long

strNulo = Chr(0)
lngLongitud = Len(Cadena)

' Devolverá una matriz bidimensional _
  si existe algún separador
If InStr(Cadena, strNulo) > 0 Then
    aCadenas = Split(Cadena, strNulo)
Else
    ' Si no hay datos devuelve _
      un array con caracteres vacíos
    ReDim aResultado(0 To 0, 0 To 1)
    aResultado(0, 0) = ""
    aResultado(0, 1) = ""
    DescomponerCadenaConNulos = aResultado
    Exit Function
End If

lngDatos = (UBound(aCadenas) + 1)
ReDim aResultado(0 To lngDatos, 0 To 1)

For i = 0 To lngDatos - 1
    aLinea = Split(aCadenas(i), "=")
    aResultado(i, 0) = aLinea(0)
    aResultado(i, 1) = aLinea(1)
Next i

DescomponerCadenaConNulos = aResultado

End Function
```

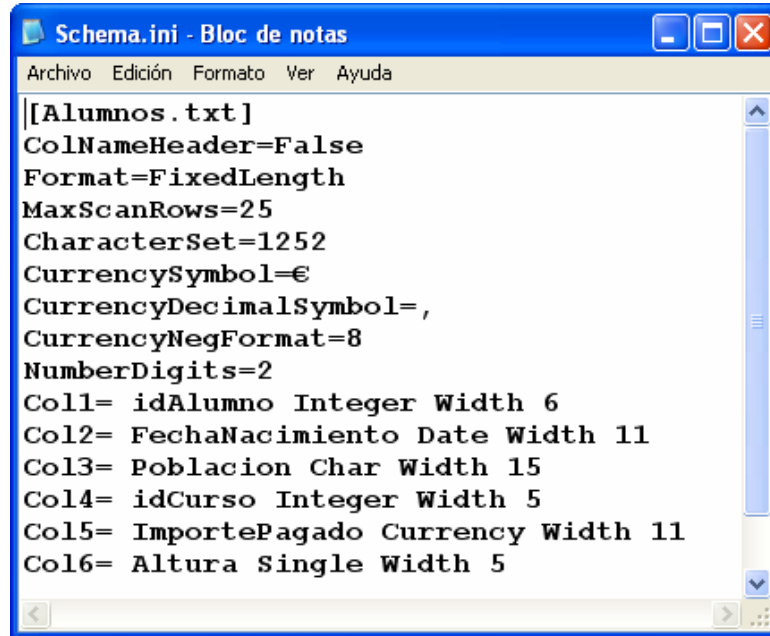
Lectura de una sección

Vamos a probar estas 2 últimas funciones.

Para ello utilizaremos el fichero **Schema.ini** que habíamos construido para usarlo en la exportación – importación de datos, en un punto anterior de esta entrega.

No debemos olvidar que sólo es un caso muy especial de fichero **ini**.

En este momento, en la carpeta de mi ordenador, el fichero **Schema.ini** contiene las siguientes líneas:



Vemos que sólo tiene una sección llamada **Alumnos.txt**

Crearemos un procedimiento que muestre los datos de esa sección en el fichero Schema.ini.

```
Public Sub MuestraDatosSchema()
    Const conTabulacion As Long = 30
    Dim aDatos As Variant
    Dim i As Long
    Dim strSeccion As String

    strSeccion = "Alumnos.txt"
    ' Llamamos a la función LeeSeccionINI
    aDatos = LeeSeccionINI("Schema.ini", strSeccion)
    ' Muestra los datos en la ventana Inmediato
    Debug.Print
    Debug.Print "Contenido del fichero Schema.ini"
    Debug.Print "Sección: [" & strSeccion & "]"
    Debug.Print "Clave", Tab(conTabulacion); "Valor"
    Debug.Print String(conTabulacion + 20, "-")
    For i = 0 To UBound(aDatos, 1)
        Debug.Print aDatos(i, 0),
        Debug.Print Tab(conTabulacion); aDatos(i, 1)
    Next i

End Sub
```

Tras ejecutar este procedimiento nos mostrará lo siguiente en la ventana de depuración o Inmediato.

Inmediato	
MuestraDatosSchema	
Contenido del fichero Schema.ini	
Sección: [Alumnos.txt]	
Clave	Valor

ColNameHeader	False
Format	FixedLength
MaxScanRows	25
CharacterSet	1252
CurrencySymbol	€
CurrencyDecimalSymbol	,
CurrencyNegFormat	8
NumberDigits	2
Col1	idAlumno Integer Width 6
Col2	FechaNacimiento Date Width 11
Col3	Poblacion Char Width 15
Col4	idCurso Integer Width 5
Col5	ImportePagado Currency Width 11
Col6	Altura Single Width 5

Que, como podemos comprobar, efectivamente se corresponde con los datos contenidos en el fichero **Schema.ini**.

Escritura de una sección completa

Para poder escribir una sección completa, usaremos la función API

WritePrivateProfileSection.

Recordemos la declaración de esta función:

```
Declare Function WritePrivateProfileSection _
    Lib "kernel32" _
    Alias "WritePrivateProfileSectionA" ( _
    ByVal lpApplicationName As String, _
    ByVal lpString As String, _
    ByVal lpFileName As String _
    ) As Long
```

lpApplicationName Sección en la que vamos a escribir las **Claves y Valores**

lpString Lista que contiene las claves y sus Valores.

lpFileName es el fichero en el que se va a escribir los datos.

Al procedimiento a crear lo vamos a llamar **EscribeSeccionINI**.

El parámetro **Datos**, contendrá un **array** semejante al devuelto por la función anterior **LeeSeccionINI**. Por tanto será de dos dimensiones conteniendo datos de tipo **String**.

Crearemos también la función **ConstruyeCadenaConNulos** que hará la función inversa a la que hace la función anterior **DescomponerCadenaConNulos**; es decir **ConstruyeCadenaConNulos** extraerá los datos del **array Datos** y los añadirá a una cadena del tipo **Clave_i=Valor_i**, teniendo como separador el carácter **ASCII 0**.

Al final de la cadena pondrá 2 caracteres **Ascii 0**.

```
Public Sub EscribeSeccionINI( _
    ByVal FicheroINI As String, _
    ByVal Seccion As String, _
    ByVal Datos As Variant, _
    Optional ByVal Carpeta As String)
    ' Presuponemos que se le pasa un Array de _
    ' 2 dimensiones, de tipo String
    ' aDatos(1 to N, 0 to 1) As String
    ' Para grabar los datos vamos a usar la función API _
    ' WritePrivateProfileSection
    Dim strCadenaDatos As String
    Dim lngResultado As Long

    ' Como en los casos anteriores _
    ' efectuamos una cierta validación
    Carpeta = CarpetaValida(Carpeta)
    FicheroINI = FicheroValido(FicheroINI)

    ' Pasamos los datos de Datos a strCadenaDatos
    strCadenaDatos = ConstruyeCadenaConNulos(Datos)

    ' Llamamos a la función API WritePrivateProfileSection
    lngResultado = WritePrivateProfileSection( _
        Seccion, _
        strCadenaDatos, _
        Carpeta & FicheroINI)
End Sub
```

Si la sección no existiera en el fichero, la crearía.

Si la sección ya existiera en el fichero, sustituirá todos los valores anteriores por los nuevos.

La función **ConstruyeCadenaConNulos** que convierte un **array** bidimensional a una cadena con separadores nulos, es la siguiente

```
Public Function ConstruyeCadenaConNulos( _
    Datos As Variant _
) As String
    Dim strDatos As String
    Dim i As Long
    Dim lngIndicemayor As Long
    Dim strNulo As String

    strNulo = Chr(0)
```



```
' Vamos añadiendo los sucesivos eslabones
For i = 0 To UBound(Datos)
    strDatos = strDatos _
        & Datos(i, 0) & "=" & Datos(i, 1) _
        & strNulo
Next i
' Añadimos un nulo adicional al final de la cadena
ConstruyeCadenaConNulos = strDatos & strNulo
End Function
```

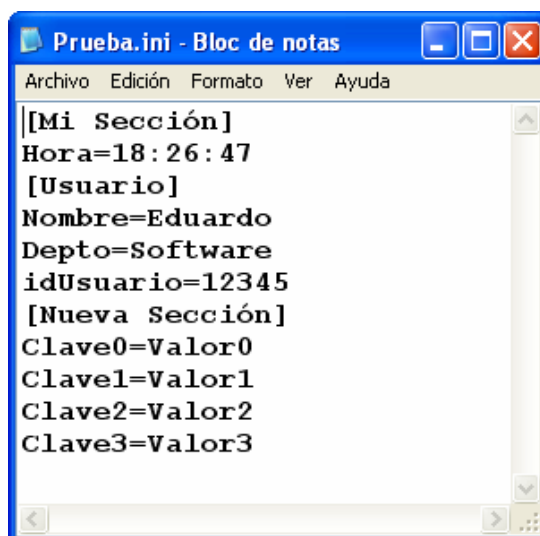
Este procedimiento pone a prueba **EscribeSeccionINI**.

```
Public Sub PruebaGrabacionSeccion()
    Const conDatos As Long = 5
    Dim aDatos(conDatos - 1, 1) As String
    Dim strSeccion As String
    Dim i As Long

    strSeccion = "Nueva Sección"
    For i = 0 To conDatos - 1
        aDatos(i, 0) = "Clave" & CStr(i)
        aDatos(i, 1) = "Valor" & CStr(i)
    Next i

    Call EscribeSeccionINI( _
        "Prueba.ini", strSeccion, aDatos)
End Sub
```

El resultado será



El Registro de Windows.

Hasta la aparición de Windows 95, la forma habitual de almacenar los parámetros de configuración de una aplicación, era la utilización de ficheros **ini**, tal y como hemos visto en los puntos anteriores de esta entrega.

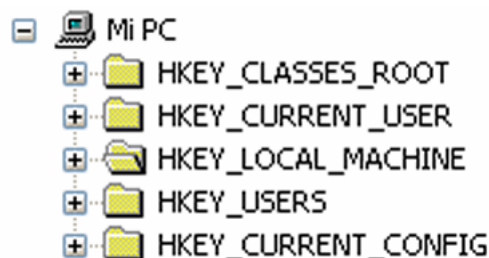
Desde la versión de Windows 95, tenemos a nuestro alcance una herramienta mucho más potente, que evita además la proliferación de ficheros ini, por todas las carpetas de las aplicaciones instaladas. Esta herramienta es el fichero **Regedit.exe**.

Este programa nos permite editar el **Registro de Configuraciones de Windows**, conocido también por su nombre reducido "**El registro**".

Normalmente suele hallarse en la carpeta **C:\Windows**.

El registro de Windows se divide en varios bloques principales.

Mi versión actual de Windows (XP Profesional - Service Pack 2) tiene los siguientes bloques:



Cada uno de los bloques almacena los parámetros de las diversas aplicaciones instaladas, y por supuesto de muchos de los virus, troyanos o programas SpyWare que hayan podido infectar nuestro ordenador.

Para poder leer y escribir en esas secciones debemos utilizar una serie de procedimientos API que nos suministra el propio Windows.

La utilización de estos procedimientos API la veremos en una sección de esta publicación en la que profundizaremos en esta y otras posibilidades no contempladas por el propio VBA.

Escritura y lectura del registro con VBA

Aunque VBA, por sí mismo, tiene limitado el acceso al registro, sí que existe una sección específica del mismo en la que podemos almacenar y leer nuestros parámetros.

Esta sección es

```
HKEY_CURRENT_USER\Software\VB and VBA Program Settings\
```

En esta carpeta del registro podremos almacenar y leer los datos que necesitemos, en unas ramas que partirán de ella con el nombre que le demos a nuestra aplicación.

Por ejemplo, supongamos que tenemos una aplicación llamada Gestión de Socios, y queremos que cada vez que arranquemos el programa averiguar el código del último socio creado, y la última fecha en la que se ha utilizado el programa.

Podríamos crear una sub-Carpeta, que cuelgue de la principal, llamada **Gestión De Socios**, creando una sección llamada **Valores Finales**, y en ella poner las claves **Último Socio** y **Fecha Utilización**, asignando su correspondiente valor.

El procedimiento a usar para grabar los datos en las claves es el **SaveSetting**.

Para Leer los datos podemos usar las funciones **GetSetting** y **GetAllSetting**.

Para borrar las claves y sus datos deberemos utilizar **DeleteSetting**.

Me permito sugeriros que hagáis **copia de seguridad del registro** antes de manipularlo, más que nada por el desastre que podría ocurrir tras un manejo inadecuado.

Instrucción SaveSetting

Actualiza o crea una entrada para una aplicación en el registro de configuración de Windows.

Su sintaxis es

```
SaveSetting NombreAplicación, sección, clave, valor
```

NombreAplicación es el nombre de la subcarpeta que queremos poner colgando de **VB and VBA Program Settings** y que normalmente tiene como nombre el de la propia aplicación.

sección es el nombre de la sección donde queremos grabar el valor de la clave.

valor es el valor, o la expresión que devuelve un valor a grabar.

Supongamos que en nuestro programa **Gestión De Socios** queremos almacenar dentro del registro de Windows los valores **1011** como último socio, y **3 de septiembre de 2005** como fecha de última utilización.

Para ello podemos crear el procedimiento **GrabarDatosDeSalida** al que llamaremos inmediatamente antes de cerrar el programa, pasándole los parámetros adecuados.

Podría ser algo tan simple como esto:

```
Public Sub GrabarDatosDeSalida( _  
                                ByVal Clave As String, _  
                                ByVal Valor As Variant)  
  
    SaveSetting "Gestión De Socios", _  
                "Valores finales", _  
                Clave, _  
                Valor  
  
End Sub
```

Inmediatamente antes de cerrar la aplicación, en alguna parte del código, se hará la siguiente llamada:

```
GrabarDatosDeSalida "Último Socio", 1011  
GrabarDatosDeSalida "Fecha Utilización", #9/3/2005#
```

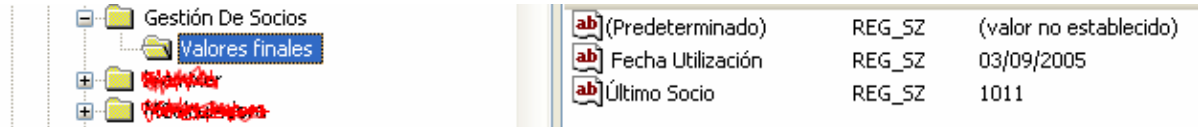
Normalmente en vez de introducir los valores concretos se efectuaría utilizando las variables que los contienen, como parámetros.

Supongamos que el número de socio lo hemos guardado, inmediatamente después de crearlo, en la variable **lngNumeroSocio**, y la fecha de ejecución del programa en la variable **dateEjecucion**.

```
GrabarDatosDeSalida "Último Socio", lngNumeroSocio
```

GrabarDatosDeSalida " Fecha Utilización", datEjecucion

Tras ejecutar estas líneas, si editáramos el registro, podríamos comprobar que se han creado las siguientes entradas:



Función GetSetting

Devuelve el valor de una clave del registro.

Su sintaxis es

```
GetSetting NombreAplicación, sección, clave[,por defecto]
```

Los parámetros **NombreAplicación**, **sección** y **clave**, son los mismos que los explicados en la Instrucción **SaveSetting**.

El parámetro opcional por defecto, permite definir el valor que devolvería la función, si no se encontrara la clave en el registro. Si se omitiera equivaldría a poner una cadena de longitud cero.

Una vez grabados los datos en el punto anterior, al arrancar la aplicación podríamos ejecutar un procedimiento semejante al siguiente, para recuperar los valores del registro.

```
Public Sub CargarDatosDeEntrada()
    Dim datUltimaFecha As Date
    Dim strUltimaFecha As String
    Dim lngSocio As Long

    lngSocio = CLng(GetSetting("Gestión De Socios", _
        "Valores finales", _
        "Último Socio", _
        "0"))
    datUltimaFecha = CDate(GetSetting("Gestión De Socios", _
        "Valores finales", _
        "Fecha Utilización", _
        CStr(Date)))
    Debug.Print "Último socio grabado " _
        & CStr(lngSocio)
    Debug.Print "Última fecha de utilización " _
        & CStr(Date)
End Sub
```

Función GetAllSettings

Devuelve el valor de todas las claves de una sección del registro.

Su sintaxis es

```
GetSetting NombreAplicación, sección
```

Los parámetros **NombreAplicación**, **sección** y **clave**, son los mismos que los explicados en la Instrucción **SaveSetting**.

Las claves son guardadas en una variable de tipo **Array** de dos dimensiones basada en cero, declarada inicialmente como del tipo **Variant**.

Una vez cargados los datos en el array, para poder acceder a sus valores, podremos utilizar un bucle que vaya desde el índice cero, hasta el valor del índice mayor (función **UBound**).

Usando como valor el 0 en el segundo índice podremos obtener los nombres de las claves.

Si utilizamos como para el segundo índice el 1, obtendremos los valores de las claves.

Veamos un ejemplo que aclarará este galimatías.

```
Public Sub MostrarDatosRegistro()

    Dim aDatos As Variant
    Dim i As Long
    aDatos = GetAllSettings ("Gestión De Socios", _
        "Valores finales")
    For i = 0 To UBound(aDatos, 1)
        Debug.Print "Clave " & CStr(i) _
            & " - " _
            & aDatos(i, 0) & ": " & aDatos(i, 1)
    Next i
End Sub
```

La línea cabecera del bucle, en vez de basarla directamente en el índice 0 podríamos usar la función **LBound** que nos devuelve el índice más bajo del **array**.

```
For i = LBound(aDatos, 1) To UBound(aDatos, 1)
```

Tras ejecutarse este procedimiento se mostrará en la ventana inmediato, lo siguiente:

```
Clave 0 - Último Socio: 1011
Clave 1 - Fecha Utilización: 03/09/2005
```

Instrucción DeleteSetting

Elimina una sección completa o una clave en el registro de configuración de Windows.

Su sintaxis es

```
DeleteSetting NombreAplicación, sección[, clave]
```

Los parámetros son los mismos que los de los procedimientos anteriores.

El nombre de la clave es opcional, si no se pusiera se borraría la sección completa.

Por ejemplo

```
DeleteSetting "Gestión De Socios", _
    "Valores finales", _
```

"Fecha Utilización"

Borraría la clave "Fecha Utilización".

Si hubiéramos ejecutado

```
DeleteSetting "Gestión De Socios", _  
"Valores finales"
```

Se borraría por completo la sección "Valores finales".

Grabar, leer y borrar cualquier Sección – Clave del registro

Como ya hemos indicado, los procedimientos anteriores son capaces de trabajar con secciones y claves ubicadas a partir de la rama del registro

```
HKEY_CURRENT_USER\Software\VB and VBA Program Settings\  
Para poder escribir en cualquier otra sección hay que utilizar APIS de Windows, como son:
```

RegOpenKeyEx	CreateRegistryKey
RegCloseKey	RegDeleteKey
RegCreateKeyEx	Etc...

Estos procedimientos los veremos más adelante en un capítulo específicamente dedicado al API de Windows.

Notas sobre este capítulo:

No quisiera desmoralizar a un posible lector, por el tipo de funciones que hemos analizado. En principio, la utilización de las funciones API y las librerías de Acceso a Datos **DAO** y **ADO**, las veremos más adelante. Cuando abordemos esos temas la comprensión de lo aquí expuesto en principio debería ser inmediata.

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
```

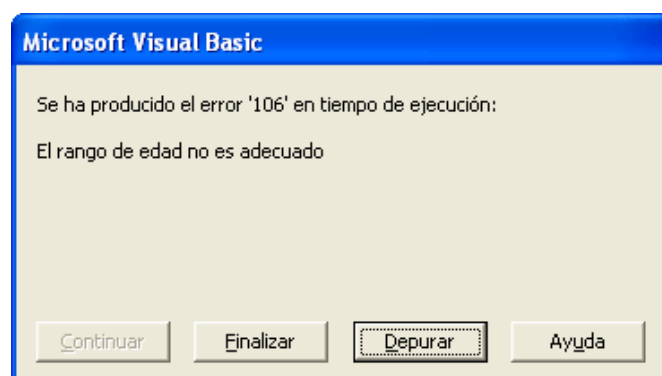
Se supone que el resto del código permanece igual.

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:

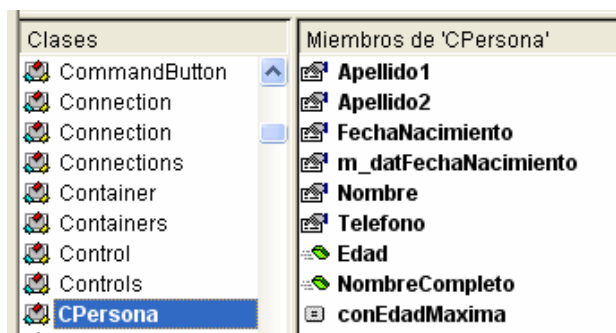
```
Public Property Get Edad() As Long
    Edad = (Date - FechaNacimiento) / 365.2425
End Property
```

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

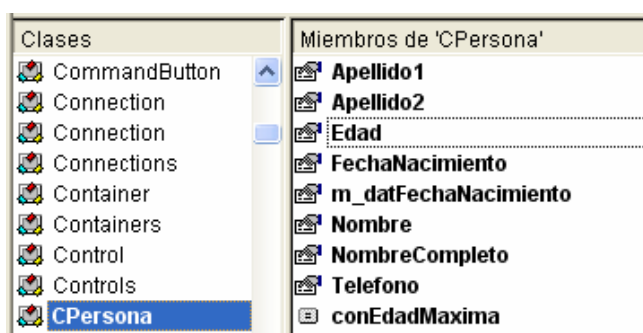
Antes de efectuar este cambio, estos procedimientos

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.

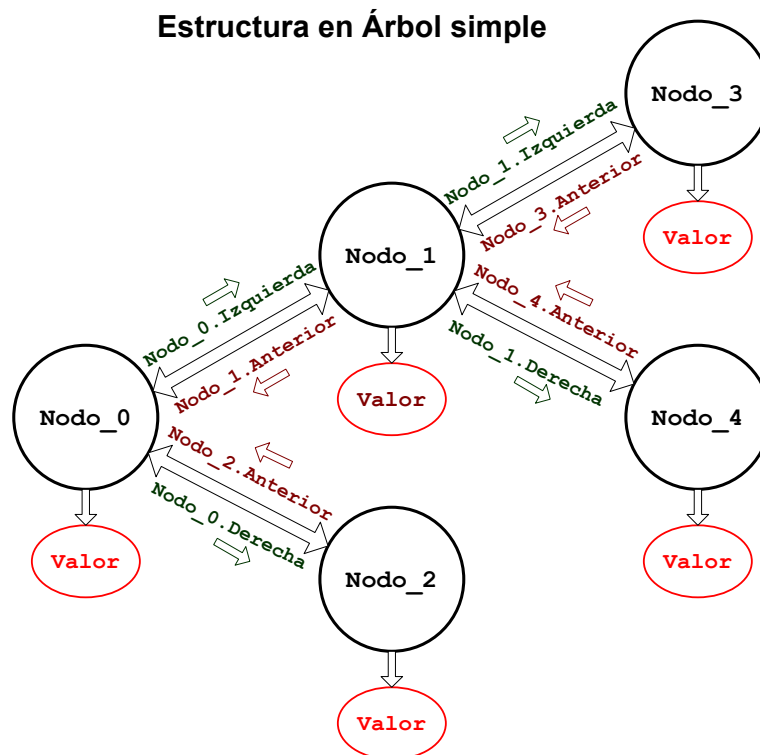
Estableceremos la comunicación entre Nodos, de forma que será el nodo el eslabón que servirá para desplazarse por el Árbol.

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
```

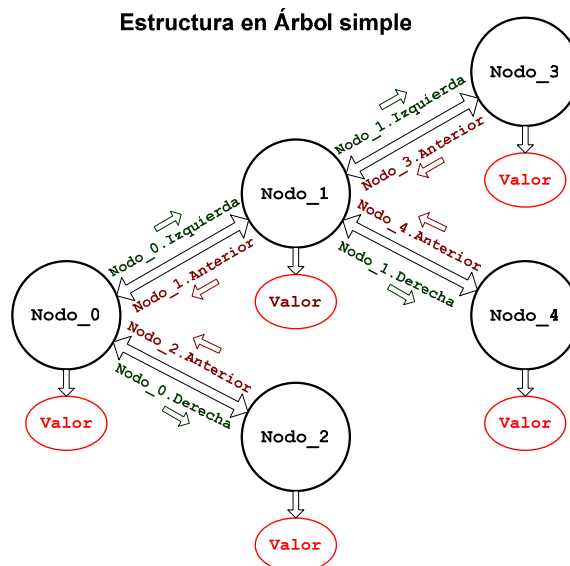
```

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 n° " & 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
```

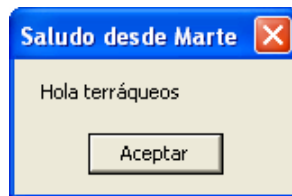
```
Private Sub cmdSaludo_Click()
```

```
    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.

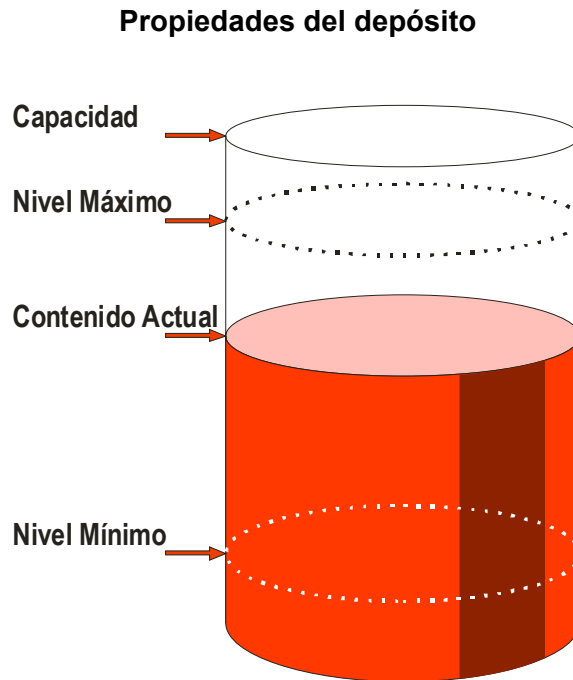
- Capacidad del depósito
- Contenido actual del depósito
- Nivel máximo de seguridad
- Nivel mínimo de seguridad

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**.

```

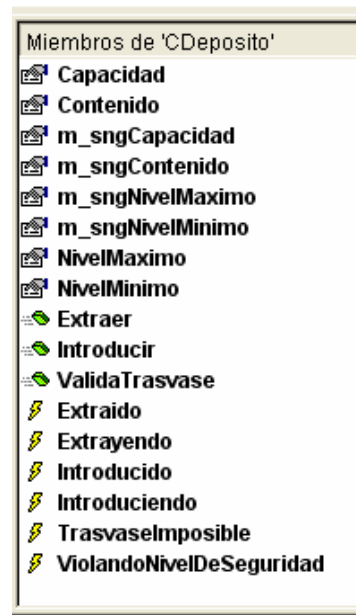
m_sngContenido = m_sngContenido + Volumen
    RaiseEvent Introducido (Volumen)

```

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

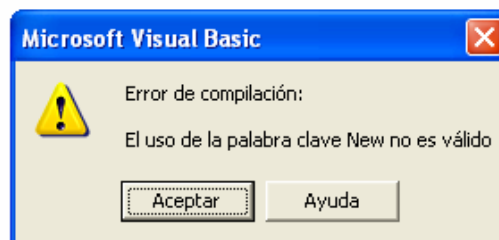
```
Dim WithEvents Deposito As CDeposito
```

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_Extraendo**.

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

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

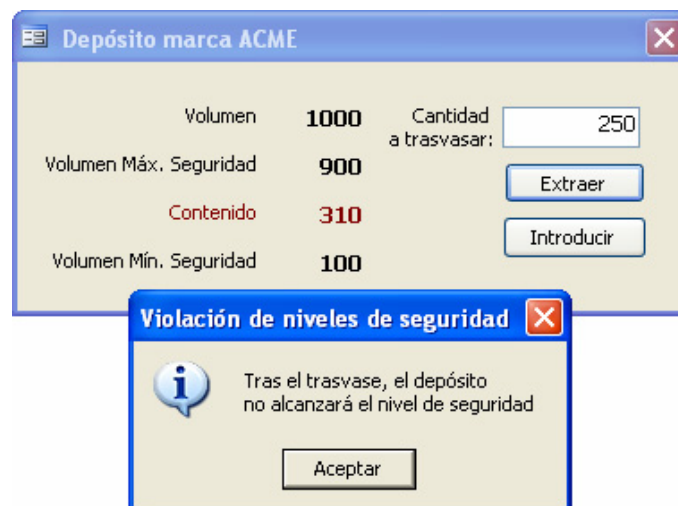
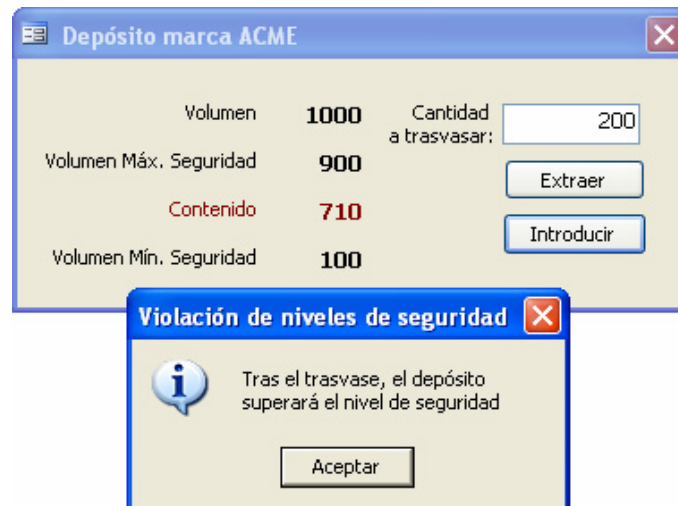
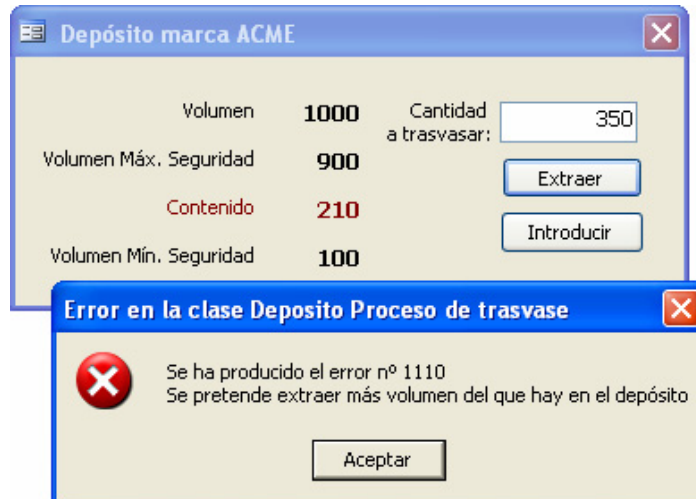
 550 es demasiado volumen para trasvasarlo de una vez

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 inmediatamente 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

```

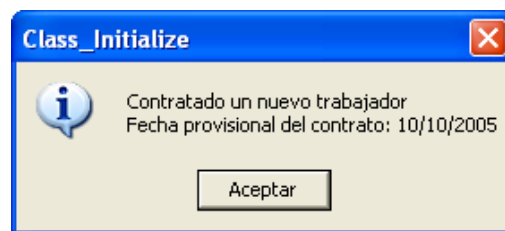
Para probar la clase y comprobar los efectos de los eventos, escribimos en un módulo normal el siguiente código:

```

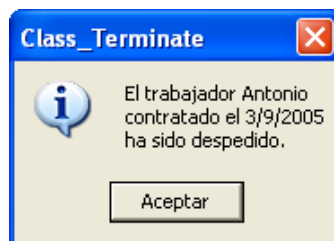
Public Sub PruebaCTrabajador()
    Dim Empleado As New CTrabajador
    Empleado.Nombre = "Antonio"
    Empleado.FechaContrato = #9/3/2005#
End Sub

```

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.

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.








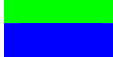


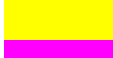
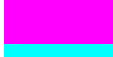
*Como aclaración a esto último, comentaré que el color que se ve por el monitor, es la suma de tres colores primarios, el **Rojo**, el **Verde** y el **Azul**.*





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 (**R**ed), Verde (**G**reen) y Azul (**B**lue).

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)

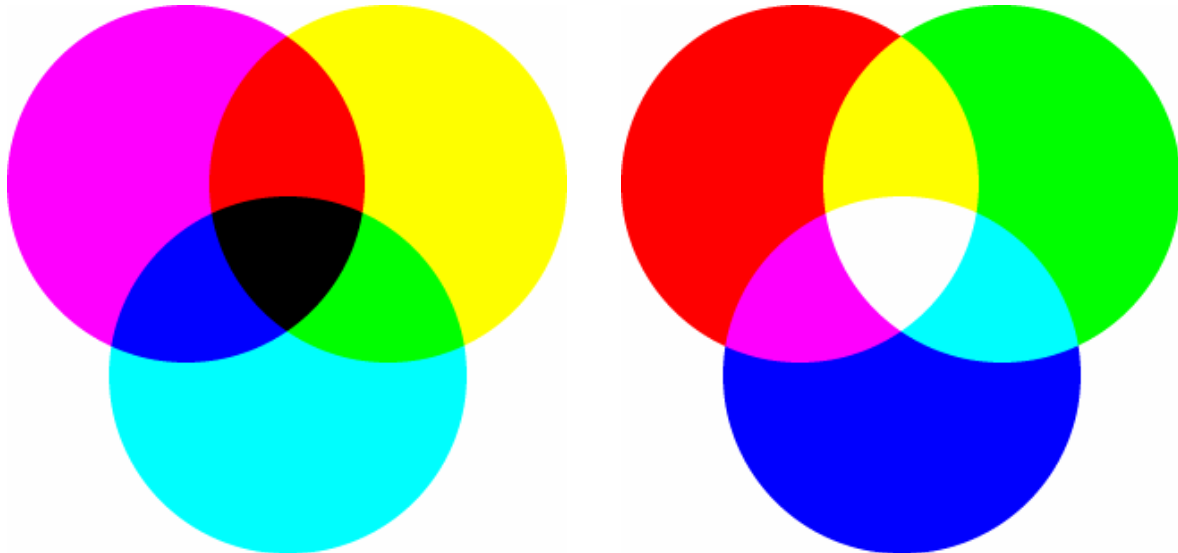
Int(15106112 / 65536) → 230 (azul)

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.

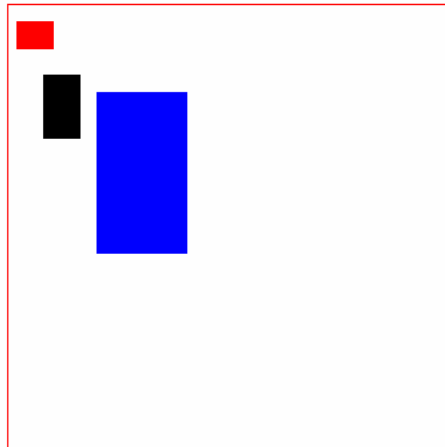
Se nos abrirá el módulo de clase del formulario en el manejador del evento **Format**

```
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:



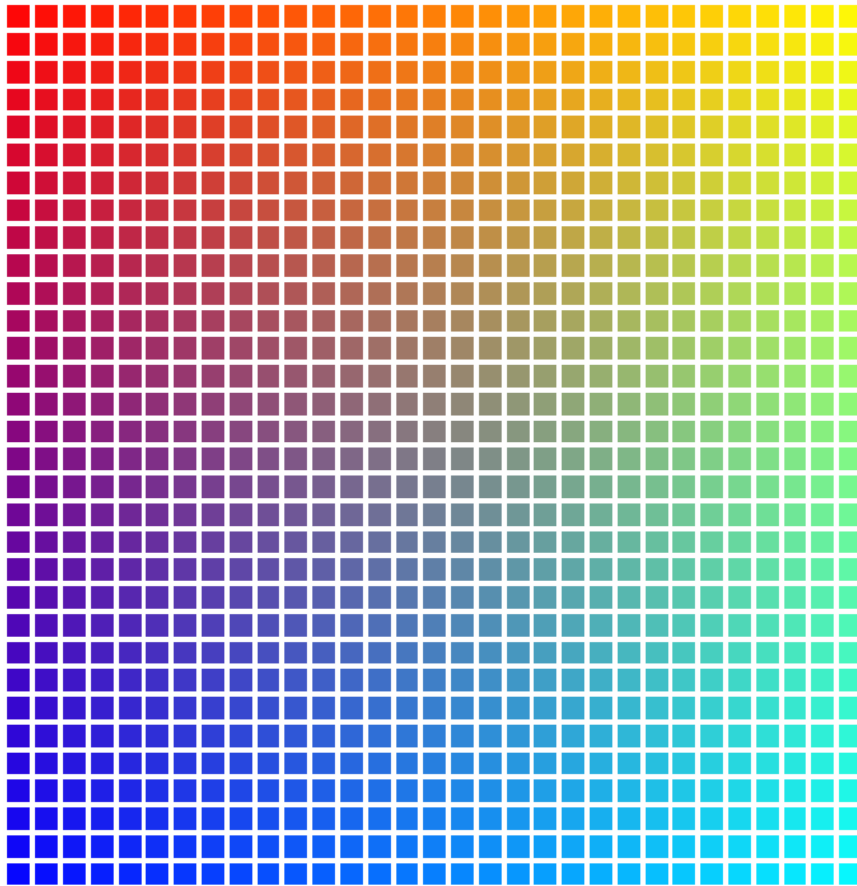
Como lo anterior era una prueba, eliminamos las sentencias **Line** anteriores y escribimos:

```
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.

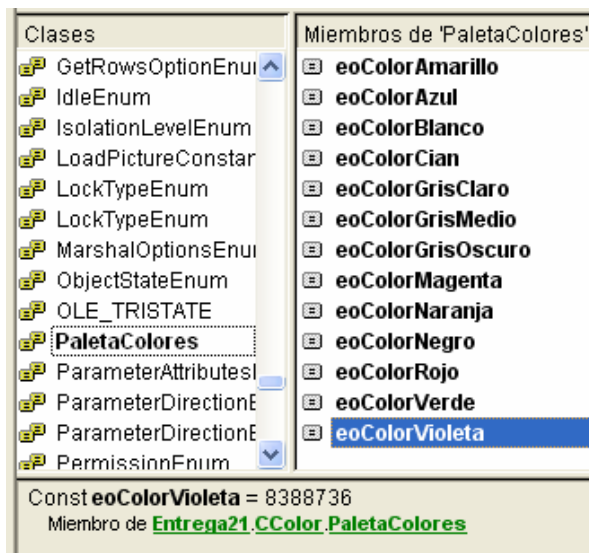
Enumeraciones en un módulo de clase

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"
```

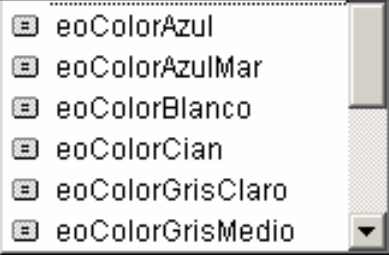
```
Case eoColorAzulMar
    NombreColor = "Azul marino claro"
Case eoColorVioleta
    NombreColor = "Violeta"
Case eoColorBlanco
    NombreColor = "Blanco"
Case eoColorGrisClaro
    NombreColor = "Gris Claro"
Case eoColorGrisMedio
    NombreColor = "Gris Medio"
Case eoColorGrisOscuro
    NombreColor = "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:

```
Public Sub PruebaEnumeración()
    Debug.Print NombreColor(|
End Sub
```

NombreColor(**Color As PaletaColores**) As String



- eoColorAzul
- eoColorAzulMar
- eoColorBlanco
- eoColorCian
- eoColorGrisClaro
- eoColorGrisMedio

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; advirtiéndole 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 heredará 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 distinguan 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 **Mauallar**

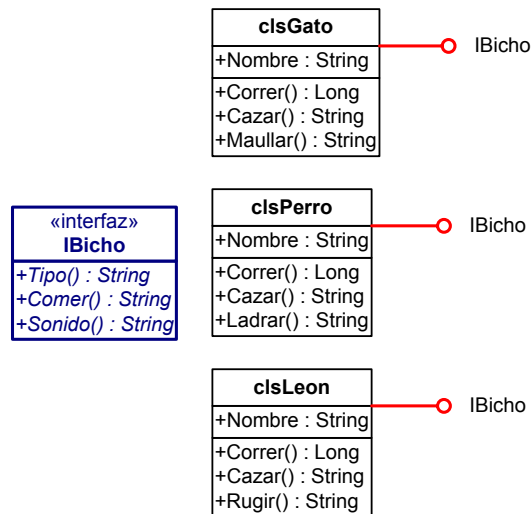
La clase **clsPerro** mostrará **Nombre**, **Correr**, **Cazar** y **Ladrar**

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

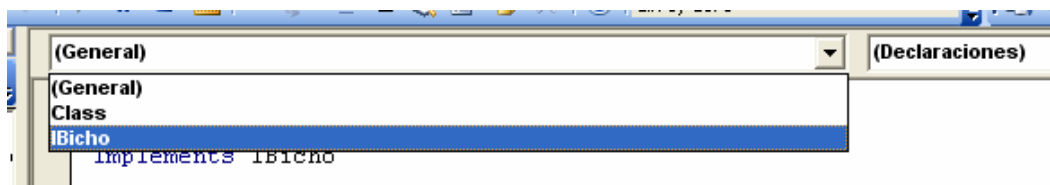


Vamos ahora a crear la clase **clsGato**, para lo que insertamos el correspondiente módulo de clase.

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

Implements IBicho

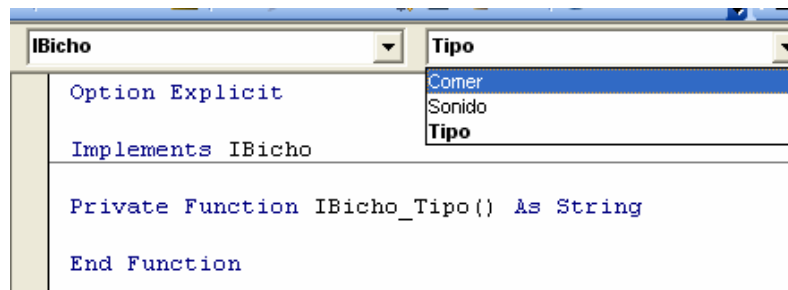
Tras esto, en el editor de código podemos seleccionar la interfaz **IBicho** del gato, y el editor nos escribirá la estructura del método **IBicho_Tipo**.



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 seleccionarlás 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**.

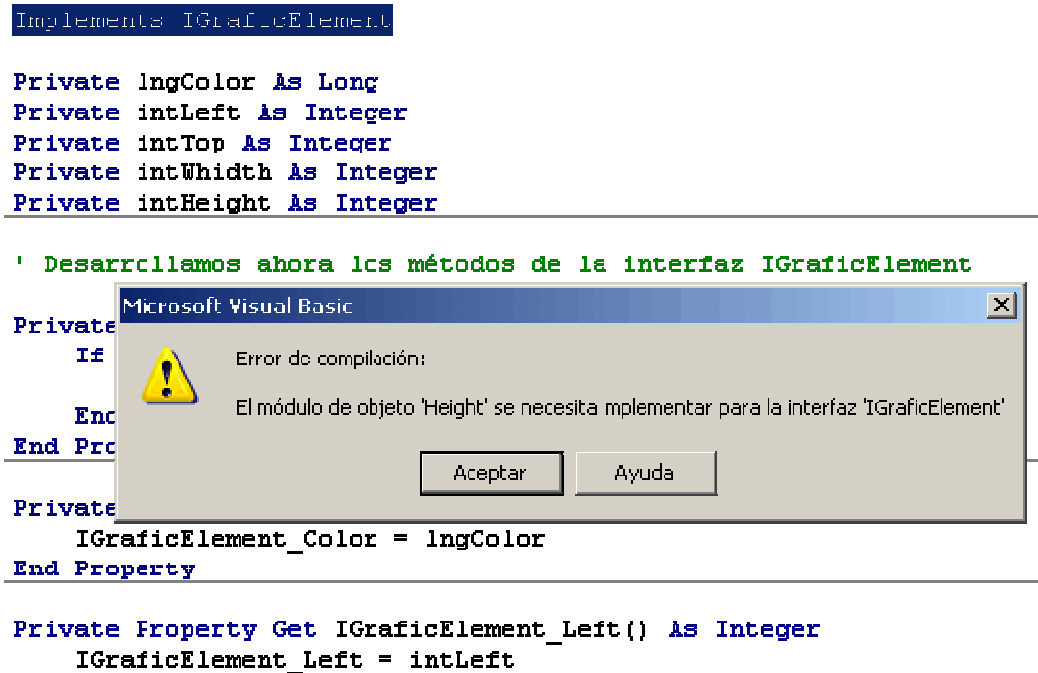
Para dibujar una elipse, un círculo, un sector circular ó elíptico, usaremos el método **Circle**.

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:

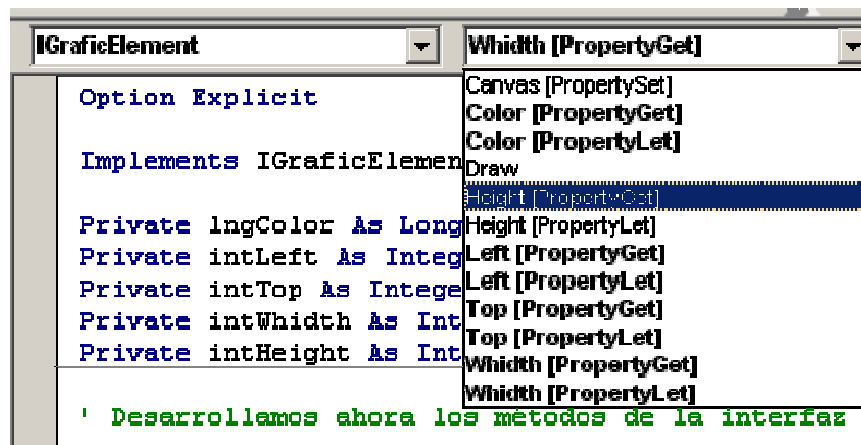
No entra en el objetivo de este trabajo mostrar cómo desarrollar controles ActiveX con Visual Basic, con C++ o con cualquier otra plataforma de desarrollo de software..

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
    End If
    X1 = X0 + intWhidth
End Sub
```



```
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.

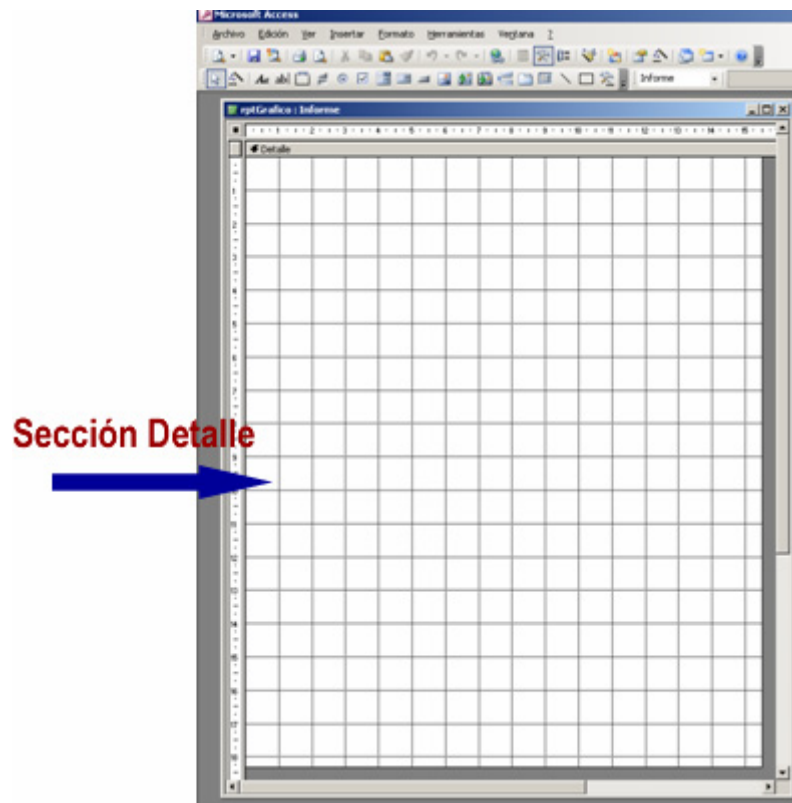
Ya he comentado que, en Access, el único objeto que admite métodos gráficos es el objeto Report.

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
        .Whidth = 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
```

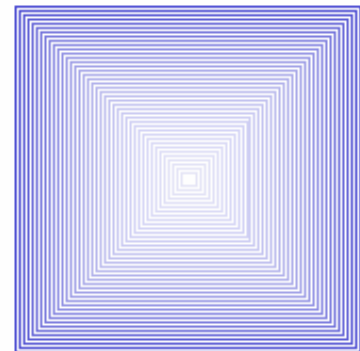
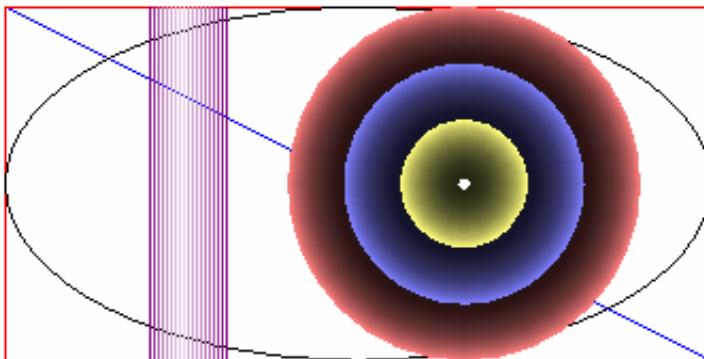
```
Set objGrafico = Elipse
With objGrafico
    ColorObjeto.Color = eoColorRojo
    ColorObjeto.AclaraColor 2 * CambioLuz
    .Left = 2000
    .Whidth = 2500
    .Height = 2500
For i = 1 To 2 * Bucles
    .Color = ColorObjeto.Color
    .Top = .Top + Incremento / 6
    .Left = .Left + Incremento / 6
    .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**, **Rectángulo** y **Elipse**, que a su vez son instancias de las clases **clsRecta**, **clsRectángulo** 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 `clsRectangulo`.

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 `Rectangulo` 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

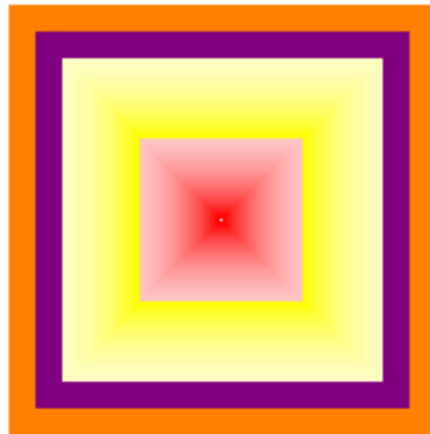
Al igual que en el caso anterior creamos un nuevo informe en el que aumentaremos la altura de la sección detalle y aprovechamos los eventos del mismo.

```
Private Sub Detalle_Print( _
    Cancel As Integer, _
    PrintCount As Integer)
    Const Escala As Integer = 30
    Dim intLado As Integer
    Dim i As Integer

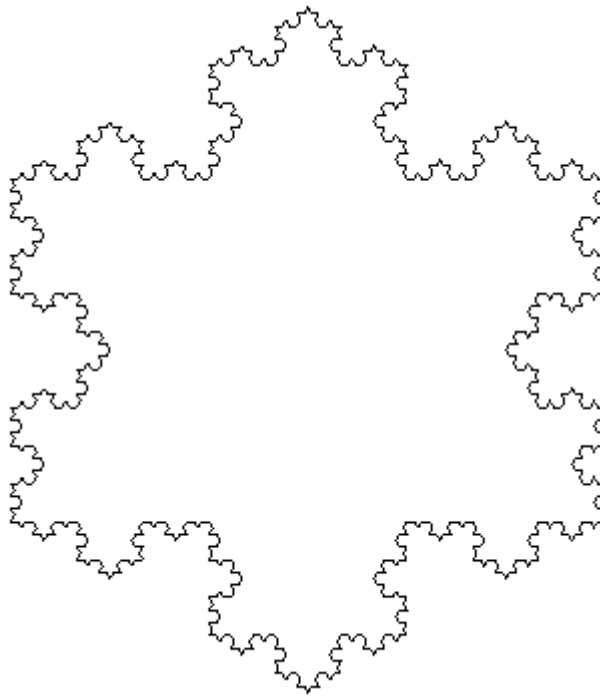
    objColor.Color = eoColorRojo
    For i = 1 To 30
        intLado = Escala * i
        With Cuadrado
            .Color = objColor.Color
            .Side = intLado
            .Left = 2500 - intLado / 2
            .Top = .Left
```

```
        .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.

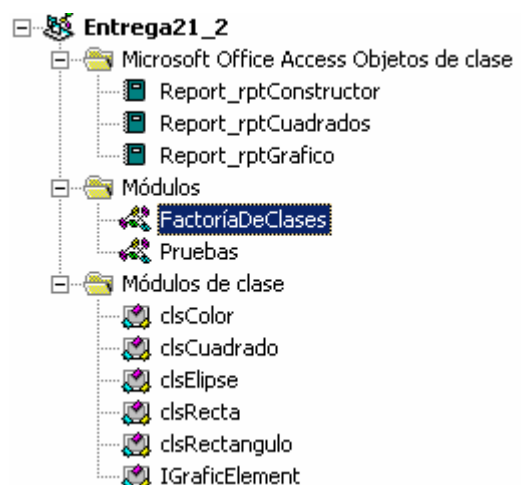
Factoría de Clases

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.



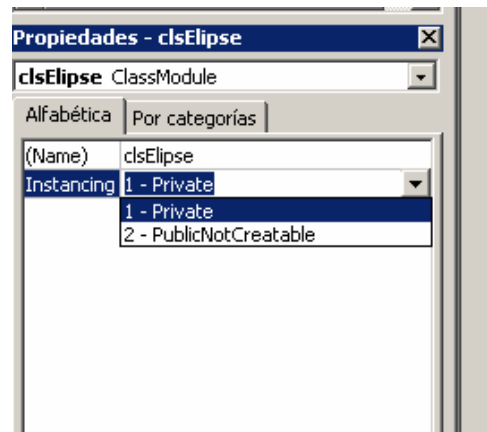
Propiedad Instancing de la clase

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.

Comencemos a programar con
VBA - Access

Entrega **22**

Objetos de Access
Formularios (1)

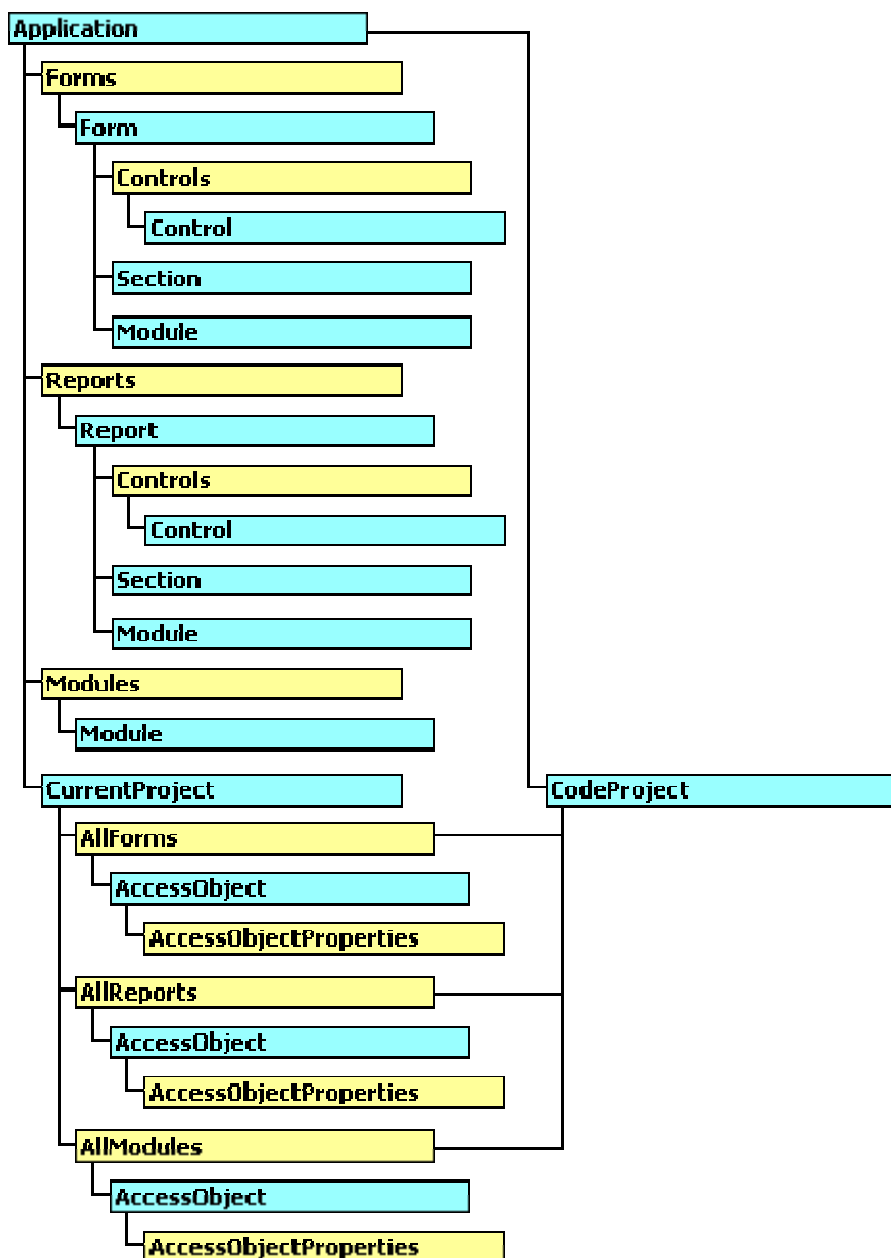
Estructura de los Objetos de Access

Una aplicación Access está compuesta de una gran cantidad de objetos que se interrelacionan.

Algunos de ellos son objetos individuales, pero otros están agrupados en Colecciones.

En la cúspide de todos ellos está el objeto **Application**

El objeto **Application** hace referencia a la aplicación de Access que está activa en ese momento en el ordenador.



Del objeto **Application** descienden una gran variedad de objetos y colecciones

- **CodeData**: Objetos guardados en la base de datos de códigos por la aplicación (servidor) de origen (Jet o SQL).
- **CodeProject**: Proyecto para la base de datos de códigos de Microsoft. Una de sus colecciones es **AllForms**, que referencia a cada formulario.

- **CurrentData**: Objetos guardados en la base de datos activa.
- **CurrentProject**: Proyecto actual de Access. Una de sus colecciones es **AllForms**.
- **DataAccessPages** (Colección): Páginas de acceso a datos abiertas actualmente.
- **DefaultWebOptions**: Atributos globales en el nivel de aplicación en una página Web.
- **DoCmd**: Objeto para ejecutar las acciones de Microsoft Access desde Visual Basic. (Ya lo hemos visto en el capítulo 16)
- **Forms** (Colección): Formularios abiertos de Access.
- **Modules** (Colección): Módulos estándar y de clase abiertos.
- **Printers** (Colección): Representan a las impresoras disponibles en el sistema.
- **References** (Colección): Las referencias establecidas actualmente.
- **Reports** (Colección): Informes abiertos.
- **Screen**: Formulario, informe o control que tiene el enfoque actualmente.

La estructura de objetos representada en el gráfico anterior es sólo un resumen del total de objetos propios de Access.

Los objetos representados en color amarillo, con nombre en plural, representan colecciones.

Por ejemplo la colección **Forms** contiene objetos del tipo **Form**, que a su vez contiene una colección **Controls** de objetos **Control**.

Un objeto **Control** tiene la colección **Properties** aunque no está representada en el gráfico.

Esta colección, como su nombre indica, contiene las propiedades de ese control.

La colección **Properties** la poseen, no sólo los controles; también la tienen los objetos **Form**, **Subform**, **Report**, **Section**, y como veremos más adelante, los objetos de acceso a datos de **DAO** y **ADO**.

Diferentes objetos pueden tener el mismo tipo de colecciones; por ejemplo vemos que los objetos **Form** y **Report** poseen la colección **Controls** que contiene a sus respectivos controles.

También podemos ver el paralelismo existente entre el objeto **CurrentProject** y **CodeProject**, ambos pertenecientes al objeto **Application**.

En ésta y próximas entregas, estudiaremos los objetos presentados en el gráfico de la página anterior.

El objeto **DoCmd**, no representado en el gráfico, ya lo estudiamos en el capítulo 16, aunque lo seguiremos utilizando en ésta y en las próximas entregas.

Formularios

De todos los objetos de Access vamos a empezar a trabajar con los formularios.

Un formulario es el elemento básico para la introducción de datos y su mantenimiento, por parte del usuario final de nuestra aplicación.

También puede servir para mostrar avisos, al estilo de un Cuadro de mensaje, como formulario de inicio de una aplicación, como formulario del tipo “Acerca de...” o incluso como contenedor de otro subformulario u objetos ActiveX.

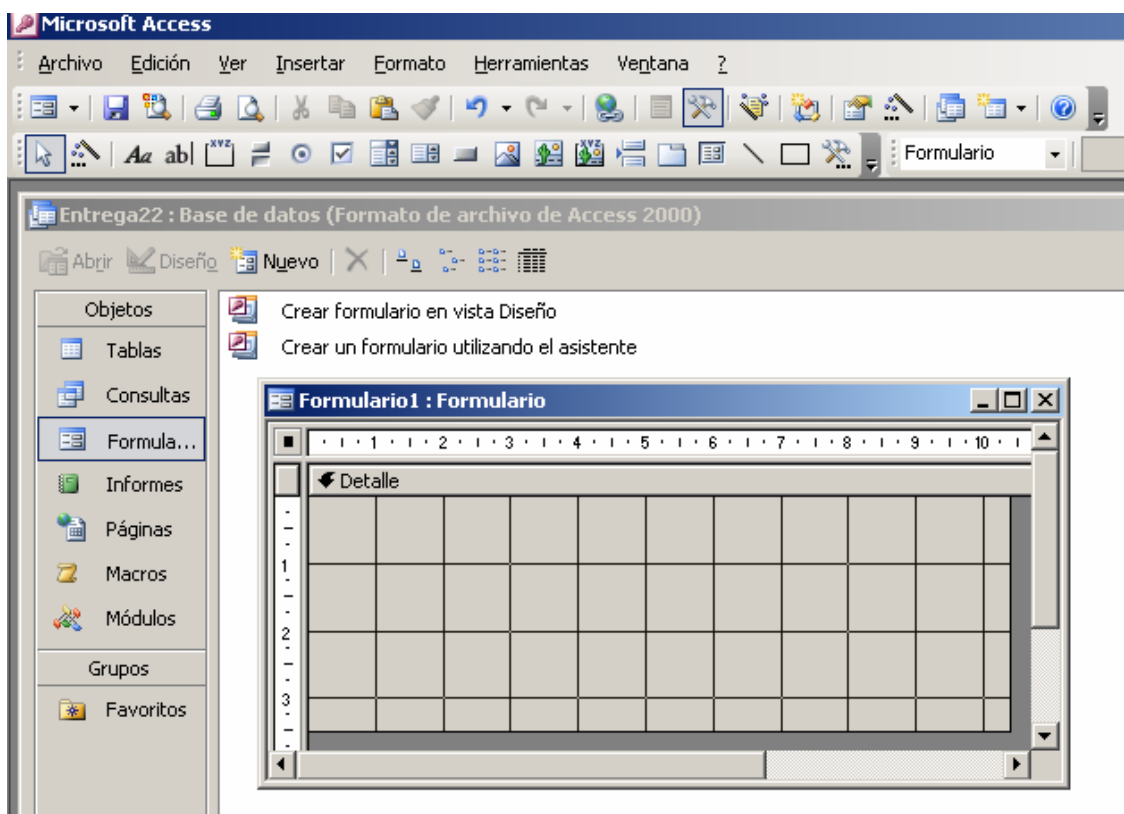
Ya hemos comentado que el objeto **Form**, que representa a un formulario, es un elemento de la colección **Forms**.

Vemos también que un objeto **Form** contiene como atributo el objeto **Module**.

El objeto **Module** representa el módulo de clase del formulario, en el que podemos, al igual que en una “clase normal” definir propiedades y métodos personalizados.

Para efectuar las pruebas de código, vamos a crear un formulario “sencillo” y a trabajar con su código de clase.

Para simplificar más las cosas no lo enlazaremos con ningún origen de datos.

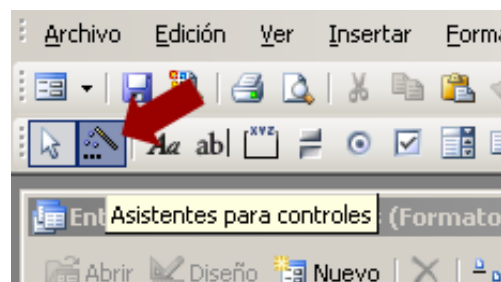


Eventos de un formulario

Ahora vamos a poner un botón que nos servirá para cerrar el formulario, una vez abierto.

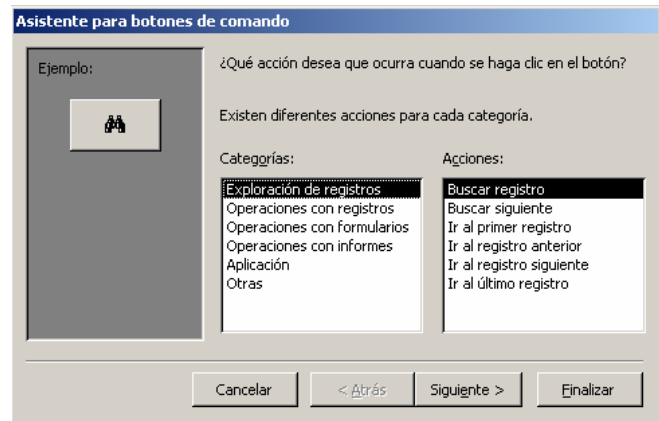
Usaremos el [Asistente para Controles].

Compruebe que está activada la “Varita mágica”

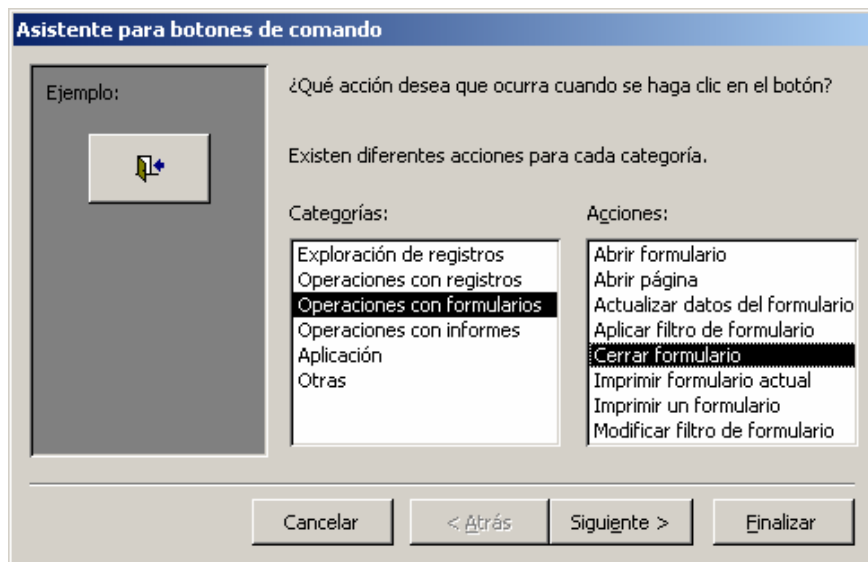


Seleccionamos el Botón de Comando y lo dibujamos en la esquina inferior derecha del formulario.

Tras esto se nos abre el asistente y nos pregunta qué queremos hacer con el botón:



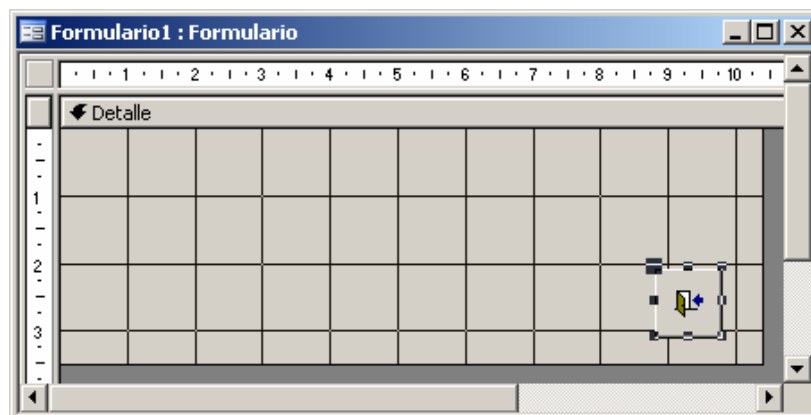
Seleccionamos en la lista izquierda **Operaciones con Formularios**, y en la derecha **Cerrar formulario**.



Como imagen seleccionamos Salir (la puerta con la flecha).

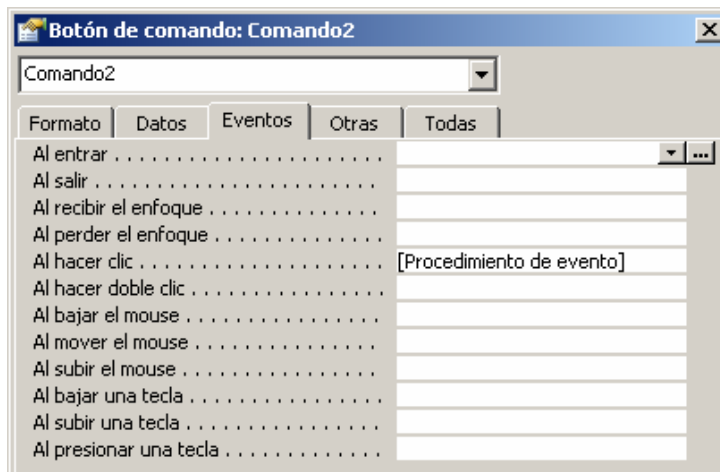
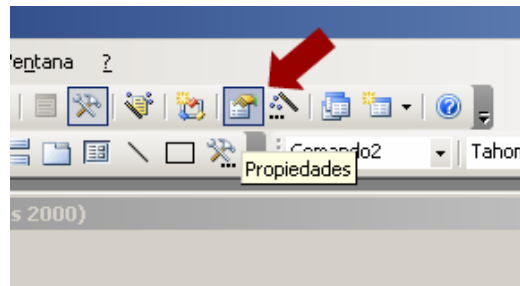
Tras esto pasan dos cosas:

El botón se nos muestra en el formulario, con el gráfico de la puerta y la flecha:



Pero, aparte de esta imagen, más o menos estética, ha ocurrido algo muy importante en el módulo de código asociado al formulario.

Si teniendo seleccionado el botón, abrimos el editor de Propiedades, pulsando el botón [Propiedades] del menú, seleccionamos la pestaña [Eventos] y en ella podemos ver que el evento Al hacer clic tiene asociado un procedimiento de evento en el código de clase asociado al formulario.



Si pinchamos con el ratón en la palabra [Procedimiento de evento], se nos abre un botón con puntos suspensivos a su derecha.

Pulsando en ese botón se nos abriría el editor de código teniendo posicionado el cursor en el procedimiento correspondiente.

También podríamos ver el código pulsando en el botón [**Código**] del menú:



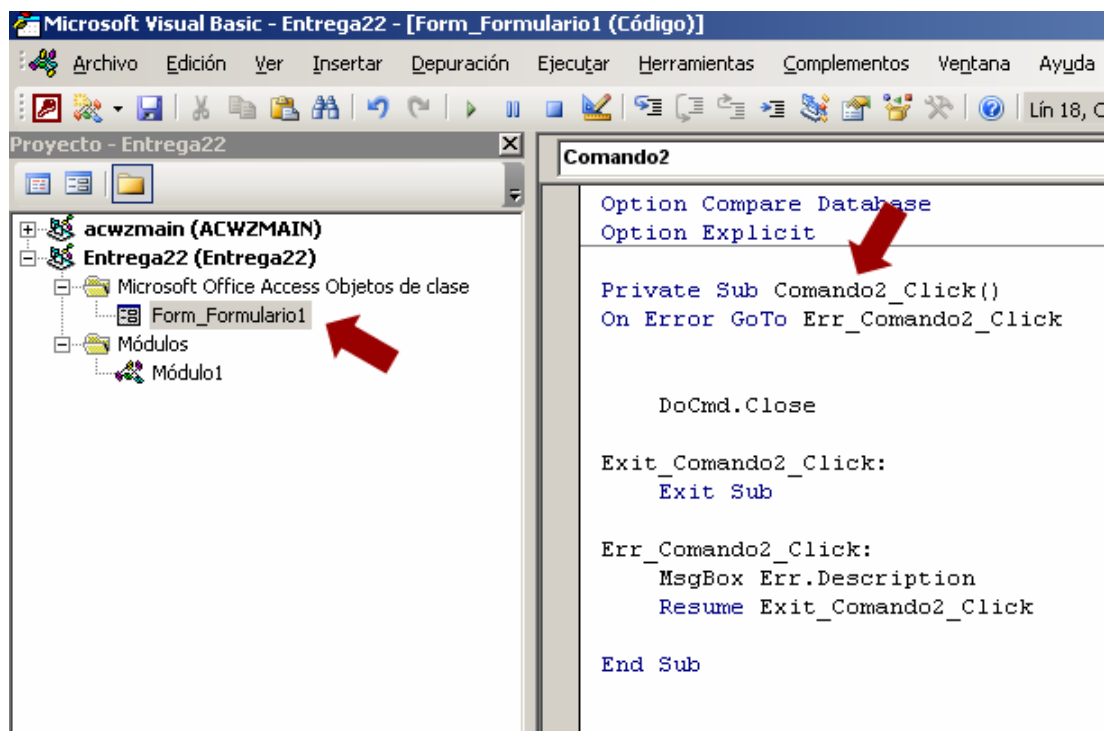
En ambos casos se nos abrirá el editor de código, y en él podremos apreciar una serie de hechos:

En la ventana del **Explorador de proyectos**, vemos que de la carpeta Microsoft Office Objetos de clase, cuelga el objeto **Form_Formulario1**.

Asociado a este objeto podemos ver el código de su módulo de clase, en el que ha aparecido el procedimiento

```
Sub Comando2_Click ()
```

Este procedimiento se ejecutará cada vez que presionemos sobre el botón de nombre **Comando2**. Este botón es el que hemos colocado en el formulario.



La parte del león de este procedimiento es la línea **DoCmd.Close** que lo que hace es cerrar el formulario actual, como ya vimos en el capítulo 16.

Además, de forma automática se han creado varias líneas de código para la gestión de posibles errores.

Como vimos en el capítulo 11, la primera línea indica la etiqueta de la línea a la que debe saltar el código si se produjera un error; en este caso a la línea **Err_Comando2_Click:**

Una vez que salte a sea línea, el código hará que de muestre la descripción del error, tras lo que anulará el error y efectuará un salto a la etiqueta **Exit_Comando2_Click:** desde donde saldrá del procedimiento mediante **Exit Sub**.

Podemos cambiar ese código.

Una cosa que no me hace especialmente feliz es el nombre del botón.

En general me gustan más nombres que sean descriptivos, por ejemplo **cmdCerrar**.

Tampoco me gustan las etiquetas del control de errores, que ha creado el asistente por lo que voy a modificar el código para quede así:

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

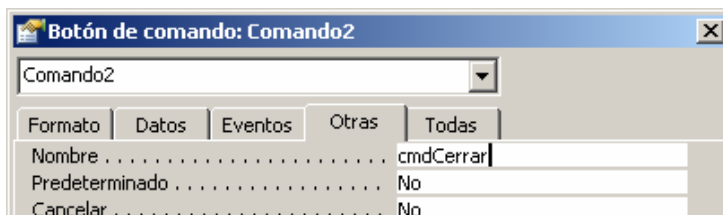
Lo primero que he hecho ha sido cambiar el nombre del procedimiento gestor del evento **Click** del botón. Esto es así porque quiero que el botón se llame **cmdCerrar**. **cmd** indica que es un botón y **Cerrar** indica el procedimiento que ejecuta, Cerrar el formulario.

Al haber eliminado el procedimiento **Comando2_Click()**, hemos eliminado también el enlace entre el evento Clic del botón y su procedimiento gestor.

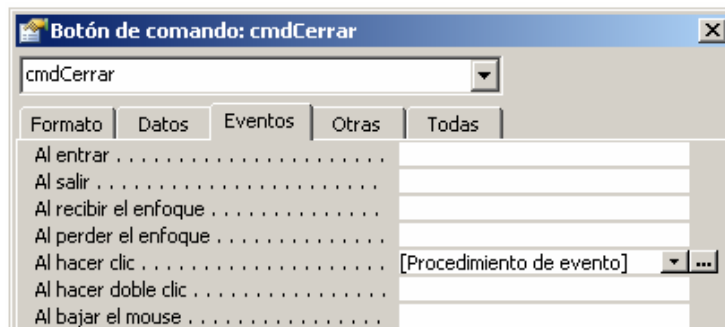
Por lo tanto es algo que deberemos hacer manualmente.

Nos volvemos al diseño del formulario cerrando la ventana del editor de código, por ejemplo pulsando en el aspa superior derecha de la ventana.

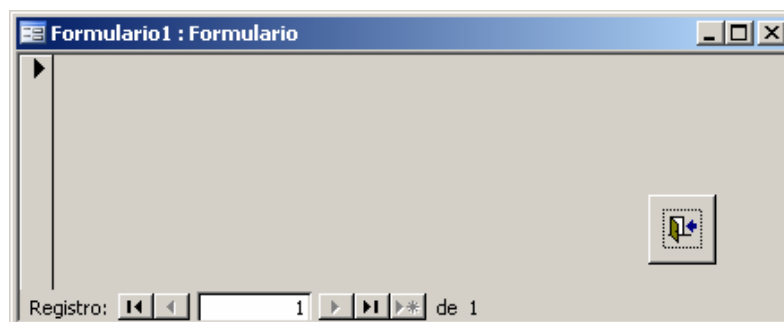
Volvemos a abrir la ventana de propiedades, y lo primero que vamos a hacer es cambiar el nombre del botón, cosa que haremos pulsando en la pestaña [Otras] seleccionando la propiedad Nombre y escribiendo **cmdCerrar**.



A continuación asignamos el procedimiento **cmdCerrar_Click()** al evento **Al hacer clic** del botón, como hemos visto en un párrafo anterior.



Guardamos los cambios y abrimos el formulario.



No es nada espectacular, pero podemos comprobar que al apretar el botón, el formulario se cierra, que es lo que en principio queríamos.

Durante la vida de un formulario ocurren una serie de eventos.

Es interesante saber en qué orden se producen éstos, tanto al abrirse el formulario, como al cerrarse.

Para ello vamos a seleccionar una serie de eventos del formulario y a escribir un pequeño código que indicará en qué orden se han producido estos eventos.

Primero creamos, a nivel del módulo, una variable que se irá incrementando cuando se produzcan determinados eventos.

Vamos a seleccionar los siguientes eventos del formulario:

- | | | |
|---------------------|----------------------|---|
| ▪ Activate | Al activar | Cuando el formulario pasa a ser ventana activa |
| ▪ Close | Al cerrar | Al cerrarse y retirarse de la pantalla |
| ▪ Current | Al activar registro | Cuando se enfoca un registro como actual |
| ▪ Deactivate | Al desactivar | Cuando la ventana del formulario pierde el foco |
| ▪ GotFocus | Al recibir el foco | Cuando el formulario recibe el foco |
| ▪ Load | Al cargar | Al abrir un formulario y mostrar sus registros |
| ▪ LostFocus | Al perder el foco | Cuando el formulario pierde el foco |
| ▪ Open | Al abrir | Al abrir pero antes de mostrar el primer registro |
| ▪ Resize | Al cambiar el tamaño | Al abrir un formulario o cambiar de tamaño |
| ▪ Unload | Al descargar | Al cerrar un formulario, antes de desaparecer |

Vamos asociando cada uno de los eventos y escribimos su código

El código sería tan simple como éste:

```
Option Compare Database
Option Explicit

Dim intNumero As Integer

Private Sub Form_Activate()
    MuestraOrden "Activate"
End Sub

Private Sub Form_Close()
    MuestraOrden "Close"
End Sub

Private Sub Form_Current()
    MuestraOrden "Current"
End Sub

Private Sub Form_Deactivate()
    MuestraOrden "Deactivate"
End Sub

Private Sub Form_GotFocus()
    MuestraOrden "GotFocus"
End Sub

Private Sub Form_Load()
    MuestraOrden "Load"
```



```
End Sub

Private Sub Form_LostFocus ()
    MuestraOrden " LostFocus "
End Sub

Private Sub Form_Open(Cancel As Integer)
    MuestraOrden "Open"
End Sub

Private Sub Form_Resize()
    MuestraOrden "Resize"
End Sub

Private Sub Form_Unload(Cancel As Integer)
    MuestraOrden "Unload"
End Sub

Private Sub MuestraOrden(ByVal Evento As String)
    intNumero = intNumero + 1
    Debug.Print CStr(intNumero) & " " & Evento
End Sub

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

¿Qué hace este código?

Cuando se produce cualquiera de los eventos incluidos en el mismo, éste llama al procedimiento **MuestraOrden**, pasándole como parámetro el nombre del evento.

Este procedimiento incrementa la variable **intNumero** e imprime su valor, junto con el nombre del evento pasado, en la ventana **Inmediato**.

Con este simple procedimiento podemos averiguar en qué orden se van produciendo los eventos.

Grabamos, abrimos el formulario y lo cerramos presionando el botón de cierre del formulario **cmdCerrar**.

Si abrimos el editor de código para ver qué ha escrito en la ventana **inmediato**, veremos lo siguiente:

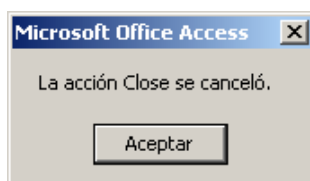
- 1 Open
- 2 Load
- 3 Resize
- 4 Activate
- 5 Current
- 6 Click del botón
- 7 Unload
- 8 Deactivate
- 9 Close

Hasta que presionamos el botón, evento número 6, vemos que el primer evento generado es el **Open**. A continuación el **Load**. El evento **Resize** se genera la primera vez que se “dibuja” el formulario. A continuación se activa el formulario (**Activate**) y por último trata de mostrar el posible registro activo (**Current**).

Tras presionar el botón de cierre, se genera el evento **Unload** a continuación se desactiva el formulario (**Deactivate**), y justo antes de que se cierre y se descargue de memoria, el evento **Close**.

Si nos fijamos en el evento Unload, vemos que incluye el parámetro Cancel.

Si en el evento, asignáramos a la variable cancel el valor **True**, detendríamos la descarga del mismo, y nos mostraría un aviso de que la acción **Close** se canceló.



Esto nos impediría cerrar el formulario.

Para salir del bucle en el que nos metería, podríamos pulsar en el botón diseño del formulario, abrir la ventana de su módulo de clase y eliminar, poner a **False** o dejar comentada la línea

```
' Cancel = True
```

Si nos fijamos en la lista de eventos generados, nos puede sorprender que no se ha generado ni el evento **GotFocus** ni el **LostFocus**.

¿Quiere decir que el formulario, como tal en ningún momento recibe el foco, y por tanto tampoco lo pierde?

Puede sorprender la respuesta, pero en este caso es así.

¿Por qué?

Porque el que recibe el foco es el botón como único control capaz de recibir el foco.

El formulario, al contrario que un Botón de comando, no posee la propiedad **TabStop** (Punto de tabulación) por lo que el botón tiene prioridad a la hora de recibir el foco.

Las secciones del formulario tampoco tienen esta propiedad. El que sí la tiene es un formulario insertado como Subformulario.

¿Qué pasa si en el botón le ponemos la propiedad **Punto de tabulación** al valor no?

En el editor del formulario seleccionamos el botón y ponemos su propiedad a no.

Abrimos y cerramos el formulario; el resultado mostrado por la ventana inmediato es:

- 1 Open
- 2 Load
- 3 Resize
- 4 Activate
- 5 GotFocus
- 6 Current
- 7 Click del botón
- 8 GotFocus
- 9 Unload
- 10 LostFocus
- 11 Deactivate
- 12 Close

Como el botón tiene desactivada la propiedad Punto de tabulación, y no hay ningún otro control que la tenga activada, es el propio formulario el que recibe el foco.

Incluso no se termina de perder totalmente el foco, ya que cuando presionamos el botón, no se genera el evento **LostFocus**, aunque sí se vuelve a generar inmediatamente después el evento **GotFocus**.

Si no nos damos cuenta de lo dicho en los puntos anteriores, podríamos tener la sorpresa de que no se ejecutara el código que diseñáramos para el evento **GotFocus** o **LostFocus** del formulario.

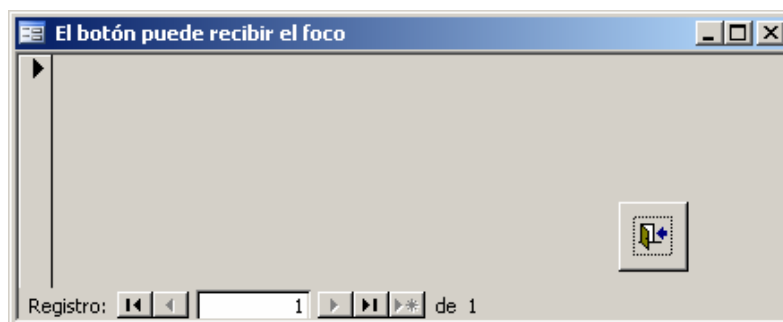
Para dejar el formulario como estaba ponemos volvemos a restituir el valor de la propiedad **TabStop** del botón a **True**, pero esta vez lo vamos a hacer mediante código.

Para ello aprovecharemos el gestor del primer evento que se produce en el formulario, en concreto el evento **Open**.

Su código quedará así:

```
Private Sub Form_Open(Cancel As Integer)
    MuestraOrden "Open"
    cmdCerrar.TabStop = True
    Caption = " El botón puede recibir el foco"
End Sub
```

Abrimos el formulario y lo volvemos a cerrar.



Si lo observamos vemos ahora, que en vez del anodino título **Formulario1: Formulario** aparece el mensaje **El botón puede recibir el foco**.

Si miramos lo que ha escrito en la ventana Inmediato, veremos que formulario ya no recibe el foco y se generan los 9 eventos iniciales, en vez de los 12 que se generaban con la propiedad **TabStop** puesta a false en el botón.

La propiedad **Caption** del formulario, de lectura y escritura, devuelve ó establece el texto que se muestra en la barra de título del formulario.

Esta propiedad controla el texto que aparece en objetos como formularios, botones de comando, páginas de objetos **TabControl** o controles ficha, etiquetas, botones de alternar e informes.

Como vemos, podemos cambiar, de una forma sencilla, las propiedades de un formulario y sus controles, en tiempo de ejecución.

Esto nos abre inmensas posibilidades para el diseño de nuestras aplicaciones.

Crear propiedades y métodos públicos en el módulo de clase del formulario.

Hemos dicho que el código asociado a un formulario es su código de clase, por lo tanto podemos escribir nuestras propias propiedades y métodos en él.

Vamos a crear la propiedad **Numero** que contendrá un dato numérico que asociaremos al formulario.

Para ello crearemos un nuevo formulario al que llamaremos **FormularioNumerado**.

Le pondremos un botón de cierre, como en el formulario anterior, con su mismo nombre y gestor del evento Al hacer Clic.

Como inciso comentaré que la clase del formulario que acabamos de crear tiene por nombre **Form_FormularioNumerado**

En el código del módulo de la clase le pondremos una variable privada de tipo numérico y crearemos la propiedad **Numero**.

El código de este nuevo formulario será:

```
Option Compare Database
Option Explicit

Dim intNumero As Integer

Public Property Get Numero() As Integer
    Numero = intNumero
End Property

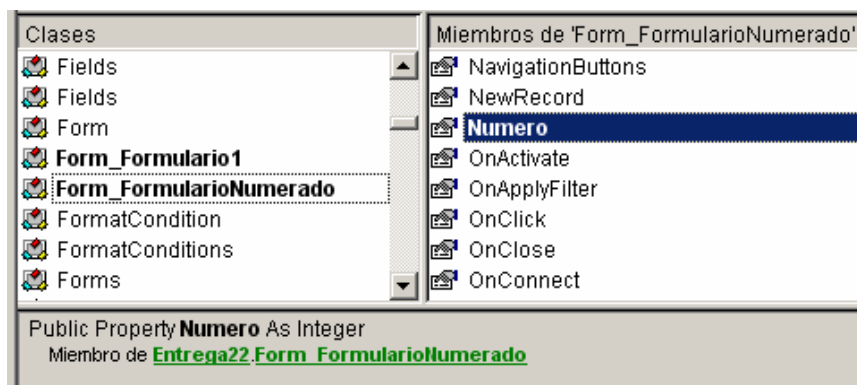
Public Property Let Numero(ByVal NuevoNumero As Integer)
    intNumero = NuevoNumero
    Caption = "Formulario N° " &
        & Format(.Numero, "000")
End Property
```

```

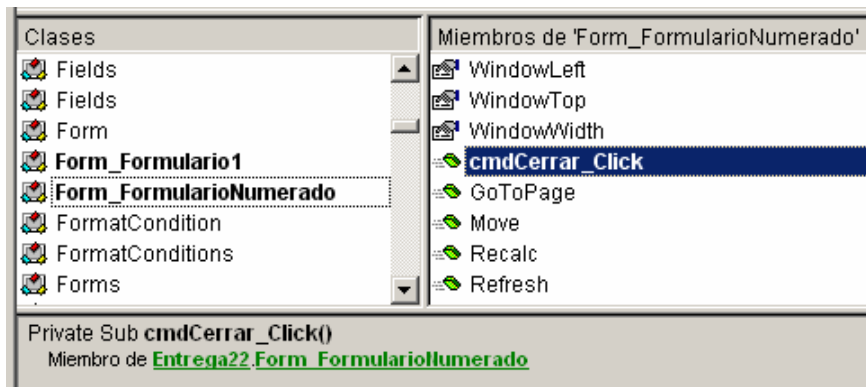
Private Sub cmdCerrar_Click()
On Error GoTo HayError
    DoCmd.Close
Salir:
    Exit Sub
HayError:
    MsgBox Err.Description
    Resume Salir
End Sub

```

Si abrimos el Examinador de objetos, vemos que en la clase del formulario ha aparecido la propiedad **Numero**.



Igualmente podríamos comprobar que existe el procedimiento privado **cmdCerrar_Click**



Instanciar un formulario

Existen varias formas de instanciar un formulario, o lo que es lo mismo, asignar un formulario concreto a una variable.

Por cierto, una variable que vaya a hacer referencia a un formulario debe ser del tipo **Variant**, **Object** o **Form**.

Como vimos en capítulos anteriores, el tipo **Variant** es el más genérico de todos, admitiendo casi cualquier cosa. El tipo **object** admite prácticamente cualquier tipo de objeto, pero al contrario que el **Variant**, no puede admitir valores que no sean objetos.

La vinculación en tiempo de ejecución, usando variables del tipo **Variant** u **Object** genera una serie de inconvenientes, como un control más impreciso de los posibles errores, un código menos eficiente y la falta de ayuda "en línea" al escribir el código.

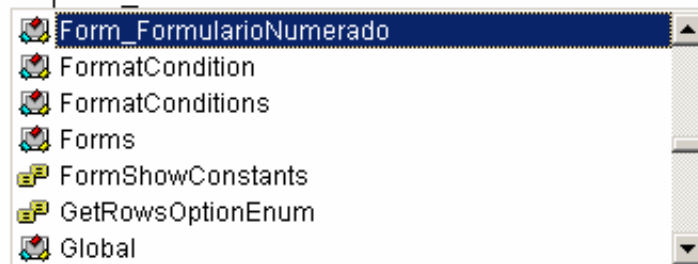
Por ello, cuando a una variable hay que asignarle un objeto concreto, es mejor declararla como del tipo de ese objeto; en nuestro caso del tipo **Form**; y mejor aún como Ya hemos dicho que nuestro formulario es un objeto del tipo **Form_FormularioNumerado**, podríamos declarar una variable de su tipo, con lo que para activarlo bastaría con asignarlo a la variable con **Set** y **New**.

Para comprobarlo, vamos a crear un módulo estándar y en él crearemos una variable del tipo **Form_FormularioNumerado**,

Fijémonos que el asistente en línea, nos lo muestra como una de las posibilidades, acompañándolo con el icono que define a las clases.

```
Option Compare Database
Option Explicit
```

```
Dim MiFormulario As Form_FormularioNumerado
```



Vamos ahora a crear un procedimiento que presente una instancia de ese formulario y le asigne algunas propiedades

```
Option Compare Database
Option Explicit
```

```
Public MiFormulario As Form_FormularioNumerado
```

```
Public Sub CargaFormulario()
```

```
    ' Creamos la instancia del formulario
```

```
    Set MiFormulario = New Form_FormularioNumerado
```

```
    With MiFormulario
```

```
        .Numero = 1
```

```
        .Caption = "Formulario N° " & _
            & Format(.Numero, "000")
```

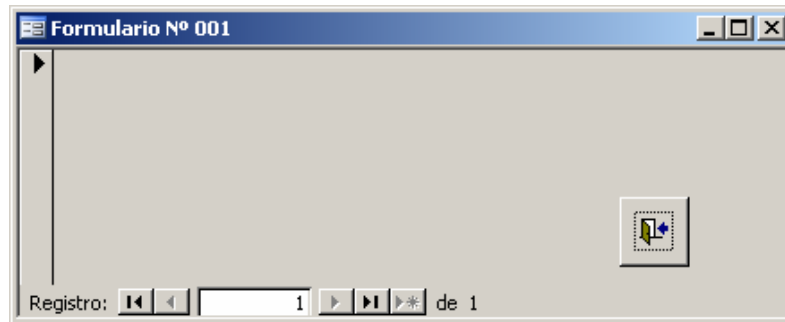
```
    End With
```

```
End Sub
```

Ejecutamos el procedimiento y aparentemente no pasa nada.

En realidad sí ha pasado. Lo que ocurre es que el formulario está cargado pero no está visible.

Si a continuación del ejecutar el procedimiento **CargaFormulario** ejecutamos en la ventana inmediato la línea **MiFormulario.Visible=True**, el formulario se nos aparecerá.

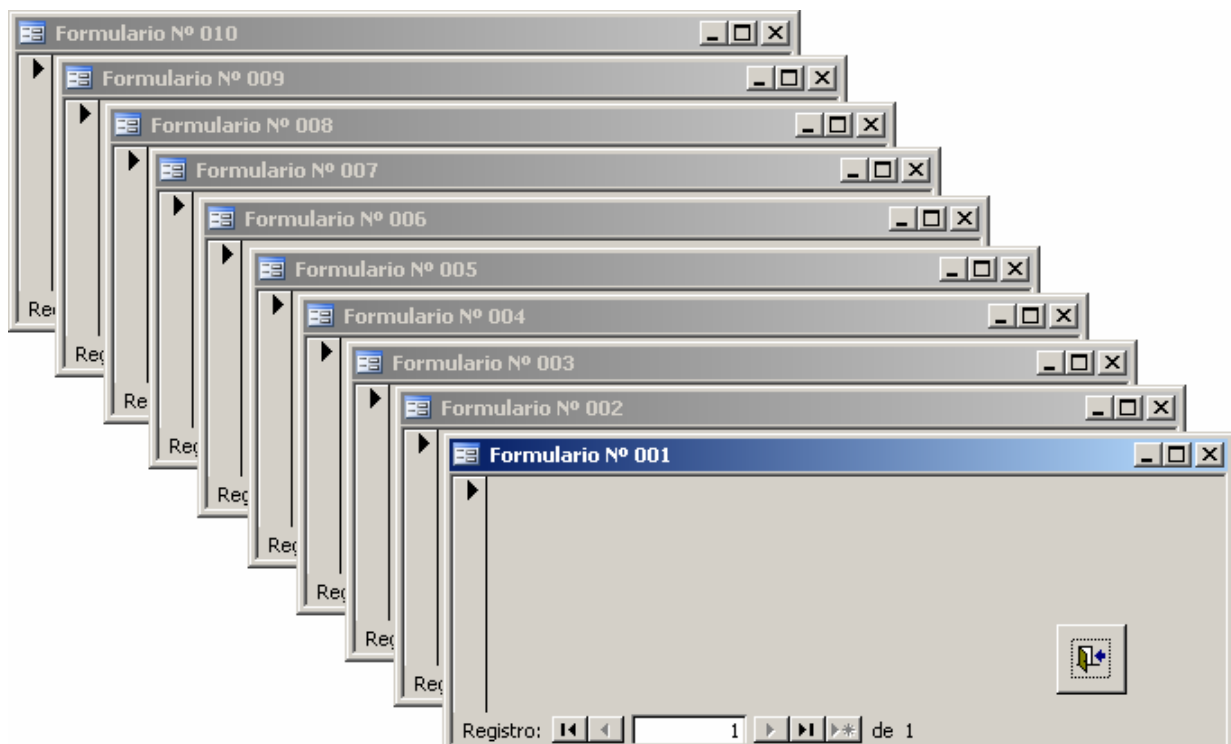


Múltiples instancias de un formulario

También, como con el resto de las clases, podríamos crear Instancias múltiples de un formulario. Por ejemplo, si tuviéramos el formulario Clientes, podríamos crear 5 instancias, cada una de ellas mostrándonos los datos de un cliente diferente.

Vamos a ver cómo podríamos hacerlo con nuestro formulario. En un módulo ponemos

```
Public aFormularios(1 To 10) As Form
Public Sub FormulariosMultiples()
    Dim i As Integer
    For i = 1 To 10
        Set aFormularios(i) = New Form_FormularioNumerado
        With aFormularios(i)
            .Caption = " Formulario N° " & Format(i, "000")
            .Visible = True
        End With
    Next i
End Sub
```

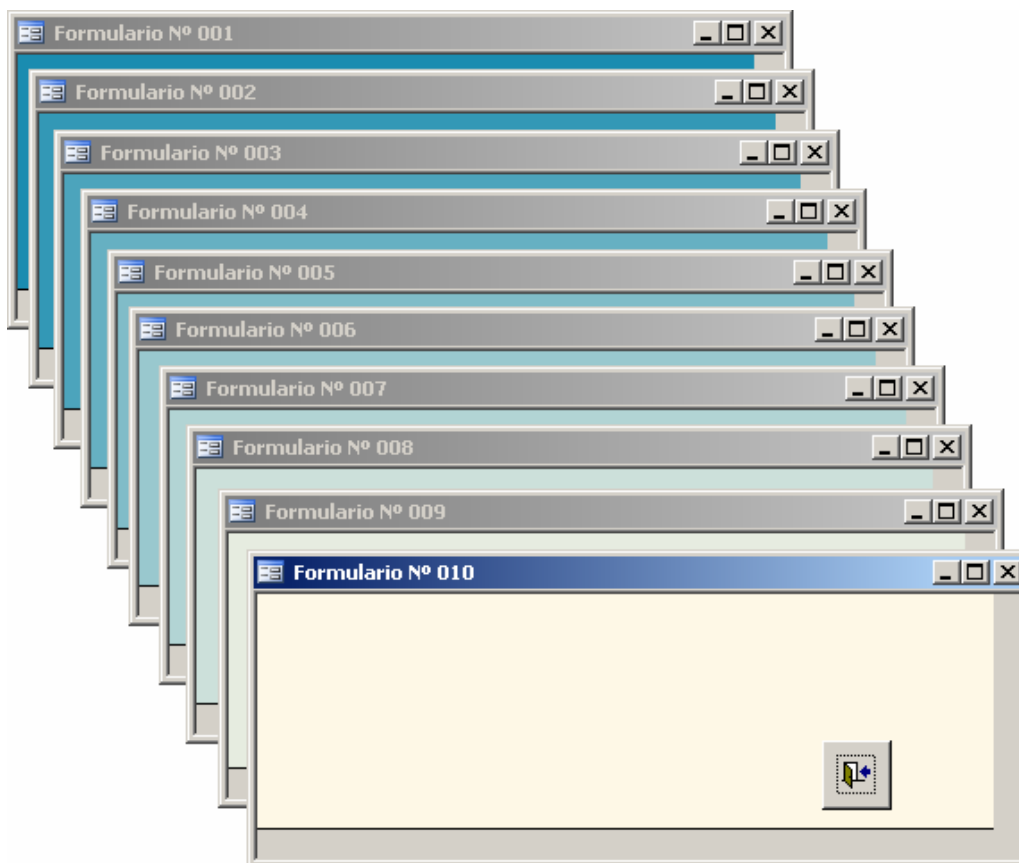


En el caso anterior hemos creado diez formularias, cada uno de ellos con una barra de título diferente.

En realidad los formularios se muestran todos en la misma posición, pero los he desplazado individualmente para que aparezcan tal como aparecen en la imagen.

Con esto hemos visto que tenemos libertad para cambiar las propiedades individuales de cada formulario.

En concreto casi cualquier propiedad del formulario que sea de escritura, por ejemplo:



Las variaciones de color en la sección Detalle se ha realizado de la siguiente forma:

```
Public aFormularios(1 To 10) As Form

Public Sub FormulariosMultiples()
    Dim i As Integer
    Dim lngColor As Long

    For i = 1 To 10
        Set aFormularios(i) = New Form_FormularioNumerado
        With aFormularios(i)
            ' Pongo el título del formulario
            .Caption = " Formulario Nº " & Format(i, "000")
            ' Color de la sección Detalle
            lngColor = RGB(25.5 * i, 128 + 12 * i, 170 + 6 * i)
        End With
    Next i
End Sub
```



```

.Detalle.BackColor = lngColor
' Elimino el selector de registros
.RecordSelectors = False
' Elimino los botones de navegación
.NavigationButtons = False
' Hago visible el formulario
.Visible = True

End With
Next i

```

End Sub

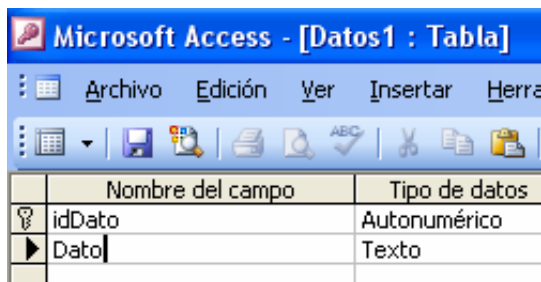
Conexión con datos en un formulario

Un formulario puede existir sin estar conectado a ningún tipo de datos.

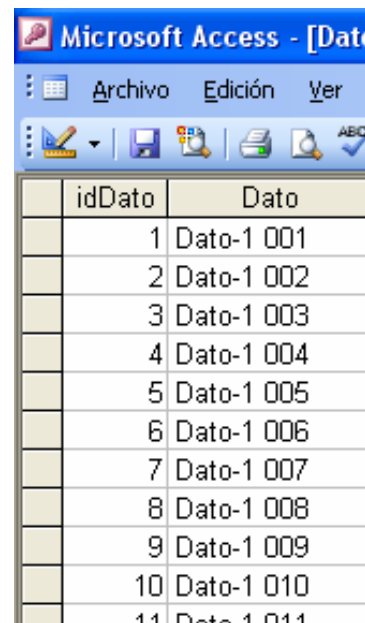
Pero a su vez podemos establecer la conexión de un formulario con un origen de datos por código en tiempo de ejecución.

E incluso, como veremos cuando estudiemos la biblioteca de **ADO**, podríamos enlazarlo a un conjunto de datos (**Recordset**) que exista únicamente en memoria.

Para comprobarlo, vamos a crear la tabla **Datos1** con los siguientes campos



Nombre del campo	Tipo de datos
idDato	Autonumérico
Dato	Texto

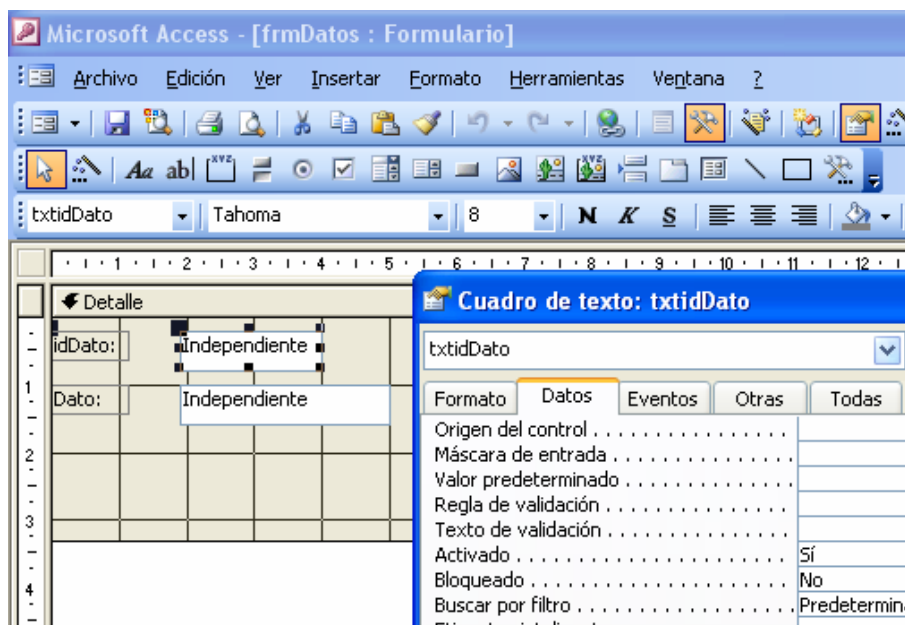


idDato	Dato
1	Dato-1 001
2	Dato-1 002
3	Dato-1 003
4	Dato-1 004
5	Dato-1 005
6	Dato-1 006
7	Dato-1 007
8	Dato-1 008
9	Dato-1 009
10	Dato-1 010
11	Dato-1 011

Para ver el efecto Creamos una serie de registros

A continuación creamos un formulario, de nombre **frmDatos**, con dos cuadros de texto. Sus nombres serán **txtidDato** y **txtdato**.

Al cuadro de texto que contendrá el campo **idDato** (autonumérico) le cambiaremos su propiedad **Activado (Enabled)** y **Bloqueado (Locked)**, para que no se pueda acceder, desde el formulario al campo **idDato**, que es **autonumérico**



Pero esto lo haremos por código en el evento **Al cargar (OnLoad)** del formulario.

```
Me.txtidDato.Enabled = False
```

```
Me.txtidDato.Locked = True
```

Para asignar una tabla, o consulta, a un formulario, utilizaremos la propiedad **Origen del registro (RecordSource)**.

Esto lo podemos hacer también en el evento Al cargar.

A la propiedad le asignaremos una cadena que puede contener,

El nombre de una tabla

El nombre de una consulta guardada

Una cadena SQL.

En nuestro caso serían igualmente válidas las siguientes opciones:

```
Me.RecordSource = "Datos"
```

```
Me.RecordSource = "Select * From Datos1;"
```

```
Me.RecordSource = "Select idDato, Dato From Datos1;"
```

```
Me.RecordSource = "Select Datos1.idDato, Datos1.Dato From Datos1;"
```

```
Me.RecordSource = "Select [idDato], [Dato] From Datos1;"
```

Respecto a la palabra **Me**, que estamos utilizando podemos recordar que la vimos cuando analizábamos las clases.

Igual que entonces, **Me** representa el objeto creado mediante la clase, es decir, representa al propio formulario.

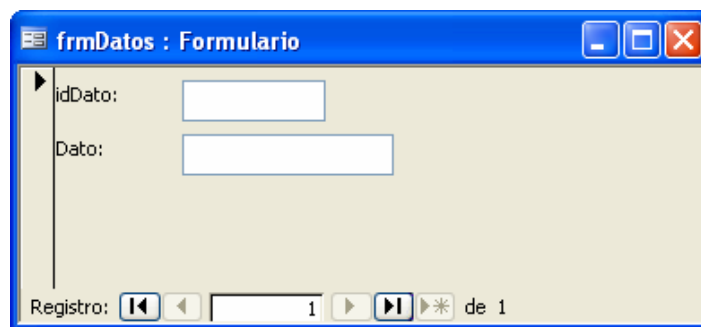
Por eso, si escribimos **Me** y a continuación el punto, nos aparece la ayuda en línea que nos suministra el editor de Visual Basic.

No es estrictamente necesario utilizarla.

Las siguientes instrucciones serían tan válidas, como los utilizadas en las líneas anteriores.

```
txtidDato.Enabled = False
txtidDato.Locked = True
RecordSource = "Select idDato, Dato From Datos1;"
-----
-----
```

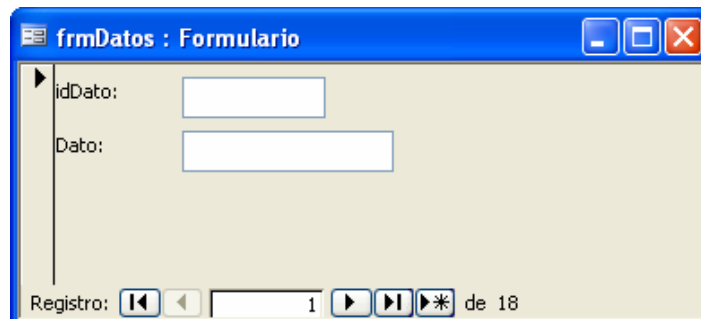
Si abrimos el formulario, sin asignarle un origen de datos, tendrá un aspecto semejante a éste:



Veamos cómo cambia al asignarle un origen de datos.

En el evento, **Al cargar** del formulario, escribimos lo siguiente:

```
Private Sub Form_Load()
    Dim strSQLDatos As String
    txtidDato.Enabled = False
    txtidDato.Locked = True
    strSQLDatos = "Select idDato, Dato From Datos1;"
    RecordSource = strSQLDatos
End Sub
```



Vemos que ahora nos indica que tenemos un determinado número de registros, por lo que podemos suponer que efectivamente está conectado a la tabla Datos.

Pero todavía no vemos nada en los cuadros de texto.

La conexión entre los cuadros de texto y los correspondientes campos de la tabla, debe efectuarse después de que hayamos conectado la tabla al formulario.

En un control, el campo al que se conecta, lo determina la propiedad **Origen del control (ControlSource)**.

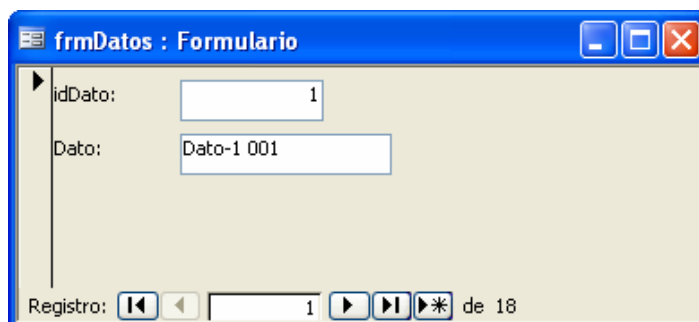
Su sintaxis es así

```
NombreDelControl.ControlSource = NombreDelCampo
```

Para ello modificaremos el código del evento **Al cargar** de la siguiente manera:

```
Private Sub Form_Load()  
    Dim strSQLDatos As String  
    txtidDato.Enabled = False  
    txtidDato.Locked = True  
    strSQLDatos = "Select idDato, Dato From Datos;"  
    RecordSource = strSQLDatos  
    txtidDato.ControlSource = "idDato"  
    txtDato.ControlSource = "Dato"  
End Sub
```

Con lo que el formulario se verá de una forma semejante a esta:



Podemos comprobar que ahora sí tenemos acceso a los datos.

Cambio, en tiempo de ejecución del origen de datos.

Vamos a crear ahora una segunda tabla a la que vamos a llamar **Datos2**.

Para simplificar haremos que esta nueva tabla contenga los mismos campos que la tabla **Datos**.

Cambiaremos el contenido de la columna datos para que podamos apreciar la diferencia.

Yo le he puesto valores del tipo

Dato-2 001

Dato-2 002

Dato-2 003

Para comprobar el proceso, copiamos el formulario anterior y le ponemos como nombre **frmDatosCambiados**.

Al copiar el formulario, se copiará con sus propiedades y código asociado.

A continuación, a este nuevo formulario le añadiremos dos botones que serán los que efectúen el “cambiao” al hacer **Clic** sobre ellos.

A estos botón le pondremos por nombre **cmdTabla1** y **cmdTabla2**.

Si tenemos activado el botón del [**Asistente para Controles**] (la varita mágica que hemos visto en un punto anterior, lo desactivamos para tener un control completo del proceso.

En cada uno de los eventos **Al hacer clic** , escribimos respectivamente lo siguiente.

```
Private Sub cmdTabla1_Click()  
    RecordSource = "Select idDato, Dato From Datos1;"  
End Sub
```

```
Private Sub cmdTabla2_Click()  
    RecordSource = "Select idDato, Dato From Datos2;"  
End Sub
```

Sólo con esto podemos comprobar que cambiamos el origen de datos para el formulario simplemente modificando la cadena de SQL que define el conjunto de datos con el que queremos trabajar, utilizando simplemente la propiedad **RecordSource**.

Más adelante veremos la gran utilidad que nos brinda esta propiedad.

Cambio, en tiempo de ejecución del origen de datos, con nombre de campos diferentes.

En los puntos anteriores, hemos conectado un formulario a dos tablas diferentes, con sólo pulsar un botón.

El problema era muy sencillo de resolver, ya que los campos tienen el mismo nombre en las dos tabla.

Pero ¿cómo se resuelve el caso en el que los nombres de los campos de las tablas no sean iguales?

Volveremos a usar la propiedad **ControlSource** del control Cuadro de texto (**TextBox**).

Vamos a comprobar todo lo dicho

Creamos la tabla Provincias, con los campos

idProvincia y **Provincia**.

Copiamos otra vez el formulario y le ponemos como nombre **frmContTresOrigenes**.

Además le añadimos un nuevo botón de nombre **cmdProvincias**.

Vamos a hacer que cuando se pulse el botón **cmdProvincias** se muestren las provincias, y cuando se pulsen los otros botones, se muestren sus respectivas tablas.

Además vamos a hacer que la barra del título del formulario muestre el nombre de la tabla conectada usando la propiedad **Caption** del formulario.

Vamos a cambiar también el nombre de las etiquetas asociadas a los cuadros de texto, poniéndoles como nombres respectivos, **lblidDato** y **lblDato**.

El prefijo **lbl** nos ayuda a saber que ese nombre se corresponde al de una etiqueta (Label).

Para conseguir nuestro objetivo, añadimos código para gestionar el evento Clic del nuevo botón, y modificamos el código anterior.

El código completo quedará así:

```
Option Compare Database
Option Explicit

Private Sub Form_Load()
    Dim strSQLDatos As String
    txtidDato.Enabled = False
    txtidDato.Locked = True
    cmdTabla1_Click
End Sub

Private Sub cmdTabla1_Click()
    AjustaCamposTablas
    Caption = "Tabla Datos1"
    RecordSource = "Select idDato, Dato From Datos1;"
End Sub

Private Sub cmdTabla2_Click()
    AjustaCamposTablas
    Caption = "Tabla Datos2"
    RecordSource = "Select idDato, Dato From Datos2;"
End Sub

Private Sub cmdProvincias_Click()
    RecordSource = "Provincias"
    Caption = "Tabla Provincias"
    txtidDato.ControlSource = "idProvincia"
    txtDato.ControlSource = "Provincia"
    txtidDato.Format = "00"
    If lblidDato.Caption <> "Clave:" Then
        lblidDato.Caption = "Clave:"
        lblidDato.Caption = "Provincia:"
    End If
End Sub

Private Sub AjustaCamposTablas()
    txtidDato.ControlSource = "idDato"
    txtDato.ControlSource = "Dato"
    txtidDato.Format = "#,##0"
    If lblidDato.Caption <> "idDato:" Then
        lblidDato.Caption = "idDato:"
    End If
End Sub
```

```
        lblidDato.Caption = "Dato:"  
    End If  
End Sub
```

El código es lo suficientemente sencillo como para que el lector, si ha seguido estas entregas, lo pueda entender sin dificultad.

Sólo una matización:

Podemos ver que desde el evento Load, se llama directamente al gestor del evento Clic del botón `cmdTabla1` mediante la instrucción:

```
cmdTabla1_Click
```

Esto es así porque un gestor de evento no deja de ser un procedimiento `Sub` más, y por tanto se puede llamar de forma directa.

El formulario mostrará la información que seleccionemos mediante los botones:

Tabla Datos1

idDato:

Dato:

Registro: de 18

Tabla Datos2

idDato:

Dato:

Registro: de 19

Tabla Provincias

Provincia:

Dato:

Registro: de 52

Ejercicio.

Aprovechando lo visto en el punto **Formularios múltiples**, le sugiero al lector que cree varias instancias del mismo formulario, cada una de ellas con un origen distinto de datos.

Próxima entrega.

En la próxima entrega jugaremos con algunas de las propiedades de los formularios, y digo jugaremos, ya que diseñaremos un formulario panel para jugar al **Bingo** o **Lotería**, en casa, que incluso nombre el número de las bolas conforme vayan saliendo.

Cuando vemos los informes, completaremos el programa de Bingo con la posibilidad de imprimir los cartones.

Comencemos a programar con
VBA - Access

Entrega **23**

Objetos de Access
Formularios (2)

Propiedades de los formularios

Cuando estudiamos las clases vimos que en una clase podemos definir

- **Propiedades**
- **Métodos**
- **Eventos.**

Una propiedad es una característica de un objeto de una clase determinada que podemos leer, escribir o ambas cosas.

Los métodos son procedimientos definidos en una clase que pueden devolver valores, como en el caso de las funciones, o sólo ejecutar una serie de instrucciones de código.

Los métodos pueden recibir parámetros, tanto por valor como por referencia.

Pueden devolver datos simples, como cadenas, valores numéricos, booleanos, etc... u objetos.

Lo dicho en el punto anterior es perfectamente aplicable a las propiedades.

Las propiedades de los formularios a las que podemos acceder, o establecer, afectan:

- Al diseño y aspecto del propio formulario, como su tamaño, posición, color...
- A los orígenes de datos a los que puede estar conectado, como vimos en la entrega anterior
- Otras características, como la forma en que se imprimirá en una impresora láser

Por la extensión del tema no podremos dar un repaso a todas las propiedades de los formularios, pero sí hablaremos de las más interesantes.

Aconsejo al lector que acuda a la ayuda de Access, donde podrá encontrar la información completa de cada una de las propiedades, incluyendo líneas de código.

Propiedades de Formato

Para hablar de las propiedades trataré de ajustarme a cómo están estructuradas en la ventana de propiedades de la versión de Access 2003.

Propiedad Título (Caption)

En un formulario controla el texto que aparecerá en la barra de título de un formulario.

Es una propiedad de tipo String y es de lectura – escritura.

Esta propiedad ya la usamos en el ejemplo del capítulo anterior

```
Caption = "Tabla Provincias"
```

Recordemos que la línea anterior es equivalente a:

```
Me.Caption = "Tabla Provincias"
```

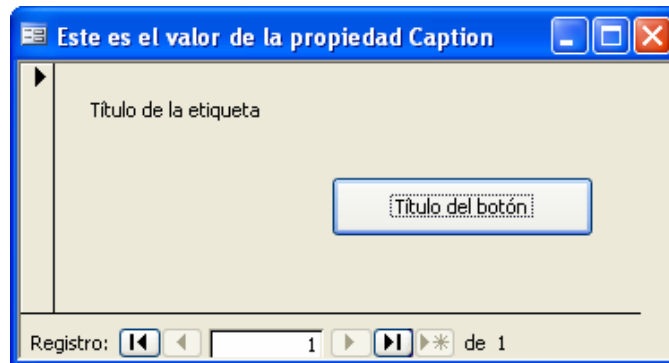
Como veremos más adelante hay otra serie de objetos que también poseen esta propiedad, como las etiquetas, informes, botones de comando, etc...

Vamos a poner en un formulario nuevo una etiqueta de nombre **lblTitulo** y un botón de comando de nombre **cmdTitulo**.

En el evento **Al cargar** del formulario escribimos:

```
Private Sub Form_Load()
    Caption = "Este es el valor de la propiedad Caption"
    lblTitulo.Caption = "Título de la etiqueta"
    cmdTitulo.Caption = "Título del botón"
End Sub
```

Al abrir el formulario veremos algo como esto:



Propiedad Presentación Predeterminada (DefaultView)

Especifica qué modo de presentación tendrá el formulario en el momento en que se abra.

Sus posibles valores son

Modo de Presentación	Valor	Descripción
Formulario simple	0	Valor Predeterminado. Muestra un registro cada vez.
Formularios continuos	1	Muestra múltiples registros, cada uno en su propia copia de la sección de detalle del formulario.
Hoja de datos	2	Muestra los datos del formulario organizados en filas y columnas como una hoja de cálculo correspondiéndose las filas a los registros y las columnas a los campos.
PivotTable	3	* Muestra el formulario como una tabla dinámica.
PivotChart	4	** Muestra el formulario como un gráfico dinámico.

(*) (**) Más adelante hablaremos sobre qué son las tablas y gráficos dinámicos.

De momento saber que es un sistema parametrizable para la presentación sintetizada de los datos agrupándolos por determinados valores.

Estas propiedades sólo pueden establecerse por código cuando el formulario está en modo diseño. Si tratamos de hacerlo en tiempo de ejecución nos dará el error N° 2136.

Puede que más de uno se sorprenda, pero en un formulario por código no sólo podemos modificar sus características, sino incluso añadirle controles.

Algunos de estos procesos sólo pueden realizarse si ponemos el formulario en modo diseño.

Más adelante veremos todo esto con más detalle.

Propiedad AllowFormView

Controla si se permite mostrar el formulario en el modo de presentación **Formulario simple**.

Propiedad AllowDatasheetView

Controla si se permite el modo de presentación **Hoja de datos**

.Estas dos últimas propiedades son de tipo **Booleano**.

Propiedad Barras de Desplazamiento (ScrollBars)

Mediante esta propiedad podemos especificar la aparición o no, tanto de la barra de desplazamiento vertical como de la horizontal.

Es aplicable a formularios y cuadros de texto, teniendo en cuenta que los cuadros de texto sólo pueden tener barras verticales

Sus posibles valores son

Barras	Valor	Descripción
Ninguna	0	No aparecen barras de desplazamiento. Para los cuadros de texto es la opción predeterminada
Sólo horizontal	1	Barra de desplazamiento horizontal en el formulario
Sólo vertical	2	Aparece una barra de desplazamiento vertical.
Ambas	3	Es el valor predeterminado para los formularios. En el formulario aparecen barras de desplazamiento horizontal y vertical. No se aplica a cuadros de texto.

Propiedad Selectores de registro (RecordSelectors)

Permite Mostrar u ocultar la barra vertical selectora de registros.

Vamos a poner en un formulario un botón de comando con el nombre **cmdSelectores** y comprobemos el resultado cada vez que lo presionamos.

```
Private Sub cmdSelectores_Click()
    RecordSelectors = Not RecordSelectors
    If RecordSelectors Then
        cmdSelectores.Caption = "Ocultar selector de registro"
    Else
        cmdSelectores.Caption = "Mostrar selector de registro"
    End If
End Sub
```

Propiedad Botones de desplazamiento (NavigationButtons)

Permite Mostrar u ocultar los botones de desplazamiento entre registros y las etiquetas que indican su número.

Ponemos un botón con el nombre `cmdBotonesDeDesplazamiento` y comprobemos el resultado cada vez que lo presionamos.

```
Private Sub cmdBotonesDeDesplazamiento_Click()  
    NavigationButtons = Not NavigationButtons  
    If NavigationButtons Then  
        cmdBotonesDeDesplazamiento.Caption = _  
            "Ocultar Botones de Desplazamiento"  
    Else  
        cmdBotonesDeDesplazamiento.Caption = _  
            "Mostrar Botones de Desplazamiento"  
    End If  
End Sub
```

Otras Propiedades de formato de tipo Boolean que pueden establecerse en tiempo de ejecución.

- **Separadores de registros (DividingLines)** especifica si habrá separadores de registros separando las secciones de un formulario o los registros mostrados en un formulario continuo.
- **Formulario movable (Moveable)** establece si el formulario se podrá o no desplazar por la pantalla.
- **Propiedad Activado (Enabled)** esta propiedad define si un objeto está activado y por tanto podrá responder a eventos.
- **Propiedad Bloqueado (Locked)** esta propiedad especifica si se pueden editar los datos de un control en la vista Formulario. Es muy útil para mostrar datos en un cuadro de texto, que no queremos que el usuario pueda cambiar.

Si queremos que se visualice en un formulario un campo, por ejemplo de tipo auto numérico, que ni queremos ni necesitamos que el usuario pueda editarlo, es habitual poner, para ese control, la propiedad **Enabled** a **False**, y la propiedad **Locked** a **True**.

Con eso podemos mantener la estética del formulario.

Yo personalmente suelo además poner el fondo como transparente.

Este es un ejemplo de formato para un TextBox, en el evento **Al cargar** del formulario.

```
Private Sub Form_Load()  
    With txtidDato  
        .Enabled = False      ' Sin activar
```

```

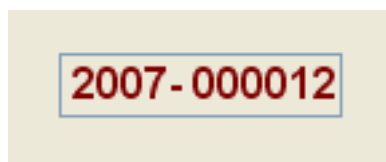
.Locked = True           ' Bloqueado
.BackStyle = 0          ' Fondo transparente
.BorderStyle = 1       ' Borde fino
.ForeColor = 128       ' Color Burdeos del texto
.FontName = "Arial"    ' Fuente de tipo Arial
.FontSize = 12         ' Tamaño de fuente 12 puntos
.FontBold = True       ' Fuente en negrita

End With

```

```
End Sub
```

Así se vería el control:



Como hemos visto en el ejemplo, en cualquier momento podemos cambiar la forma como se muestra un control.

Animo al lector a que haga el siguiente ejemplo

Un formulario con una etiqueta y dos botones.

Cuando se pulsa uno de los botones aumenta el tamaño del texto.

Cuando se pulse el otro botón disminuiría su tamaño..

Propiedad Imagen (Picture)

Mediante esta propiedad podemos establecer la imagen que queremos mostrar como fondo del formulario, de un botón, informe, etc...

Admite distintos tipos de ficheros gráficos

Su sintaxis es

```
Objeto.Picture = "RutaYNombreDelFicheroGráfico "
```

```
Me.Picture = "c:\Imágenes\MiFoto.jpg"
```

```
cmdPaletas.Picture = "c:\Imágenes\Paleta.gif"
```

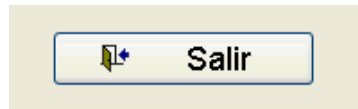
```
Picture = "c:\Imágenes\Escudo.bmp"
```

Nota:

Los Botones de Access no admiten mostrar simultáneamente un texto y una imagen.

Esto no es un problema insoluble, ya que podemos crearnos un fichero gráfico que incluya el texto, con cualquier herramienta de diseño.

Por ejemplo, aquí tenemos un botón con el gráfico típico de salida y el texto.



Podríamos incluso hacer cambiar el gráfico cuando se pulse el botón, y cuando se vuelve a soltar.

Para ello podemos aprovechar los eventos

- **Al bajar el Mouse (MouseDown)** que se produce al presionar un botón del ratón sobre el botón
- **Al subir el Mouse (MouseUp)** se produce al levantar el botón del ratón sobre el botón después de haberlo presionado

```
Private Sub cmdMiBoton_MouseDown( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
    cmdMiBoton.Picture = "C:\Pograma\BotonPresionado.bmp"
End Sub
```

```
Private Sub cmdMiBoton_MouseUp( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
    cmdMiBoton.Picture = "C:\Pograma\BotonNormal.bmp"
End Sub
```

Eventos del ratón MouseDown y MouseUp

En el controlador de estos eventos, podemos averiguar

- Mediante el parámetro **Button**, qué botón hemos presionado pudiendo tomar estos valores:

acLeftButton	→	1	Botón izquierdo del ratón
acRightButton	→	2	Botón derecho
acMiddleButton	→	4	Botón central

Podemos controlar también si hemos presionado, de forma simultánea alguna de estas teclas

- **Mayúscula** → 1
- **Control.** → 2
- **Alt** → 4

Y cualquier combinación de ellas (es la suma de los valores anteriores)

Este código nos mostrará en un cuadro de mensaje si se ha presionado alguna de las teclas anteriores mientras se pulsaba un botón del ratón.

```
Private Sub cmdMiBoton_MouseDown( _  
    Button As Integer, _  
    Shift As Integer, _  
    X As Single, Y As Single)  
    Select Case Shift  
        Case 0  
            MsgBox "Sólo el ratón"  
        Case 1  
            MsgBox "Tecla Mayúscula"  
        Case 2  
            MsgBox "Tecla Control"  
        Case 3  
            MsgBox "Teclas Mayúscula y Control"  
        Case 4  
            MsgBox "Tecla Alt"  
        Case 5  
            MsgBox "Teclas Mayúscula y Alt"  
        Case 6  
            MsgBox "Teclas Control y Alt"  
        Case 7  
            MsgBox "Teclas Mayúscula Control y Alt"  
    End Select  
End Sub
```

Estas combinaciones nos permiten definir diferentes comportamientos de un control al pulsarlo con el ratón dependiendo de si se ha pulsado simultáneamente una tecla **Mayúscula Control** o **Alt**, o incluso combinaciones de ellas.

Los parámetros **X** e **Y** contienen las coordenadas del ratón, en **Twips** respecto a la esquina superior izquierda del botón.

Más adelante veremos qué clase de unidad es un **Twip**.

El parámetro no es específico del evento **MouseDown**.

Por ejemplo, un cuadro de texto posee el evento **KeyDown**, que también posee el parámetro **Shift**, al que se accedería de forma similar a lo visto con **MouseDown**.

```
Private Sub txtDato_KeyDown( _  
    KeyCode As Integer, _
```


`Shift As Integer)`

End Sub

Gestión de colores en un control

Propiedad Estilo del fondo (BackStyle)

Controla si el fondo de un control es opaco ó transparente.

0	Transparente	El control completo o parte de él serán transparentes
1	Opaco	El fondo del control tendrá el color definido por la propiedad Color de fondo (BackColor)

Propiedad Color de fondo (BackColor)

La propiedad que gobierna el color de fondo de la mayoría de los objetos es la propiedad **BackColor**.

Esta propiedad admite valores de tipo Long, en el rango:

del RGB(0,0,0) → 0 (Negro)

al valor RGB(255,255,255) → 16.777.215 → (Blanco)

Otra forma de acceder a los colores es mediante la función **QBColor (Índice)**, que admite para el parámetro Índice, valores del 0 al 15

<u>Índice</u>	<u>Color</u>	<u>Índice</u>	<u>Color</u>
0	Negro	8	Gris
1	Azul	9	Azul claro
2	Verde	10	Verde claro
3	Aguamarina	11	Aguamarina claro
4	Rojo	12	Rojo claro
5	Fucsia	13	Fucsia claro
6	Amarillo	14	Amarillo claro
7	Blanco	15	Blanco brillante

Los veteranos que hayáis programado con aquellos “entrañables” lenguajes como:

GW Basic, Q Basic, T Basic, etc... identificaréis enseguida estos parámetros.

Y hay una tercera, que es la utilización de las constantes definidas en el módulo **ColorConstants**

Estas son:

vbBlack	→	Negro	→	0
vbBlue	→	Azul	→	16.711.680
vbCyan	→	Cyan (Azul primario)	→	16.776.960
vbGreen	→	Verde	→	65.280
vbMagenta	→	Magenta (Rojo primario)	→	16.711.935
vbRed	→	Rojo	→	255
vbWhite	→	Blanco	→	16.777.215
vbYellow	→	Amarillo	→	65.535

Propiedad Color del texto (**ForeColor**)

Esta propiedad define el color que va a mostrar el texto de un control.

Hay otras propiedades que definen aspectos del color en un objeto

BorderColor

Como su propio nombre indica, define el color de los bordes de un objeto.

Fijémonos en el siguiente código:

```
Private Sub Form_Load()  
    With txtidDato  
        .Enabled = False  
        .Locked = True  
        .BackStyle = 0  
        .BackColor = vbYellow  
        .BorderStyle = 1  
        .ForeColor = vbBlue  
        .FontName = "Arial"  
        .FontSize = 12  
        .FontBold = True  
        .BorderWidth = 2  
        .BorderColor = vbRed  
    End With  
End Sub
```

Define un cuadro de texto (**TextBox**) en el que lo que escribamos se verá en color azul, tendrá un borde rojo y el fondo del mismo será de color amarillo.

Propiedad Color de los bordes (BorderColor)

Define el color que van a mostrar los bordes de un control.

El grosor de los bordes lo define la propiedad **Ancho de los bordes (BorderWidth)**

Otras propiedades para el formato de objetos**Propiedad Estilo de los Bordes (BorderStyle)**

Es una propiedad que funciona de forma diferente, si se aplica a un formulario o se aplica a un control.

En un **formulario** puede tomar los siguientes valores

0	Ningún borde	No aparece ningún borde ni ningún elemento relacionado con los mismos. No se puede cambiar el tamaño del formulario.
1	Bordes delgados	El formulario tiene un borde fino y puede mostrar los elementos relacionados. No se puede cambiar su tamaño.
2	Bordes ajustables	Es el valor predeterminado de Access. Puede mostrar los elementos relacionados con el borde, y ajustar su tamaño con el cursor.
3	Cuadro de Diálogo	El borde del formulario se pone de tamaño doble. Nos muestra una barra de título, un botón cerrar y un menú control. El formulario no puede ni maximizarse, ni minimizarse ni cambiar de tamaño. Además puede mostrarse como Modal (por encima de todas las demás ventanas) poniendo sus propiedades Emergente (PopUp) y Modal al valor True .

En un **control** sus valores pueden ser

0	Transparente	Predeterminado sólo para etiquetas, gráficos y sub-informes
1	Sólido	Línea sólida (Predeterminado)
2	Guiones	Línea de guiones
3	Guiones cortos	Línea de guiones cortos
4	Puntos	Línea de puntos
5	Puntos separados	Línea de puntos separados
6	Guión punto	Línea con una combinación de punto y raya
7	Guión-punto-punto	Línea con una combinación de raya punto y punto
8	Sólido doble	Líneas dobles

Propiedad ancho de los bordes (BorderWidth)

Especifica el ancho del borde de un control.

Es de tipo Byte. Valores admitidos:

0	Trazo fino	Predeterminado sólo para etiquetas, gráficos y sub-informes
1 a 6	De 1 a 6 ptos.	Ancho en puntos

En la ayuda de Access puede encontrar información completa de cómo hacer efectiva esta propiedad.

Ejercicio:

Vamos a crear un formulario en el que vamos a colocar tres barras que funcionarán al estilo de las **Barras de Progreso (ProgressBar)**.

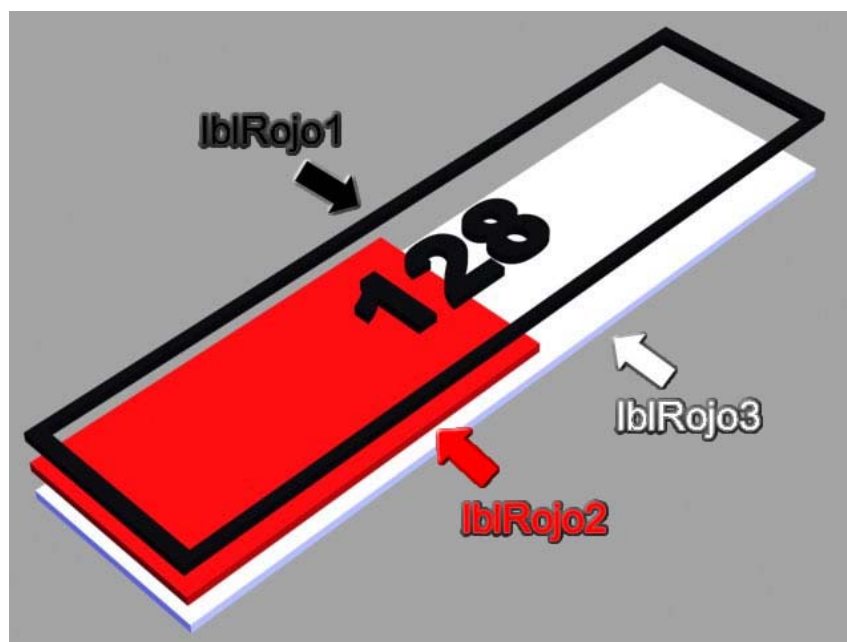
Haremos que al desplazar las barras, varíe la proporción de Rojo, Verde y Azul del fondo del formulario, con lo que irá variando su color.

Hay varios aspectos interesantes de este ejercicio.

Uno de ellos es que no vamos a utilizar ningún control **ActiveX**, como los que vienen en la librería **MsComCtlLib**, para realizar las barras de desplazamiento.

Éstas las vamos a crear nosotros.

Fijémonos en el siguiente gráfico:



Para imitar la barra utilizaremos 3 etiquetas colocadas una encima de la otra.

La inferior **lblColor3** tendrá el fondo de color blanco.

La intermedia **lblColor2** será de color Rojo, Verde ó Azul, e irá cambiando su anchura cuando pulsemos con el cursor en la etiqueta superior y lo desplazemos.

La superior **lblColor1**, a diferencia de las otras dos, será transparente, además de poseer un borde negro y mostrar un valor numérico, del **0** al **255**, también en negro.

“El viaje más largo empieza con el primer paso” (proverbio chino)

Para empezar con el proyecto, haremos nuestras primeras pruebas.

En un formulario colocamos una etiqueta con el texto Rojo, ocupando casi todo el ancho del mismo.

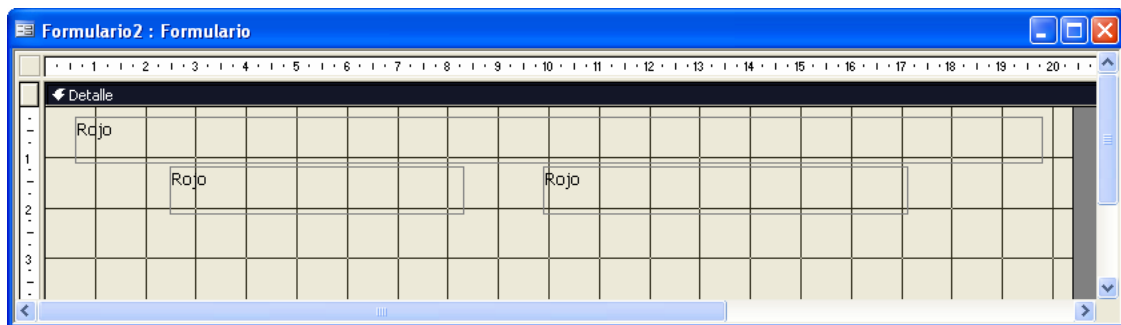
Le ponemos como nombre **IbIRojo1**

Seleccionamos la etiqueta, la copiamos y la pegamos dos veces.

A estas dos nuevas etiquetas les ponemos como nombres **IbIRojo2** y **IbIRojo3**.

Estas dos últimas las hacemos más pequeñas y las ponemos cerca de la primera, pero por debajo de ella.

Quedarán con un aspecto similar a éste.



Ahora seleccionamos la etiqueta **IbIRojo1** y pulsamos en la opción **Traer al frente** del menú **Formato**.

A continuación seleccionamos la etiqueta **IbIRojo3** y pulsamos en la opción **Traer al frente** del menú **Formato**.

Con esto ya hemos conseguido que su orden se adapte al del gráfico anterior.

El siguiente paso es adaptar el tamaño de las etiquetas.

La inferior será igual a la superior, y la del medio tendrá una anchura mitad.

Además las colocaremos una encima de la otra.

Propiedad Izquierda (Left)

Especifica la distancia de la parte izquierda de un objeto a la parte izquierda del objeto que lo contiene.

Podemos especificar, por ejemplo la distancia de una etiqueta respecto del borde izquierdo de un formulario o de un informe. Es de Lectura y Escritura.

Existe otra propiedad semejante:

Propiedad Superior (Top)

Especifica la distancia de la parte superior de un objeto al borde superior del objeto que lo contiene.

Ambas son de tipo **Long** para los informes y de tipo Integer para los formularios.

Propiedades Alto y Ancho (Height) y (Width)

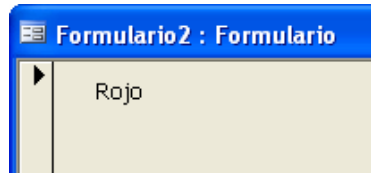
Height especifica la altura de un objeto

Width especifica su anchura

Vamos a utilizar estas propiedades para colocar y dimensionar correctamente las etiquetas.

En el evento **Al cargar** del formulario escribimos:

Abrimos el formulario, y no parece que suceda nada espectacular, ya que sólo nos muestra la palabra Rojo.



Pero esa es precisamente la demostración de que hemos conseguido una parte de lo que queríamos y es que las tres etiquetas están colocadas exactamente una encima de la otra.

Ahora vamos a mostrar las etiquetas, para lo que modificaremos el código.

En vez de ponerlo todo en el evento Al Cargar haremos que desde este evento llame al procedimiento **FormateaEtiquetas** que es el que hará todo el trabajo:

Si queremos mantener un "histórico" de los pasos que vamos a dar, guardamos este primer formulario con el nombre, **frmEtiquetas01**.

Y esta nueva versión mejorada, la grabamos con el nombre **frmEtiquetas02**.

```
Private Sub Form_Load()
    FormateaEtiquetas
End Sub
```

```
Private Sub FormateaEtiquetas()
    With lblRojo1
        .BorderStyle = 1           ' Borde de línea sólida
        .BorderWidth = 3          ' Anchura de 3 puntos
        .BackStyle = 0            ' Fondo transparente
        .TextAlign = 2            ' Texto centrado
        .Caption = 128            ' Texto inicial de la etiqueta
        .FontSize = 18            ' Tamaño de la fuente
        .FontBold = True          ' Texto en negrita
    End With
```

```
    With lblRojo2
        .Height = lblRojo1.Height
        .Width = lblRojo1.Width / 2
        .Caption = ""
        .BackStyle = 1           ' Fondo opaco
        .BackColor = vbRed       ' Color rojo para el fondo
        ' Salvo la anchura, sus propiedades geométricas _
        ' las ponemos como las de lblRojo1
        .Width = lblRojo1.Width / 2
```

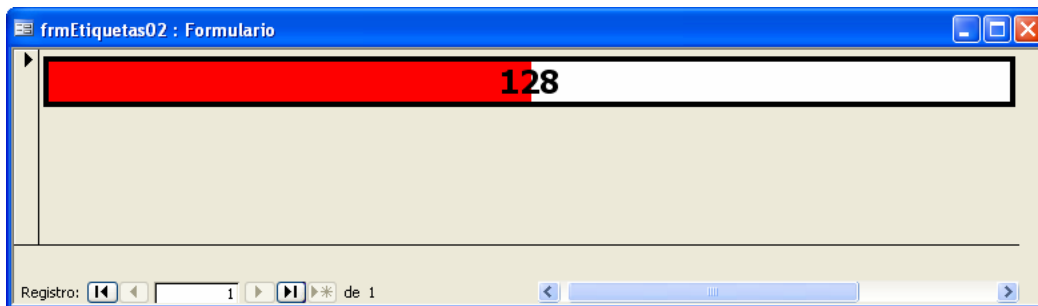
```

.Height = lblRojo1.Height
.Left = lblRojo1.Left
.Top = lblRojo1.Top
End With

With lblRojo3
.Height = lblRojo1.Height
.Width = lblRojo1.Width
.Caption = ""
.BackStyle = 1           ' Fondo opaco
.BackColor = vbWhite    ' Color blanco
' Igualamos sus propiedades geométricas _
  con las de lblRojo1
.Width = lblRojo1.Width
.Height = lblRojo1.Height
.Left = lblRojo1.Left
.Top = lblRojo1.Top
End With
End Sub

```

El resultado de este código es éste:



En el código aparece la propiedad **TextAlign**

Propiedad Alineación del texto (TextAlign)

Con esta propiedad podemos definir la colocación del texto que aparece en un control

0	General	Texto a la izquierda, números y fechas a la derecha
1	Izquierda	Todos los datos a la izquierda
2	Centro	Centrado
3	Derecha	Alineación a la derecha
4	Distribuir	El contenido se reparte por la anchura del control. Ojo: No es lo mismo que la Justificación del texto de Word.

El valor pasado a la propiedad **TextAlign** ha sido 2, con lo que vemos el valor 128 centrado en la etiqueta.

Antes de continuar vamos a dar formato al formulario.

Primero ponemos la propiedad Estilo de los bordes al modo Cuadro de diálogo, desde su ventana Propiedades.

A continuación, por código haremos las siguientes operaciones:

- Quitar la barra de desplazamiento que aparece en horizontal
- Quitar el selector de registros
- Quitar los botones de desplazamiento
- Quitar el separador de registros

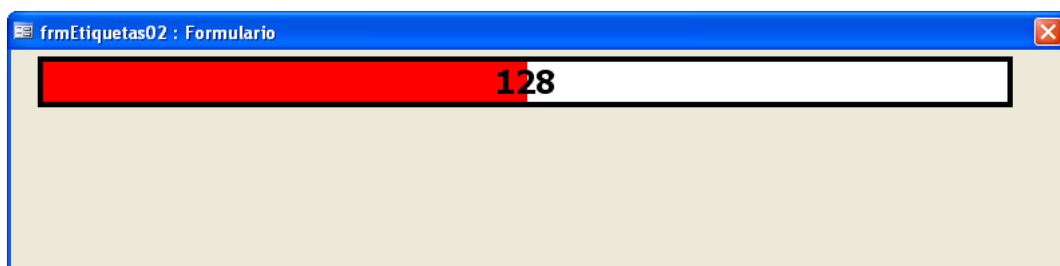
Para ello crearemos el procedimiento **FormateaFormulario**, que será llamado desde el evento Al Cargar.

Para conseguirlo el código quedará así:

```
Private Sub Form_Load()  
    FormateaEtiquetas  
    FormateaFormulario  
End Sub
```

```
Private Sub FormateaFormulario()  
    ScrollBars = 0           ' Sin barras de desplazamiento  
    RecordSelectors = False ' Sin selector de registros  
    NavigationButtons = False ' Sin Botones desplazamiento  
    DividingLines = False  ' Sin Separador de registros  
    BorderStyle = 3        ' Estilo cuadro de diálogo  
End Sub
```

Con estas modificaciones el formulario quedará así:



Vemos que el aspecto del formulario ha quedado mucho más limpio.

Tengo que reconocer que todas estas operaciones podrían haberse realizado sin escribir una sola línea de código, mediante la vista propiedades del formulario, pero al hacerlo de esta forma he pretendido dos cosas:

- Mostrar cómo se puede hacer por código
- Demostrar el alto grado de control que podemos tener sobre el diseño del mismo

El paso siguiente es conseguir que la barra roja se desplace cuando presionemos sobre las etiquetas y desplazemos el cursor sobre ellas.

El principal evento que vamos a utilizar es el **MouseMove**.

Vamos a analizarlo con más detenimiento:

Análisis de eventos

Antes de seguir vamos a analizar los gestores de los eventos que necesitaremos para conseguir nuestro objetivo.

Evento MouseMove

Este evento se produce cuando el usuario desplaza el ratón por encima de un objeto que lo genere.

La sintaxis de su gestor es:

```
Private Sub NombreDelobjeto_MouseMove( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, _
    Y As Single)
```

Si nos fijamos, los parámetros son los mismos que los del evento que hemos analizado en uno de los puntos anteriores.

Por el momento, el parámetro que nos interesa es el **X**, que define la coordenada horizontal de la punta del ratón, respecto al borde izquierdo del control.

Lo interesante de esta coordenada es que, al estar en el mismo tipo de unidades que la propiedad `Width` de las etiquetas, podemos tomar su valor para definir la anchura de la etiqueta roja **lblRojo02**.

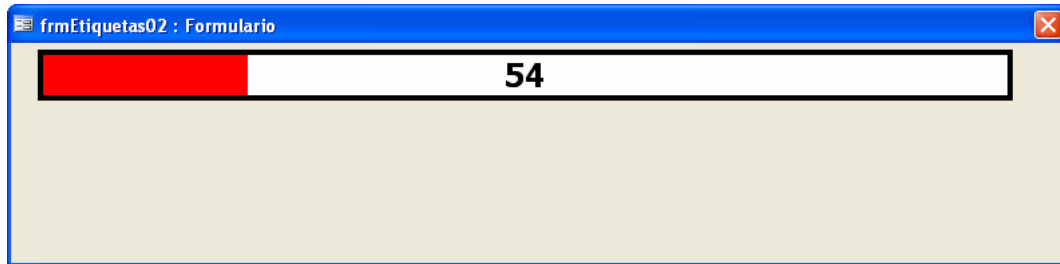
Añadimos este código para controlar el evento **MouseMove** de la etiqueta **lblRojo02**.

```
Private Sub lblRojo01_MouseMove( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, _
    Y As Single)

    Dim bytRojo As Byte
    Dim lngAnchoEtiquetas As Long
    lngAnchoEtiquetas = lblRojo01.Width
    If X <= lblRojo01.Width And X >= 0 Then
        lblRojo02.Width = X
        bytRojo = Round(255 * X / lngAnchoEtiquetas)
        lblRojo01.Caption = bytRojo
    End If
End Sub
```

Abrimos el formulario, y vemos que ya hemos conseguido la parte más importante del objetivo, que el conjunto simule una barra de progreso, que aumente o disminuya cuando

pasemos el ratón por encima, y que el valor presentado en la etiqueta superior vaya de 0 a 255, en función de la posición del ratón.



Pero aquí faltan todavía cosas.

Hemos quedado que la barra debe moverse cuando movamos el ratón por encima, sólo si tenemos presionada la tecla izquierda del mismo.

En cambio se desplaza aunque no tengamos presionada la tecla izquierda.

Esto significa que en el gestor del evento, antes de cambiar el tamaño de la etiqueta, o que la etiqueta superior muestre el valor numérico, se debe comprobar si la tecla izquierda está presionada.

La comprobación la efectuaremos leyendo la variable Boleana **blnBotonBajado**, definida a nivel del módulo de clase del formulario.

Esta variable se pondrá a **True** cuando presionemos el botón izquierdo del ratón, y se pondrá a **False**, cuando levantemos el botón.

Para ello nos aprovecharemos del evento **MouseDown** de la etiqueta **lblRojo1**, y del evento **MouseUp** de la misma.

Option Explicit

```
Private blnBotonBajado As Boolean
```

```
- - - - -  
- - - - -
```

```
Private Sub lblRojo1_MouseDown( _  
    Button As Integer, _  
    Shift As Integer, _  
    X As Single, Y As Single)  
    blnBotonBajado = True  
End Sub
```

```
Private Sub lblRojo1_MouseUp( _  
    Button As Integer, _  
    Shift As Integer, _  
    X As Single, Y As Single)
```

```

        blnBotonBajado = False
    End Sub

```

Con esto ya hemos conseguido que las etiquetas funcionen como deseábamos.

¿Del todo?

Si nos fijamos en su funcionamiento, vemos que si presionamos el botón sobre la etiqueta, sin desplazarlo por encima de la misma, la barra roja no cambia hasta que movamos el botón.

Podemos evitar esta “pequeña deficiencia”, llamando directamente al gestor del evento **MouseMove** desde el gestor del evento **MouseDown**.

Modificaremos el evento de la siguiente forma:

```

Private Sub lblRojo1_MouseDown( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
    ' Comprobamos que el botón pulsado es el izquierdo
    If Button = acLeftButton Then
        blnBotonBajado = True
        ' Llamamos al gestor del evento MouseMove _
        de lblRojo1
        lblRojo1_MouseMove Button, Shift, X, Y
    End If
End Sub

```

Un asunto interesante de esto es comprobar que podemos llamar al gestor de un evento cualquiera como si fuera un procedimiento normal.

Por supuesto le deberemos pasar los parámetros adecuados, si es que los posee.

En este caso mediante la línea

```

lblRojo1_MouseMove Button, Shift, X, Y

```

le pasamos los mismos parámetros que recibe del evento **MouseDown** de **lblRojo1**

Antes de seguir con nuestro “pequeño proyecto” haremos unas pequeñas modificaciones, como poner la declaración de las variables **bytRojo** y **lngAnchoEtiquetas** a nivel del módulo, en vez de en el procedimiento `lblRojo1_MouseMove`.

Con esto el código del módulo de clase del formulario quedará:

Código intermedio del proyecto

```

Option Compare Database
Option Explicit

Private blnBotonBajado As Boolean
Dim bytRojo As Byte
Dim lngAnchoEtiquetas As Long

```

```
Private Sub Form_Load()  
    FormateaEtiquetas  
    FormateaFormulario  
End Sub  
  
Private Sub lblRojo1_MouseDown( _  
    Button As Integer, _  
    Shift As Integer, _  
    X As Single, Y As Single)  
    ' Comprobamos que el botón pulsado es el izquierdo  
    If Button = acLeftButton Then  
        blnBotonBajado = True  
        ' Llamamos al gestor del evento MouseMove _  
        de lblRojo1  
        lblRojo1_MouseMove Button, Shift, X, Y  
    End If  
End Sub  
  
Private Sub lblRojo1_MouseUp( _  
    Button As Integer, _  
    Shift As Integer, _  
    X As Single, Y As Single)  
    blnBotonBajado = False  
End Sub  
  
Private Sub lblRojo1_MouseMove( _  
    Button As Integer, _  
    Shift As Integer, _  
    X As Single, Y As Single)  
    lngAnchoEtiquetas = lblRojo1.Width  
    If blnBotonBajado Then  
        If X <= lblRojo1.Width And X >= 0 Then  
            lblRojo2.Width = X  
            bytRojo = Round(255*X / lngAnchoEtiquetas)  
            lblRojo1.Caption = bytRojo  
        End If  
    End If  
End Sub
```

```
Private Sub FormateaFormulario()  
    ' Sin barras de desplazamiento  
    ScrollBars = 0  
    'Sin selector de registros  
    RecordSelectors = False  
    ' Sin Botones de desplazamiento  
    NavigationButtons = False  
    ' Sin Separador de registros  
    DividingLines = False  
End Sub  
  
Private Sub FormateaEtiquetas()  
    With lblRojo1  
        .BorderStyle = 1           ' Borde de línea sólida  
        .BorderWidth = 3           ' Anchura de 3 puntos  
        .BackStyle = 0             ' Fondo transparente  
        .TextAlign = 2             ' Texto centrado  
        .Caption = 128             ' Texto inicial _  
                                   de la etiqueta  
        .FontSize = 18            ' Tamaño de la fuente  
        .FontBold = True          ' Texto en negrita  
    End With  
  
    With lblRojo2  
        .Height = lblRojo1.Height  
        .Width = lblRojo1.Width / 2  
        .Caption = ""  
        .BackStyle = 1           ' Fondo opaco  
        .BackColor = vbRed       ' Color rojo para el fondo  
        ' Salvo la anchura, ponemos _  
        ' sus propiedades geométricas _  
        ' como las de lblRojo1  
        .Width = lblRojo1.Width / 2  
        .Height = lblRojo1.Height  
        .Left = lblRojo1.Left  
        .Top = lblRojo1.Top  
    End With  
  
    With lblRojo3  
        .Height = lblRojo1.Height
```

```
.Width = lblRojo1.Width
.Caption = ""
.BackStyle = 1           ' Fondo opaco
.BackColor = vbWhite    ' Color blanco
' Igualamos sus propiedades geométricas _
  con las de lblRojo1
.Width = lblRojo1.Width
.Height = lblRojo1.Height
.Left = lblRojo1.Left
.Top = lblRojo1.Top
End With
End Sub
```

Ya hemos conseguido una parte importante de nuestro proyecto.

Ahora guardamos el formulario **frmEtiquetas02** y para la versión siguiente lo guardamos como **frmEtiquetas03**.

Creación de una barra de progreso para los colores básicos Verde y Azul.

Vamos a poner seis etiquetas nuevas

```
lblVerde1   lblVerde2   lblVerde3
lblAzul1    lblAzul2    lblAzul3
```

A continuación modificamos el evento Load de la siguiente manera:

```
Private Sub Form_Load()
    FormateaEtiquetas "Rojo"
    FormateaEtiquetas "Verde"
    FormateaEtiquetas "Azul"
    FormateaFormulario
End Sub
```

Al procedimiento `FormateaEtiquetas` le estamos pasando un parámetro que antes no tenía.

Para que no nos de error, deberemos por tanto modificar la cabecera y el diseño del mismo.

Un inciso previo:

¿Cuál es el objetivo de todo esto?

Si utilizara el mismo sistema que en el código inicial, sería preciso repetir el procedimiento tres veces, una para cada color de etiquetas.

Nos podemos ahorrar todo este montón de código, si somos capaces de diseñar en el procedimiento un sistema que permita manejar las diferentes etiquetas de forma dinámica.

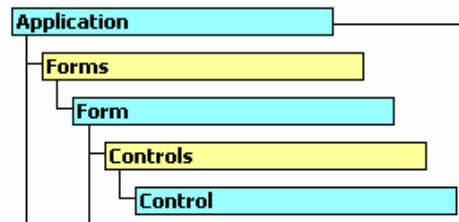
Asignación dinámica de controles desde la colección Controls.

Cuando analizamos las colecciones, vimos que había dos formas de recuperar un elemento de las mismas.

La primera era utilizar el índice del elemento contenido. `NombreColección (Índice)`

La segunda era utilizando una clave única de tipo **string** creada en el momento en que se añadía un elemento a la colección.

Con los controles de un formulario sucede algo semejante.



Además podemos realizar un recorrido por los mismos mediante la estructura **For Each...**

Vemos que el formulario tiene la colección **Controls**.

Esta colección comienza con el índice cero.

Acceso mediante **For Each ...**

En un módulo estándar vamos a escribir el siguiente código:

```
Public Sub ListaControles1(Formulario As Form)
    Dim ctl As Control
    For Each ctl In Formulario
        Debug.Print ctl.Name
    Next ctl
End Sub
```

Si llamamos a este procedimiento mediante

```
ListaControles1 Form_frmEtiquetas03
```

en mi caso el procedimiento escribe en la ventana inmediato lo siguiente:

```
lblRojo3
lblRojo2
lblRojo1
lblVerde3
lblVerde2
lblVerde1
lblAzul3
lblAzul2
lblAzul1
```

Acceso mediante un índice: Controls (Índice)

En el mismo módulo que en el procedimiento anterior escribimos:

```
Public Sub ListaControles2(Formulario As Form)
    Dim i As Integer
    Dim ctl As Control
    For i = 0 To Formulario.Controls.Count - 1
        Set ctl = Formulario.Controls(i)
        Debug.Print ctl.Name
    Next i
End Sub
```

Si llamamos al procedimiento mediante

```
ListaControles2 Form_frmEtiquetas03
```

El resultado es el mismo que en el procedimiento **ListaControles1**.

Acceso mediante el nombre del control: Controls (Nombre)

Se puede acceder a un control mediante su nombre.

Por ejemplo, este procedimiento muestra el nombre y la anchura de cada control accediendo a él mediante su nombre:

```
Public Sub ListaControles3(Formulario As Form)
    Dim ctl As Control
    Dim ctlNombre As Control
    Dim strNombre As String
    For Each ctl In Formulario
        strNombre = ctl.Name
        Set ctlNombre = Formulario.Controls(strNombre)
        Debug.Print ctlNombre.Name, ctlNombre.Width
    Next ctl
End Sub
```

Si ejecutamos **ListaControles3 Form_frmEtiquetas03** me muestra:

lblRojo3	4091
lblRojo2	3296
lblRojo1	10886
lblVerde3	4091
lblVerde2	3296
lblVerde1	7196
lblAzul3	4091
lblAzul2	3296
lblAzul1	7376

Vemos que efectivamente estamos asignando a la variable `ctlNombre`, de tipo `Control`, cada uno de los controles obtenidos desde la colección `Controls` a la que se le pasa el parámetro del nombre del control.

¿Cómo afecta esto a nuestro proyecto?

Sencillamente, si le pasamos al procedimiento `FormateaEtiquetas` una cadena que nos permita construir el nombre de la etiqueta, podremos ajustar el formato de cada etiqueta sin tener que repetir todo el código del mismo.

Podría ser esto:

```
Private Sub FormateaEtiquetas (ByVal NombreColor As String)
    Const SEPARACION As Long = 500
    Dim lngTop As Long
    Dim lngColor As Long
    Dim lbl As Label
    Dim strNombreEtiqueta As String

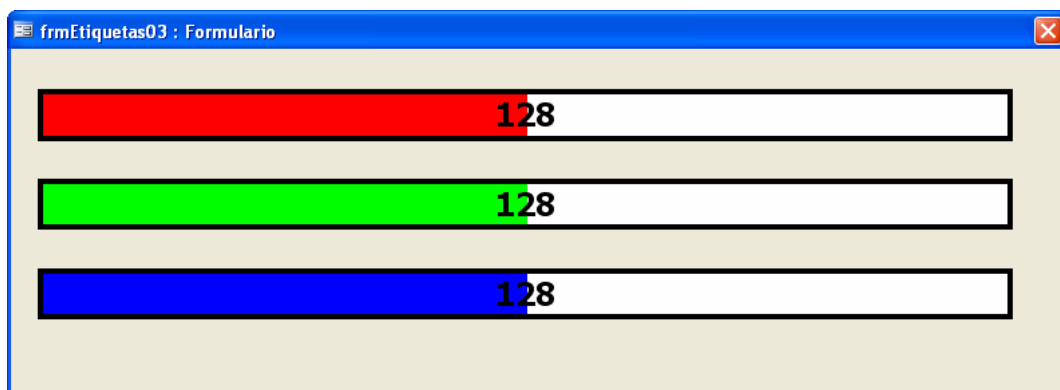
    Select Case NombreColor
    Case "Rojo"
        lngTop = SEPARACION
        lngColor = vbRed
    Case "Verde"
        lngTop = 3 * SEPARACION
        lngColor = vbGreen
    Case "Azul"
        lngTop = 5 * SEPARACION
        lngColor = vbBlue
    End Select

    strNombreEtiqueta = "lbl" & NombreColor & "1"
    Set lbl = Controls(strNombreEtiqueta)
    With lbl
        .Top = lngTop
        .Width = lblRojol.Width
        .BorderStyle = 1           ' Borde de línea sólida
        .BorderWidth = 3          ' Anchura de 3 puntos
        .BackStyle = 0            ' Fondo transparente
        .TextAlign = 2            ' Texto centrado
        .Caption = 128            ' Texto inicial de la etiqueta
        .FontSize = 18            ' Tamaño de la fuente
        .FontBold = True          ' Texto en negrita
    End With
End Sub
```

```
strNombreEtiqueta = "lbl" & NombreColor & "2"
Set lbl = Controls(strNombreEtiqueta)
With lbl
    .Top = lngTop
    .Caption = ""
    .BackStyle = 1          ' Fondo opaco
    .BackColor = lngColor  ' Color para el fondo
    ' Salvo la anchura, sus propiedades geométricas _
    ' como las de lblRojo1
    .Width = lblRojo1.Width / 2
    .Height = lblRojo1.Height
    .Left = lblRojo1.Left
End With

strNombreEtiqueta = "lbl" & NombreColor & "3"
Set lbl = Controls(strNombreEtiqueta)
With lbl
    .Caption = ""
    .BackStyle = 1          ' Fondo opaco
    .BackColor = vbWhite   ' Color blanco
    ' Igualamos sus propiedades geométricas _
    ' con las de lblRojo1
    .Width = lblRojo1.Width
    .Height = lblRojo1.Height
    .Left = lblRojo1.Left
    .Top = lngTop
End With
End Sub
```

Este es el resultado final:



Lógicamente sólo será operativa la barra roja, ya que no hemos definido los eventos para el resto de las barras.

Esta tarea es ya muy sencilla; prácticamente “copiar y pegar”.

Aunque no del todo.

Para controlar si tenemos pulsada la tecla sobre cada barra, crearemos tres variables booleanas y dos variable de tipo **Byte** adicionales.

```
Option Compare Database
Option Explicit

Private blnBotonRojoBajado As Boolean
Private blnBotonVerdeBajado As Boolean
Private blnBotonAzulBajado As Boolean
Dim bytRojo As Byte
Dim bytVerde As Byte
Dim bytAzul As Byte
Dim lngAnchoEtiquetas As Long
```

Cambiamos la variable en el código anterior, y añadimos los eventos para las otras barras

Este sería el código para las barras Verdes

```
Private Sub lblVerde1_MouseDown( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
    ' Comprobamos que el botón pulsado es el izquierdo
    If Button = acLeftButton Then
        blnBotonRojoBajado = True
        ' Llamamos al gestor del evento _
        MouseMove de lblRojol
        lblVerde1_MouseMove Button, Shift, X, Y
    End If
End Sub

Private Sub lblVerde1_MouseUp( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
    blnBotonRojoBajado = False
End Sub

Private Sub lblVerde1_MouseMove( _
    Button As Integer, _
```

```

        Shift As Integer, _
        X As Single, _
        Y As Single)
    If blnBotonRojoBajado Then
        If X <= lblRojo1.Width And X >= 0 Then
            lblVerde2.Width = X
            bytVerde = Round(255 * X / lngAnchoEtiquetas)
            lblVerde1.Caption = bytVerde
            ColoreaFormulario
        End If
    End If
End Sub

```

Fíjese el lector que al final del procedimiento `lblVerde1_MouseMove` he añadido una llamada al procedimiento `ColoreaFormulario`.

Este procedimiento será el que se encargue de colorear el formulario.

Su código es tan sencillo como este:

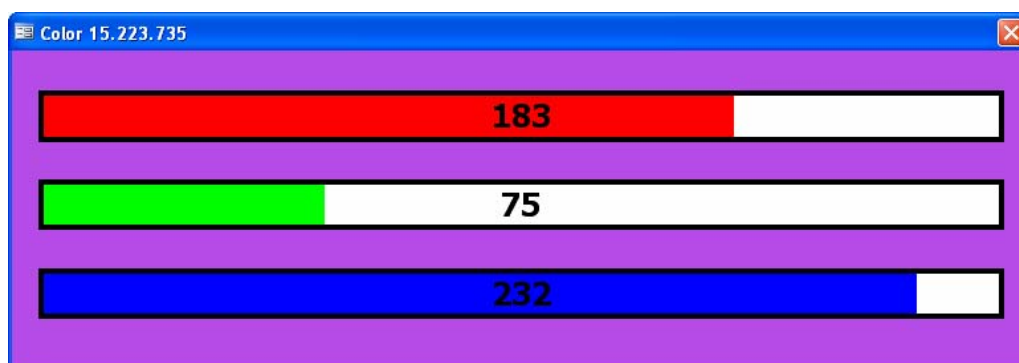
```

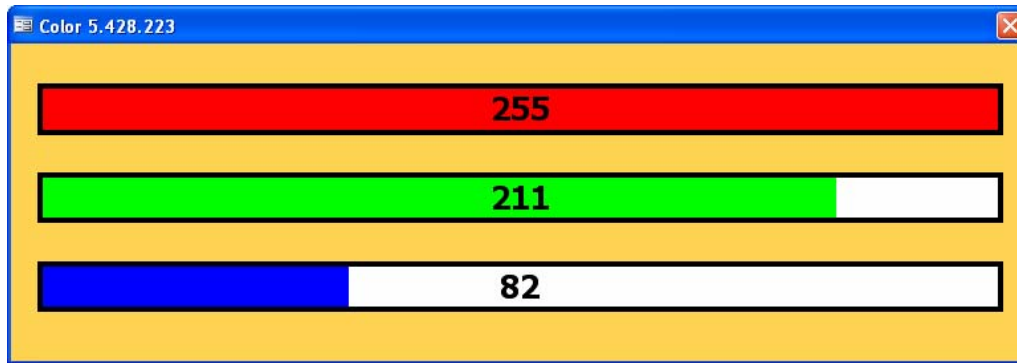
Private Sub ColoreaFormulario()
    Dim lngColor As Long
    lngColor = RGB(bytRojo, bytVerde, bytAzul)
    Detalle.BackColor = lngColor
    Caption = "Color " & Format(lngColor, "#,##0")
End Sub

```

Tras todo esto el formulario nos servirá para analizar la variación de los colores, en función de la proporción de cada color básico.

A continuación muestro diferentes resultados de ajustar las barras





Código final del proyecto

Aunque resulte un tanto reiterativo, aquí pongo el código completo del módulo de clase del formulario.

```
Option Compare Database
```

```
Option Explicit
```

```
Private blnBotonRojoBajado As Boolean
```

```
Private blnBotonVerdeBajado As Boolean
```

```
Private blnBotonAzulBajado As Boolean
```

```
Dim bytRojo As Byte
```

```
Dim bytVerde As Byte
```

```
Dim bytAzul As Byte
```

```
Dim lngAnchoEtiquetas As Long
```

```
Private Sub Form_Load()
```

```
    bytRojo = 128
```

```
    bytVerde = 128
```

```
    bytAzul = 128
```

```
    FormateaEtiquetas "Rojo"
```

```
    FormateaEtiquetas "Verde"
```

```
    FormateaEtiquetas "Azul"
```

```
    FormateaFormulario
```

```
    lngAnchoEtiquetas = lblRojol.Width
```

```
    ColoreaFormulario
```

```
End Sub
```

```
Private Sub lblRojol_MouseDown( _
```

```
    Button As Integer, _
```

```
    Shift As Integer, _
```

```
    X As Single, Y As Single)
```

```
    ' Comprobamos que el botón pulsado es el izquierdo
```

```
    If Button = acLeftButton Then
```

```
        blnBotonRojoBajado = True
```

```
        ' Llamamos al gestor del evento MouseMove de lblRojol
```

```
        lblRojol_MouseMove Button, Shift, X, Y
    End If
End Sub

Private Sub lblRojol_MouseUp( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
    blnBotonRojoBajado = False
End Sub

Private Sub lblRojol_MouseMove( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, _
    Y As Single)
    lngAnchoEtiquetas = lblRojol.Width
    If blnBotonRojoBajado Then
        If X <= lblRojol.Width And X >= 0 Then
            lblRojol2.Width = X
            bytRojo = Round(255 * X / lngAnchoEtiquetas)
            lblRojol1.Caption = bytRojo
            ColoreaFormulario
        End If
    End If
End Sub

Private Sub lblVerde1_MouseDown( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
    ' Comprobamos que el botón pulsado es el izquierdo
    If Button = acLeftButton Then
        blnBotonRojoBajado = True
        ' Llamamos al gestor del evento MouseMove de lblRojol
        lblVerde1_MouseMove Button, Shift, X, Y
    End If
End Sub

Private Sub lblVerde1_MouseUp( _
    Button As Integer, _
    Shift As Integer, _
    X As Single, Y As Single)
```

```
        blnBotonRojoBajado = False
    End Sub

    Private Sub lblVerde1_MouseMove( _
        Button As Integer, _
        Shift As Integer, _
        X As Single, _
        Y As Single)
        If blnBotonRojoBajado Then
            If X <= lblRojo1.Width And X >= 0 Then
                lblVerde2.Width = X
                bytVerde = Round(255 * X / lngAnchoEtiquetas)
                lblVerde1.Caption = bytVerde
                ColoreaFormulario
            End If
        End If
    End Sub

    Private Sub lblAzul1_MouseDown( _
        Button As Integer, _
        Shift As Integer, _
        X As Single, Y As Single)
        ' Comprobamos que el botón pulsado es el izquierdo
        If Button = acLeftButton Then
            blnBotonAzulBajado = True
            ' Llamamos al gestor del evento MouseMove de lblAzul1
            lblAzul1_MouseMove Button, Shift, X, Y
        End If
    End Sub

    Private Sub lblAzul1_MouseUp( _
        Button As Integer, _
        Shift As Integer, _
        X As Single, Y As Single)
        blnBotonAzulBajado = False
    End Sub

    Private Sub lblAzul1_MouseMove( _
        Button As Integer, _
        Shift As Integer, _
        X As Single, _
        Y As Single)
        lngAnchoEtiquetas = lblAzul1.Width
    End Sub
```

```
        If blnBotonAzulBajado Then
            If X <= lblAzul1.Width And X >= 0 Then
                lblAzul2.Width = X
                bytAzul = Round(255 * X / lngAnchoEtiquetas)
                lblAzul1.Caption = bytAzul
                ColoreaFormulario
            End If
        End If
    End Sub

Private Sub FormateaFormulario()
    ScrollBars = 0           ' Sin barras de desplazamiento
    RecordSelectors = False  ' Sin selector de registros
    NavigationButtons = False ' Sin Botones de desplazamiento
    DividingLines = False   ' Sin Separador de registros
End Sub

Private Sub FormateaEtiquetas(ByVal NombreColor As String)
    Const SEPARACION As Long = 500
    Dim lngTop As Long
    Dim lngColor As Long
    Dim lbl As Label
    Dim strNombreEtiqueta As String

    Select Case NombreColor
    Case "Rojo"
        lngTop = SEPARACION
        lngColor = vbRed
    Case "Verde"
        lngTop = 3 * SEPARACION
        lngColor = vbGreen
    Case "Azul"
        lngTop = 5 * SEPARACION
        lngColor = vbBlue
    End Select

    strNombreEtiqueta = "lbl" & NombreColor & "1"
    Set lbl = Controls(strNombreEtiqueta)
    With lbl
        .Top = lngTop
        .Width = lblRojol.Width
        .BorderStyle = 1           ' Borde de línea sólida
        .BorderWidth = 3          ' Anchura de 3 puntos
    End With
End Sub
```



```
.BackStyle = 0           ' Fondo transparente
.TextAlign = 2           ' Texto centrado
.Caption = 128           ' Texto inicial de la etiqueta
.FontSize = 18           ' Tamaño de la fuente
.FontBold = True        ' Texto en negrita
End With

strNombreEtiqueta = "lbl" & NombreColor & "2"
Set lbl = Controls(strNombreEtiqueta)
With lbl
    .Top = lngTop
    .Caption = ""
    .BackStyle = 1        ' Fondo opaco
    .BackColor = lngColor ' Color para el fondo
    ' Salvo la anchura, sus propiedades geométricas _
    ' como las de lblRojo1
    .Width = lblRojo1.Width / 2
    .Height = lblRojo1.Height
    .Left = lblRojo1.Left
End With

strNombreEtiqueta = "lbl" & NombreColor & "3"
Set lbl = Controls(strNombreEtiqueta)
With lbl
    .Caption = ""
    .BackStyle = 1        ' Fondo opaco
    .BackColor = vbWhite  ' Color blanco
    ' Igualamos sus propiedades geométricas _
    ' con las de lblRojo1
    .Width = lblRojo1.Width
    .Height = lblRojo1.Height
    .Left = lblRojo1.Left
    .Top = lngTop
End With
End Sub

Private Sub ColoreaFormulario()
    Dim lngColor As Long
    lngColor = RGB(bytRojo, bytVerde, bytAzul)
    Detalle.BackColor = lngColor
    Caption = "Color " & Format(lngColor, "#,##0")
End Sub
```

Recapitulemos

En esta entrega hemos analizado las características de algunas propiedades del formulario y de sus controles

Hemos visto cómo podemos manejar una serie de eventos

Cómo podemos, con un poco de imaginación, y bastante trabajo, emular un control mezcla de una barra de progreso y otro que se suele llamar **Slider**.

Hemos vuelto a ver cómo podemos gestionar los colores con varias técnicas diferentes.

Hemos repasado la asignación dinámica de controles a variables, creando instancias de los mismos, utilizando para ello la colección **Controls** del formulario..

Nos hemos construido un formulario que puede resultarnos “divertido”, e incluso útil cuando queramos manejar colores.

Sugerencia:

En una entrega anterior creamos la clase **clsColor** que permitía controlar los colores, e incluso generaba eventos cuando éste cambiaba.

En el código anterior la llamada al procedimiento **ColoreaFormulario** se realiza desde el gestor del evento **MouseMove** de las etiquetas superiores.

Quedaría mucho más “elegante” gestionar los colores mediante una instancia de la clase **clsColor**, y que sea el evento que se desencadena desde la clase, cuando se cambia su color, el que se encargue de dibujar el color del formulario y el **Caption** de las etiquetas superiores.

Nota:

En la entrega anterior, la 22, indicaba que en esta entrega se iba a desarrollar un formulario para jugar al bingo.

A la hora de redactar esta entrega he considerado que sería preciso, antes de abordar un proyecto de esas características, profundizar un poco en el manejo de las propiedades del formulario y los controles utilizando el código VBA.

Me comprometo a que en la próxima entrega desarrollaremos el proyecto.

Y espero que esa entrega se realice en breve.