

# Curso de



# Avanzado

SEPTIEMBRE 1999

<b>INTRODUCCIÓN AL MANUAL .....</b>	<b>6</b>
<b>1. FORMULARIOS Y EVENTOS.....</b>	<b>7</b>
INTRODUCCIÓN .....	7
CONOCIMIENTOS TEÓRICOS .....	8
<i>Formularios, controles y eventos</i> .....	8
<i>Ciclo de vida de un formulario</i> .....	8
<i>La ventana Debug</i> .....	9
EJEMPLO PROPUESTO.....	10
EJEMPLO RESUELTO.....	11
<i>Formulario 1</i> .....	11
<i>Formulario 2</i> .....	12
<b>2. IDENTIFICACIÓN DE OBJETOS (CONTROLES) EN UN FORMULARIO .....</b>	<b>13</b>
INTRODUCCIÓN .....	13
CONOCIMIENTOS TEÓRICOS .....	14
<i>Cómo utilizar controles: eventos y propiedades</i> .....	14
<i>Controles comunes: Label, TextBox, CommandButton, CheckBox</i> .....	14
<i>Identificar el tipo de un control</i> .....	15
<i>Colección Controls de un formulario</i> .....	15
EJEMPLO PROPUESTO.....	16
EJEMPLO RESUELTO.....	17
<b>3. CLASES Y OBJETOS (I).....</b>	<b>18</b>
INTRODUCCIÓN .....	18
CONOCIMIENTOS TEÓRICOS .....	19
<i>Los objetos en Visual Basic</i> .....	19
<i>Concepto de encapsulación (Public v.s. Private)</i> .....	19
<i>Utilización de variables de objeto</i> .....	19
<i>Instrucción Property</i> .....	20
<i>Variables estáticas</i> .....	21
EJEMPLO PROPUESTO.....	22
EJEMPLO RESUELTO.....	23
<i>Formulario</i> .....	23
<i>Clase Ficha</i> .....	23
<b>4. CLASES Y OBJETOS (II) .....</b>	<b>25</b>
INTRODUCCIÓN .....	25
CONOCIMIENTOS TEÓRICOS .....	26
<i>Cómo crear colecciones de objetos: la clase Collection</i> .....	26
<i>Propiedades y métodos del objeto Collection</i> .....	26
<i>Cómo acceder a los elementos de una colección: claves e índices</i> .....	27
<i>Agregar elementos a una colección</i> .....	27
<i>Eliminación de elementos de una colección</i> .....	28
<i>Recuperación de elementos de una colección</i> .....	28
<i>Control HScrollBar</i> .....	29
<i>Más conceptos de clases: eventos Initialize y Terminate</i> .....	30
EJEMPLO PROPUESTO.....	31
<i>Objetivo</i> .....	31
<i>Desarrollo del ejemplo</i> .....	32
EJEMPLO RESUELTO .....	34
<i>Formulario</i> .....	34
<i>Clase Agenda</i> .....	37
<b>5. TECNOLOGÍA ACTIVEX.....</b>	<b>39</b>
INTRODUCCIÓN .....	39

CONOCIMIENTOS TEÓRICOS .....	40
<i>Qué es un componente ActiveX</i> .....	40
<i>Tipos de componentes ActiveX</i> .....	40
<i>Creación dinámica de matrices</i> .....	41
<i>Preservar el contenido de las matrices dinámicas</i> .....	41
<i>Matrices de controles</i> .....	42
<i>Publicar un control ActiveX en un página Web</i> .....	42
EJEMPLO PROPUESTO .....	44
<i>Objetivo</i> .....	44
<i>Desarrollo del ejemplo</i> .....	44
EJEMPLO RESUELTO .....	46
<b>6. ACCESO A BASES DE DATOS (DAO Y JET).....</b>	<b>48</b>
INTRODUCCIÓN .....	48
CONOCIMIENTOS TEÓRICOS .....	49
<i>DAO y Jet</i> .....	49
<i>Estructura de una aplicación de bases de datos en Visual basic con DAO y Jet</i> .....	49
<i>El modelo de objetos DAO</i> .....	50
<i>Programación con DAO y Jet</i> .....	50
EJEMPLO PROPUESTO .....	54
<i>Objetivo</i> .....	54
<i>Desarrollo del ejemplo</i> .....	55
EJEMPLO RESUELTO .....	57
<i>ModDefiniciones</i> .....	57
<i>FrmAgenda</i> .....	57
<i>ClaAgenda</i> .....	60
<i>ClaDAO_Jet</i> .....	61
<b>7. ACCESO A BASES DE DATOS (DAO Y ODBCIRECT).....</b>	<b>66</b>
INTRODUCCIÓN .....	66
CONOCIMIENTOS TEÓRICOS .....	67
<i>Opciones de utilización de ODBC con DAO</i> .....	67
<i>Estructura de una aplicación de bases de datos en Visual basic con DAO y ODBCDirect</i> .....	67
<i>Programación con DAO y ODBCDirect</i> .....	67
EJEMPLO PROPUESTO .....	69
<i>Objetivo</i> .....	69
<i>Desarrollo del ejemplo</i> .....	69
EJEMPLO RESUELTO .....	70
<i>ClaDAO_ODBCDirect</i> .....	70
<b>8. ACCESO A BASES DE DATOS (RDO) .....</b>	<b>74</b>
INTRODUCCIÓN .....	74
CONOCIMIENTOS TEÓRICOS .....	75
<i>El modelo de objetos RDO</i> .....	75
<i>Estructura de una aplicación de bases de datos en Visual basic con RDO</i> .....	75
<i>Programación con RDO</i> .....	76
EJEMPLO PROPUESTO .....	78
<i>Objetivo</i> .....	78
<i>Desarrollo del ejemplo</i> .....	78
EJEMPLO RESUELTO .....	79
<i>ClaRDO</i> .....	79
<b>9. EL REGISTRO DE WINDOWS .....</b>	<b>83</b>
INTRODUCCIÓN .....	83
CONOCIMIENTOS TEÓRICOS .....	84
<i>Qué es el registro de Windows</i> .....	84
<i>Cómo acceder al registro de Windows</i> .....	84
<i>Estructura del registro de Windows</i> .....	84
<i>Cómo utilizar el API de Windows</i> .....	85
EJEMPLO PROPUESTO .....	89

Objetivo.....	89
Desarrollo del ejemplo .....	89
EJEMPLO RESUELTO.....	91
<i>FrmPresentacion</i> .....	91
<i>FrmRegistro</i> .....	93
<i>ModRegistro</i> .....	93
<b>APÉNDICE A: ESPECIFICACIONES, LIMITACIONES Y FORMATOS DE ARCHIVOS DE VISUAL BASIC.....</b>	<b>97</b>
REQUISITOS DEL SISTEMA PARA APLICACIONES DE VISUAL BASIC .....	97
LIMITACIONES DE LOS PROYECTOS .....	97
LIMITACIONES DE CONTROLES .....	97
<i>Número total de controles</i> .....	97
<i>Limitaciones para determinados controles</i> .....	98
LIMITACIONES DE CÓDIGO .....	98
<i>Procedimientos, tipos y variables</i> .....	98
<i>Tabla de entradas de módulo</i> .....	98
LIMITACIONES DE LOS DATOS .....	99
<i>Datos de formulario, módulos estándar y módulos de clase</i> .....	99
<i>Procedimientos, tipos y variables</i> .....	99
<i>Tipos definidos por el usuario</i> .....	99
<i>Espacio de pila</i> .....	99
<i>Limitaciones de los recursos del sistema</i> .....	99
<i>Recursos de Windows</i> .....	99
FORMATOS DE ARCHIVOS DE PROYECTO.....	100
<i>Extensiones de archivos de proyecto</i> .....	100
<i>Archivos varios y de tiempo de diseño</i> .....	100
<i>Archivos de tiempo de ejecución</i> .....	101
<i>Estructuras de formularios</i> .....	101
<i>Número de versión</i> .....	101
<i>Descripción del formulario</i> .....	102
<i>Bloques de control</i> .....	102
<i>Orden de los bloques de control</i> .....	102
<i>Bloques de control incrustados</i> .....	103
<i>Controles de menú</i> .....	103
<i>Teclas de método abreviado</i> .....	104
<i>Comentarios en la descripción del formulario</i> .....	104
<i>Propiedades de la descripción de un formulario</i> .....	104
<i>Sintaxis</i> .....	104
<i>Valores binarios de propiedades</i> .....	104
<i>Propiedad Icon</i> .....	105
<i>Propiedad TabIndex</i> .....	105
<i>Unidades de medida</i> .....	105
<i>Valores de colores</i> .....	105
<i>Objetos de propiedades</i> .....	106
<i>Código de Basic</i> .....	106
<i>Errores al cargar archivos de formulario</i> .....	106
<i>Mensajes de registro de error en la carga de un formulario</i> .....	106
<i>Formato del archivo de proyecto (.vbp)</i> .....	108
<b>APÉNDICE B: CONVENCIONES DE CODIFICACIÓN.....</b>	<b>110</b>
¿POR QUÉ EXISTEN LAS CONVENCIONES DE CODIFICACIÓN?.....	110
CONVENCIONES DE CODIFICACIÓN MÍNIMAS .....	110
CONVENCIONES DE NOMBRES DE OBJETOS .....	110
PREFIJOS SUGERIDOS PARA CONTROLES .....	111
PREFIJOS SUGERIDOS PARA LOS OBJETOS DE ACCESO A DATOS (DAO) .....	112
PREFIJOS SUGERIDOS PARA MENÚS.....	112
SELECCIÓN DE PREFIJOS PARA OTROS CONTROLES .....	113
CONVENCIONES DE NOMBRES DE CONSTANTES Y VARIABLES .....	113
<i>Prefijos de alcance de variables</i> .....	114

<i>Constantes</i> .....	114
<i>Variables</i> .....	114
<i>Tipos de datos de variables</i> .....	115
<i>Nombres descriptivos de variables y procedimientos</i> .....	115
<i>Tipos definidos por el usuario</i> .....	115
CONVENCIONES DE CODIFICACIÓN ESTRUCTURADA .....	115
<i>Convenciones de comentarios al código</i> .....	115
<i>Dar formato al código</i> .....	116
<i>Agrupación de constantes</i> .....	117
<i>Operadores &amp; y +</i> .....	117
<i>Crear cadenas para MsgBox, InputBox y consultas SQL</i> .....	117

## Introducción al manual

Visual Basic se ha convertido en un importante entorno de desarrollo que cubre todos los aspectos de la programación profesional, desde las aplicaciones financieras a la construcción de componentes para Internet. En la actualidad puede considerarse como la herramienta por excelencia para la programación en entornos Windows, gracias a su potencia, comparable a la de lenguajes tradicionalmente considerados superiores como C o C++, pero fundamentalmente por su sencillez y facilidad de uso.

Este manual está dirigido a aquellas personas que, teniendo nociones básicas de Visual Basic, deseen adquirir conocimientos sólidos de los principios y técnicas involucradas en el desarrollo de aplicaciones con este entorno de desarrollo. En concreto, en este curso se darán las bases para adquirir conocimientos en:

- Clases y objetos
- Tecnología active X
- Acceso a bases de datos (DAO, JET, ODBC-DIRECT, RDO)
- El registro de Windows

Si bien en todos los capítulos se expondrán los conocimientos teóricos que los autores hemos estimado suficientes para desenvolverse con soltura en cada tema propuesto, este manual no pretende ser un libro de referencia de Visual Basic (para eso existen multitud de libros técnicos, o la propia ayuda de Visual Basic). Los autores hemos preferido limitar en la medida de lo posible la carga teórica del manual, potenciando el aspecto práctico con la inclusión de ejemplos propuestos y resueltos, totalmente funcionales, que ayudarán al lector en la comprensión de los contenidos del curso.

Todos los capítulos de este manual siguen la misma estructura:

- Una introducción en la que se expondrán los temas a tratar y el objetivo del capítulo.
- Conocimientos teóricos del capítulo.
- Un ejemplo en el que se ponen en práctica los conocimientos recién adquiridos
- Y por supuesto, una solución al ejemplo propuesto.

Visual Basic proporciona una gran cantidad de elementos que permiten aumentar la potencia de nuestras aplicaciones considerablemente. Aunque el estudio individual de dichos objetos se escapa al propósito de este curso, en los ejemplos iremos proponiendo el uso de diferentes tipos de objetos para ir aumentando nuestro *vocabulario* en Visual Basic.

## 1. Formularios y eventos

### Introducción

El primer paso para crear una aplicación con Visual Basic es crear la interfaz, la parte visual de la aplicación con la que va a interactuar el usuario. Los formularios y controles son los elementos de desarrollo básicos que se usan para crear la interfaz; son los objetos con los que se trabaja para desarrollar la aplicación.

Los formularios son objetos que exponen las propiedades que definen su apariencia, los métodos que definen su comportamiento y los eventos que definen la forma en que interactúan con el usuario. Mediante el establecimiento de las propiedades del formulario y la escritura de código de Visual Basic para responder a sus eventos se personaliza el objeto para cubrir las necesidades de la aplicación.

Todos los lenguajes de programación para Windows comparten la misma característica: la orientación a eventos. Las aplicaciones creadas para Windows no siguen una estructura de ejecución lineal. Esto implica que un programa no va a seguir una secuencia fija de instrucciones, ejecutándose línea a línea, sino que debe responder a las acciones que el usuario, el sistema operativo u otras aplicaciones realizan sobre él; estas acciones se conocen en Visual Basic como eventos.

Todos los elementos que podemos utilizar en la construcción de una aplicación en Visual Basic (formularios, controles ActiveX, etc) reciben unos eventos predeterminados como **GetFocus**, **LostFocus** o **Activate**. Podemos controlar estos eventos introduciendo el código que deseamos en la correspondiente subrutina de atención al evento.

El objetivo de este capítulo es recordar los conocimientos básicos, pero imprescindibles, para el manejo de formularios en Visual Basic. Haremos hincapié en el concepto de formulario como objeto, con sus propiedades, métodos y eventos. Se comprobará que un formulario sólo se elimina de memoria cuando se eliminan todas las referencias al mismo, exactamente igual que cualquier objeto. Comentaremos especialmente los eventos que puede recibir un formulario, tanto en el proceso de carga como en la ejecución de la aplicación y en el proceso de descarga del mismo.

## Conocimientos teóricos

### Formularios, controles y eventos

Los formularios y controles de Visual Basic son objetos que exponen sus propios métodos, propiedades y eventos. Las propiedades se pueden considerar como atributos de un objeto, los métodos como sus acciones y los eventos como sus respuestas.

Un objeto de uso diario como el globo de un niño tiene también propiedades, métodos y eventos. Entre las propiedades de un globo se incluyen atributos visibles como el peso, el diámetro y el color. Otras propiedades describen su estado (inflado o desinflado) o atributos que no son visibles, como su edad. Por definición, todos los globos tienen estas propiedades; lo que varía de un globo a otros son los valores de estas propiedades.

Un globo tiene también métodos o acciones inherentes que puede efectuar. Tiene un método inflar (la acción de llenarlo de helio) o un método desinflar (expeler su contenido) y un método elevarse (si se deja escapar). De nuevo, todos los globos pueden efectuar estos métodos.

Los globos tienen además respuestas predefinidas a ciertos eventos externos. Por ejemplo, un globo respondería al evento de pincharlo desinflándose o al evento de soltarlo elevándose en el aire.

Como objetos que son, los formularios pueden ejecutar métodos y responder a eventos.

El evento **Resize** de un formulario se desencadena siempre que se cambia el tamaño de un formulario, ya sea por una acción del usuario o a través del código. Esto permite realizar acciones como mover o cambiar el tamaño de los controles de un formulario cuando han cambiado sus dimensiones.

El evento **Activate** se produce siempre que un formulario se convierte en el formulario activo; el evento **Deactivate** se produce cuando otro formulario u otra aplicación se convierte en activo. Estos eventos son adecuados para iniciar o finalizar acciones del formulario. Por ejemplo, en el evento **Activate** podríamos escribir código para resaltar el texto de un determinado cuadro de texto; con el evento **Deactivate** podríamos guardar los cambios efectuados en un archivo o en una base de datos.

Para hacer visible un formulario se invoca el método **Show**. Por ejemplo, la sentencia

```
Form1.Show
```

mostraría el formulario Form1. Invocar el método **Show** tiene el mismo efecto que establecer a **True** la propiedad **Visible** del formulario.

### Ciclo de vida de un formulario

Comprender el *ciclo de vida* de un formulario es básico para el programador de Visual Basic, puesto que son estos elementos los que permitirán al usuario interactuar con la aplicación. Sólo comprendiendo qué acciones podrán realizarse sobre los formularios y en qué momentos podremos controlar adecuadamente la ejecución de nuestros programas.

En la vida de un formulario puede pasar por cinco estados diferentes. En orden cronológico son:

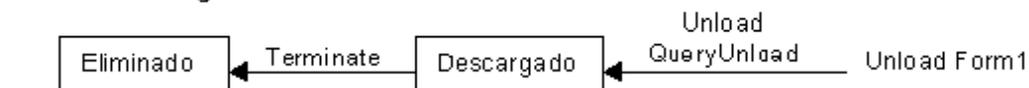
- **Creado:** el formulario existe como objeto, pero todavía no es visible. Cuando un formulario está siendo creado, se genera el evento **Initialize**. En esta fase del proceso de carga de un formulario no podemos actuar sobre los objetos que lo componen.

- **Cargado:** ya podemos acceder a los controles del formulario, pero éste no es visible. Un formulario se encuentra en este estado en el proceso de carga, cuando se oculta o cuando se establece a *False* su propiedad *Visible*. Cuando se inicia el proceso de carga de un formulario, se genera el evento **Load**.
- **Mostrado:** el estado normal de un formulario. Podemos acceder a todos los elementos que lo componen y podemos actuar sobre ellos, pues el formulario es visible. Durante este estado, cada vez que hay que repintar el formulario (por ejemplo, cuando una ventana que se ha situado sobre éste desaparece) se genera el evento **Paint**. Además, si el formulario recupera el foco después de haberlo perdido (por ejemplo, al cambiar a otra aplicación con Alt+Tab y volver después al formulario), se genera el evento **Activate**.
- **Descargado:** una vez terminado este proceso el formulario no es visible, y sus componentes no son accesibles. El proceso de descarga se produce cuando se cierra el formulario (pulsando el botón , ejecutando el método *Unload* del formulario, etc). El proceso de descarga de un formulario, a su vez, generará dos eventos: **QueryUnload** y **Unload** (en este orden). En los dos eventos podemos detener el proceso de descarga estableciendo el valor del parámetro **Cancel** a cualquier número distinto de 0. En general, aprovecharemos estos eventos para liberar la memoria ocupada por los elementos que pertenecen al formulario antes de que este se cierre, salvar el trabajo que esté pendiente de guardado, etc.
- **Eliminado:** aunque un formulario esté descargado, mientras existe cualquier referencia al mismo no será eliminado totalmente de memoria. Cuando se produce esta circunstancia (el formulario está descargado y nadie hace referencia a él) se genera el evento **Terminate**, y

**Proceso de carga de un formulario:**



**Proceso de descarga de un formulario:**



el formulario desaparece de memoria.

## La ventana Debug

Si bien este elemento del entorno de desarrollo de Visual Basic no tiene una relación directa con el tema que se trata en este capítulo, queremos recordar su funcionalidad, puesto que para el programador es una herramienta importante en el proceso de desarrollo de una aplicación.

En dicho proceso es necesario disponer de un elemento donde poder mostrar mensajes, valores de variables, etc. que nos ayuden en la depuración de los programas. El objeto **Debug** de Visual Basic nos permite escribir en la pantalla Inmediato (**Debug→Inmediate Window**) utilizando el método **Print**.

También podemos comprobar el valor de una variable si utilizamos la sentencia

`?nombre_variable`

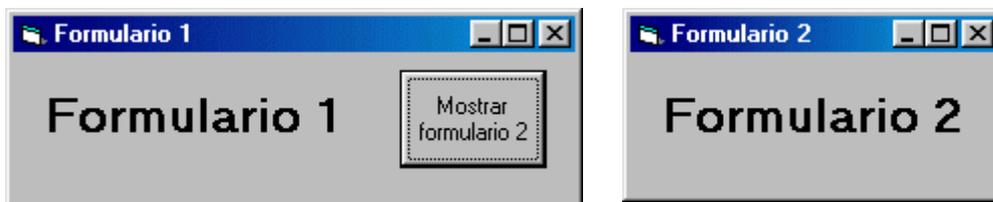
en la ventana Inmediato, o establecer dicho valor con la sentencia

`nombre_variable=valor`

### **Ejemplo propuesto**

Con este ejemplo veremos los estados por los que pasa un formulario durante su ciclo de vida. Además, comprobaremos cómo un formulario no se elimina de memoria hasta que no se liberan todas las referencias al mismo en el programa.

1. Crearemos los siguientes formularios



2. En el formulario 1, insertaremos código en las subrutinas Initialize, Load, Activate, Paint, QueryUnload, Unload y Terminate, para que cada vez que se lance el evento correspondiente mostremos en la ventana Debug el formulario sobre el que se ha generado el evento y el nombre del evento.
3. Añadiremos código en el Click del botón para mostrar el formulario 2. También mantendremos una referencia al formulario 2 (para evitar que se lance su evento Terminate al cerrarse).
4. Insertaremos código en las subrutinas Load, Unload y Terminate del formulario 2, para mostrar en la pantalla Debug el formulario y el evento que se produce en cada caso.

## Ejemplo resuelto

### Formulario 1

```
' Ejemplo 1 del Curso de Visual Basic Avanzado
'
' El objetivo de este ejemplo es reconocer los eventos que un
' formulario puede recibir, así como el orden en el que se
' reciben
'
Option Explicit

Private Sub Command1_Click()
    Dim FrmNuevoFormulario As Form

    ' Guardamos un enlace al formulario 2, para que no
    ' salte su Terminate cuando lo cerremos (saltará
    ' el Terminate del formulario 2 cuando salte el
    ' Terminate del formulario 1 y se pierda la referencia
    ' al segundo formulario)
    Set FrmNuevoFormulario = Formulario2
    FrmNuevoFormulario.Show
End Sub

Private Sub Form_Activate()
    Debug.Print "Formulario1: Evento Activate"
End Sub

Private Sub Form_Initialize()
    Debug.Print "Formulario1: Evento Initialize"
End Sub

Sub Form_Load()
    Debug.Print "Formulario1: Evento Load"
End Sub

Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
    Debug.Print "Formulario1: Evento QueryUnload"
End Sub

Private Sub Form_Paint()
    Debug.Print "Formulario1: Evento Paint"
End Sub

Private Sub Form_Terminate()
    Debug.Print "Formulario1: Evento Terminate"
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Debug.Print "Formulario1: Evento Unload"
End Sub
```

## Formulario 2

```
'  
' Ejemplo 1 del Curso de Visual Basic Avanzado  
'  
' El objetivo de este ejemplo es reconocer los eventos que un  
' formulario puede recibir, así como el orden en el que se  
' reciben  
'
```

---

Option Explicit

```
Private Sub Form_Load()  
    Debug.Print "Formulario2: Evento Load"  
End Sub
```

```
Private Sub Form_Terminate()  
    Debug.Print "Formulario2: Evento Terminate"  
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)  
    Debug.Print "Formulario2: Evento Unload"  
End Sub
```

## 2. Identificación de objetos (controles) en un formulario

### Introducción

Los controles son objetos que están contenidos en los objetos de formularios. Cada tipo de control tiene su propio conjunto de propiedades, métodos y eventos, que lo hacen adecuado para una finalidad determinada. Algunos de los controles que puede usar en las aplicaciones son más adecuados para escribir o mostrar texto, mientras que otros controles permiten tener acceso a otras aplicaciones y procesan los datos como si la aplicación remota formara parte del código.

Este capítulo presenta los conceptos básicos del trabajo con formularios y controles, y las propiedades, métodos y eventos que tienen asociados. Se explican también algunos de los controles estándar, y cómo es posible diferenciar en tiempo de ejecución los tipos de controles utilizados en nuestros formularios.

## Conocimientos teóricos

### Cómo utilizar controles: eventos y propiedades

Un evento es una acción reconocida por un formulario o un control, es decir, un suceso que ha sido provocado por el usuario, el sistema operativo u otros programas en ejecución y que repercute en un elemento de nuestra aplicación. Por ejemplo, son eventos la pulsación de una tecla o el paso del puntero de ratón sobre un objeto. A través de los eventos podemos determinar el comportamiento del objeto con su entorno.). Las aplicaciones controladas por eventos ejecutan código Basic como respuesta a un evento. Cada formulario y control de Visual Basic tiene un conjunto de eventos predefinidos. Si se produce uno de dichos eventos y el procedimiento de evento asociado tiene código, Visual Basic llama a ese código.

Aunque los objetos de Visual Basic reconocen automáticamente un conjunto predefinido de eventos, nosotros decidimos cuándo y cómo se responderá a un evento determinado. A cada evento le corresponde una sección de código (un procedimiento de evento). Cuando deseamos que un control responda a un evento, escribimos código en el procedimiento de ese evento.

Los tipos de eventos reconocidos por un objeto varían, pero muchos tipos son comunes a la mayoría de los controles. Por ejemplo, la mayoría de los objetos reconocen el evento **Click**: si un usuario hace clic en un formulario, se ejecuta el código del procedimiento de evento **Click** del formulario; si un usuario hace clic en un botón de comando, se ejecuta el código del procedimiento de evento **Click** del botón. El código en cada caso será diferente.

Las propiedades de los controles determinan su aspecto, cómo se mostrará al usuario. Estas propiedades definen sus atributos: color, tamaño, forma, etc. Al igual que en el caso de los eventos, los objetos de Visual Basic comparten una serie de propiedades comunes, como **Name** o **Visible**, y otras específicas de su tipo.

### Controles comunes: Label, TextBox, CommandButton, CheckBox

En Visual Basic hay un conjunto de controles comunes que son usados en la gran mayoría de aplicaciones. En la siguiente tabla se muestra el uso normal de estos controles:

Control		Uso típico y descripción breve
Label (Etiqueta)		Mostrar un texto descriptivo que el usuario no puede modificar. El texto se especifica en la propiedad <b>Caption</b> .
TextBox (Cuadro de texto)		Mostrar un texto que puede ser editado. La principal propiedad es <b>Text</b> , que determina el texto que se muestra en dicho control.
CommandButton (Botón)		Comenzar o terminar un proceso. Cuando el usuario pulsa este botón, se produce el evento <b>Click</b> .
CheckBox (Casilla de activación)		Seleccionar una opción (no excluyente) de entre un conjunto de opciones. Su principal propiedad es <b>Value</b> , que indica si está seleccionada (1) o no (0).
ListBox (Cuadro de lista)		Seleccionar un valor dentro una lista. La propiedad <b>Text</b> determina el elemento seleccionado por el usuario.

## Identificar el tipo de un control

Existen dos funciones que nos permiten identificar el tipo de control que estamos tratando: **TypeName** y **TypeOf**

La función **TypeName** nos permite conocer el nombre de la clase a la que pertenece un objeto, lo que nos posibilita adecuar el código que escribimos al tipo de objeto que estemos tratando, así como crear funciones generales para diversos tipos de controles.

Si tenemos un botón (CommandButton1) en nuestro formulario, utilizando la sentencia

```
TypeName (CommandButton1)
```

Obtendríamos la cadena "CommandButton", es decir, la clase a la que pertenece el botón.

La función **TypeOf** es algo más restrictiva. Debe ir en una sentencia de condición (una sentencia del tipo *if ... then ...*). Esta función nos permite identificar la clase a la que pertenece el objeto, para poder operar en consecuencia.

Un ejemplo de uso sería:

```
If TypeOf MiControl Is CommandButton Then
...
else
...
endif
```

## Colección **Controls** de un formulario

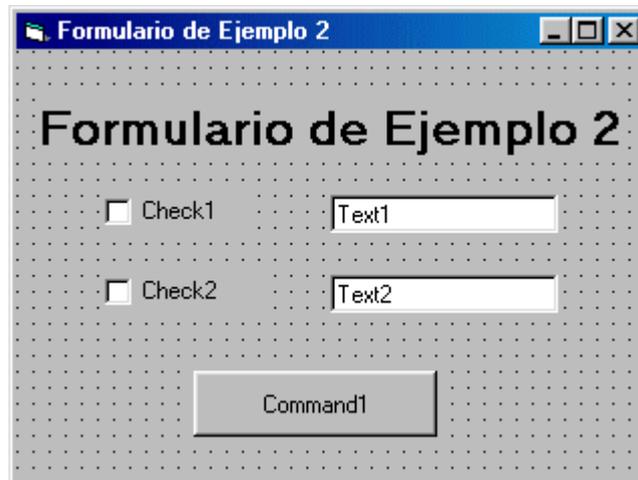
Un formulario no es más que un *lienzo* donde se *pintan* controles. Es decir, podemos entender un formulario como un objeto que posee una **colección** de controles. Esta colección se llama **Controls**.

Podemos acceder a cada uno de los controles de un formulario a través de esta colección. Sin necesidad de conocer a priori su nombre, bastaría recorrer toda la colección y hacer una rutina para lanzar el código a ejecutar para cada tipo de controles del formulario.

### **Ejemplo propuesto**

En este ejemplo distinguiremos los tipos de objetos que tiene un formulario, accediendo a las propiedades del objeto en el que nos encontremos durante el recorrido de la colección de objetos.

1. Crearemos el siguiente formulario:



2. En el Load del formulario, insertaremos código para que una vez cargado el formulario tengamos en la ventana de Debug el nombre y el tipo de todos los objetos que se han cargado.
3. En función del tipo de objeto, cambiaremos su color de fondo:<sup>1</sup>
  - Label: rojo
  - CheckBox: verde
  - TextBox: azul
  - CommandButton: amarillo

---

<sup>1</sup> Nota: para poder cambiar el color de fondo de un CommandButton, es necesario establecer a 1 (Graphical) su propiedad Style.

### **Ejemplo resuelto**

```
' Ejemplo 2 del Curso de Visual Basic Avanzado
'
' El objetivo de este ejemplo es identificar los objetos contenidos
' en un formulario y operar con ellos en función de su tipo.
'
Option Explicit

Private Sub Form_Load()
    Dim ObjControl As Object

    ' Recorremos todos los controles del formulario
    For Each ObjControl In Me.Controls
        ' En función del tipo de control, cambiaremos su color de
        ' fondo
        Select Case TypeName(ObjControl)
            Case "Label"
                ObjControl.BackColor = vbRed
            Case "TextBox"
                ObjControl.BackColor = vbBlue
            Case "CheckBox"
                ObjControl.BackColor = vbGreen
            Case "CommandButton"
                ObjControl.BackColor = vbYellow
        End Select

        ' Sacaremos su nombre y tipo de objeto a la pantalla
        ' Debug
        Debug.Print TypeName(ObjControl) & ": " & ObjControl.Name
    Next ObjControl
End Sub
```

### 3. Clases y objetos (I)

#### Introducción

Hasta ahora hemos visto los objetos que proporciona Visual Basic, pero nosotros también podemos crear objetos. En este capítulo se darán los conocimientos básicos necesarios para entender cómo podemos utilizar objetos en Visual Basic.

Cada objeto de Visual Basic se define mediante una *clase*. Podemos entender las clases como un "modelo conceptual" de los objetos. Para comprender la relación entre un objeto y su clase, podemos pensar en un cubo de playa y un castillo de arena. El cubo es la clase. Define las características de cada castillo de arena, como su tamaño y forma. Se utiliza la clase para crear (*instanciar*) objetos. Los objetos son los castillos de arena y como era de esperar todos tienen las mismas propiedades y eventos.

En el capítulo anterior hemos hablado de las propiedades y los eventos de un objeto. Pero además de estas características los objetos proporcionan un conjunto de *métodos*, que determinan las acciones que el objeto puede realizar.

La programación orientada a objetos proporciona un enfoque totalmente distinto al utilizado en la programación clásica (estructuras, tipos de datos, programación secuencial...). Bien aplicada, la programación orientada a objetos minimiza el tiempo dedicado a la depuración de los programas, y facilita la reutilización del código.

Así mismo este modelo de programación nos permite controlar qué características de nuestros objetos son *visibles* para las aplicaciones que los utilizan, lo que se denomina *encapsulación*.

## Conocimientos teóricos

### Los objetos en Visual Basic

Los objetos están *encapsulados*; es decir, contienen su propio código y sus propios datos, lo cual facilita su mantenimiento más que los métodos de escritura de código tradicionales.

Los objetos de Visual Basic tienen *propiedades*, *métodos* y *eventos*. Las propiedades son los datos que describen un objeto. Los métodos son lo que puede hacer con el objeto. Los eventos son tareas que realiza el objeto; podemos escribir código para ejecutarlo cuando se produzcan eventos.

Los objetos de Visual Basic se crean a partir de *clases*; así, un objeto se dice que es una *instancia de una clase*. La clase define las *interfaces* de un objeto, si el objeto es público y en qué circunstancias se puede crear.

Para utilizar un objeto tiene que mantener una *referencia a él* en una *variable de objeto*. El tipo de *enlace* determina la velocidad con la que se tiene acceso a los métodos de un objeto mediante la variable de objeto. Una variable de objeto puede ser un *enlace en tiempo de compilación* (el más lento) o un *enlace en tiempo de diseño*.

Al conjunto de propiedades y métodos se le llama *interfaz*. La interfaz predeterminada de un objeto de Visual Basic es una *interfaz dual* que admite las formas de enlace anteriormente mencionadas. Si una variable de objeto está *correctamente definida* (es decir, **Dim ... As nombre\_clase**), utilizará la forma más rápida de enlace.

Además de su interfaz predeterminada, los objetos de Visual Basic pueden implementar interfaces adicionales para proporcionar *polimorfismo*. El polimorfismo le permite manipular muchos tipos deferentes de objetos sin preocuparse de su tipo. Las *interfaces múltiples* son una característica del Modelo de objetos componentes (COM); permiten que los programas evolucionen con el tiempo, agregando nueva funcionalidad sin afectar al código existente.

### Concepto de encapsulación (Public v.s. Private)

Una de las características más importantes de la programación orientada a objetos es el concepto de **encapsulación**: un objeto bien desarrollado es aquel que puede ser utilizado por otros elementos de forma eficiente, pero sin que éstos conozcan cómo funciona internamente. Esto nos permite cambiar la forma en que un objeto *funciona* sin que los elementos que utilizan el objeto *noten* dicho cambio.

En Visual Basic, las propiedades y métodos de una clase pueden ser de dos tipos:

- **Public**: accesibles por cualquier elemento del proyecto.
- **Private**: accesibles sólo por los elementos de la clase donde están definidos.

### Utilización de variables de objeto

Utilizar una variable de objeto es similar a utilizar una variable convencional, pero con un paso adicional: asignar un objeto a la variable.

Primero, declararemos la variable con una sentencia

```
Dim variable As clase
```

Luego, asignaremos un objeto a esa variable, utilizando la instrucción **Set**:

**Set variable = objeto**

Pero **atención**: el objeto **ha de existir** para poder ser asignado a una variable de objeto. Es decir, previamente hemos tenido que utilizar la instrucción **New**, bien en la misma sentencia en la que declaramos la variable de objeto:

**Dim variable As New Clase**

O bien en otro lugar del código:

**Set variable = New Clase**

La sentencia **New** crea una instancia de la clase especificada (es decir, crea un objeto nuevo a partir del *modelo* que es la clase).

Cuando ya no sean de utilidad necesitamos liberar la memoria y los recursos que los objetos consumen. En la mayoría de los casos el sistema operativo se encargará de esta tarea si se nos olvida a nosotros, pero no hay que confiarse (podéis imaginar el rendimiento de una máquina en la que tengamos, por ejemplo, 5 instancias de Word al mismo tiempo...). Para liberar la memoria y los recursos que consume un objeto utilizaremos la sentencia:

**Set variable = Nothing**

### Instrucción *Property*

Como se mencionó anteriormente, es deseable que un elemento externo a un objeto no pueda acceder de forma incontrolada a sus propiedades. Podemos controlar esto creando todas las propiedades de tipo **Private**, pero... ¿y si queremos que sean accesibles por los elementos que utilicen el objeto, pero sin arriesgarnos a que éstos realicen acciones no permitidas?. Por ejemplo, podemos necesitar que el TPV (Terminal Punto de Venta) de una tienda pueda modificar el saldo de una tarjeta (lo que se correspondería con la compra de un cliente); sin embargo, no tiene sentido que dicho TPV permita al vendedor comprobar el saldo de la cuenta del cliente, o hacer una transferencia a su propia cuenta.

Visual Basic permite acceder y modificar el valor de una propiedad privada a través de los *procedimientos de propiedad* o **Property**.

Se proporcionan tres tipos de procedimientos:

Procedimiento	Propósito
<b>Property Get</b>	Devuelve el valor de una propiedad
<b>Property Let</b>	Establece el valor de una propiedad
<b>Property Set</b>	Establece el valor de una propiedad de objeto (es decir, una propiedad que contiene una referencia a un objeto)

Normalmente se define una pareja **Property Get/Property Let** por cada propiedad privada de una clase. **Property Get** nos permite acceder al valor almacenado en dicha propiedad, mientras que **Property Let** nos posibilita asignarle un valor.

**Property Set** también permite establecer el valor de una propiedad privada. La diferencia entre este procedimiento y **Property Let** es el modo en el que se asigna valor a la

propiedad. Visual Basic llama a **Property Set** si se utiliza la instrucción **Set** y a **Property Let** en caso contrario.

Esta propiedad se utiliza para mantener el estándar de los objetos, ya que para asignarles un valor se usa la palabra reservada **Set**.

### Variables estáticas

Aunque este tema no está directamente relacionado con el tema del capítulo, queremos recordar este concepto para poder aplicarlo en el ejemplo propuesto.

La vida de una variable está directamente relacionada con la forma y el lugar donde se declara. El valor de las variables declaradas a nivel de módulo y públicas se preserva durante todo el tiempo de ejecución. Sin embargo, las variables locales declaradas con **Dim** sólo existen mientras se ejecuta el procedimiento en el cual se han declarado. La próxima vez que se ejecute el procedimiento se reinicializarán todas sus variables locales.

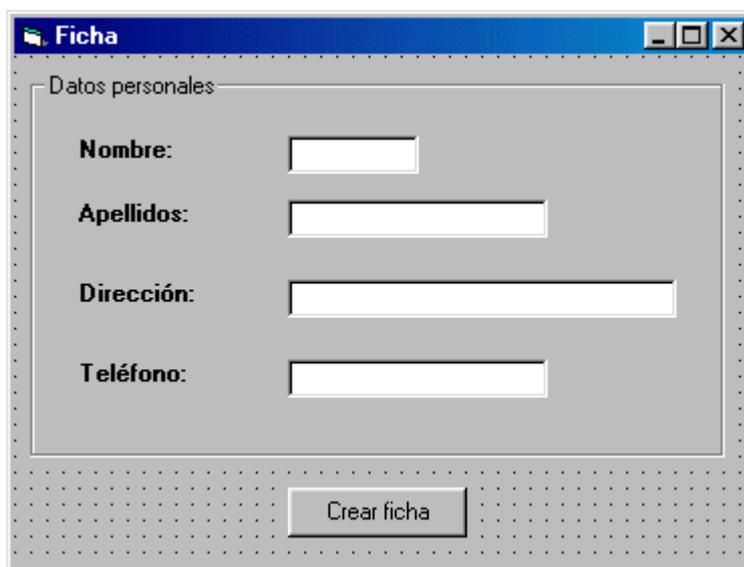
Sin embargo es posible preservar el valor de una variable local si se declara como *estática*, utilizando la palabra clave **Static**:

```
Static nombre_variable as tipo_dato
```

### Ejemplo propuesto

El objetivo de este ejemplo es conocer el proceso de creación e instanciación de una clase en Visual Basic. También se pretende que el lector se familiarice con el *ciclo de vida* de un objeto y la necesidad de liberar la memoria asociada con cada uno de los objetos creados. Para practicar estos conceptos crearemos una agenda telefónica básica.

1. Diseñaremos el siguiente formulario:



The image shows a screenshot of a Visual Basic form titled "Ficha". The form has a blue title bar with standard Windows window controls. Inside the form, there is a container labeled "Datos personales" which contains four text boxes with labels: "Nombre:", "Apellidos:", "Dirección:", and "Teléfono:". Below the text boxes, there is a button labeled "Crear ficha". The form has a dotted grid background.

2. Definiremos una clase *Ficha* con cinco propiedades: *nombre*, *apellidos*, *dirección*, *teléfono* y *número*. Para cada una de las propiedades crearemos sus correspondientes métodos de propiedad **Property Get** y **Property Let**.
3. Implementaremos también un método *Muestra()* que muestre mediante un **msgbox** el valor de las propiedades de un objeto *Ficha* creado previamente.
4. En el formulario, una vez establecidas las propiedades de la ficha (nombre, apellidos...) se pulsará el botón para crear una nueva ficha. Será necesario crear un nuevo objeto de la clase *Ficha* y establecer sus propiedades.
5. Para establecer la propiedad número de la ficha, crearemos una variable estática que incrementaremos en una unidad cada vez que se pulse el botón, de forma que nunca creemos dos fichas con el mismo número.
6. Una vez creada la nueva ficha, llamaremos a su método *Muestra()*, para que nos enseñe el valor de sus propiedades.
7. Por último, no podemos olvidarnos de **destruir el objeto creado** igualándolo a **Nothing**.

## Ejemplo resuelto

### Formulario

```
' Ejemplo 3 del Curso de Visual Basic Avanzado
'
' El objetivo de este ejemplo es familiarizarnos con el uso de objetos.
```

---

Option Explicit

```
Private Sub ButCrearFicha_Click()
    Dim ObjFicha As ClaFicha
    Static IntNumFicha As Integer

    ' Creamos un nuevo objeto de la clase ficha
    Set ObjFicha = New ClaFicha

    ' Incrementamos el número de fichas creadas
    IntNumFicha = IntNumFicha + 1

    ' Establecemos las propiedades de la nueva ficha
    ObjFicha.StrNombre = TxtNombre
    ObjFicha.StrApellidos = TxtApellidos
    ObjFicha.StrDireccion = TxtDireccion
    ObjFicha.StrTelefono = TxtTelefono
    ObjFicha.IntNumero = IntNumFicha

    ' Llamamos al método MetMostrar del objeto para
    ' que muestre los datos de la ficha
    ObjFicha.MetMostrar

    ' Eliminamos el objeto creado
    Set ObjFicha = Nothing
End Sub
```

### Clase Ficha

```
' Ejemplo 3 del Curso de Visual Basic Avanzado
'
' Definición de la clase FICHA, todos los objetos fichas tendrán esta interface.
```

---

Option Explicit

```
' Propiedades de la clase ClaFicha
Private VarIntNumero As Integer
Private VarStrNombre As String
Private VarStrApellidos As String
Private VarStrDireccion As String
```

```
Private VarStrTelefono As String

' Métodos para acceder a la propiedad VarIntNumero
Public Property Get IntNumero() As String
    IntNumero = VarIntNumero
End Property

Public Property Let IntNumero(ParIntNumero As String)
    VarIntNumero = ParIntNumero
End Property

' Métodos para acceder a la propiedad VarStrNombre
Public Property Get StrNombre() As String
    StrNombre = VarStrNombre
End Property

Public Property Let StrNombre(ParStrNombre As String)
    VarStrNombre = ParStrNombre
End Property

' Métodos para acceder a la propiedad VarStrApellidos
Public Property Get StrApellidos() As String
    StrApellidos = VarStrApellidos
End Property

Public Property Let StrApellidos(ParStrApellidos As String)
    VarStrApellidos = ParStrApellidos
End Property

' Métodos para acceder a la propiedad VarStrDireccion
Public Property Get StrDireccion() As String
    StrDireccion = VarStrDireccion
End Property

Public Property Let StrDireccion(StrDireccion As String)
    VarStrDireccion = StrDireccion
End Property

' Métodos para acceder a la propiedad VarStrTelefono
Public Property Get StrTelefono() As String
    StrTelefono = VarStrTelefono
End Property

Public Property Let StrTelefono(ParStrTelefono As String)
    VarStrTelefono = ParStrTelefono
End Property

' Método para mostrar los datos de una ficha
Public Sub MetMostrar()
    MsgBox "Número: " & VarIntNumero & vbCrLf & _
        "Nombre: " & VarStrNombre & vbCrLf & _
        "Apellidos: " & VarStrApellidos & vbCrLf & _
        "Dirección: " & VarStrDireccion & vbCrLf & _
        "Teléfono: " & VarStrTelefono, , "Ficha"
End Sub
```

## 4. Clases y objetos (II)

### Introducción

Hasta ahora hemos aprendido cómo se crea e instancia una clase sencilla, y los conceptos más importantes de la programación orientada a objetos en Visual Basic.

En este capítulo aprenderemos a desarrollar clases más complejas utilizando colecciones de clases más sencillas. Para ello aprenderemos a usar colecciones de clases, y haremos hincapié en la optimización en tiempo y recursos del sistema al hacer búsquedas, inserciones y/o borrados mediante *índices o claves* de una colección.

También se pretende que el lector se familiarice con el uso de controles menos utilizados como la barra de desplazamiento horizontal (HScrollBar).

## Conocimientos teóricos

### Cómo crear colecciones de objetos: la clase **Collection**

Una colección es una forma de agrupar elementos relacionados. Visual Basic proporciona una gran cantidad de colecciones predefinidas (**Forms**, **Controls**, etc), pero además permite la creación de colecciones personalizadas utilizando la clase genérica **Collection**.

Los objetos **Collection** almacenan cada elemento en una variable **Variant**. De esta forma, la lista de cosas que se pueden agregar a un objeto **Collection** es igual que la lista de cosas que se pueden almacenar en una variable **Variant**. Entre ellas se incluyen los tipos de datos, los objetos y las matrices estándar, pero no los tipos definidos por el usuario.

Las variables **Variant** siempre ocupan 16 bytes, independientemente de lo que contienen<sup>®</sup>, por lo que el uso de objetos **Collection** no es tan eficiente como el uso de matrices. Sin embargo, no es necesario aplicar **ReDim** a un objeto **Collection**, lo que produce un código más limpio y más fácil de mantener. Además, los objetos **Collection** ofrecen un acceso por clave extremadamente rápido, que no ofrecen las matrices.

A pesar del tamaño de las variables **Variant**, habrá muchas ocasiones en las que tenga más sentido utilizar un objeto **Collection** para almacenar todos los tipos de datos enumerados anteriormente. Sin embargo, siempre tenemos que tener presente que si bien los objetos **Collection** nos permitirán escribir un código limpio y fácil de mantener, estamos pagando el precio de almacenar los elementos en variables **Variant**.

### Propiedades y métodos del objeto **Collection**

Los objetos **Collection** disponen de propiedades y métodos que se puede utilizar para insertar, eliminar y recuperar los elementos de la colección.

Propiedad o método	Descripción
Método <b>Add</b>	Agrega elementos a la colección.
Propiedad <b>Count</b>	Devuelve el número de elementos de la colección. Sólo lectura.
Método <b>Item</b>	Devuelve un elemento, por índice o por clave.
Método <b>Remove</b>	Elimina un elemento de la colección, por índice o por clave.

Estas propiedades y métodos sólo proporcionan los servicios más básicos de las colecciones. Por ejemplo, el método **Add** no puede comprobar el tipo de objeto que se agrega a la colección para asegurar que la colección contenga un único tipo de objeto. Podemos proporcionar una funcionalidad más robusta, y propiedades, métodos y eventos adicionales, si creamos nuestra propia clase de colección, como haremos en el ejemplo correspondiente a este capítulo.

<sup>®</sup> Más exactamente, una variable **Variant** ocupa siempre 16 bytes *incluso si los datos se encuentran almacenados en cualquier otra parte*. Por ejemplo, si asignamos una cadena o una matriz a una variable de tipo **Variant**, ésta contiene un puntero a una copia de la cadena o de los datos de la matriz. En los sistemas de 32 bits sólo se utilizan como puntero 4 bytes de la variable **Variant** y realmente no hay ningún dato en ella. Si almacenamos un objeto, la variable de tipo **Variant** contiene la referencia de objeto, como si fuera una variable de objeto. Igual que ocurre con las cadenas y las matrices, solamente se utilizan 4 bytes. Los tipos de datos numéricos se almacenan ocupando siempre los 16 bytes.

## Cómo acceder a los elementos de una colección: claves e índices

Los servicios básicos de agregar, eliminar y recuperar elementos de una colección dependen de claves e índices. Una *clave* es un valor **String**. Puede ser un nombre, un número de documento de identidad, un número de la seguridad social o simplemente un dato de tipo **Integer** convertido en tipo **String**. El método **Add** nos permite asociar una clave con un elemento, como se describe más adelante.

Un *índice* es un dato de tipo **Long** entre uno (1) y el número de elementos de la colección<sup>®</sup>. Podemos controlar el valor inicial del índice de un elemento mediante los parámetros **before** y **after**, pero su valor puede cambiar si agregamos o eliminamos otros elementos.

Podemos utilizar el índice para recorrer los elementos de una colección. Por ejemplo, si insertamos el código siguiente en el **Load** de un formulario aparecería en la pantalla **Debug** el nombre de los controles que están pegados en él:

```
Dim IntContador as Integer
For IntContador = 0 To Controls.Count - 1
    Debug.Print Controls(IntContador).Name & vbCrLf
Next
```

Existe otra forma de acceder a los elementos de una colección que proporciona un mayor rendimiento: utilizar la sentencia **For Each**. Esta opción es notablemente más rápida que la iteración mediante índice, aunque esto no es cierto en todas las implementaciones de colecciones: depende de cómo almacena la colección los datos internamente. Podemos codificar de nuevo el ejemplo anterior utilizando esta sentencia:

```
Dim VarDato as Variant
For Each VarDato In Controls
    Debug.Print VarDato.Name & vbCrLf
Next VarDato
```

## Agregar elementos a una colección

Para agregar un elemento a una colección utilizaremos el método **Add**. La sintaxis es la siguiente:

*colección*.**Add** (*elemento As Variant* [, *clave As Variant*] [, *before As Variant*] [, *after As Variant*] )

Por ejemplo, imaginemos un objeto Persona que tiene una propiedad DNI que lo identifica de forma única, y una colección de personas denominada ColPersonas. Para agregar un objeto nuevo a la colección utilizaríamos la sentencia:

```
ColPersonas.Add Persona, Persona.DNI
```

---

<sup>®</sup> Una colección puede tener *base cero* o *base uno*, dependiendo de su índice inicial (lo primero quiere decir que el índice del primer elemento de la colección es cero y lo segundo quiere decir que es uno). Ejemplos de colecciones con base cero son las colecciones **Forms** y **Controls**. El objeto **Collection** es un ejemplo de colección con base uno.

Las colecciones más antiguas de Visual Basic suelen tener base cero, mientras que las más recientes suelen tener base uno. Es más intuitivo utilizar las colecciones con base uno, ya que su índice va desde uno a **Count**, donde **Count** es la propiedad que devuelve el número de elementos de una colección.

Por otra parte, el índice de una colección con base cero va desde cero al valor de la propiedad **Count** menos uno.

Se supone que la propiedad DNI es un tipo **String**. Si esta propiedad es un número (por ejemplo, un tipo **Long**), utilizaríamos la función **CStr** para convertirlo al valor **String** requerido por las claves:

```
ColPersonas.Add Persona, CStr(Persona.DNI)
```

El método **Add** acepta argumentos con nombre. Para agregar una persona nueva como tercer elemento de la colección, podemos escribir:

```
ColPersonas.Add Persona, Persona.DNI, after:=2
```

Podemos utilizar los argumentos con nombre **before** y **after** para mantener ordenada una colección de objetos. Por ejemplo, `before:=1` inserta un elemento al principio de la colección, ya que los objetos **Collection** están basados en uno.

## Eliminación de elementos de una colección

Para eliminar un elemento de una colección, utilizaremos el método **Remove**. La sintaxis es la siguiente:

```
colección.Remove índice
```

El argumento *índice* puede ser la posición del elemento que desea eliminar o su clave. Si la clave del tercer elemento de la colección de personas es "1234567W", podemos utilizar cualquiera de estas dos instrucciones para eliminarlo:

```
ColPersonas.Remove 3
```

o bien

```
ColPersonas.Remove "1234567W"
```

## Recuperación de elementos de una colección

Para recuperar un elemento de una colección, utilizaremos el método **Item**. La sintaxis es la siguiente:

```
[Set] variable = colección.Item(índice)
```

Como ocurre con el método **Remove**, el índice puede ser la posición dentro de la colección o la clave del elemento. En el mismo ejemplo que para el método **Remove**, cualquiera de estas dos instrucciones recuperará el tercer elemento de la colección:

```
Set Persona = ColPersonas.Item(3)
```

o bien

```
Set Persona = ColPersonas.Item("1234567W")
```

Si utilizamos números enteros como claves, debemos emplear la función **CStr** para convertirlos a cadenas antes de pasarlos a los métodos **Item** o **Remove**. Los objetos **Collection** siempre asumen que un número entero es un índice<sup>⊗</sup>.

El método **Item** es el método predeterminado de los objetos **Collection**, por lo que podemos omitirlo cuando queramos acceder a un elemento de una colección. De esta forma, el ejemplo de código anterior también se puede escribir de la siguiente manera:

```
Set Persona = ColPersonas(3)
```

o bien

```
Set Persona = ColPersonas("1234567W")
```

Los objetos **Collection** mantienen sus números de índice automáticamente al agregar y eliminar elementos. El índice numérico de un elemento dado puede variar a lo largo del tiempo. No es conveniente, por tanto, guardar un valor numérico de índice y esperar recuperar el mismo elemento más tarde. Lo mejor es utilizar claves para este propósito.

## Control HScrollBar

Las barras de desplazamiento proporcionan un medio sencillo de recorrer largas listas de elementos o grandes cantidades de información al desplazarse horizontal o verticalmente dentro de una aplicación o de un control.

Las barras de desplazamiento son un elemento común de la interfaz de Windows 95 y de Windows NT.



Barra de desplazamiento horizontal (**HScrollBar**)



Barra de desplazamiento vertical (**VScrollBar**)

Los controles **HScrollBar** y **VScrollBar** no son igual que las barras de desplazamiento incorporadas que se pueden encontrar en Windows o que las adjuntas a los cuadros de texto, los cuadros de lista, los cuadros combinados o los formularios MDI de Visual Basic. Dichas barras de desplazamiento aparecen automáticamente cuando la aplicación o el control contiene más información de la que se puede presentar con el tamaño actual de la ventana (o bien, en el caso de los cuadros de texto y los formularios MDI, cuando la propiedad **ScrollBars** está establecida a **True**).

En las versiones anteriores de Visual Basic, las barras de desplazamiento se solían utilizar como dispositivos de entrada. Sin embargo, las recomendaciones de la interfaz de Windows sugieren ahora que se utilicen como dispositivos de entrada los controles **Slider** en lugar de los controles **ScrollBar**. El control **Slider** de Windows 95 se incluye en la Edición Profesional y la Edición Empresarial de Visual Basic.

---

<sup>⊗</sup> No es conveniente permitir que los objetos **Collection** decidan si el valor que están recibiendo es un índice o una clave. Si deseamos que un valor se interprete como clave y la variable que contiene el valor no es de tipo **String**, debemos utilizar **CStr** para convertirla. Si deseamos que un valor se interprete como índice y la variable que contiene el valor no es de un tipo de datos numéricos, debemos utilizar **CLng** para convertirla.

Los controles **ScrollBar** siguen siendo útiles en Visual Basic porque proporcionan capacidad de desplazamiento a las aplicaciones o a los controles que no la proporcionan de forma automática.

Los controles **ScrollBar** utilizan los eventos **Scroll** y **Change** para supervisar el movimiento del cuadro de desplazamiento a lo largo de la barra de desplazamiento.

Evento	Descripción
Change	Ocurre después de mover el cuadro de desplazamiento.
Scroll	Ocurre al mover el cuadro de desplazamiento. No ocurre si se hace clic en las flechas o en la barra de desplazamiento.

El evento **Scroll** proporciona acceso al valor de la barra de desplazamiento cuando ésta se arrastra. El evento **Change** se produce después de liberar el cuadro de desplazamiento o cuando se hace clic en la barra o en las flechas de desplazamiento.

La propiedad **Value** (que de forma predeterminada es 0) es un valor entero que se corresponde con la posición del cuadro de desplazamiento dentro de la barra de desplazamiento. Cuando la posición del cuadro de desplazamiento tiene el valor mínimo, éste se mueve a la posición situada más a la izquierda (en las barras de desplazamiento horizontal) o a la posición más alta (en las barras de desplazamiento vertical). Cuando el cuadro de desplazamiento tiene el valor máximo, se mueve a la posición situada más a la derecha o a la más baja. De forma similar, un valor a medio camino entre los límites del intervalo coloca el cuadro de desplazamiento en la mitad de la barra de desplazamiento.

Además de utilizar los clics del ratón para cambiar el valor de la barra de desplazamiento, el usuario también puede arrastrar el cuadro de desplazamiento a cualquier posición de la barra. El valor resultante depende de la posición del cuadro de desplazamiento, pero siempre se encuentra dentro del intervalo definido por las propiedades **Min** y **Max** establecidas por el usuario.

## Más conceptos de clases: eventos Initialize y Terminate

Todas las clases proporcionan dos eventos predefinidos: **Initialize** y **Terminate**.

Normalmente el procedimiento de evento **Initialize** contiene el código que se ejecuta en el momento de la creación del objeto.

El evento **Terminate** suele contener el código necesario para liberar el objeto cuando éste se destruye. Supone una buena norma de programación asegurarnos, utilizando este método, que toda la memoria utilizada por un objeto se libera en el momento de su destrucción.

## Ejemplo propuesto

### Objetivo

El objetivo es crear una agenda de teléfonos en la que podamos consultar fácilmente las fichas, así como insertar nuevas fichas o borrar fichas existentes.

Para ello, crearemos una nueva clase **ClaAgenda**:

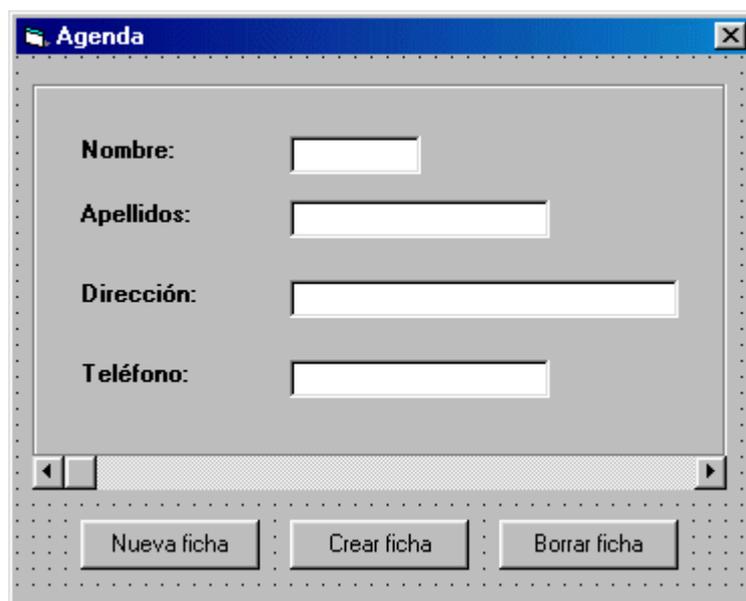
- **Propiedades:**
  - **ColFichas**: colección de las fichas almacenadas en la agenda
- **Métodos:**
  - **Property IntNumFichas**: devuelve el número de fichas almacenadas en la agenda
  - **Function CrearFicha(StrNombre, StrApellidos, StrDireccion, StrTelefono) as Integer**: crea una nueva ficha y la inserta en la colección ColFichas. Devuelve el número de la ficha.
  - **Function BorrarFicha(IntNumFicha)**: borra la ficha cuyo número se pasa como argumento.
  - **Function Ficha(IntIndice) As ClaFicha**: devuelve la ficha correspondiente a la posición IntIndice en la colección de fichas ColFichas
  - **Sub Class\_Terminate()**: eliminaremos todas las fichas de la colección ColFichas, así como la propia colección, antes de que el objeto se elimine de memoria

En el formulario podremos realizar **tres acciones**:

- **Crear una nueva ficha**: primero pulsaremos el botón **Nueva ficha**, con lo que se borrarán los cuadros de texto y se incrementará el límite superior de la barra de scroll. Una vez introducidos los datos de la nueva ficha, pulsaremos el botón **Crear ficha**, donde se llamará al método **CrearFicha** de la clase **ClaAgenda**, obteniendo el número de ficha resultante, que nos servirá para poder borrarla en caso necesario.
- **Consultar una ficha existente**: utilizando la barra de scroll podremos consultar todas las fichas de la agenda. Cada vez que cambie el valor de la barra de scroll, actualizaremos el formulario para mostrar los datos de la ficha correspondiente (por ejemplo, si el valor de la barra de scroll es 3, mostraremos la **cuarta** ficha de la colección; hay que recordar que el límite inferior de la barra de scroll es 0, mientras que una colección empieza en 1). Cada vez que cambiemos de ficha, actualizaremos el número de la ficha actual.
- **Borrar la ficha actual**: utilizaremos el número de ficha actual para llamar al método **BorrarFicha** de la clase **ClaAgenda**. Decrementaremos el límite máximo de la barra de scroll, y actualizaremos el formulario para que muestra los datos de la ficha siguiente o anterior a la borrada.

## Desarrollo del ejemplo

1. Crearemos el siguiente formulario:



2. Crearemos una clase *Agenda* con una propiedad privada *ColFichas* de tipo **Collection**, que no tendrá métodos **Property** asociados. Crearemos un método **Property Get** llamado **IntNumFichas** que nos devuelva el número de fichas de la colección *ColFichas*.
3. Implementaremos un método público *CrearFicha* que recibe como argumentos el nombre, apellido, dirección y teléfono y crea una ficha nueva (utilizaremos la clase *Ficha* definida en el ejemplo 3), asignándole un número (este número será una especie de “campo autonumérico”, que generaremos incrementalmente). Además de crear la nueva ficha, la introduce en la colección *ColFichas*, poniendo como clave el número de la ficha (como la clave tiene que ser una cadena, hay que hacer la conversión a string de dicho número).
4. Implementaremos también un método público *BorrarFicha*, que recibe como argumento la clave de la ficha que queremos eliminar. Este método elimina la ficha de memoria y también de la colección.
5. También crearemos un método público *Ficha*, que recibe como argumento el índice de la ficha dentro de la colección *ColFichas* y devuelve la ficha correspondiente.
6. Como último elemento de la clase, en el evento **Terminate** liberaremos todo el espacio de memoria que podamos tener reservado para las fichas y la colección.
7. En el formulario crearemos un objeto de la clase *Agenda*, y una variable *IntNumFichaActual* que indicará el número de la ficha cuyos datos aparecen en el formulario. Como estas variables van a ser utilizadas en todas las funciones del formulario, las crearemos como variables globales del formulario.
8. Al pulsar el botón “Nueva ficha”, debemos aumentar la propiedad **Max** de la barra de scroll, posicionarnos en el último elemento y limpiar los datos (nombre, apellidos, dirección y teléfono), preparando el formulario para insertar una nueva ficha. (¡Ojo!: para la primera ficha, no es necesario incrementar el límite máximo de la barra de scroll, ni limpiar el formulario).

9. Una vez insertados los datos de la nueva ficha, pulsaremos el botón “Crear ficha”, donde llamaremos al método CrearFicha del objeto Agenda y actualizaremos el título del frame, donde pondrá “Ficha N” (siendo N el número de la nueva ficha, que obtendremos como valor de retorno del método CrearFicha).
10. Cada vez que cambie la barra de scroll (evento **Change**) debemos actualizar los datos del formulario, mostrando la información de la ficha correspondiente en la colección ColFichas del objeto Agenda. Crearemos una función ActualizarBarraScroll que se encargará de ello (**¡Ojo!**: la función debe comprobar si hay fichas en la colección, y también si estamos insertando una ficha nueva, en cuyo caso en lugar de actualizar los datos del formulario, los limpiará.). También tenemos que obtener el número de la ficha actual.
11. Cuando pulsemos el botón “Borrar ficha” llamaremos al método BorrarFicha del objeto Agenda pasándole el número de la ficha actual. Una vez borrada, decrementaremos el valor máximo de la barra de scroll (**¡Ojo!**: cuidado cuando borramos la última ficha) y actualizaremos los datos del formulario llamando a la función ActualizarBarraScroll.
12. Para terminar, sólo nos falta controlar cuando están habilitados los cuadros de texto, los botones y demás elementos para que quede lo más coherente posible.

## Ejemplo Resuelto

### Formulario

```
' Ejemplo 4 del Curso de Visual Basic Avanzado
'
' Definición del formulario, este será el interfaz del usuario.
'
Option Explicit

Dim ObjAgenda As New ClaAgenda
Dim IntNumFichaActual As Integer

Private Sub ButBorrarFicha_Click()

    ' Confirmamos el borrado
    If MsgBox("¿Desea eliminar la ficha actual?", vbYesNo, "Borrado de ficha") = vbNo Then
        Exit Sub
    End If

    ' Borramos la ficha actual
    ObjAgenda.BorrarFicha IntNumFichaActual

    ' Actualizamos el número de elementos de la barra de scroll
    If ObjAgenda.IntNumFichas > 0 Then
        HScroll1.Max = ObjAgenda.IntNumFichas - 1
    Else
        HScroll1.Max = 0
    End If

    ' Actualizamos la barra de scroll
    ActualizarBarraScroll

End Sub

Private Sub ButCrearFicha_Click()

    ' Creamos una nueva ficha con los datos introducidos por el
    ' usuario, y obtenemos el código de la ficha creada
    IntNumFichaActual = ObjAgenda.CrearFicha(TxtNombre, TxtApellidos, TxtDireccion, _
        TxtTelefono)

    ' Actualizamos el título del frame
    Frame1.Caption = "Ficha " & IntNumFichaActual

    ' Deshabilitamos el botón de crear ficha y el frame
    ButCrearFicha.Enabled = False
    Frame1.Enabled = False

    ' Habilitamos los botones de nueva ficha y borrar
    ' ficha, así como la barra de scroll
    ButNuevaFicha.Enabled = True
    ButBorrarFicha.Enabled = True
    HScroll1.Enabled = True
```

```

End Sub

Private Sub ButNuevaFicha_Click()
    ' Actualizamos la barra de scroll
    If ObjAgenda.IntNumFichas = 0 Then
        ' Con la primera ficha creada, habilitamos
        ' la barra de scroll
        HScroll1.Enabled = True
        Frame1.Enabled = True
    Else
        ' Incrementamos el número de elementos de
        ' la barra de scroll
        HScroll1.Max = HScroll1.Max + 1
        HScroll1.Value = HScroll1.Max
    End If

    ' Habilitamos el botón de crear ficha y el frame
    ButCrearFicha.Enabled = True
    Frame1.Enabled = True

    ' Deshabilitamos los botones de nueva ficha y borrar
    ' ficha, así como la barra de scroll
    ButNuevaFicha.Enabled = False
    ButBorrarFicha.Enabled = False
    HScroll1.Enabled = False
End Sub

Private Sub Form_Load()
    ' Deshabilitamos el frame, la barra de scroll,
    ' el botón de crear ficha y el botón de borrar ficha
    Frame1.Enabled = False
    HScroll1.Enabled = False
    ButCrearFicha.Enabled = False
    ButBorrarFicha.Enabled = False

    ' Establecemos los límites de la barra de scroll
    HScroll1.Max = 0
End Sub

Private Sub Form_Unload(Cancel As Integer)
    ' Eliminamos la agenda
    Set ObjAgenda = Nothing
End Sub

Private Sub HScroll1_Change()
    ' Actualizamos la barra de scroll
    ActualizarBarraScroll
End Sub

Private Function ActualizarBarraScroll()
    Dim ObjFicha As ClaFicha

    ' Comprobamos si no hay fichas o estamos creando una ficha nueva
    If HScroll1.Max = ObjAgenda.IntNumFichas Then
        ' Limpiamos los datos del frame
        TxtNombre = vbNullString
        TxtApellidos = vbNullString
        TxtDireccion = vbNullString
        TxtTelefono = vbNullString
    End If
End Function

```

```
' Actualizamos el título del frame
Frame1.Caption = vbNullString

' Deshabilitamos el botón de borrado
ButBorrarFicha.Enabled = False
Else
' Obtenemos la ficha correspondiente de la agenda
' (buscamos por índice, no por código de ficha)
Set ObjFicha = ObjAgenda.Ficha(HScroll1.Value + 1)

' Sacamos los datos de la ficha
TxtNombre = ObjFicha.StrNombre
TxtApellidos = ObjFicha.StrApellidos
TxtDireccion = ObjFicha.StrDireccion
TxtTelefono = ObjFicha.StrTelefono

' Actualizamos el código de la ficha actual
IntNumFichaActual = ObjFicha.IntNumFicha

' Actualizamos el frame
Frame1.Caption = "Ficha " & IntNumFichaActual
End If
End Function
```

## Clase Agenda

```
' Ejemplo 4 del Curso de Visual Basic Avanzado
'
' Definición de la clase AGENDA, aquí estarán los datos de cada agenda y los métodos
' para actuar con ellos.
'
'
Option Explicit

Private ColFichas As New Collection ' Colección de fichas de la agenda

' Método para obtener el número de fichas de la agenda
Public Property Get IntNumFichas() As Integer
    IntNumFichas = ColFichas.Count
End Property

' Método para crear una ficha nueva en la agenda
Public Function CrearFicha(ByVal StrNombre As String, _
    ByVal StrApellidos As String, ByVal StrDireccion As String, _
    ByVal StrTelefono As String) As Integer
    Static IntNumTotalFichas As Integer
    Dim ObjFicha As ClaFicha

    ' Incrementamos el "campo autonúmerico" IntNumTotalFichas
    IntNumTotalFichas = IntNumTotalFichas + 1

    ' Creamos un nuevo objeto de la clase ficha
    Set ObjFicha = New ClaFicha

    ' Establecemos las propiedades de la nueva ficha
    ObjFicha.StrNombre = StrNombre
    ObjFicha.StrApellidos = StrApellidos
    ObjFicha.StrDireccion = StrDireccion
    ObjFicha.StrTelefono = StrTelefono
    ObjFicha.IntNumFicha = IntNumTotalFichas

    ' Insertamos la nueva ficha en la colección de fichas, poniendo
    ' como key el código de la ficha, para poder buscarla después
    ' por código (convertimos a string el código porque el key
    ' tiene que ser un string)
    ColFichas.Add ObjFicha, CStr(ObjFicha.IntNumFicha)

    ' Devolvemos el código de la ficha creada
    CrearFicha = IntNumTotalFichas
End Function

' Método para borrar una ficha de la agenda
Public Function BorrarFicha(ByVal IntNumFicha As Integer)
    Dim ObjFicha As ClaFicha

    ' Obtenemos la ficha correspondiente de la colección
    ' de fichas (para poder buscar por key, tenemos que
    ' convertir el número a string, pues en caso contrario
    ' buscamos por índice)
```

```
Set ObjFicha = ColFichas.Item(CStr(IntNumFicha))

' Eliminamos la ficha de la colección
ColFichas.Remove (CStr(IntNumFicha))

' Eliminamos la ficha
Set ObjFicha = Nothing
End Function

' Método para eliminar todas las fichas de una agenda cuando
' éste es eliminada
Private Sub Class_Terminate()
    Dim ObjFicha As ClaFicha

    ' Eliminamos todas las fichas de la colección
    For Each ObjFicha In ColFichas
        Set ObjFicha = Nothing
    Next ObjFicha
End Sub

' Método para obtener una ficha específica de la agenda (no busca
' por key sino por índice)
Public Function Ficha(ByVal IntIndice As Integer) As ClaFicha
    Set Ficha = ColFichas.Item(IntIndice)
End Function
```

## 5. Tecnología ActiveX

### Introducción

El objetivo de este capítulo es mostrar al lector el proceso de creación de componentes ActiveX, y su utilización en un proyecto. También se pretende que el lector se familiarice con el proceso de publicación de un control ActiveX en una página Web, como muestra de la versatilidad de este tipo de componentes.

Con este capítulo se hace una introducción al curso *Active X*, que es la continuación de los cursos Visual Basic 5.0 (básico y avanzado).

## Conocimientos teóricos

### Qué es un componente ActiveX

Un componente ActiveX es un archivo .EXE, .DLL u .OCX que cumple la especificación ActiveX para proporcionar objetos. Para entendernos, un control ActiveX es una extensión del cuadro de herramientas de Visual Basic. Los controles ActiveX se usan como cualquiera de los controles estándar incorporados, como el control CheckBox. Cuando agregamos un control ActiveX a un programa, pasa a formar parte del entorno de desarrollo y de tiempo de ejecución y proporciona nueva funcionalidad a la aplicación.

Los controles ActiveX incrementan la capacidad del programador de Visual Basic conservando algunos métodos, propiedades y eventos ya familiares, como la propiedad **Name**, que se comportan como cabría esperar. Pero, además, los controles ActiveX incorporan métodos y propiedades que aumentan enormemente la flexibilidad y capacidad del programador de Visual Basic.

La tecnología ActiveX permite a los programadores ensamblar estos componentes software reutilizables en sus aplicaciones.

Los controles ActiveX, que se llamaban antes controles OLE, son elementos estándar de interfaz de usuario que nos permiten ensamblar rápidamente formularios y cuadros de diálogo. Los controles ActiveX también dan vida a Internet, agregando una nueva y atractiva funcionalidad a las páginas del World Wide Web.

Visual Basic siempre ha presentado diversos controles que podíamos utilizar en nuestras aplicaciones. Ahora podemos crear nuestros propios controles para utilizarlos con Visual Basic y otras herramientas de programación.

Diseñar un control ActiveX puede resultar tan fácil como diseñar un formulario de Visual Basic: podemos utilizar los comandos gráficos de Visual Basic con los que estamos familiarizados para dibujar el control o bien crear un grupo de controles con los controles existentes.

Los controles ActiveX se pueden depurar en proceso, de forma que puede pasar directamente desde el código del formulario de prueba al código del proyecto de control ActiveX.

Visual Basic facilita crear paquetes de controles ActiveX perfeccionados al agregar a los controles páginas de propiedades, constantes con nombre y eventos.

Podemos compilar nuestros controles ActiveX directamente en el archivo ejecutable de nuestra aplicación o en archivos .ocx, que se pueden utilizar con herramientas de programación como Visual Basic y Microsoft Visual C++, con productos para el usuario final como Microsoft Office y en Internet.

### Tipos de componentes ActiveX

Existen tres tipos de componentes ActiveX:

- **Controles ActiveX:** Los controles ActiveX son elementos estándar de interfaz de usuario que permiten construir rápidamente aplicaciones, como si de un *puzzle* se tratara.

Visual Basic proporciona un conjunto de controles predefinidos, y permite la creación de controles de usuario. Por ejemplo, podríamos hacer una caja de texto que controlara si los caracteres introducidos son numéricos, mostrando los números introducidos con la coma de los decimales o de los millares, y utilizarlo como el convencional *textbox*.

- **ActiveX DLL (librerías):** Los componentes ActiveX proporciona código que se puede volver a utilizar en forma de objetos. Una aplicación (*cliente*) puede crear un objeto a partir de un componente ActiveX (*servidor*) y utilizar sus propiedades y métodos. Los componentes ActiveX DLL se ejecutan dentro del espacio de direcciones del programa que los utiliza.
- **ActiveX EXE:** Un componente ActiveX EXE es muy similar a un componente ActiveX DLL. La única diferencia es que un componente ActiveX EXE se ejecuta en su propio espacio de direcciones, lo que permite que el *cliente* puede ejecutar un método del componente y continuar con su ejecución mientras el componente realiza el trabajo, manteniendo el control de la ejecución del programa principal si se quedara bloqueada la llamada a una función del Active X. La pega es que gasta mas recursos, pues la comunicación entre el *cliente* y el *servidor* es más compleja.

## Creación dinámica de matrices

Hay ocasiones en las que no conocemos a priori lo grande que debe ser una matriz. Puede que deseemos poder cambiar el tamaño de la matriz en tiempo de ejecución.

Una matriz dinámica se puede cambiar de tamaño en cualquier momento. Las matrices dinámicas son una de las características más flexibles y cómodas de Visual Basic, y nos ayudan a administrar de forma eficiente la memoria. Por ejemplo, podemos utilizar una matriz grande durante un tiempo corto y liberar memoria del sistema cuando no necesite volver a utilizar la matriz.

La alternativa consiste en declarar la matriz con el mayor tamaño posible y pasar por alto los elementos de la matriz que no necesitemos. Sin embargo, esta solución, si se utiliza demasiado, puede hacer que el sistema operativo funcione con muy poca memoria.

El primer paso en la creación de una matriz dinámica es su declaración: en un primer momento, la matriz no tendrá ninguna dimensión. Por ejemplo, la siguiente sentencia define un *array* de enteros, pero no especifica cuántos elementos va a contener:

```
Dim IntMatriz() As Integer
```

Para asignar el número real de elementos utilizaremos la instrucción **ReDim**. Por ejemplo, con la siguiente sentencia permitiremos que el *array* anterior almacene 10 enteros:

```
ReDim IntMatriz (0 To 9)
```

La instrucción **ReDim** puede aparecer sola en un procedimiento. A diferencia de las instrucciones **Dim** y **Static**, **ReDim** es una instrucción ejecutable; hace que la aplicación realice una acción en tiempo de ejecución.

La instrucción **ReDim** acepta la misma sintaxis que se utiliza en las matrices fijas. Cada **ReDim** puede cambiar el número de elementos, así como los límites inferior y superior de cada dimensión. Sin embargo, si la matriz ha sido declarada con un número de elementos determinado, no podremos utilizar la sentencia **ReDim** para alterarlo. Por ejemplo, si definimos la matriz anterior de esta forma:

```
Dim IntMatriz(5) As Integer
```

No podremos utilizar posteriormente la sentencia **ReDim** para alterar su tamaño.

## Preservar el contenido de las matrices dinámicas

Cada vez que ejecutamos la instrucción **ReDim** se perderán todos los valores almacenados en ese momento en la matriz. Visual Basic restablece los valores al valor **Empty** (en matrices

**Variant**), a cero (en matrices numéricas), a una cadena de longitud cero (en matrices de cadenas) o a **Nothing** (en matrices de objetos).

Esto resulta muy útil cuando deseemos preparar la matriz para contener datos nuevos o cuando deseemos reducir el tamaño de la matriz para que ocupe menos memoria. Puede que a veces nos interese cambiar el tamaño de la matriz sin perder los datos de la misma. Para ello podemos utilizar **ReDim** con la palabra clave **Preserve**. Por ejemplo, podemos ampliar una matriz en un elemento sin perder los valores de los elementos existentes mediante la función **UBound** para hacer referencia al límite superior:

```
ReDim Preserve IntMatriz(UBound(IntMatriz) + 1)
```

Sólo se puede cambiar el límite superior de la última dimensión de una matriz multidimensional cuando se utiliza la palabra clave **Preserve**; si cambiamos alguna otra dimensión o el límite inferior de la última dimensión, se producirá un error en tiempo de ejecución. Así pues, podemos utilizar un código como el siguiente:

```
ReDim Preserve IntMatriz(10, UBound(IntMatriz, 2) + 1)
```

Pero no podemos utilizar este código:

```
ReDim Preserve IntMatriz(UBound(IntMatriz, 1) + 1, 10)
```

## Matrices de controles

Visual Basic permite crear matrices de controles dinámicamente y situar los nuevos controles a voluntad en un formulario. Supongamos que tenemos un control llamado MiCtl (por ejemplo, un *CommandButton*). Para crear un nuevo control, utilizaríamos la sentencia:

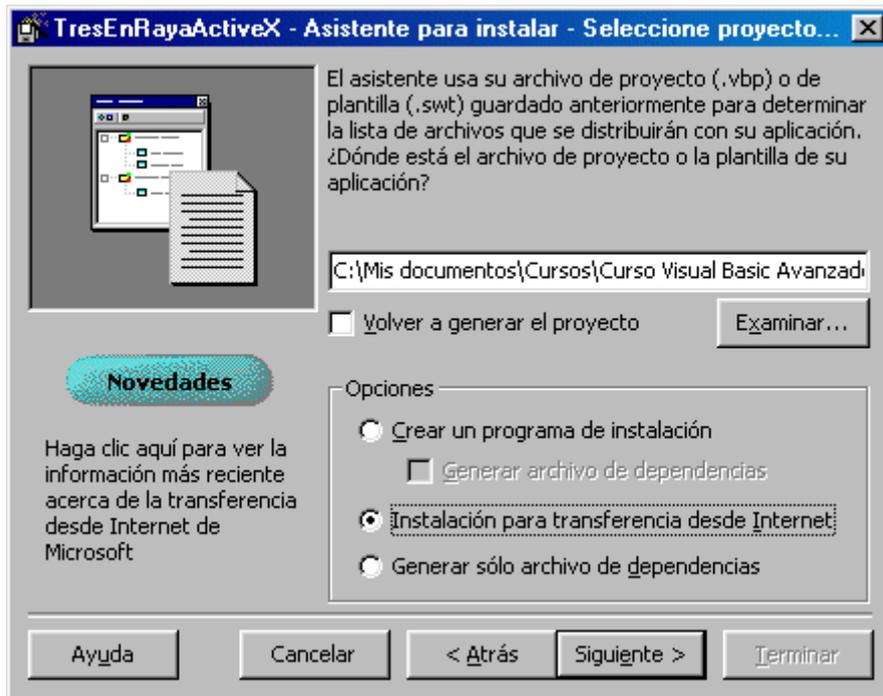
```
Load MiCtl(N)
```

donde N es el índice del nuevo control. Una vez creado, podemos situar el nuevo control en la posición que deseemos estableciendo apropiadamente su propiedad **Top** y **Left**, y hacerlo visible estableciendo a **True** la propiedad **Visible**.

## Publicar un control ActiveX en un página Web

Una vez creado el control ActiveX, hay que seguir los siguientes pasos:

1. Generar el archivo .OCX correspondiente **y guardar el proyecto después**.
2. Ejecutar el **Asistente para instalar aplicaciones**.
3. Seleccionar el proyecto correspondiente al control ActiveX, marcando la opción **Instalación para transferencia desde Internet**.
4. Durante el proceso de creación de los archivos, especificaremos el directorio donde se crearán los archivos, así como el lugar desde donde el cliente que abra la página Web donde estará el control ActiveX bajará los archivos .CAB de los componentes de nuestro control (si es necesario). También contestaremos afirmativamente a la pregunta de si queremos incluir la DLL de la página de propiedades (necesario si queremos ejecutar el control desde fuera de Visual Basic, como es el caso).



El asistente nos creará principalmente dos archivos con el nombre del control ActiveX, uno con extensión .CAB que contiene el propio control (y otros elementos utilizados por él, si procede) y otro con extensión .HTM en el que se muestra un ejemplo de página HTML que utiliza el control ActiveX.

## Ejemplo propuesto

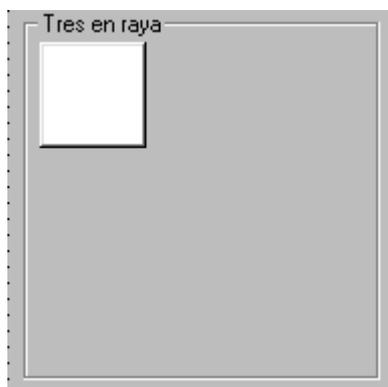
### Objetivo

El objetivo es crear un control ActiveX que simule el juego de las *Tres en raya*. Una vez creado, generaremos una página Web donde pegaremos el control con este ejemplo podemos ver la potencia que tiene esta herramienta, ya que con el uso de un control podemos tener una aplicación bastante complicada corriendo detrás, siendo muy fácil el implantación para usos posteriores.

### Desarrollo del ejemplo

1) Crearemos un control ActiveX **TresEnRaya**:

- **Interfaz:**
  - Simularemos un tablero de 3x3, pero sólo crearemos el *frame* que lo contiene y **UN** botón. Estableceremos las propiedades del botón (color, nombre...), especialmente la propiedad **Index=0** (va a ser el primero de una matriz de controles).



#### Propiedades:

- **StrTurno**: almacena el carácter correspondiente al turno actual ("O" o "X")
- **IntNumeroJugadas**: recoge el número de movimientos realizados
- **Métodos:**
  - **Sub UserControl\_Initialize**: a partir de un único botón creado en tiempo de diseño en el control ActiveX, generará una matriz ordenada de 3x3 botones (índices de 0 a 8).
  - **Sub InicializarFormulario**: inicializará el caption de los botones, el turno y el número de jugadas.
  - **Function HayTresEnRaya() As Boolean**: comprobará cuando hay tres en raya, en función del caption de los botones que es donde nos indica el jugador que lo eligió (O o X).
  - **Sub ButCelda\_Click(Index As Integer)**: realizará todo el proceso del juego (comprobar si hay tres en raya, comprobar si el botón ya ha sido pulsado, y en caso contrario cambiar el caption del mismo, comprobar si se han marcado ya todas las casillas, incrementar el número de jugadas realizadas...)

2) Una vez creado el control, abriremos **un proyecto nuevo** (de tipo EXE estándar) en el que añadiremos un formulario en blanco **y nuestro proyecto TresEnRaya**. Pegaremos el control TresEnRaya, que aparecerá en la caja de herramientas, y depuraremos el control abierto como proyecto, ya que si no tendríamos que depurarlo ejecutándolo directamente, y para solucionar un posible error volvernos al VB para realizar la modificación, recompilar y hacer el ejecutable, mientras que de esta forma con reescribir y ejecutar tenemos el problema resuelto.

- 3) Cuando el control funcione bien, abriremos el proyecto inicial, generaremos el archivo TresEnRaya.ocx, **guardaremos el proyecto** y cerraremos Visual Basic.
- 4) Ejecutaremos el Asistente para instalar aplicaciones y crearemos los ficheros necesarios para publicar nuestro control en una página Web.

### Ejemplo resuelto

```
'
' Ejemplo 5 del Curso de Visual Basic Avanzado
'
' Código del control de TRES EN RAYA, funciones necesarias para el control de la partida
' y control de movimientos de los jugadores.
'
Option Explicit

Dim StrTurno As String
Dim IntNumeroJugadas As Integer

Private Sub ButCelda_Click(Index As Integer)
    If ButCelda(Index).Caption = vbNullString Then
        ' Pintamos la marca en la ButCelda correspondiente
        ButCelda(Index).Caption = StrTurno

        ' Comprobamos si hay 3 en línea
        If HayTresEnRaya() Then
            MsgBox "¡Has ganado!", vbOKOnly Or vbExclamation, "Tres en Raya"
            InicializarFormulario
            Exit Sub
        End If

        ' Cambiamos de StrTurno
        If StrTurno = "O" Then
            StrTurno = "X"
        Else
            StrTurno = "O"
        End If

        ' Comprobamos si se han realizado todos los movimientos
        ' posibles

        If IntNumeroJugadas = 9 Then
            If MsgBox("Ha habido un empate. ¿Quieres jugar otra vez?", vbYesNo Or vbQuestion, "Tres en
Raya") = vbYes Then
                InicializarFormulario
            End If
        End If

        ' Incrementamos el número de movimientos realizados
        IntNumeroJugadas = IntNumeroJugadas + 1
    End If
End Sub

Private Function HayTresEnRaya() As Boolean
    Dim Intl, IntJ As Integer

    ' Filas horizontales
    For Intl = 0 To 2
        If ButCelda(3 * Intl).Caption <> vbNullString And _
            ButCelda(3 * Intl).Caption = ButCelda(3 * Intl + 1).Caption And _
            ButCelda(3 * Intl).Caption = ButCelda(3 * Intl + 2).Caption Then
            HayTresEnRaya = True
            Exit Function
        End If
    Next

    ' Filas verticales
    For Intl = 0 To 2
        If ButCelda(Intl).Caption <> vbNullString And _
```

```

        ButCelda(IntI).Caption = ButCelda(3 + IntI).Caption And _
        ButCelda(IntI).Caption = ButCelda(6 + IntI).Caption Then
        HayTresEnRaya = True
        Exit Function
    End If
Next

' Diagonales
If ButCelda(0).Caption <> vbNullString And _
    ButCelda(0).Caption = ButCelda(4).Caption And _
    ButCelda(0).Caption = ButCelda(8).Caption Then
    HayTresEnRaya = True
    Exit Function
End If
If ButCelda(2).Caption <> vbNullString And _
    ButCelda(2).Caption = ButCelda(4).Caption And _
    ButCelda(2).Caption = ButCelda(6).Caption Then
    HayTresEnRaya = True
    Exit Function
End If

HayTresEnRaya = False
End Function

Public Sub InicializarFormulario()
    Dim IntI As Integer

    ' Limpiamos las celdas
    For IntI = 0 To 8
        ButCelda(IntI).Caption = vbNullString
    Next

    ' Inicializamos el StrTurno
    StrTurno = "X"

    ' Inicializamos el número de movimientos
    IntNumeroJugadas = 1
End Sub

Private Sub UserControl_Initialize()
    Dim IntI, IntJ As Integer

    ' Creamos los nuevos botones
    For IntI = 1 To 8
        Load ButCelda(IntI)
        ButCelda(IntI).Visible = True
    Next

    ' Colocamos los botones
    For IntI = 0 To 2
        ' Colocamos el botón izquierdo de la fila correspondiente
        ButCelda(3 * IntI).Top = ButCelda(3 * IntI).Top + IntI * ButCelda(0).Height

        ' Colocamos los dos botones restantes de la fila
        For IntJ = 1 To 2
            ButCelda(3 * IntI + IntJ).Top = ButCelda(3 * IntI).Top
            ButCelda(3 * IntI + IntJ).Left = ButCelda(3 * IntI + IntJ - 1).Left + ButCelda(3 * IntI + IntJ - 1).Width
        Next
    Next

    ' Inicializamos el tablero
    InicializarFormulario
End Sub

```

## 6. Acceso a bases de datos (DAO y Jet)

### Introducción

En Visual Basic existen múltiples métodos de acceso a bases de datos. En este capítulo, y en los dos siguientes, se pretende familiarizar al lector con tres de los métodos más utilizados para acceso a bases de datos, tanto locales como remotas. Utilizar uno u otro depende de nuestras necesidades: unos métodos son más aptos para aplicaciones sencillas sin conexión a bases de datos externas, mientras que otros aportan un mayor rendimiento al ser utilizados contra servidores inteligentes como SQL Server.

Visual Basic proporciona también un conjunto de controles *enlazados a datos* (como los controles **DBGrid**, **DBCombo**, **DBList**, **RemoteData**...) que facilitan el acceso y el procesamiento de la información almacenada en una base de datos. Dichos controles no van a ser tratados en este manual, por dos razones: la primera razón es que estos controles no nos permiten definir su funcionamiento para controlar cómo acceden a la base de datos, lo que en la mayoría de las ocasiones suele ser un requisito fundamental, y la segunda razón es que una vez asimile el lector los tres capítulos que vamos a presentar, no debería tener ningún problema en profundizar por sí mismo, si fuera necesario, en su funcionamiento.

Este capítulo mostrará al lector en qué consiste y cómo se utiliza el primero de los métodos de acceso a datos, DAO (Data Access Objects), para acceder a bases de datos locales a través del motor Jet. De forma paralela se pretende que el lector sea consciente de la ventaja del uso de clases para la reutilización del código.

## Conocimientos teóricos

### DAO y Jet

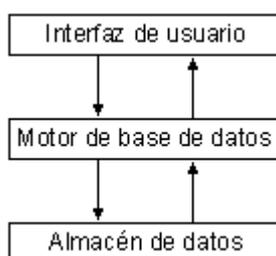
El modelo DAO es una interfaz de programación que posibilita un control total de la base de datos. Es decir, DAO facilita una colección de clases de objetos que proporcionan las propiedades y los métodos que nos permitirán llevar a cabo todas las operaciones necesarias para administrar una base de datos: funciones para crear bases de datos, definir tablas, campos e índices, establecer relaciones entre las tablas, desplazarse por la base de datos y crear consultas sobre ella, etc.

El motor de base de datos **Jet** convierte estas operaciones, definidas sobre objetos de acceso de datos, en operaciones físicas que se efectúan directamente sobre los propios archivos de las bases de datos y que controlan todos los mecanismos de interfaz con las distintas bases de datos compatibles. Hay tres categorías de bases de datos que Visual Basic reconoce a través de DAO y del motor Jet:

- Bases de datos **nativas** de Visual Basic: aquellas que utilizan el mismo formato que Microsoft Access. El motor Jet crea y manipula directamente estas bases de datos, que proporcionan la máxima flexibilidad y velocidad.
- Bases de datos **externas**: las que utilizan el método de acceso secuencial indexado (ISAM), como dBASE III, dBASE IV, Microsoft FoxPro, Paradox, archivos de texto, hojas de cálculo de Microsoft Excel o Lotus, etc.
- Bases de datos **ODBC**: bases de datos cliente-servidor que cumplen el estándar ODBC. Aunque es posible trabajar contra una base de datos utilizando Jet y ODBC, existen otros métodos más recomendables que veremos en los ejemplos 7 y 8.

### Estructura de una aplicación de bases de datos en Visual basic con DAO y Jet

Una aplicación de bases de datos en Visual Basic que utiliza DAO y Jet consta de tres partes, como se muestra en la siguiente figura:



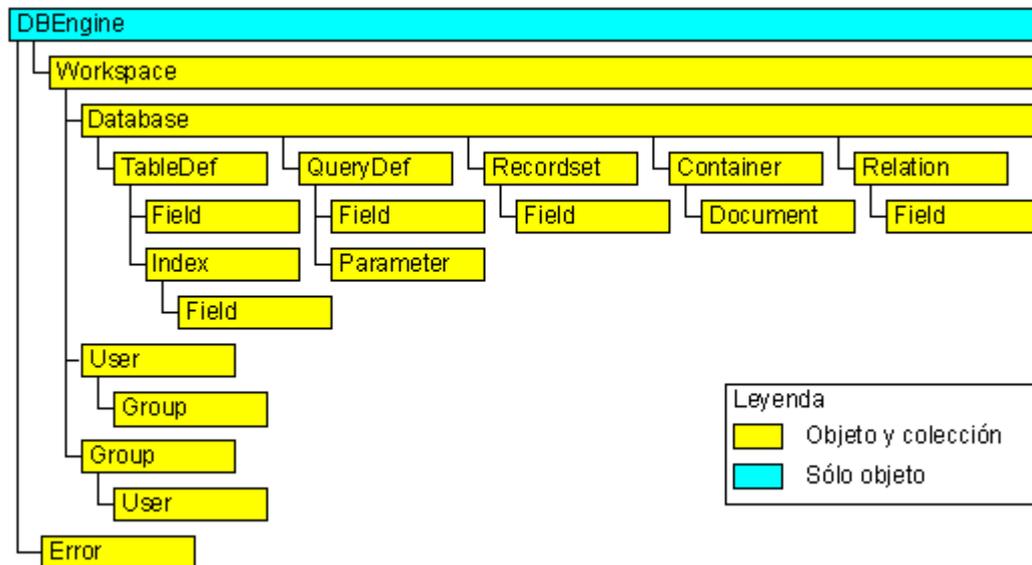
El interfaz de usuario es lo que el usuario ve y con lo que interactúa con la base de datos. Como se ve en la figura, todo acceso a la base de datos se realiza a través del motor de base de datos (Jet) ya que es aquí donde están los métodos y funciones de acceso a base de datos.

El motor de base de datos es un intermediario entre el interfaz de usuario y el almacén de datos, lo que permite abstraer a la aplicación de la base de datos utilizada, facilitando la reutilización del código.

## El modelo de objetos DAO

El modelo de objetos de acceso a datos es la interfaz orientada a objetos del motor de base de datos Jet. Se trata de una jerarquía de clases que corresponde a una visión lógica de un sistema de base de datos relacional (la base de datos propiamente dicha, las tablas definidas en ella y sus campos, índices, etc). Estas clases se utilizan para crear objetos de acceso a datos que se defieren a la base de datos en particular que se desea manipular.

Las clases de los objetos de acceso a datos se organizan en una jerarquía, en la que la mayoría de las clases pertenecen a una clase de colección, que a su vez pertenece a otra clase en la jerarquía. Aquí se muestra la jerarquía completa de DAO.



## Programación con DAO y Jet

En las siguientes líneas vamos a comentar cómo programar una aplicación que acceda a una base de datos Jet a través de DAO, explicando las acciones más comunes que deberemos implementar. Existen muchas posibilidades que no comentaremos. Sin embargo, el ejemplo propuesto es lo suficientemente completo como para servir a los propósitos de la mayoría.

Si programamos con Visual Basic 5.0, es necesario incluir en el proyecto una referencia a **Microsoft DAO 3.5 Object Library**.

Como ejemplo vamos a utilizar una base de datos creada en Access y guardada en 'c:\bd\MiBD.mdb'. La base de datos contiene una sola tabla, MiTabla, cuya estructura es la siguiente:

Nombre del campo	Tipo
DNI	Texto
Nombre	Texto
Apellidos	Texto

Ahora veremos la estructura de un programa en el que se mantiene esta base de datos.

• **Apertura de la base de datos:**

El objeto **DBEngine** contiene y controla todos los otros objetos en la jerarquía de los objetos de DAO. Es único por aplicación (no pueden crearse otros objetos DBEngine). Utilizaremos el **Workspace** (espacio de trabajo) por defecto de este objeto para abrir la base de datos, especificando el *path* completo del archivo:

```
Dim VarBD as Database
Set VarBD = DBEngine.Workspaces(0).OpenDatabase("c:\bd\MiBD.mdb")
```

• **Utilización de Recordsets:**

Un objeto **Recordset** representa los registros de una tabla o los registros que resultan de ejecutar una consulta. Es análogo a los cursores utilizados en otros lenguajes. Existen varios tipos, cada uno de los cuales tiene sus ventajas e inconvenientes:

- **dbOpenTable:** tiene acceso a la base de datos directamente, es decir, que tiene acceso a la tabla directamente. Las búsquedas son más rápidas que con otros tipos de **Recordset**, pero no permite uniones ni puede ser utilizado con otros modelos de acceso a datos como RDO.
- **dbOpenSnapshot:** contiene una copia fija de los datos tal y como existen en el momento en el que se accedió. Consume menos recursos que otros **Recordset** (**dbOpenTable**, **dbOpenDynaset**), por lo que puede ejecutar consultas y devolver datos más rápidamente, sobre todo si utiliza ODBC. No es actualizable, por lo que no recoge los cambios que puedan realizarse en la base de datos mientras se utiliza, es óptimo para usos de lectura de datos.
- **dbOpenDynaset:** es un conjunto de referencias a los registros de una o más tablas. Los cambios realizados en la base de datos son automáticamente actualizados en un **Recordset** de este tipo. Sin embargo, las búsquedas son más lentas ya que las acciones que se hacen no solo se ejecutan en la memoria, si no que se reflejan a la vez en la base de datos.
- **dbOpenForwardOnly:** es un tipo de **Snapshot** en el que sólo se permite el desplazamiento hacia delante sobre el conjunto de resultados obtenido en una consulta. Es el **Recordset** que menos funcionalidad proporciona, con la ventaja de que es el más rápido.

En el siguiente cuadro se recogen los tipos de **Recordset** que deben utilizarse para realizar las acciones más comunes, de forma que se maximice en la medida de lo posible el rendimiento de la aplicación:

Acción	Recordset recomendado
Consultar los datos contenidos en una sola tabla	<b>dbOpenTable</b>
Insertar un registro nuevo en una tabla	<b>dbOpenTable</b>
Recorrer un <b>Recordset</b> utilizando <b>MoveNext</b>	<b>dbOpenForwardOnly</b>
Buscar un registro utilizando <b>Find</b>	<b>dbOpenSnapshot</b>
Buscar un registro con <b>Find</b> y borrarlo con <b>Delete</b>	<b>dbOpenDynaset</b>

**Siempre** hay que cerrar los **Recordsets** abiertos una vez dejen de ser útiles utilizando el método **Close**. No cerrar un **Recordset** abierto puede suponer estar trabajando con datos no actualizados, no poder abrir o cerrar transacciones, etc.

En las siguientes líneas se exponen varios ejemplos de uso de **Recordsets** utilizando en muchos casos sentencias **específicas** de DAO para acceder a bases de datos Jet. Este tipo de sentencias están pensadas para el acceso a bases de datos locales a través del motor Jet, y **no deben** ser utilizadas para acceder a servidores de bases de datos no localizados en el cliente (es más, en ocasiones no pueden ser utilizadas fuera de este contexto). Su utilización en este tipo de entornos puede suponer una importante bajada en el rendimiento de la aplicación, siendo preferible el uso de DAO con ODBCDirect o RDO, como veremos en los siguientes ejemplos.

### Ejemplos de utilización de *Recordsets*:

Para insertar un registro nuevo (DNI="12345678", Nombre="Anastasio", Apellidos="Pérez Martín") en la tabla 'MiTabla':

```
Dim RsetTabla as Recordset

Set RsetTabla = VarBD.OpenRecordset("MiTabla",dbOpenTable)
With RsetTabla
    .AddNew
    !DNI = "12345678"
    !Nombre = "Anastasio"
    !Apellidos = "Pérez Martín"
    .Update
End With
```

Para contar el número de elementos en la tabla MiTabla utilizaremos la propiedad **RecordCount** del **Recordset**:

```
Dim RsetTabla as Recordset
Dim NumElementos as integer

Set RsetTabla = VarBD.OpenRecordset("MiTabla",dbOpenTable)
NumElementos = RsetTabla.RecordCount
```

Para mostrar en la pantalla **Debug** los datos de cada registro de la tabla "MiTabla" utilizaremos el método **MoveNext** del **Recordset**, que obtiene el siguiente registro del conjunto. Accederemos a los campos del registro utilizando la sintaxis **Recordset.Abierto!Nombre\_delcampo**. La propiedad **EOF** del **Recordset** nos indicará cuando se ha llegado al final del conjunto de registros:

```
Dim RsetTabla as Recordset

Set RsetTabla = VarBD.OpenRecordset("MiTabla",dbOpenForwardOnly)
With RsetTabla
    Do Until .EOF
        Debug.Print "DNI: " & !DNI & vbCRLF & _
            "Nombre: " & !Nombre & vbCRLF & _
            "Apellidos: " & !Apellidos
        .MoveNext
    Loop
End With
```

Para buscar los datos del registro con DNI="12345678" utilizaremos el método **FindFirst** del **Recordset**, que obtiene el **primer registro** del conjunto de registros del **Recordset** que cumple una cierta condición, una vez encontrado el registro mostraremos los datos en el **debug**. Utilizaremos la propiedad **NoMatch** para saber si se ha encontrado el registro buscado, ya que será **true** si se ha llegado al final del recorset sin encontrar un registro con los datos especificados:

```
Dim RsetTabla as Recordset

Set RsetTabla = VarBD.OpenRecordset("MiTabla",dbOpenSnapshot)
With RsetTabla
    .FindFirst "DNI=12345678"
    If .NoMatch Then
        Debug.Print "Registro no encontrado"
    Else
        Debug.Print "DNI: " & !DNI & vbCRLF & _
            "Nombre: " & !Nombre & vbCRLF & _
```

```
        "Apellidos: " & !Apellidos
    End If
End With
```

Para borrar el registro con DNI="12345678" utilizaremos el método **FindFirst** y el método **Delete** del **Recordset**, que elimina el registro actual del conjunto de registros:

```
Dim RsetTabla as Recordset

Set RsetTabla = VarBD.OpenRecordset("MiTabla",dbOpenSnapshot)
With RsetTabla
    .MoveLast
    .FindFirst "DNI=12345678"
    If Not .NoMatch Then .Delete
End With
```

Hemos visto los métodos de acceso a registros, ahora veremos los métodos de movimiento dentro del recordset, el uso es similar a los de acceso a los registros, por eso sólo los enunciaremos:

**MoveFirst:** El cursor se sitúa en el primer registro.

**MoveLast:** El registro en el que nos situamos es el último del recordset.

**MoveNext:** Nos vamos al siguiente registro al que nos encontremos.

**MovePrevious:** Retrocedemos un registro en el recordset.

**Move #n:** Avanzamos n registros a partir de la posición actual.

Es importante controlar la situación en la que nos encontremos en el recordset a la hora de desplazarnos, ya que el acceso a una posición que no esté dentro del recordset produciría un error en tiempo de ejecución.

## Ejemplo propuesto

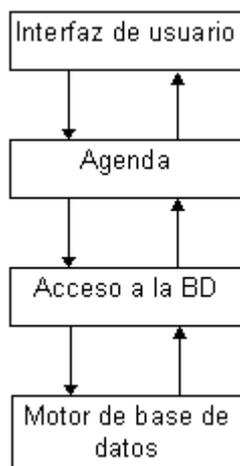
### Objetivo

El objetivo de este ejemplo es ampliar el ejemplo de la agenda de forma que los datos estén ahora almacenados en una base de datos.

La aplicación se diseñará en tres capas:

- *Interfaz de usuario* (FrmAgenda): capa que permite al usuario interactuar con el programa (idéntico al utilizado en la agenda que ya tenemos).
- *Clase agenda* (ClaAgenda): capa intermedia entre el interfaz de usuario y la clase de acceso a la base de datos.
- *Clase de acceso a la base de datos* (ClaDAO\_Jet): capa que se comunica con el motor de base de datos. El objetivo es que esta capa sea lo suficientemente independiente del tipo de acceso a la base de datos elegido (DAO con Jet, DAO con ODBCDirect o RDO) como para permitir que sea sustituido por otro sin cambiar la capa superior.

Cada capa sólo podrá relacionarse con su capa inmediata superior e inferior, como muestra el dibujo:



El objetivo es aislar cada capa de la implementación de las demás, de forma que las capas inferiores puedan ser cambiadas sin que las modificaciones repercutan en las capas superiores<sup>®</sup>. En particular, el objetivo que se pretende en este ejemplo y los dos siguientes es crear una capa de acceso a la base de datos diferente en cada ejemplo, pero utilizando el mismo interfaz de usuario y la misma agenda.

---

<sup>®</sup> Esta independencia entre las diferentes capas es lo que confiere su potencia al modelo de desarrollo estructurado en capas, permitiendo una gran flexibilidad en el desarrollo y mantenimiento de las aplicaciones.

## Desarrollo del ejemplo

Crearemos la base de datos para la agenda (bdAgenda.mdb) en Access, que contendrá una única tabla **Agenda** cuya arquitectura se muestra en la tabla siguiente:

Campo	Tipo	Propiedades
NumFicha	Autonumérico	Campo clave
Nombre	Texto	
Apellidos	Texto	
Direccion	Texto	
Telefono	Texto	

El lector utilizará los siguientes elementos<sup>®</sup>:

- *ClaFicha*: ya definida en anteriores ejemplos.
- *ModDefiniciones*: módulo con las definiciones utilizadas en la aplicación.
- *FrmAgenda*: formulario correspondiente a la capa de interfaz.

Desarrollaremos la clase correspondiente a la capa de agenda (ClaAgenda). Contendrá un objeto privado ObjAccesoBD, de tipo ClaDAO\_Jet, a través del cual podrá interactuar con la base de datos. Esta clase proporcionará los siguientes métodos:

- **Public Property Get IntNumFichas() As Integer**: Método para obtener el número de fichas de la agenda. Llamará al método NumFichas del objeto ObjAccesoBd.
- **Public Function CrearFicha(ByVal StrNombre As String, ByVal StrApellidos As String, ByVal StrDireccion As String, ByVal StrTelefono As String) As Integer**: Método para crear una ficha nueva en la agenda. Creará un objeto de la clase ClaFicha y se lo pasará al método GuardarFicha del objeto ObjAccesoBD.
- **Public Function BorrarFicha(ByVal IntNumFicha As Integer)**: Método para borrar una ficha de la agenda. Utilizará el método BorrarFicha del objeto ObjAccesoBD.
- **Public Function Ficha(Optional ByVal IntIndice As Integer, Optional ByVal IntNumFicha As Integer) As ClaFicha**: Método para obtener una ficha específica de la agenda, ya sea por índice o por número de ficha. Utilizará el método ObtenerFicha del objeto ObjAccesoBD.
- **Private Sub Class\_Terminate()**: Al eliminar la agenda es necesario cerrar la base de datos y eliminar el objeto ObjAccesoBD.

Desarrollaremos también la clase correspondiente a la capa de acceso a la base de datos, que utilizará DAO y Jet (ClaDAO\_Jet). Esta capa proporcionará los siguientes métodos:

- **Public Property Get NumFichas() As Integer**: Método para obtener el número de fichas de la tabla Agenda.
- **Public Function AbrirAgenda() AsCodigoRetorno**: Método para abrir la base de datos.
- **Public Function CerrarAgenda() AsCodigoRetorno**: Método para cerrar la base de datos.

<sup>®</sup> Se recomienda al lector, para facilitar la comprensión de estos capítulos dedicados al acceso a bases de datos desde Visual Basic, que utilice los códigos proporcionados en el manual para estos tres elementos, y desarrolle personalmente las clases que implementan las demás capas.

- **Public Function GuardarFicha(ByVal ObjFicha As ClaFicha, Optional ByRef IntNumFicha As Integer) As CodigoRetorno:** Método para guardar una ficha. Devuelve el número de ficha asignado por la base de datos como campo autonumérico.
- **Public Function ObtenerFicha(ByRef ObjFicha As ClaFicha, Optional ByVal IntIndice As Integer, Optional ByVal IntNumFicha As Integer) As CodigoRetorno:** Método para obtener una ficha específica de la tabla Agenda. Permite buscar por índice o por número de ficha.
- **Public Function BorrarFicha(ByVal IntNumFicha) As CodigoRetorno:** Método para borrar una ficha de la tabla Agenda.
- **Private Sub Class\_Initialize():** Al crear un objeto de esta clase, debe de llamarse automáticamente a su correspondiente método AbrirAgenda.

Seguiremos unas normas sencillas junto con unos comentarios breves, claros y concisos que facilitarán la posterior comprensión del programa.

Todos los métodos de la clase de acceso a la base de datos incluirán al comienzo la sentencia **On error goto TratamientoError**, con su correspondiente sección de tratamiento de error, para interceptar los posibles errores del motor de la base de datos. Además, todas las funciones devolverán un código de retorno especificado por el tipo **CódigoRetorno** definido en el módulo ModDefiniciones, que determinará si han podido realizar correctamente su función.

Los métodos de la clase de acceso a la base de datos utilizarán el motor Jet para acceder a la base de datos, a través de las funciones que DAO proporciona.

### Ejemplo resuelto

Utilizaremos la clase **ClaFicha** ya definida.

### **ModDefiniciones**

```
' Ejemplo 6 del Curso de Visual Basic Avanzado
'
' Archivo con el estándar de devolución de las funciones.
```

```
Option Explicit
```

```
Public Enum CodigoRetorno
    Error = 0
    Ok = 1
End Enum
```

### **FrmAgenda**

```
' Ejemplo 6 del Curso de Visual Basic Avanzado
'
' Interfaz del usuario con la base de datos, aquí están los métodos de mantenimiento de
' la base de datos.
```

```
Option Explicit
```

```
Dim ObjAgenda As New ClaAgenda
Dim IntNumFichaActual As Integer
```

```
Private Sub ButBorrarFicha_Click()
```

```
    ' Confirmamos el borrado
    If MsgBox("¿Desea eliminar la ficha actual?", vbYesNo, "Borrado de ficha") = vbNo Then
        Exit Sub
    End If
```

```
    ' Borramos la ficha actual
    ObjAgenda.BorrarFicha IntNumFichaActual
```

```
    ' Actualizamos el número de elementos de la barra de scroll
```

```
    If ObjAgenda.IntNumFichas > 0 Then
        HScroll1.Max = ObjAgenda.IntNumFichas - 1
    Else
        HScroll1.Max = 0
    End If
```

```
    ' Actualizamos la barra de scroll
    ActualizarFormulario
```

```
End Sub
```

```
Private Sub ButCrearFicha_Click()
```

```
' Creamos una nueva ficha con los datos introducidos por el
' usuario, y obtenemos el código de la ficha creada
IntNumFichaActual = ObjAgenda.CrearFicha(TxtNombre, TxtApellidos, TxtDireccion, _
    TxtTelefono)

' Comprobamos si no ha podido insertar una nueva ficha, lo que
' sucederá si IntNumFichaActual=0
If IntNumFichaActual = 0 Then
    ' Decrementamos el número de elementos de la barra de
    ' scroll
    HScroll1.Max = HScroll1.Max - 1

    ' Actualizamos el formulario
    ActualizarFormulario
    Exit Sub
End If

' Actualizamos el título del frame
Frame1.Caption = "Ficha " & IntNumFichaActual

' Deshabilitamos el botón de crear ficha y el frame
ButCrearFicha.Enabled = False
Frame1.Enabled = False

' Habilitamos los botones de nueva ficha y borrar
' ficha, así como la barra de scroll
ButNuevaFicha.Enabled = True
ButBorrarFicha.Enabled = True
HScroll1.Enabled = True
End Sub

Private Sub ButNuevaFicha_Click()
    Dim IntNumFichas As Integer

    ' Actualizamos la barra de scroll
    IntNumFichas = ObjAgenda.IntNumFichas
    If IntNumFichas = 0 Then
        ' Con la primera ficha creada, habilitamos
        ' la barra de scroll
        HScroll1.Enabled = True
        Frame1.Enabled = True
    Else
        ' Establecemos el número de elementos de
        ' la barra de scroll
        HScroll1.Max = HScroll1.Max + 1
        HScroll1.Value = HScroll1.Max
    End If

    ' Habilitamos el botón de crear ficha y el frame
    ButCrearFicha.Enabled = True
    Frame1.Enabled = True

    ' Deshabilitamos los botones de nueva ficha y borrar
    ' ficha, así como la barra de scroll
    ButNuevaFicha.Enabled = False
    ButBorrarFicha.Enabled = False
    HScroll1.Enabled = False
End Sub

Private Sub Form_Load()
```

```
Dim IntNumFichas As Integer

' Comprobamos si no hay datos en la agenda para
' deshabilitar las opciones
IntNumFichas = ObjAgenda.IntNumFichas
If IntNumFichas = 0 Then
    ' Deshabilitamos el frame, la barra de scroll,
    ' el botón de crear ficha y el botón de borrar ficha
    Frame1.Enabled = False
    HScroll1.Enabled = False
    ButCrearFicha.Enabled = False
    ButBorrarFicha.Enabled = False

    ' Establecemos los límites de la barra de scroll
    HScroll1.Max = 0
Else
    ' Establecemos el máximo de la barra de scroll
    HScroll1.Max = IntNumFichas - 1

    ' Actualizamos el formulario
    ActualizarFormulario
End If
End Sub

Private Sub Form_Unload(Cancel As Integer)
    ' Eliminamos la agenda
    Set ObjAgenda = Nothing
End Sub

Private Sub HScroll1_Change()
    ' Actualizamos la barra de scroll
    ActualizarFormulario
End Sub

Private Function ActualizarFormulario()
    Dim ObjFicha As ClaFicha

    ' Comprobamos si no hay fichas o estamos creando una ficha nueva
    If HScroll1.Max = ObjAgenda.IntNumFichas Then
        ' Limpiamos los datos del frame
        TxtNombre = vbNullString
        TxtApellidos = vbNullString
        TxtDireccion = vbNullString
        TxtTelefono = vbNullString

        ' Actualizamos el título del frame
        Frame1.Caption = vbNullString

        ' Deshabilitamos el botón de borrado
        ButBorrarFicha.Enabled = False
    Else
        ' Obtenemos la ficha correspondiente de la agenda
        ' (buscamos por índice, no por código de ficha)
        Set ObjFicha = New ClaFicha
        Set ObjFicha = ObjAgenda.Ficha(HScroll1.Value + 1)

        ' Sacamos los datos de la ficha
        If Not ObjFicha Is Nothing Then
            TxtNombre = ObjFicha.StrNombre
            TxtApellidos = ObjFicha.StrApellidos
        End If
    End If
End Function
```

```
TxtDireccion = ObjFicha.StrDireccion
TxtTelefono = ObjFicha.StrTelefono

' Actualizamos el código de la ficha actual
IntNumFichaActual = ObjFicha.IntNumFicha

' Eliminamos el objeto ficha creado
Set ObjFicha = Nothing

' Actualizamos el frame
Frame1.Caption = "Ficha " & IntNumFichaActual
End If
End If
End Function
```

## ClaAgenda

```
'
' Ejemplo 6 del Curso de Visual Basic Avanzado
'
' Métodos para acceder a los métodos de la clase, que serán los que tienen acceso a la
' base de datos.
'
Option Explicit

Private ObjAccesoBD As New ClaDAO_Jet 'Objeto de acceso a la base de datos

' Método para obtener el número de fichas de la agenda
Public Property Get IntNumFichas() As Integer
    IntNumFichas = ObjAccesoBD.NumFichas
End Property

' Método para crear una ficha nueva en la agenda
Public Function CrearFicha(ByVal StrNombre As String, _
    ByVal StrApellidos As String, ByVal StrDireccion As String, _
    ByVal StrTelefono As String) As Integer
    Static IntNumFicha As Integer
    Dim ObjFicha As ClaFicha

    ' Creamos un nuevo objeto de la clase ficha
    Set ObjFicha = New ClaFicha

    ' Establecemos las propiedades de la nueva ficha
    ObjFicha.StrNombre = StrNombre
    ObjFicha.StrApellidos = StrApellidos
    ObjFicha.StrDireccion = StrDireccion
    ObjFicha.StrTelefono = StrTelefono

    ' Insertamos la nueva ficha en la base de datos
    If ObjAccesoBD.GuardarFicha(ObjFicha, IntNumFicha) = Ok Then
        ' Devolvemos el código de la ficha creada
        CrearFicha = IntNumFicha
    Else
        ' Devolvemos 0, porque es un valor que nunca aparecerá
        ' en un campo autonumérico
        CrearFicha = 0
        MsgBox "Imposible agregar una nueva ficha"
    End If
End Function
```

```
' Eliminamos el objeto Ficha creado
Set ObjFicha = Nothing
End Function

' Método para borrar una ficha de la agenda
Public Function BorrarFicha(ByVal IntNumFicha As Integer)
    Dim ObjFicha As ClaFicha

    If ObjAccesoBD.BorrarFicha(IntNumFicha) = Error Then
        MsgBox "Imposible borrar la ficha nº " & IntNumFicha
    End If

End Function

' Al eliminar la agenda, cerraremos la base de datos y
' eliminaremos el objeto de acceso a la base de datos
Private Sub Class_Terminate()
    If ObjAccesoBD.CerrarAgenda() = Error Then
        MsgBox "Imposible cerrar la agenda"
    End If
    Set ObjAccesoBD = Nothing
End Sub

' Método para obtener una ficha específica de la agenda, ya sea
' por índice o por número de ficha
'
' NOTA: la función crea un nuevo objeto Ficha, que habrá que
' eliminar en la función de llamada
Public Function Ficha(Optional ByVal IntIndice As Integer, _
    Optional ByVal IntNumFicha As Integer) As ClaFicha
    Dim ObjFicha As New ClaFicha

    ' Comprobamos si se busca por índice
    If Not IsMissing(IntIndice) Then
        If ObjAccesoBD.ObtenerFicha(ObjFicha, IntIndice) = Error Then
            MsgBox "Imposible obtener la ficha de la posición nº " & IntIndice
            Exit Function
        End If
    ElseIf Not IsMissing(IntNumFicha) Then
        If ObjAccesoBD.ObtenerFicha(ObjFicha, , IntNumFicha) = Error Then
            MsgBox "Imposible obtener la ficha nº " & IntNumFicha
            Exit Function
        End If
    End If
    Set Ficha = ObjFicha
    Set ObjFicha = Nothing
End Function
```

## ClaDAO\_Jet

```
'
' Ejemplo 6 del Curso de Visual Basic Avanzado
'
' Clase de acceso a bases de datos utilizando DAO y
' funciones EXCLUSIVAS de Jet
'
```

```
Option Explicit

Private VarBD As Database
Private VarPathBD As String
Private Const ConNombreTabla = "Agenda"

Public Property Get NumFichas() As Integer
    On Error GoTo TratamientoError
    Dim RsetTabla As Recordset

    ' Abrimos un recordset de tipo dbOpentable
    Set RsetTabla = VarBD.OpenRecordset(ConNombreTabla, dbOpenTable)

    ' Obtenemos el número de elementos
    NumFichas = RsetTabla.RecordCount

    ' Cerramos el Recordset
    RsetTabla.Close

Exit Property

TratamientoError:
    NumFichas = -1
End Property

Public Function AbrirAgenda() As CodigoRetorno
    On Error GoTo TratamientoError

    ' Abrimos la base de datos, utilizando el Workspace
    ' por defecto
    Set VarBD = DBEngine.Workspaces(0).OpenDatabase(VarPathBD)

    AbrirAgenda = Ok
Exit Function

TratamientoError:
    AbrirAgenda = Error
End Function

Public Function CerrarAgenda() As CodigoRetorno
    On Error GoTo TratamientoError

    ' Cerramos la base de datos
    VarBD.Close

    CerrarAgenda = Ok
Exit Function

TratamientoError:
    CerrarAgenda = Error
End Function

Public Function GuardarFicha(ByVal ObjFicha As ClaFicha, _
    Optional ByRef IntNumFicha As Integer) As CodigoRetorno
    On Error GoTo TratamientoError
    Dim RsetTabla As Recordset

    ' Abrimos un recordset de tipo dbOpenTable (sólo funciona
    ' en Jet)
    Set RsetTabla = VarBD.OpenRecordset(ConNombreTabla, dbOpenTable)
```

```

With RsetTabla
    ' Añadimos una nueva ficha
    .AddNew
    !Nombre = ObjFicha.StrNombre
    !Apellidos = ObjFicha.StrApellidos
    !Direccion = ObjFicha.StrDireccion
    !Telefono = ObjFicha.StrTelefono

    ' Obtenemos el campo NumFicha
    IntNumFicha = !NumFicha
    .Update
End With

' Cerramos el recordset
RsetTabla.Close

GuardarFicha = Ok
Exit Function

TratamientoError:
    GuardarFicha = Error
End Function

Public Function ObtenerFicha(ByRef ObjFicha As ClaFicha, _
    Optional ByVal IntIndice As Integer, _
    Optional ByVal IntNumFicha As Integer) As CodigoRetorno
    On Error GoTo TratamientoError
    Dim RsetTabla As Recordset

    ' Comprobamos si hay que buscar por índice
    If Not IsMissing(IntIndice) Then
        ' Abrimos un recordset de tipo dbOpenForwardOnly, para poder
        ' movernos con MoveNext (lo abrimos ForwardOnly porque sólo
        ' necesitamos desplazarnos hacia delante)
        Set RsetTabla = VarBD.OpenRecordset(ConNombreTabla, dbOpenForwardOnly)

        ' Recorremos el Recordset hasta encontrar la ficha indicada
        With RsetTabla
            Do Until .EOF
                If .RecordCount = IntIndice Then
                    Exit Do
                End If
                .MoveNext
            Loop
        End With

        ' Comprobamos si hay que buscar por número de ficha
        ElseIf Not IsMissing(IntNumFicha) Then
            ' Abrimos un recordset de tipo dbOpenSnapshot, para poder
            ' buscar con FindFirst
            Set RsetTabla = VarBD.OpenRecordset(ConNombreTabla, dbOpenSnapshot)
            With RsetTabla
                ' Llenamos el Recordset
                .MoveLast

                ' Encontramos el primer registro con ese número
                .FindFirst "NumFicha=" & IntNumFicha
            End With
        End If
    End If

```

```

' Comprobamos si la hemos encontrado, o hemos llegado al
' final del Recordset sin encontrarla
With RsetTabla
    If .EOF Or .NoMatch = True Then
        ObtenerFicha = Error
    Else
        ' Guardamos los datos obtenidos
        ObjFicha.StrNombre = !Nombre
        ObjFicha.StrApellidos = !Apellidos
        ObjFicha.StrDireccion = !Direccion
        ObjFicha.StrTelefono = !Telefono
        ObjFicha.IntNumFicha = !NumFicha
        ObtenerFicha = Ok
    End If
End With

' Cerramos el Recordset
RsetTabla.Close

Exit Function

TratamientoError:
    ObtenerFicha = Error
End Function

Public Function BorrarFicha(ByVal IntNumFicha) As CodigoRetorno
    On Error GoTo TratamientoError

    Dim RsetTabla As Recordset

    ' Abrimos un recordset de tipo dbOpenDynaset, para poder
    ' buscar con FindFirst y borrar con Delete
    Set RsetTabla = VarBD.OpenRecordset(ConNombreTabla, dbOpenDynaset)

    ' Buscamos la ficha correspondiente
    With RsetTabla
        ' Llenamos el Recordset
        .MoveLast

        ' Encontramos el primer registro con ese número
        .FindFirst "NumFicha=" & IntNumFicha

        ' Comprobamos si hemos encontrado algún registro
        If .NoMatch Then
            BorrarFicha = Error
        Else
            ' Borraremos la ficha obtenida
            .Delete
            BorrarFicha = Ok
        End If

        ' Cerramos el Recordset
        .Close
    End With

    BorrarFicha = Ok
Exit Function

TratamientoError:
    BorrarFicha = Error

```

```
End Function

Private Sub Class_Initialize()
    ' Establecemos el path donde está el archivo de
    ' la base de datos
    VarPathBD = App.Path & "\bdAgenda.mdb"

    ' Abrimos la base de datos
    AbrirAgenda
End Sub
```

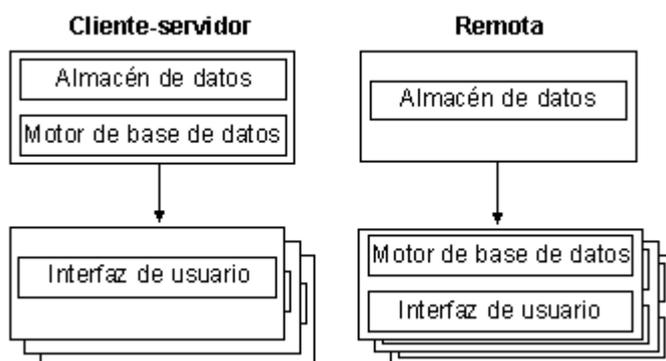
## 7. Acceso a bases de datos (DAO y ODBCDirect)

### Introducción

Hasta ahora hemos explicado cómo podemos acceder a una base de datos en local a través del motor Jet. En este capítulo y en el siguiente mostraremos cómo se puede acceder a un servidor de bases de datos.

Al trabajar con bases de datos localizadas en un servidor podemos diferenciar dos métodos de trabajo, dependiendo de la localización del motor de base de datos:

- **cliente-servidor:** el motor de base de datos está en el servidor.
- **remota:** el motor está en el cliente



Normalmente se utiliza el modelo cliente-servidor para acceder a bases de datos localizadas en un servidor, ya que aporta bastantes ventajas:

- Operaciones más fiables y robustas, puesto que existe un único servidor de base de datos que interactúa con todos los clientes.
- Notable aumento del rendimiento de algunas operaciones, especialmente cuando las estaciones de trabajo de los usuarios son equipos de gama baja.
- Reducción del tráfico de la red gracias a una transmisión de datos más eficiente. Sólo se transfieren los datos que la aplicación necesita.
- Características críticas como los registros de transacciones, las capacidades de copia de seguridad complejas, las matrices de discos redundantes y las herramientas de recuperación de fallos.

La forma más común de acceder a un servidor de bases de datos es a través de ODBC. ODBC es una capa intermedia entre las aplicaciones que se ejecutan en el cliente y el servidor de bases de datos. El controlador ODBC del cliente recibe peticiones de la aplicación, las traduce a peticiones ODBC y las envía al servidor. El servidor responde al controlador ODBC del cliente, y éste pasa la respuesta a la aplicación. La ventaja de usar ODBC es la independencia del SGBD (Sistema Gestor de Bases de Datos) utilizado, pudiendo cambiar éste realizando cambios mínimos en el código de la aplicación.

## Conocimientos teóricos

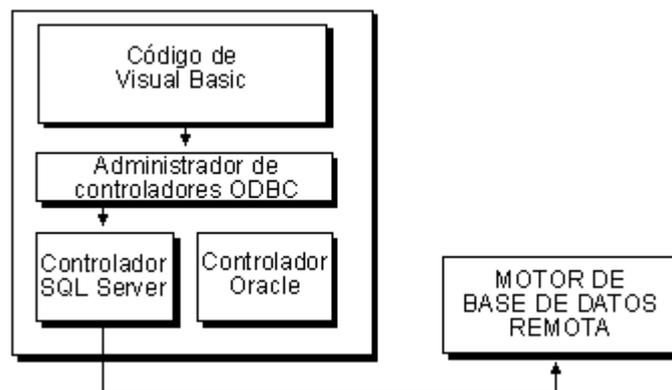
### Opciones de utilización de ODBC con DAO

Existen dos formas de acceder a orígenes de datos remotos utilizando DAO:

- **Jet-ODBC:** utiliza el motor Jet para acceder al SGBD.
- **ODBCDirect:** no utiliza el motor Jet. En realidad no es más que un interfaz DAO con las bibliotecas de RDO, lo que permite un rendimiento mayor que al utilizar Jet-ODBC con la pega de que no puede utilizar todas las características de Jet.

### Estructura de una aplicación de bases de datos en Visual basic con DAO y ODBCDirect

Una aplicación de bases de datos en Visual Basic que utiliza DAO y ODBCDirect se comunica con el SGBD a través del controlador ODBC correspondiente, como se muestra en la figura:



### Programación con DAO y ODBCDirect

En las siguientes líneas vamos a comentar cómo programar una aplicación que acceda a una base de datos a través de DAO y ODBCDirect, explicando las acciones más comunes que deberemos implementar. Al igual que en el capítulo anterior, existen muchas posibilidades que no comentaremos.

Si programamos con Visual Basic 5.0, es necesario incluir en el proyecto una referencia a **Microsoft DAO 3.5 Object Library**.

Como ejemplo utilizaremos la base de datos descrita en el capítulo anterior. Para acceder a dicha base de datos necesitamos crear un DSN (Data Source Name) desde **Panel de Control → ODBC 32 bits**, que llamaremos MiDSN.

#### Variables utilizadas:

En los siguientes ejemplos utilizaremos dos variables, VarWks y VarConexion, para definir el espacio de trabajo y la conexión con el SGBD, respectivamente. Además, para establecer la cadena de conexión ODBC utilizaremos la constante ConCadenaConexion:

```
Dim VarWks As Workspace
Dim VarConexion As Connection
Dim Const ConCadenaConexion = "ODBC;DATABASE=MiBD;UID=;PWD=;DSN=MiDSN"
```

### Conexión con el SGBD:

Primero necesitamos crear un espacio de trabajo específico para ODBC utilizando la sentencia **CreateWorkspace**. Una vez creado, estableceremos la conexión con el SGBD utilizando el método **OpenConnection** de dicho espacio de trabajo.

```
Set VarWks = CreateWorkspace("", "", "", dbUseODBC)
Set VarConexion = VarWks.OpenConnection("", , , ConCadenaConexion)
```

### Utilización de Recordsets:

Los **Recordsets** creados con ODBCdirect son, en realidad, **Resultsets** de RDO. La forma de trabajar con estos **Recordsets** será crear la consulta SQL que genere el conjunto de registros buscado y pasársela al método **OpenRecordset** de la conexión creada.

### Ejemplos de utilización de Recordsets:

Para sacar en la pantalla **Debug** los datos de cada registro de la tabla "MiTabla" utilizaremos el método **OpenRecordset** de la conexión, pasándole la consulta SQL que obtiene todos los datos de la tabla. Para acceder a cada uno de los campos del registro actual, utilizaremos la colección **Fields** del **Recordset**, que contiene todos los campos del registro actual.

```
Dim RsetDatos As Recordset

Set RsetDatos = VarConexion.OpenRecordset("select * from MiTabla")

With RsetDatos
    Do While Not .EOF
        Debug.Print "DNI: " & .Fields(0).Value & vbCRLF & _
            "Nombre: " & .Fields(1).Value & vbCRLF & _
            "Apellidos: " & .Fields(2).Value
        .MoveNext
    Loop
End With
```

### Sentencias que no devuelven datos:

Cuando queremos realizar una acción sobre la base de datos que no devuelve ningún conjunto de registros (insert, update, delete, etc) utilizaremos el método **Execute** de la conexión, pasándole la sentencia SQL a ejecutar. Para determinar si la acción se ha realizado correctamente, consultaremos la propiedad **RowsAffected** de la conexión.

### Ejemplos de ejecución de sentencias que no devuelven datos:

Por ejemplo, si queremos eliminar el registro con DNI="12345678":

```
VarConexion.Execute "delete from MiTabla where DNI='12345678'"
If VarConexion.RecordsAffected = 1 Then
    MsgBox "Registro borrado"
Else
    MsgBox "Error al borrar el registro"
End If
```

## Ejemplo propuesto

### Objetivo

El objetivo de este ejemplo es cambiar la capa de acceso a la base de datos desarrollada en el ejemplo anterior para que utilice DAO y ODBCDirect en lugar de Jet para acceder a la misma base de datos, pero a través de ODBC.

### Desarrollo del ejemplo

Crearemos un DSN para la base de datos (se llamará "Agenda"), y cambiaremos la clase ClaDAO\_Jet por otra clase ClaDAO\_ODBCDirect, que constará de los mismos métodos, pero utilizará ODBCDirect para acceder a través de ODBC a la base de datos.

Una vez creada esta clase, el único cambio a realizar en la aplicación es sustituir en la clase agenda (ClaAgenda) la sentencia

```
Private ObjAccesoBD As New ClaDAO_Jet
```

por

```
Private ObjAccesoBD As New ClaDAO_ODBCDirect
```

### Ejemplo resuelto

Utilizaremos el formulario **FrmAgenda**, el módulo **ModDefiniciones** y las clase **ClaFicha** y **ClaAgenda** (con la modificación comentada).

#### **ClaDAO\_ODBCDirect**

```
'
' Ejemplo 7 del Curso de Visual Basic Avanzado
'
' Clase de acceso a bases de datos utilizando DAO_ODBCDirect
'
'
' Clase de acceso a bases de datos utilizando DAO y
' ODBCDirect

Option Explicit

Private VarWks As Workspace
Private VarConexion As Connection
Private Const ConCadenaConexion = "ODBC;DATABASE=Agenda;UID=;PWD=;DSN=Agenda"

Public Property Get NumFichas() As Integer
    On Error GoTo TratamientoError
    Dim RsetDatos As Recordset
    Dim CadenaConexion As String

    ' Construimos la sentencia SQL
    CadenaConexion = "select count(*) from agenda"

    ' Ejecutamos la consulta
    Set RsetDatos = VarConexion.OpenRecordset(CadenaConexion)

    ' Obtenemos el número de fichas
    NumFichas = RsetDatos.Fields(0).Value

    ' Cerramos el Recordset
    RsetDatos.Close

    Exit Property

TratamientoError:
    NumFichas = -1
End Property

Public Function AbrirAgenda() As CodigoRetorno
    On Error GoTo TratamientoError

    ' Creamos el objeto Workspace de ODBCDirect
    Set VarWks = CreateWorkspace("", _
        "", "", dbUseODBC)

    ' Abrimos la conexión utilizando la cadena
    ' de conexión predefinida
    Set VarConexion = VarWks.OpenConnection("", , , _
        ConCadenaConexion)
```

```

AbrirAgenda = Ok
Exit Function

TratamientoError:
    AbrirAgenda = Error
End Function

Public Function CerrarAgenda() As CodigoRetorno
    On Error GoTo TratamientoError

    ' Cerramos la base de datos
    VarConexion.Close

    ' Cerramos el Workspace
    VarWks.Close

    CerrarAgenda = Ok
    Exit Function

TratamientoError:
    CerrarAgenda = Error
End Function

Public Function GuardarFicha(ByVal ObjFicha As ClaFicha, _
    Optional ByRef IntNumFicha As Integer) As CodigoRetorno
    On Error GoTo TratamientoError
    Dim RsetDatos As Recordset
    Dim CadenaSQL As String

    ' Construimos la sentencia SQL
    CadenaSQL = "insert into agenda " & _
        "(Nombre, Apellidos, Direccion, Telefono) " & _
        "values (" & _
        """" & ObjFicha.StrNombre & """, " & _
        """" & ObjFicha.StrApellidos & """, " & _
        """" & ObjFicha.StrDireccion & """, " & _
        """" & ObjFicha.StrTelefono & """)"

    ' Ejecutamos la consulta
    VarConexion.Execute CadenaSQL

    ' Comprobamos si se ha realizado la inserción
    If VarConexion.RecordsAffected <> 1 Then
        GuardarFicha = Error
        Exit Function
    End If

    ' Obtenemos el número de ficha asignado por el servidor
    ' NOTA: en realidad, dado que el campo clave es el número
    ' de ficha, esta sentencia puede devolver más de un
    ' registro, pero no lo tendremos en cuenta
    CadenaSQL = "select numficha from agenda " & _
        "where " & _
        "Nombre=" & ObjFicha.StrNombre & " AND " & _
        "Apellidos=" & ObjFicha.StrApellidos & " AND " & _
        "Direccion=" & ObjFicha.StrDireccion & " AND " & _
        "Telefono=" & ObjFicha.StrTelefono & ""

    Set RsetDatos = VarConexion.OpenRecordset(CadenaSQL)

```

```

If Not RsetDatos.EOF Then
    ' Obtenemos el código de la nueva ficha creada
    IntNumFicha = RsetDatos.Fields(0).Value
    GuardarFicha = Ok
Else
    GuardarFicha = Error
End If

' Cerramos el Recordset
RsetDatos.Close
Exit Function

TratamientoError:
    GuardarFicha = Error
End Function

Public Function ObtenerFicha(ByRef ObjFicha As ClaFicha, _
    Optional ByVal IntIndice As Integer, _
    Optional ByVal IntNumFicha As Integer) As CodigoRetorno
    On Error GoTo TratamientoError
    Dim RsetDatos As Recordset
    Dim CadenaConexion As String
    Dim IntFichaActual As Integer

    ' Comprobamos si hay que buscar por índice
    If Not IsMissing(IntIndice) Then
        ' Abrimos un recordset seleccionando todos los
        ' registros de la tabla
        CadenaConexion = "select * from agenda order by numficha"
        Set RsetDatos = VarConexion.OpenRecordset(CadenaConexion)

        ' Recorremos el Recordset hasta encontrar la ficha indicada
        With RsetDatos
            Do While Not .EOF
                IntFichaActual = IntFichaActual + 1
                If IntFichaActual = IntIndice Then
                    Exit Do
                End If
                .MoveNext
            Loop
        End With

        ' Comprobamos si hay que buscar por número de ficha
        ElseIf Not IsMissing(IntNumFicha) Then
            ' Abrimos un recordset seleccionando el registro
            ' directamente
            CadenaConexion = "select * from agenda " & _
                "where numficha=" & IntNumFicha & _
                "order by numficha"
            Set RsetDatos = VarConexion.OpenRecordset(CadenaConexion)
        End If

        ' Comprobamos si la hemos encontrado, o hemos llegado al
        ' final del Recordset sin encontrarla
        With RsetDatos
            If .EOF Then
                ObtenerFicha = Error
            Else
                ' Guardamos los datos obtenidos
                ObjFicha.IntNumFicha = .Fields(0).Value
                ObjFicha.StrNombre = .Fields(1).Value
            End If
        End With
    End Function

```

```
ObjFicha.StrApellidos = .Fields(2).Value
ObjFicha.StrDireccion = .Fields(3).Value
ObjFicha.StrTelefono = .Fields(4).Value
ObtenerFicha = Ok
End If
End With

' Cerramos el Recordset
RsetDatos.Close

Exit Function

TratamientoError:
    ObtenerFicha = Error
End Function

Public Function BorrarFicha(ByVal IntNumFicha) As CodigoRetorno
    On Error GoTo TratamientoError

    Dim CadenaSQL As String

    ' Construimos la sentencia SQL
    CadenaSQL = "delete from agenda " & _
        "where numficha=" & IntNumFicha

    ' Ejecutamos la sentencia SQL
    VarConexion.Execute CadenaSQL

    ' Comprobamos cuantas filas han sido afectadas
    If VarConexion.RecordsAffected = 1 Then
        BorrarFicha = Ok
    Else
        BorrarFicha = Error
    End If
    Exit Function

TratamientoError:
    BorrarFicha = Error
End Function

Private Sub Class_Initialize()
    ' Conectamos con la base de datos
    AbrirAgenda
End Sub
```

## 8. Acceso a bases de datos (RDO)

### Introducción

Como vimos en el ejemplo anterior, el acceso a servidores de bases de datos suele realizarse a través de ODBC. RDO (Remote Data Objects) es una fina capa de objetos sobre el API de ODBC, que permite utilizar llamadas al API de ODBC de forma sencilla.

RDO está diseñado para aprovechar al máximo la potencia de servidores de bases de datos inteligentes, especialmente SQL Server. Tiene multitud de opciones que permiten incrementar el rendimiento de las aplicaciones que utilicen RDO, por eso es el más utilizado en cliente-servidor.

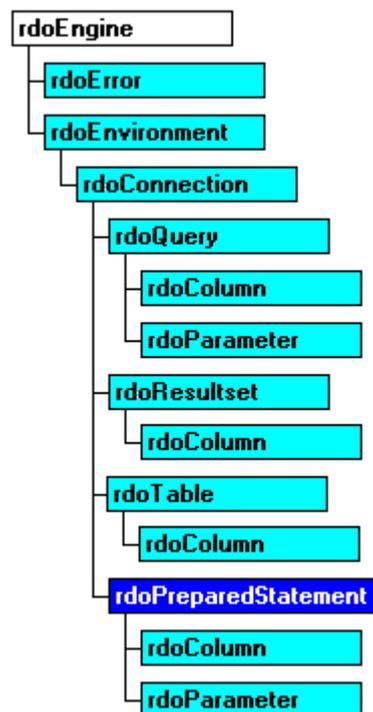
En realidad ya conocemos a grandes rasgos cómo funciona RDO, puesto que ODBCDirect es sólo una forma de utilizar RDO desde DAO.

**Conocimientos teóricos**

**El modelo de objetos RDO**

Los objetos y las colecciones de RDO proporcionan un marco para utilizar código con el fin de crear y manipular componentes de un sistema de base de datos remota de ODBC.

Al igual que DAO, las clases de los objetos de acceso a datos se organizan en una jerarquía, en la que la mayoría de las clases pertenecen a una clase de colección, que a su vez pertenece a otra clase en la jerarquía. La figura de la derecha muestra la jerarquía completa de RDO.



**Estructura de una aplicación de bases de datos en Visual basic con RDO**

La estructura de una aplicación de bases de datos en Visual Basic que utiliza RDO es casi idéntica a la de la aplicaciones que utilizan DAO con ODBCdirect. Los modelos DAO y RDO son equivalentes en cuanto a objetos, la diferencia es que están programados internamente para trabajar a través de ODBC o en local, pero para el programador, conociendo la manera de programar con uno, le debe ser fácil trabajar con el otro. En la siguiente tabla se recogen las correspondencias entre los objetos utilizados en RDO y sus objetos equivalentes en DAO:

Objeto de RDO	Tipo (si aplicable)	Objeto de DAO equivalente	Tipo (si aplicable)
rdoEngine		DBEngine	
rdoError		Error	
rdoEnvironment		Workspace	
rdoConnection		Database	
rdoTable		TableDef	
No está implementado		Index	
rdoResultset		Recordset	
	No implementado		Tipo table
	Tipo Keyset		Tipo Dynaset
	Tipo static (l/e)		Tipo Snapshot (s/l)
	Tipo dynamic		(ninguno)
	Tipo forward-only		Tipo forward-only
	(sin cursor)		(ninguno)
rdoColumn		Field	
rdoQuery		QueryDef	
rdoParameter		Parameter	
No implementado		Relation	
No implementado		Group	
No implementado		User	

## Programación con RDO

En las siguientes líneas vamos a comentar cómo programar una aplicación que acceda a una base de datos a través de RDO, explicando las acciones más comunes que deberemos implementar. En los dos capítulos anteriores comentamos que existen muchas posibilidades que no hemos tratado. Esto se aplica especialmente a RDO, pues proporciona muchas posibilidades en función del SGBD con el que trabajemos (sobre todo con SQL Server), como consultas asíncronas, cursores del lado del servidor, ejecución de procedimientos en el servidor, etc.

Si programamos con Visual Basic 5.0, es necesario incluir en el proyecto una referencia a **Microsoft Remote Data Object 2.0**.

Utilizaremos la misma base de datos descrita en el capítulo anterior, así como el mismo DSN.

### Variables utilizadas:

En los siguientes ejemplos utilizaremos una variable `VarConexion` que representará la conexión con el SGBD y la misma constante para la cadena de conexión ODBC que utilizamos en el capítulo anterior:

```
Dim VarConexion As New rdoConnection
Dim Const ConCadenaConexion = "ODBC;DATABASE=MiBD;UID=;PWD=;DSN=MiDSN"
```

### Conexión con el SGBD:

Estableceremos la conexión con el SGBD utilizando el método **EstablishConnection** de la conexión. Antes de abrir la conexión, tenemos que especificar el tipo de cursor a utilizar utilizando la propiedad **CursorDriver** de la conexión (especificaremos **rdUseOdbc**), así como la cadena de conexión usando la propiedad **Connect**:

```
With VarConexion
    .CursorDriver = rdUseOdbc
    .Connect = ConCadenaConexion
    .EstablishConnection rdDriverNoPrompt
End With
```

### Utilización de Resultsets:

Como ya dijimos en el capítulo anterior, los **Recordsets** creados con `ODBCDirect` son en realidad **Resultsets** de RDO. Por lo tanto, la forma de trabajar con estos **Resultsets** será muy semejante a la que ya conocemos. Para abrir un **Resultset**, crearemos la consulta SQL que genere el conjunto de registros buscado y utilizaremos el método **OpenResultset** de la conexión para lanzarla.

### Ejemplo de utilización de Resultsets:

Para sacar en la pantalla **Debug** los datos de cada registro de la tabla "MiTabla" utilizaremos el método **OpenResultset** de la conexión, pasándole la consulta SQL que obtiene todos los datos de la tabla. Para acceder a cada uno de los campos del registro actual, utilizaremos la colección **rdoColumns** del **Resultset**, que contiene todos los campos del registro actual.

```
Dim RsetDatos As rdoResultset

Set RsetDatos = VarConexion.OpenResultset("select * from MiTabla")

With RsetDatos
    Do While Not .EOF
        Debug.Print "DNI: " & .rdoColumns(0).Value & vbCRLF & _
```

```
        "Nombre: " & .rdoColumns(1).Value & vbCRLF & _  
        "Apellidos: " & .rdoColumns(2).Value  
        .MoveNext  
    Loop  
End With
```

**Sentencias que no devuelven datos:**

La forma de ejecutar sentencias que no devuelven datos en RDO es idéntica a la que ya hemos visto con ODBCDirect.

## Ejemplo propuesto

### Objetivo

El objetivo de este ejemplo es cambiar la capa de acceso a la base de datos desarrollada en el ejemplo anterior para que utilice RDO en lugar de DAO y ODBCdirect para acceder a la misma base de datos a través de ODBC.

### Desarrollo del ejemplo

Utilizaremos el DSN creado en el ejemplo anterior y cambiaremos la clase ClaDAO\_ODBCDirect por una nueva clase ClaRDO, que constará de los mismo métodos pero utilizará RDO para acceder a través de ODBC a la base de datos.

Una vez creada esta clase, el único cambio a realizar en la aplicación es sustituir en la clase agenda (ClaAgenda) la sentencia

```
Private ObjAccesoBD As New ClaDAO_ODBCDirect
```

por

```
Private ObjAccesoBD As New ClaRDO
```

### Ejemplo resuelto

Utilizaremos el formulario **FrmAgenda**, el módulo **ModDefiniciones** y las clase **ClaFicha** y **ClaAgenda** (con la modificación comentada).

#### **ClaRDO**

```
'
' Ejemplo 8 del Curso de Visual Basic Avanzado
'
' Clase de acceso a bases de datos utilizando RDO
'
Option Explicit

Private VarConexion As New rdoConnection
Private Const ConCadenaConexion = "ODBC;DATABASE=Agenda;UID=;PWD=;DSN=Agenda"

Public Property Get NumFichas() As Integer
    On Error GoTo TratamientoError
    Dim RsetDatos As rdoResultset
    Dim CadenaConexion As String

    ' Construimos la sentencia SQL
    CadenaConexion = "select count(*) from agenda"

    ' Ejecutamos la consulta
    Set RsetDatos = VarConexion.OpenResultset(CadenaConexion)

    ' Obtenemos el número de fichas
    NumFichas = RsetDatos.rdoColumns(0).Value

    ' Cerramos el rdoresultset
    RsetDatos.Close

    Exit Property

TratamientoError:
    NumFichas = -1
End Property

Public Function AbrirAgenda() As CodigoRetorno
    On Error GoTo TratamientoError

    ' Abrimos la conexión utilizando la cadena
    ' de conexión predefinida
    With VarConexion
        .CursorDriver = rdUseOdbc
        .Connect = ConCadenaConexion
        .EstablishConnection rdDriverNoPrompt
    End With

    AbrirAgenda = Ok
    Exit Function

TratamientoError:
    AbrirAgenda = Error
End Function
```

```

Public Function CerrarAgenda() As CodigoRetorno
    On Error GoTo TratamientoError

    ' Cerramos la base de datos
    VarConexion.Close

    CerrarAgenda = Ok
    Exit Function

TratamientoError:
    CerrarAgenda = Error
End Function

Public Function GuardarFicha(ByVal ObjFicha As ClaFicha, _
    Optional ByRef IntNumFicha As Integer) As CodigoRetorno
    On Error GoTo TratamientoError
    Dim RsetDatos As rdoResultset
    Dim CadenaSQL As String

    ' Construimos la sentencia SQL
    CadenaSQL = "insert into agenda " & _
        "(Nombre, Apellidos, Direccion, Telefono) " & _
        "values (" & _
        """" & ObjFicha.StrNombre & """, " & _
        """" & ObjFicha.StrApellidos & """, " & _
        """" & ObjFicha.StrDireccion & """, " & _
        """" & ObjFicha.StrTelefono & """)"

    ' Ejecutamos la consulta
    VarConexion.Execute CadenaSQL

    ' Comprobamos si se ha realizado la inserción
    If VarConexion.RowsAffected <> 1 Then
        GuardarFicha = Error
        Exit Function
    End If

    ' Obtenemos el número de ficha asignado por el servidor
    ' NOTA: en realidad, dado que el campo clave es el número
    ' de ficha, este sentencia puede devolver más de un
    ' registro, pero no lo tendremos en cuenta
    CadenaSQL = "select numficha from agenda " & _
        "where " & _
        "Nombre=" & ObjFicha.StrNombre & " AND " & _
        "Apellidos=" & ObjFicha.StrApellidos & " AND " & _
        "Direccion=" & ObjFicha.StrDireccion & " AND " & _
        "Telefono=" & ObjFicha.StrTelefono & ""

    Set RsetDatos = VarConexion.OpenResultset(CadenaSQL)

    If Not RsetDatos.EOF Then
        ' Obtenemos el código de la nueva ficha creada
        IntNumFicha = RsetDatos.rdoColumns(0).Value
        GuardarFicha = Ok
    Else
        GuardarFicha = Error
    End If

    ' Cerramos el rdoresultset
    RsetDatos.Close

```

Exit Function

TratamientoError:

    GuardarFicha = Error

End Function

Public Function ObtenerFicha(ByRef ObjFicha As ClaFicha, \_

    Optional ByVal IntIndice As Integer, \_

    Optional ByVal IntNumFicha As Integer) As CodigoRetorno

    On Error GoTo TratamientoError

    Dim RsetDatos As rdoResultset

    Dim CadenaConexion As String

    Dim IntFichaActual As Integer

    ' Comprobamos si hay que buscar por índice

    If Not IsMissing(IntIndice) Then

        ' Abrimos un rdoresultset seleccionando todos los

        ' registros de la tabla

        CadenaConexion = "select \* from agenda order by numficha"

        Set RsetDatos = VarConexion.OpenResultset(CadenaConexion)

        ' Recorremos el rdoresultset hasta encontrar la ficha indicada

        With RsetDatos

            Do While Not .EOF

                IntFichaActual = IntFichaActual + 1

                If IntFichaActual = IntIndice Then

                    Exit Do

                End If

                .MoveNext

            Loop

        End With

    ' Comprobamos si hay que buscar por número de ficha

    ElseIf Not IsMissing(IntNumFicha) Then

        ' Abrimos un rdoresultset seleccionando el registro

        ' directamente

        CadenaConexion = "select \* from agenda " & \_

        "where numficha=" & IntNumFicha & \_

        "order by numficha"

        Set RsetDatos = VarConexion.OpenResultset(CadenaConexion)

    End If

    ' Comprobamos si la hemos encontrado, o hemos llegado al

    ' final del rdoresultset sin encontrarla

    With RsetDatos

        If .EOF Then

            ObtenerFicha = Error

        Else

            ' Guardamos los datos obtenidos

            ObjFicha.IntNumFicha = .rdoColumns(0).Value

            ObjFicha.StrNombre = .rdoColumns(1).Value

            ObjFicha.StrApellidos = .rdoColumns(2).Value

            ObjFicha.StrDireccion = .rdoColumns(3).Value

            ObjFicha.StrTelefono = .rdoColumns(4).Value

            ObtenerFicha = Ok

        End If

    End With

    ' Cerramos el rdoresultset

    RsetDatos.Close

```
Exit Function

TratamientoError:
  ObtenerFicha = Error
End Function

Public Function BorrarFicha(ByVal IntNumFicha) As CodigoRetorno
  On Error GoTo TratamientoError

  Dim CadenaSQL As String

  ' Construimos la sentencia SQL
  CadenaSQL = "delete from agenda " & _
    "where numficha=" & IntNumFicha

  ' Ejecutamos la sentencia SQL
  VarConexion.Execute CadenaSQL

  ' Comprobamos cuantas filas han sido afectadas
  If VarConexion.RowsAffected = 1 Then
    BorrarFicha = Ok
  Else
    BorrarFicha = Error
  End If
  Exit Function

TratamientoError:
  BorrarFicha = Error
End Function

Private Sub Class_Initialize()
  ' Conectamos con la base de datos
  AbrirAgenda
End Sub
```

## 9. El registro de Windows

### Introducción

El objetivo de este capítulo es introducir al lector en un tema complejo como es el tratamiento del registro de Windows. Se explicará cómo se estructura el registro de Windows, cómo podemos obtener valores almacenados en él y cómo crear nuevas claves y valores de diferentes tipos.

Pero todos los accesos al registro **deben ser controlados**, ya que del registro depende el correcto funcionamiento del S.O., por lo que cualquier cambio podría repercutir en el funcionamiento correcto del sistema.

El acceso al registro de Windows implica la utilización de las correspondientes funciones que proporciona el API de Windows, por lo que en este capítulo se explicará someramente qué es y cómo se utiliza dicho API (Application Programming Interface, o Interfaz para la Programación de Aplicaciones).

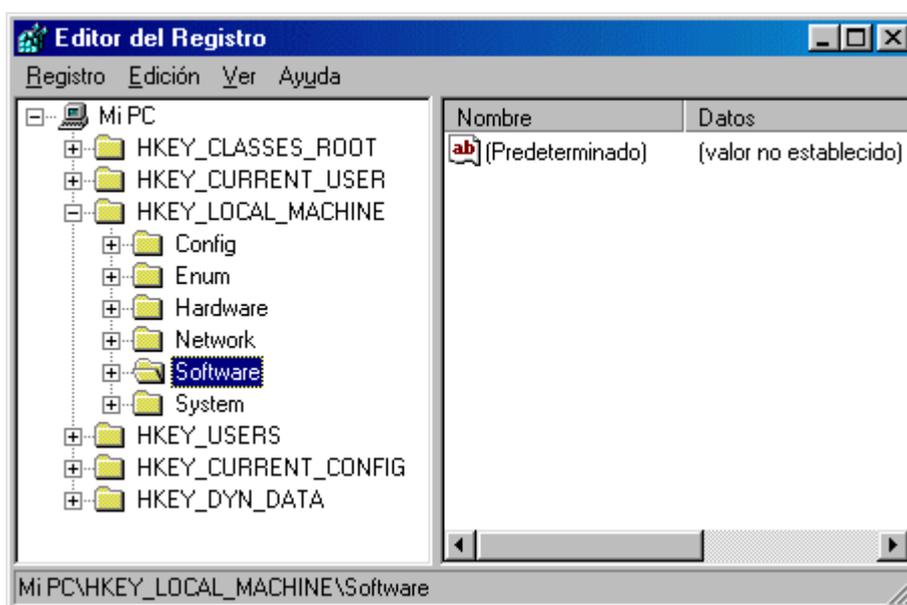
## Conocimientos teóricos

### Qué es el registro de Windows

En Windows 3.1 y en las versiones anteriores de Windows, la configuración de los programas se almacenaban normalmente en archivos .ini. En Windows NT, en Windows 95 y en las versiones posteriores de Windows, la configuración de los programas se almacena en el registro del sistema, todas las instalaciones realizadas, versiones de librerías, controles... quedan reflejados en el registro.

### Cómo acceder al registro de Windows

Podemos acceder *manualmente* a los valores almacenados en el registro de Windows utilizando la aplicación **Regedit**.



También podemos acceder al registro de Windows desde Visual Basic utilizando las funciones que proporciona el API de Windows que veremos mas adelante, pero antes profundizaremos en la definición del *registro*.

### Estructura del registro de Windows

Los elementos almacenados en el registro de Windows se dividen en dos tipos:

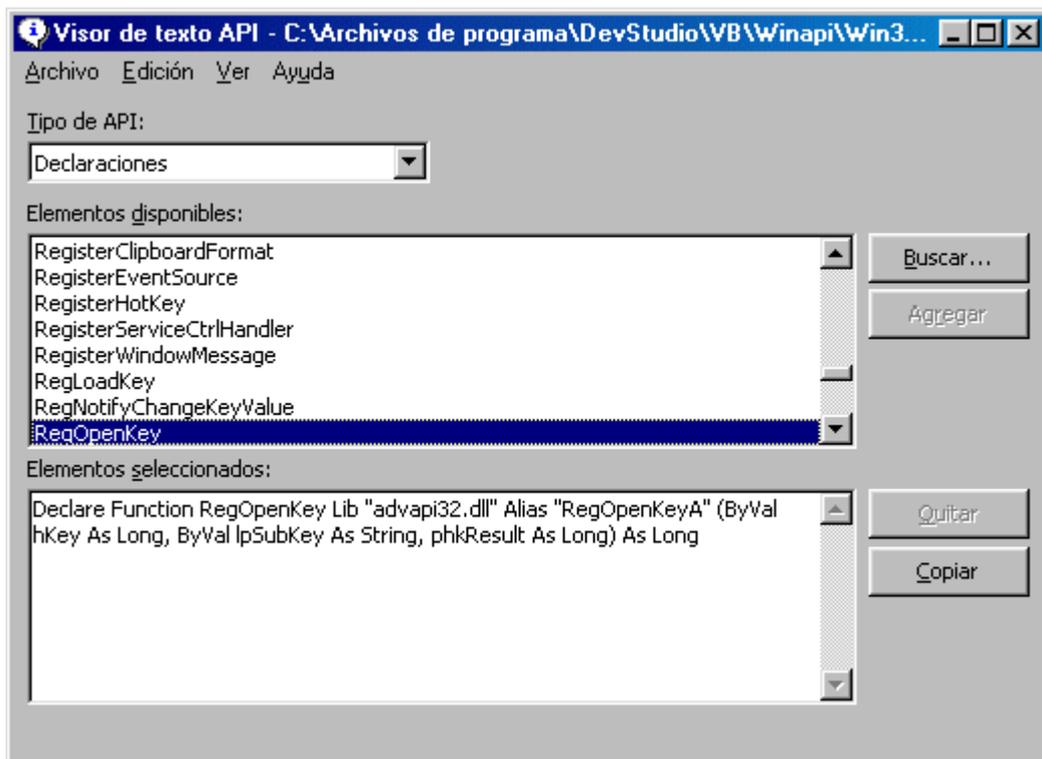
- **claves**, que se estructuran en forma de directorios.
- **valores** de cadena, binarios o DWORD.

Para acceder al valor contenido en una clave es necesario conocer el *“path”* completo de ésta (por ejemplo, para conocer el nº de versión de Windows hay que consultar la clave HKEY\_LOCAL\_MACHINE \ SOFTWARE \ Microsoft \ Windows \ CurrentVersion \ VersionNumber).

## Cómo utilizar el API de Windows

Para poder utilizar las funciones que proporciona el API de Windows es necesario declarar las funciones que éste implementa, así como los posibles tipos de datos y constantes que utilicen. Podemos obtener esta información utilizando el **Visor de texto API** que proporciona Visual Basic, abriendo el archivo **Win32api.txt** que estará en el directorio **Winapi** de la instalación de Visual Basic. Esta aplicación, como se ve en la figura, nos permite conocer el nombre de las funciones que implementa el API de Windows, así como la forma de declararlas, los argumentos que utilizan, los valores de retorno, etc.

Por ejemplo, si quisiéramos utilizar la función `RegOpenKey`, que nos permite abrir una clave del registro, buscaríamos el nombre de la función en el cuadro de lista superior, haríamos doble click sobre el nombre y copiaríamos la declaración en nuestro código de Visual Basic (en realidad, para utilizar esta función necesitamos obtener también ciertas constantes que indican las *ramas* del registro donde vamos a buscar, entre otras).



Pero el problema que tienen los visores API es que no proporcionan una breve explicación del contenido de el API seleccionada, por lo que muchas veces hay que usar la intuición para encontrar una que nos solucione el problema.

En el siguiente cuadro se muestran algunas de las funciones que proporciona el API de Windows para acceso al registro y su correspondiente declaración:

Función	Descripción	Declaración
RegOpenKey	Abre una clave existente	Declare Function OSRegOpenKey Lib "advapi32" Alias "RegOpenKeyA" (ByVal hKey As Long, ByVal lpSubKey As String, phkResult As Long) As Long
RegCloseKey	Cierra una clave abierta con RegOpenKey	Declare Function OSRegCloseKey Lib "advapi32" Alias "RegCloseKey" (ByVal hKey As Long) As Long

RegCreateKey	Crea una nueva clave	Declare Function OSRegCreateKey Lib "advapi32" Alias "RegCreateKeyA" (ByVal hKey As Long, ByVal lpszSubKey As String, phkResult As Long) As Long
RegDeleteKey	Elimina una clave existente	Declare Function OSRegDeleteKey Lib "advapi32" Alias "RegDeleteKeyA" (ByVal hKey As Long, ByVal lpszSubKey As String) As Long
RegQueryValueEx	Obtiene el valor de una clave, especificando también su tipo	Declare Function OSRegQueryValueEx Lib "advapi32" Alias "RegQueryValueExA" (ByVal hKey As Long, ByVal lpszValueName As String, ByVal dwReserved As Long, lpdwType As Long, lpbData As Any, cbData As Long) As Long
RegSetValueEx	Establece el valor de una clave	Declare Function OSRegSetValueEx Lib "advapi32" Alias "RegSetValueExA" (ByVal hKey As Long, ByVal lpszValueName As String, ByVal dwReserved As Long, ByVal fdwType As Long, lpbData As Any, ByVal cbData As Long) As Long

Para poder utilizar estas funciones es necesario declarar también varias constantes, que se especifican en el siguiente cuadro<sup>®</sup>:

Constante	Descripción
Global Const HKEY_CLASSES_ROOT = &H80000000	Define la rama HKEY_CLASSES_ROOT
Global Const HKEY_CURRENT_USER = &H80000001	Define la rama HKEY_CURRENT_USER
Global Const HKEY_LOCAL_MACHINE = &H80000002	Define la rama HKEY_LOCAL_MACHINE
Global Const HKEY_USERS = &H80000003	Define la rama HKEY_USERS
Const ERROR_SUCCESS = 0&	Valor de error
Const REG_SZ = 1	Tipo de dato String
Const REG_BINARY = 3	Tipo de dato binario
Const REG_DWORD = 4	Tipo de dato DWORD

#### Apertura de una clave existente:

Para abrir una clave existente debemos especificar la rama donde se encuentra (HKEY\_CLASSES\_ROOT, etc) y el "path" restante de la clave que queremos abrir. Una vez abierta, nos devolverá un valor de tipo long que identifica a esa clave, y nos permitirá trabajar con ella. Utilizaremos la función **RegOpenKey**, controlando si el valor de retorno es **ERROR\_SUCCESS**, lo que indicaría que todo ha ido bien.

Por ejemplo, si queremos abrir la clave HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft utilizaríamos el siguiente código:

```
Dim hKey As Long
If OSRegOpenKey(HKEY_LOCAL_MACHINE, "SOFTWARE\Microsoft", hKey) =
    ERROR_SUCCESS Then
    MsgBox "Clave abierta"
Else
    MsgBox "Error al abrir la clave"
End If
```

<sup>®</sup> Existen más constantes de este tipo, pero sólo mencionamos las necesarias para el desarrollo del ejemplo correspondiente a este capítulo

### Cerrado de una clave abierta:

Para cerrar una clave abierta, utilizaremos la función **RegCloseKey**, pasándole el identificador de la clave abierta. También en este caso debemos comprobar si el valor devuelto por la función es ERROR\_SUCCESS. Por ejemplo, para cerrar la clave abierta en el ejemplo anterior, utilizaríamos el siguiente fragmento de código:

```
If OSRegCloseKey(hKey) = ERROR_SUCCESS Then
    MsgBox "Clave cerrada"
Else
    MsgBox "Error al cerrar la clave"
End If
```

### Creación de una clave nueva:

Para crear una clave nueva en el registro utilizaremos la función **RegCreateKey**. Esta función creará la clave nueva (o la abrirá, si existe) y nos proporcionará el identificador de la clave recién creada/abierta. Como en los casos anteriores, hemos de comprobar si devuelve ERROR\_SUCCESS. Por ejemplo, para crear la clave HKEY\_LOCAL\_MACHINE\SOFTWARE\MiClave utilizaríamos la siguiente sentencia:

```
If OSRegCreateKey(HKEY_LOCAL_MACHINE, "SOFTWARE\MiClave", hKeyNueva) =
    ERROR_SUCCESS then
    MsgBox "Clave creada"
Else
    MsgBox "Error al crear la clave"
End If
```

### Establecimiento del valor de una clave

Para establecer el valor de una clave existente utilizaremos la función **RegSetValueEx**, a la que pasaremos el identificador de la clave abierta, el nombre que vamos a darle a ese valor (si no se especifica, se entenderá que es el valor por defecto de dicha clave) y el propio valor. Es necesario concretar qué tipo de valor es (una cadena (REG\_SZ), un valor binario (REG\_BINARY) o un valor numérico (REG\_DWORD)) y qué longitud tiene.

Por ejemplo, una vez creada la clave anterior, crearemos un valor nuevo dentro de ella que se llamará MiValor y contendrá la cadena "Esto es una prueba":

```
If OSRegSetValueEx(hKeyNueva, "MiValor", 0&, REG_SZ, "Esto es una prueba",
    Len("Esto es una prueba") + 1) = ERROR_SUCCESS then
    MsgBox "Valor creado"
Else
    MsgBox "Error al crear el valor"
End If
```

Dentro de la misma clave, crearemos un nuevo valor llamado MiNuevoValor, que contendrá el número 17 (como vamos a crearlo como REG\_DWORD, el tamaño es 4):

```
If OSRegSetValueEx(hKeyNueva, "MiNuevoValor", 0&, REG_DWORD, 17, 4) =
    ERROR_SUCCESS then
    MsgBox "Valor creado"
Else
    MsgBox "Error al crear el valor"
End If
```

### Obtención de alguno de los valores de una clave:

Para obtener alguno de los valores asociados a una clave utilizaremos la función **RegQueryValueEx**, a la que pasaremos el identificador de la clave abierta, el nombre del valor que queremos obtener (si no se especifica, se entenderá que es el valor por defecto de dicha clave), una variable donde se guardará el tipo del valor recogido, un *buffer* donde se guardará el valor recogido y una variable donde se guardará el tamaño del valor recogido.

Como es posible obtener valores de tipos y tamaños diferentes, esta función se llamará dos veces: la primera con un *buffer* vacío (0&) para obtener el tipo y el tamaño del valor; la segunda con el *buffer* correspondiente a este tipo y de tamaño correcto (¡OJO!: en el caso de **REG\_SZ** debemos reservar expresamente espacio de memoria). Una vez obtenido el valor, hay que coger sólo el tamaño especificado del valor.

Por ejemplo, si queremos obtener el valor MiValor de la clave anterior, utilizaremos la siguiente sentencia (¡OJO!: el **ByVal** es importante)

```
Dim IValueType As Long
Dim StrBuf As String
Dim IDataBufSize As Long

If OSRegQueryValueEx(hKeyNueva, "MiValor", 0&, IValueType, ByVal 0&, IDataBufSize) =
  ERROR_SUCCESS Then
  If IValueType = REG_SZ Then
    StrBuf = String(IDataBufSize, " ")
    If OSRegQueryValueEx(hKey, strValueName, 0&, 0&, ByVal StrBuf,
      IDataBufSize) = ERROR_SUCCESS Then
      RegQueryStringValue = True
      StrData = Left(StrBuf, IDataBufSize - 1)
    End If
  End If
End If
```

Si queremos obtener el valor MiNuevoValor de la clave anterior, utilizaremos la sentencia:

```
Dim IValueType As Long
Dim IBuf As Long
Dim IDataBufSize As Long

IDataBufSize = 4
If OSRegQueryValueEx(hKeyNueva, "MiNuevoValor", 0&, IValueType, IBuf, IDataBufSize) =
  ERROR_SUCCESS Then
  If IValueType = REG_DWORD Then
    IData = IBuf
  End If
End If
```

### Borrado de una clave:

Para borrar una clave existente utilizaremos la función **RegDeleteKey**, a la que pasaremos el identificador de la clave abierta y la subclave que queremos borrar. Por ejemplo, para borrar la clave HKEY\_LOCAL\_MACHINE\SOFTWARE\MiClave y todo lo que a partir de ella hemos creado, utilizaríamos la sentencia:

```
If OSRegOpenKey(HKEY_LOCAL_MACHINE, "SOFTWARE", hKey) =
  ERROR_SUCCESS Then
  If OSRegDeleteKey(hKey, "MiClave") = ERROR_SUCCESS then
    MsgBox "Clave borrada"
  End If
End If
```

## Ejemplo propuesto

### Objetivo

El objetivo de este ejemplo es crear un formulario de inicio de la aplicación que hemos desarrollado, de forma que nos permita 10 usos y luego no nos deje ejecutar más la aplicación, a no ser que la registremos. Para controlar esto utilizaremos el registro de Windows, en el llevaremos el control de veces que se ha ejecutado la aplicación y guardaremos el registro de la aplicación, tal y como hacen la mayoría de las aplicaciones del mercado.

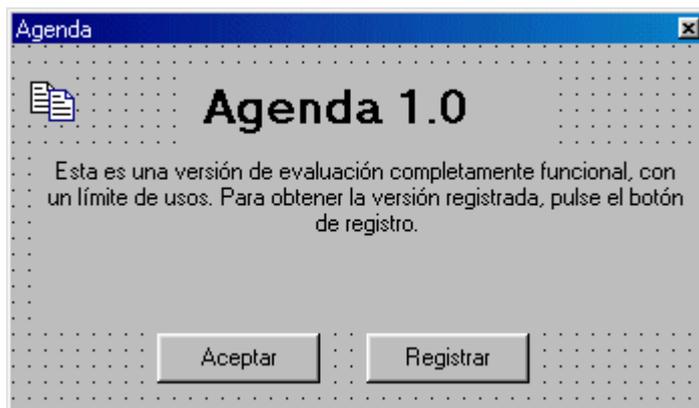
### Desarrollo del ejemplo

Crearemos un módulo ModRegistro que proporcione las siguientes funciones:

- **Function RegCreateKey(ByVal hKey As Long, ByVal lpszKey As String, phkResult As Long) As Boolean**
- **Function RegSetStringValue(ByVal hKey As Long, ByVal strValueName As String, ByVal StrData As String) As Boolean**
- **Function RegSetNumericValue(ByVal hKey As Long, ByVal strValueName As String, ByVal IData As Long, Optional ByVal fLog) As Boolean**
- **Function RegOpenKey(ByVal hKey As Long, ByVal lpszSubKey As String, phkResult As Long) As Boolean**
- **Function RegDeleteKey(ByVal hKey As Long, ByVal lpszSubKey As String) As Boolean**
- **Function RegCloseKey(ByVal hKey As Long) As Boolean**
- **Function RegQueryStringValue(ByVal hKey As Long, ByVal strValueName As String, StrData As String) As Boolean**
- **Function RegQueryNumericValue(ByVal hKey As Long, ByVal strValueName As String, IData As Long) As Boolean**

Estas funciones utilizarán sus correspondientes funciones del API de Windows, tal y como se ha explicado antes en este capítulo.

Crearemos un formulario FrmPresentacion de inicio de la aplicación como el que aparece en la siguiente figura:

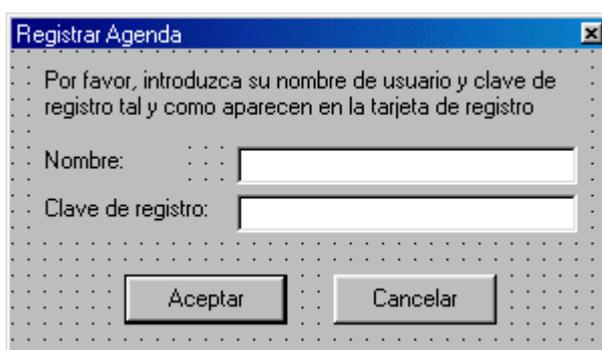


Cada vez que se ejecute la aplicación se lanzaría este formulario (para este ejemplo no habrá aplicación que lanzar: tanto si la aplicación se registra como si es está en periodo de prueba o se supera éste, el formulario se descargará y nada más).

La primera vez que se ejecute la aplicación, este formulario creará una clave HKEY\_LOCAL\_MACHINE\SOFTWARE\Agenda en el registro, y en esta clave dos valores: MaxIntentos=10 y NumIntentos=1. Actualizará el caption del label inferior para mostrar el número máximo de ejecuciones permitidas y el número actual de ejecuciones.

Cada vez que se ejecute la aplicación se incrementará NumIntentos y se comprobará si es menor que MaxIntentos, de forma que si se llega al número máximo de intentos no se permita la ejecución del programa. En cada caso se actualizará el caption.

En todo momento se permitirá registrar la aplicación, en cuyo caso se lanzará un formulario como el siguiente:



Al pulsar el botón aceptar, crearemos un nuevo valor DatosRegistro en nuestra clave, que contendrá el nombre y clave de registro separados por un guión.

La aplicación debe comprobar cada vez que se ejecuta si existe el valor DatosRegistro, en cuyo caso se supone que está registrada y no se tomará en cuenta el valor de NumIntentos y MaxIntentos del registro, permitiendo siempre lanzar la aplicación (en nuestro caso siempre se descargará el formulario).

## Ejemplo resuelto

### FrmPresentacion

```
'
' Ejemplo 9 del Curso de Visual Basic Avanzado
'
' Formulario de presentación, en él comprobamos los usos, la registración de la aplicación...
'
Option Explicit

Const ConMaxIntentos = 10

Private Sub ButAceptar_Click()
    ' Aquí comprobaríamos si se lanza la aplicación
    Unload Me
End Sub

Private Sub ButRegistrar_Click()
    Dim StrClave As String
    Dim StrCadenaRegistro As String
    Dim hkeyExistente As Long

    ' Mostramos el formulario de registro
    frmRegistro.Show vbModal

    ' Abrimos la clave del registro
    If RegOpenKey(HKEY_LOCAL_MACHINE, "SOFTWARE\Agenda", hkeyExistente) Then

        ' Comprobamos si está registrado
        StrClave = "DatosRegistro"
        If RegQueryStringValue(hkeyExistente, StrClave, StrCadenaRegistro) Then
            LabTextoPresentacion.Caption = "Versión registrada, completamente operacional." &
vbCrLf & "¡¡Disfrútela!!"
            LabNumUsos.Visible = False
            ButRegistrar.Enabled = False
            Exit Sub
        End If

        ' Cerramos la clave del registro
        RegCloseKey hkeyExistente
    End If
End Sub

Private Sub Form_Load()
    Dim hKey As Long
    Dim hkeyExistente As Long
    Dim StrClave As String
    Dim StrCadenaRegistro As String
    Dim LngNumIntentos As Long
    Dim LngMaxIntentos As Long

    ' Abrimos la clave del registro
    If RegOpenKey(HKEY_LOCAL_MACHINE, "SOFTWARE\Agenda", hkeyExistente) Then
```

```

' Comprobamos si está registrado
StrClave = "DatosRegistro"
If RegQueryStringValue(hkeyExistente, StrClave, StrCadenaRegistro) Then
    LabTextoPresentacion.Caption = "Versión registrada, completa operativa." &
vbCrLf & "¡¡Disfrútela!!"
    RegCloseKey hkeyExistente
    ButRegistrar.Enabled = False
    Exit Sub
End If

' Como no está registrado, obtenemos el número máximo de intentos
StrClave = "MaxIntentos"
If RegQueryNumericValue(hkeyExistente, StrClave, LngMaxIntentos) Then

    ' Comprobamos cuántos intentos van
    StrClave = "NumIntentos"
    If RegQueryNumericValue(hkeyExistente, StrClave, LngNumIntentos) Then

        ' Comprobamos si se ha llegado al número máximo de intentos
        If LngNumIntentos < LngMaxIntentos Then
            ' Incrementamos el número de usos y lo guardamos en el registro
            LngNumIntentos = LngNumIntentos + 1
            RegSetNumericValue hkeyExistente, "NumIntentos", LngNumIntentos

            ' Actualizamos el formulario
            LabNumUsos.Caption = "La agenda ha sido utilizada " & LngNumIntentos & "
veces (máximo " & LngMaxIntentos & ")"
        Else
            LabNumUsos.Caption = "La agenda ha sido utilizada " & LngNumIntentos & "
veces (máximo " & LngMaxIntentos & "). Registre la versión antes de continuar."
        End If
    End If
End If
RegCloseKey hkeyExistente
Else

' Como no existe, la creamos
If RegCreateKey(HKEY_LOCAL_MACHINE, "SOFTWARE\Agenda", hKey) Then

    ' Número máximo de intentos
    RegSetNumericValue hKey, "MaxIntentos", ConMaxIntentos

    ' Número de intentos realizado
    RegSetNumericValue hKey, "NumIntentos", 1

    RegCloseKey hKey

    ' Actualizamos el formulario
    LabNumUsos.Caption = "La agenda ha sido utilizada 1 vez (máximo " &
ConMaxIntentos & ")"
End If
End If

End Sub

```

## FrmRegistro

```
'
' Ejemplo 9 del Curso de Visual Basic Avanzado
'
' Formulario para introducir los datos del registro de la aplicación.
'
Option Explicit

Private Sub ButAceptar_Click()
    Dim hkeyExistente As Long
    Dim StrClave As String
    Dim StrCadenaRegistro As String

    ' Registramos la aplicación (podríamos comprobar si es auténtico)
    If RegOpenKey(HKEY_LOCAL_MACHINE, "SOFTWARE\Agenda", hkeyExistente) Then

        ' Lo registramos
        StrCadenaRegistro = Text1.Text & "-" & Text2.Text
        StrClave = "DatosRegistro"
        If RegSetStringValue(hkeyExistente, StrClave, StrCadenaRegistro) Then
            MsgBox "La aplicación ha sido registrada con éxito.", , "Registro"
            Unload Me
        End If
    End If
End Sub

Private Sub ButRegistrar_Click()
    Unload Me
End Sub
```

## ModRegistro

```
'
' Ejemplo 8 del Curso de Visual Basic Avanzado
'
' Módulo donde están las funciones de acceso al registro y las declaraciones de las API.
'
Option Explicit

' API de manipulación del regiStro (32 bits)
Declare Function OSRegCloseKey Lib "advapi32" Alias "RegCloseKey" (ByVal hKey As Long)
As Long
Declare Function OSRegCreateKey Lib "advapi32" Alias "RegCreateKeyA" (ByVal hKey As
Long, ByVal lpszSubKey As String, phkResult As Long) As Long
Declare Function OSRegDeleteKey Lib "advapi32" Alias "RegDeleteKeyA" (ByVal hKey As
Long, ByVal lpszSubKey As String) As Long
Declare Function OSRegOpenKey Lib "advapi32" Alias "RegOpenKeyA" (ByVal hKey As Long,
ByVal lpszSubKey As String, phkResult As Long) As Long
Declare Function OSRegQueryValueEx Lib "advapi32" Alias "RegQueryValueExA" (ByVal hKey
As Long, ByVal lpszValueName As String, ByVal dwReserved As Long, lpdwType As Long,
lpbData As Any, cbData As Long) As Long
Declare Function OSRegSetValueEx Lib "advapi32" Alias "RegSetValueExA" (ByVal hKey As
Long, ByVal lpszValueName As String, ByVal dwReserved As Long, ByVal fdwType As Long,
lpbData As Any, ByVal cbData As Long) As Long

Global Const HKEY_CLASSES_ROOT = &H80000000
```

```

Global Const HKEY_CURRENT_USER = &H80000001
Global Const HKEY_LOCAL_MACHINE = &H80000002
Global Const HKEY_USERS = &H80000003
Const ERROR_SUCCESS = 0&

Const REG_SZ = 1
Const REG_BINARY = 3
Const REG_DWORD = 4

' Crea (o abre si ya existe) una clave en el registro del sistema
Function RegCreateKey(ByVal hKey As Long, ByVal lpszKey As String, phkResult As Long) As Boolean
    On Error GoTo 0

    If OSRegCreateKey(hKey, lpszKey, phkResult) = ERROR_SUCCESS Then
        RegCreateKey = True
    Else
        RegCreateKey = False
    End If
End Function

' Asocia un valor con nombre (StrValueName = nombre) o sin nombre (StrValueName = "")
' con una clave del regiStro.
Function RegSetStringValue(ByVal hKey As Long, ByVal strValueName As String, ByVal StrData As String) As Boolean
    On Error GoTo 0

    If hKey = 0 Then Exit Function

    If OSRegSetValueEx(hKey, strValueName, 0&, REG_SZ, ByVal StrData, _
        Len(StrData) + 1) = ERROR_SUCCESS Then
        RegSetStringValue = True
    Else
        RegSetStringValue = False
    End If
End Function

' Asocia un valor con nombre (strValueName = nombre) o sin nombre (strValueName = "")
' con una clave del registro.
Function RegSetNumericValue(ByVal hKey As Long, ByVal strValueName As String, ByVal IData As Long, Optional ByVal fLog) As Boolean
    On Error GoTo 0

    If OSRegSetValueEx(hKey, strValueName, 0&, REG_DWORD, IData, 4) = ERROR_SUCCESS Then
        RegSetNumericValue = True
    Else
        RegSetNumericValue = False
    End If
End Function

' Abre una clave existente en el registro del sistema.
Function RegOpenKey(ByVal hKey As Long, ByVal lpszSubKey As String, phkResult As Long) As Boolean
    On Error GoTo 0

    If OSRegOpenKey(hKey, lpszSubKey, phkResult) = ERROR_SUCCESS Then
        RegOpenKey = True
    Else
        RegOpenKey = False
    End If
End Function

```

```

    End If
End Function

' Elimina una clave existente del regiStro del sistema.
Function RegDeleteKey(ByVal hKey As Long, ByVal lpszSubKey As String) As Boolean
    On Error GoTo 0

    If OSRegDeleteKey(hKey, lpszSubKey) = ERROR_SUCCESS Then
        RegDeleteKey = True
    Else
        RegDeleteKey = False
    End If
End Function

' Cierra una clave abierta del registro
Function RegCloseKey(ByVal hKey As Long) As Boolean
    On Error GoTo 0

    If OSRegCloseKey(hKey) = ERROR_SUCCESS Then
        RegCloseKey = True
    Else
        RegCloseKey = False
    End If
End Function

' Recupera los datos de cadena para un valor con nombre
' (StrValueName = nombre) o sin nombre (StrValueName = "")
' dentro de una clave del regiStro. Si el valor con
' nombre existe, pero sus datos no son una cadena, esta
' función fallará.
Function RegQueryStringValue(ByVal hKey As Long, ByVal strValueName As String, StrData
As String) As Boolean
    On Error GoTo 0

    Dim IValueType As Long
    Dim StrBuf As String
    Dim IDataBufSize As Long

    RegQueryStringValue = False
    ' Obtiene el tipo y longitud de los datos
    If OSRegQueryValueEx(hKey, strValueName, 0&, IValueType, ByVal 0&, IDataBufSize) =
ERROR_SUCCESS Then
        If IValueType = REG_SZ Then
            StrBuf = String(IDataBufSize, " ")
            If OSRegQueryValueEx(hKey, strValueName, 0&, 0&, ByVal StrBuf, IDataBufSize) =
ERROR_SUCCESS Then
                StrData = Left(StrBuf, IDataBufSize - 1)
                RegQueryStringValue = True
            End If
        End If
    End If
End Function

' Recupera los datos enteros para un valor con nombre
' (StrValueName = nombre) o sin nombre (StrValueName = "")
' dentro de una clave del regiStro. Si el valor con nombre
' existe, pero sus datos no son de tipo REG_DWORD, esta
' función fallará.
Function RegQueryNumericValue(ByVal hKey As Long, ByVal strValueName As String, IData
As Long) As Boolean

```

```
On Error GoTo 0

Dim IValueType As Long
Dim IBuf As Long
Dim IDataBufSize As Long

RegQueryNumericValue = False

' Obtiene el tipo y longitud de los datos
IDataBufSize = 4
If OSRegQueryValueEx(hKey, strValueName, 0&, IValueType, IBuf, IDataBufSize) =
ERROR_SUCCESS Then
    If IValueType = REG_DWORD Then
        IData = IBuf
        RegQueryNumericValue = True
    End If
End If
End Function
```

## APÉNDICE A: Especificaciones, limitaciones y formatos de archivos de Visual Basic

En este apéndice se describen los requisitos de sistema, las limitaciones de un proyecto de Visual Basic, los tipos de archivos que se pueden incluir en el proyecto de Visual Basic y las descripciones de los archivos de formulario (.frm) y de proyecto (.vbp).

### **Requisitos del sistema para aplicaciones de Visual Basic**

Para las aplicaciones de Visual Basic se requiere el siguiente hardware y software:

- Microsoft Windows NT 3.51 o posterior, o Microsoft Windows 95 o posterior.
- Microprocesador 80486 o superior.
- Pantalla VGA o de resolución superior compatible con Microsoft Windows.
- 8 MB de RAM para aplicaciones. (Esto variará dependiendo de las bibliotecas de tipos o los archivos DLL específicos que incluya en sus aplicaciones.)
- 16 MB de RAM para el entorno de desarrollo de Visual Basic.

### **Limitaciones de los proyectos**

Un único proyecto puede contener hasta 32.000 identificadores, que incluyen entre otros formularios, controles, módulos, variables, constantes, procedimientos, funciones y objetos. Los nombres de variables en Visual Basic no pueden tener más de 255 caracteres y los nombres de formularios, controles, módulos y clases pueden tener un máximo de 40 caracteres. Visual Basic no impone ningún límite en cuanto al número de objetos distintos de un proyecto.

### **Limitaciones de controles**

Cada control no gráfico (todos los controles excepto **Shape**, **Line**, **Image** y **Label**) utiliza una ventana. Cada ventana utiliza recursos del sistema, limitando el número total de ventanas que pueden existir al mismo tiempo. El límite exacto depende de los recursos del sistema disponibles y el tipo de controles que se utilicen.

Para reducir el consumo de recursos, utilice controles **Shape**, **Line**, **Label** e **Image** en vez de los controles **PictureBox** para crear o presentar gráficos.

### **Número total de controles**

El número máximo de controles permitidos en un único formulario depende del tipo de controles que se utilicen y de los recursos disponibles del sistema. No obstante, hay un límite fijo de 254 nombres de control por formulario. En este límite una matriz de controles sólo cuenta como uno, ya que todos los controles en la matriz comparten un único nombre de control.

El límite para índices de matrices de controles es de 0 a 32.767 para todas las versiones.

Si coloca controles uno encima de otro, como por ejemplo si utiliza varios controles de marco dentro de otros marcos, Visual Basic no aceptará más de seis niveles de controles anidados.

## Limitaciones para determinados controles

La siguiente tabla muestra las limitaciones de propiedades aplicables a determinados controles de Visual Basic:

Propiedad	Aplicable a	Limitación
<b>List</b> y <b>ListCount</b>	Controles de cuadro de lista y cuadro combinado	El número máximo de elementos es 32 KB; el límite del tamaño de cada elemento es 1 KB (1024 bytes).
<b>Text</b>	Control de cuadro de texto	El límite es de 64 KB.
<b>Caption</b>	Control de etiqueta	Limitado a 1024 bytes.
	Controles de botón de comando, casilla de verificación, marco y botón de opción	Limitados a 255 caracteres. Se truncan todos los títulos que superen ese límite. Los títulos en las propiedades de los controles personalizados tienen un límite de 32 KB.
	Control de menú	Limitado a 235 caracteres.
<b>Tag</b>	Todos los controles	Limitado sólo por la memoria disponible.
<b>Name</b>	Todos los controles	Limitado a 40 caracteres.

## Limitaciones de código

La cantidad de código que se puede cargar en un formulario, clase o módulo estándar está limitada a 65.534 líneas. Una única línea de código puede constar de 1.023 bytes como máximo. Puede haber hasta 256 espacios en blanco delante del texto en una única línea y no se pueden incluir más de veinticinco caracteres de continuación de línea ( `_` ) en una única línea lógica.

## Procedimientos, tipos y variables

No hay límite en cuanto al número de procedimientos por módulo. Cada procedimiento puede contener hasta 64 KB de código. Si un procedimiento o un módulo excede este límite, Visual Basic genera un error en tiempo de compilación. Si se encuentra con este error, puede evitarlo dividiendo los procedimientos extremadamente grandes en varios procedimientos más pequeños o trasladando las declaraciones de nivel de módulo a otro módulo.

Visual Basic utiliza tablas para almacenar los nombres de los identificadores (variables, procedimientos, constantes, etc.) en el código. Cada tabla está limitada a 64 KB.

## Tabla de entradas de módulo

Esta tabla acepta hasta 125 bytes por módulo, con un límite total de 64 KB, que da como resultado unos 400 módulos por proyecto.

## Limitaciones de los datos

Las siguientes limitaciones son aplicables a las variables en el lenguaje Visual Basic

### **Datos de formulario, módulos estándar y módulos de clase**

Cada formulario, módulo estándar y módulo de clase tiene su propio segmento de datos que puede ser como máximo de 64 KB. Este segmento de datos contiene los siguientes datos:

- Variables locales declaradas con **Static**.
- Variables a nivel de módulo que no sean matrices y cadenas de longitud variable.
- 4 bytes para cada matriz a nivel de módulo y cadena de longitud variable.

### **Procedimientos, tipos y variables**

Si un procedimiento o un módulo excede el límite de 64 KB, Visual Basic generará un error de tiempo de compilación. Si se encuentra con este error, puede evitarlo dividiendo los procedimientos extremadamente grandes en varios procedimientos más pequeños o trasladando las declaraciones a nivel de módulo a otro módulo.

### **Tipos definidos por el usuario**

Ninguna variable de un tipo definido por el usuario puede exceder los 64 KB, aunque la suma de las cadenas de longitud variable en un tipo definido por el usuario puede exceder de 64 KB (las cadenas de longitud variable sólo ocupan 4 bytes cada una en el tipo definido por el usuario; el contenido real de una cadena se almacena por separado). Los tipos definidos por el usuario se pueden definir en términos de otros tipos definidos por el usuario, pero el tamaño total de los tipos no puede exceder los 64 KB.

### **Espacio de pila**

Los argumentos y las variables locales en los procedimientos ocupan espacio de pila en tiempo de ejecución. Las variables estáticas y a nivel de módulo no ocupan espacio de pila porque se encuentran en el segmento de datos para los formularios o los módulos. Todos los procedimientos de DLL a los que se llame utilizan esta pila mientras se están ejecutando.

Visual Basic utiliza parte de la pila para sus propios usos, como el almacenamiento de valores intermedios al evaluar expresiones.

### **Limitaciones de los recursos del sistema**

Algunas limitaciones de Visual Basic y las aplicaciones creadas con él están impuestas por Microsoft Windows. Estas limitaciones pueden cambiar cuando se instala una versión diferente de Microsoft Windows.

### **Recursos de Windows**

Cada ventana abierta usa recursos del sistema (áreas de datos utilizadas por Microsoft Windows). Si se agotan los recursos del sistema, se producirá un error en tiempo de ejecución. Puede comprobar el porcentaje de recursos del sistema que quedan si elige **Acerca de** en el menú **Ayuda** del Administrador de programas o del Administrador de archivos en Windows NT

3.51 o, en Windows 95 y Windows NT 4.0, si elige **Acerca de** en el menú **Ayuda** del Explorador de Windows. Las aplicaciones también pueden llamar a la función GetFreeSystemResources de el API de Windows para reclamar recursos del sistema, cerrar ventanas (como formularios abiertos y ventanas de código, así como ventanas de otras aplicaciones) y finalizar la ejecución de aplicaciones.

### **Formatos de archivos de proyecto**

Microsoft Visual Basic utiliza y crea una serie de archivos tanto en tiempo de diseño como en tiempo de ejecución. Los archivos que el proyecto o la aplicación requerirán dependen de su alcance y funcionalidad.

### **Extensiones de archivos de proyecto**

Visual Basic crea varios archivos cuando se crea y compila un proyecto. Estos se pueden dividir como sigue: tiempo de diseño, otros desarrollos y tiempo de ejecución.

Los archivos de tiempo de diseño son los ladrillos de su proyecto: por ejemplo, módulos de Basic (.bas) y módulos de formulario (.frm).

Otros procesos y funciones del entorno de desarrollo de Visual Basic crean diversos archivos: por ejemplo, archivos de dependencias del Asistente para instalación (.dep).

### **Archivos varios y de tiempo de diseño**

La siguiente tabla muestra todos los archivos de tiempo de diseño y otros archivos que se pueden crear al desarrollar una aplicación:

<b>Extensión</b>	<b>Descripción</b>
.bas	Módulo de Basic
.cls	Módulo de clase
.ctl	Archivo de control de usuario
.ctx	Archivo binario de control de usuario
.dca	Caché de diseñador activo
.dep	Archivo de dependencias del Asistente de instalación
.dob	Archivo de formulario de documento ActiveX
.dox	Archivo binario de formulario de documento ActiveX
.dsr	Archivo de diseñador activo
.dsx	Archivo binario de diseñador activo
.frm	Archivo de formulario
.frx	Archivo binario de formulario
.log	Archivo de registro de errores de carga
.oca	Archivo de caché de biblioteca de tipos de controles
.pag	Archivo de página de propiedades
.pgx	Archivo binario de página de propiedades
.res	Archivo de recursos
.swt	Archivo de plantilla del Asistente de instalación de Visual Basic
.tlb	Archivo TypeLib de Automatización remota
.vbg	Archivo de proyecto de grupo de Visual Basic
.vbl	Archivo de control de licencia
.vbp	Archivo de proyecto de Visual Basic

.vbr	Archivo de registro de Automatización remota
.vbw	Archivo de espacio de trabajo de proyecto de Visual Basic
.vbz	Archivo de inicio del Asistente

## Archivos de tiempo de ejecución

Al compilar la aplicación, todos los archivos necesarios de tiempo de ejecución se incluyen en los archivos ejecutables de tiempo de ejecución. La siguiente tabla muestra los archivos de tiempo de ejecución:

Extensión	Descripción
.dll	Componente ActiveX en proceso
.exe	Archivo ejecutable o componente ActiveX
.ocx	Control ActiveX
.vbb	Archivo de inicio de documento ActiveX
.vbd	Archivo de estado de documento ActiveX

## Estructuras de formularios

A pesar de que muchos de los archivos de un típico proyecto de Visual Basic están en formato binario y sólo los pueden leer determinados procesos y funciones de Visual Basic o de su aplicación, los archivos de formulario (.frm) y de proyecto (.vbp) se guardan como texto ASCII. Estos archivos se pueden leer en un visor de texto (por ejemplo, el Bloc de notas).

En las siguientes secciones se describen los archivos de tiempo diseño y de tiempo de ejecución en un proyecto típico de Visual Basic y el formato de los archivos de formulario (.frm) y de proyecto (.vbp).

Los archivos de formulario de Visual Basic (.frm) se crean y se guardan en formato ASCII. La estructura de un formulario consiste en lo siguiente:

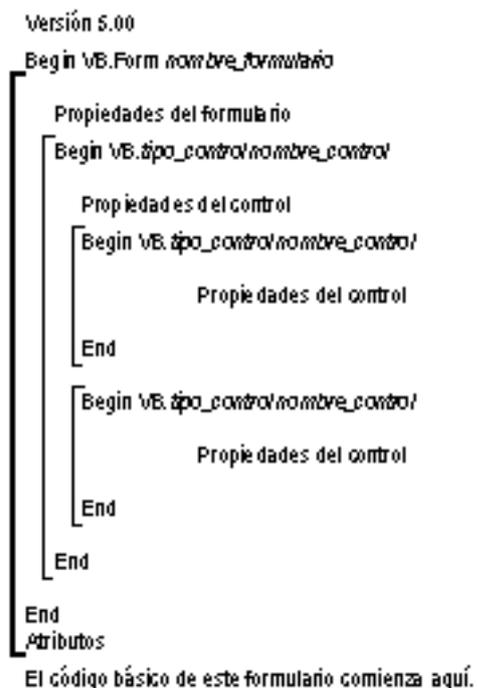
- El número de versión del formato de archivo.
- Un bloque de texto que contiene la descripción del formulario.
- Un conjunto de atributos del formulario.
- El código Basic del formulario.

La descripción del formulario contiene los valores de las propiedades del formulario. Los bloques de texto que definen las propiedades de los controles del formulario están anidados en el formulario. Los controles contenidos en otros controles tienen sus propiedades anidadas en el texto del contenedor. La figura A.1 ilustra la estructura de la descripción del formulario.

## Número de versión

El número de versión para los formularios creados con Visual Basic 5.0 para Windows es 5.00. Si en un formulario se omite el número de versión, se generará un error. Al cargar una aplicación de Visual Basic que tiene un número de versión menor que 5.00, aparece un cuadro de diálogo de advertencia para informar que el archivo se guardará en el nuevo formato.

Figura A.1 Estructura de la descripción del formulario



## Descripción del formulario

La descripción de un formulario empieza con una instrucción **Begin** y termina con una instrucción **End**. La sintaxis de la instrucción **Begin** es la siguiente:

**Begin VB.**{Form|MDIForm} *nombre\_formulario*

La instrucción **End** indica dónde termina la descripción del formulario y dónde empieza el conjunto de atributos del formulario. Sin la instrucción **End**, Visual Basic intentaría leer los atributos como si se estuvieran describiendo los controles y las propiedades del formulario, por lo que se producirían errores.

Entre las instrucciones **Begin Form** y **End** están las propiedades del formulario propiamente dicho, seguidas de las descripciones de cada control del formulario. La figura A.2 muestra la estructura anidada de la descripción del formulario con más detalle.

## Bloques de control

Un *bloque de control* consta del texto de la descripción del formulario que define las propiedades de un control individual. Al igual que la descripción del formulario, los bloques de control empiezan con una instrucción **Begin** y terminan con una instrucción **End**. La sintaxis para una instrucción **Begin** de un bloque de control es la siguiente:

**Begin clase\_control.tipo\_control** *nombre\_control*

Las propiedades para el control aparecen entre la instrucción **Begin** y la instrucción **End**.

## Orden de los bloques de control

El orden de los bloques de control determina el orden z de los controles. El *orden z* es una ordenación relativa que determina la manera en la que los controles se solapan entre sí en un formulario. El primer control de la descripción del formulario establece la parte inferior del orden



controles de menú, supone que no hay más controles de menú en el formulario y pasa por alto cualquier control de menú que encuentre durante la carga de ese formulario.

## Teclas de método abreviado

Las teclas de método abreviado son teclas que se utilizan para activar un control de menú. Los formularios ASCII usan la misma sintaxis que la instrucción **SendKeys** para definir las combinaciones de teclas: "+" = MAYÚSCULAS, "^" = CTRL y "{Fn}" = tecla de función, donde *n* es el número de la tecla. Los caracteres alfabéticos se representan a sí mismos. La sintaxis de las teclas de método abreviado es la siguiente:

Método abreviado = ^{F4} ' <CTRL><F4>

## Comentarios en la descripción del formulario

Puede agregar comentarios a la descripción del formulario. Las comillas simples (') son los delimitadores de los comentarios.

## Propiedades de la descripción de un formulario

Cuando Visual Basic guarda un formulario, organiza las propiedades en un orden predeterminado. No obstante, al crear un formulario las propiedades se pueden organizar en cualquier orden.

Cualquier propiedad que no enumere se establece a su valor predeterminado cuando se carga. Cuando Visual Basic guarda un formulario, sólo incluye aquellas propiedades que no usan sus valores predeterminados. Cada control determina si se guardan o no todas sus propiedades o solamente aquellas cuyos valores son distintos de los valores predeterminados.

## Sintaxis

Use esta sintaxis para definir las propiedades en la descripción del formulario:

*propiedad* = *valor*

Los valores de propiedad de texto deben aparecer entre comillas dobles. Las propiedades booleanas tienen un valor - 1 para **True** y 0 para **False**. Visual Basic interpreta todos los valores distintos de -1 o 0 como **True**. Las propiedades con valores enumerados incluyen sus valores numéricos con la descripción del valor incluida como comentario. Por ejemplo, la propiedad **BorderStyle** aparece así:

BorderStyle = 0 ' Ninguno

## Valores binarios de propiedades

Algunos controles disponen de propiedades cuyos valores son datos binarios, como la propiedad **Picture** del cuadro de imagen y los controles de imagen o ciertas propiedades de los controles personalizados. Visual Basic guarda todos los datos binarios de un formulario en un archivo de datos binario aparte del formulario.

Visual Basic guarda el archivo de datos binario en el mismo directorio que el formulario. El archivo de datos binario tiene el mismo nombre que el formulario, pero con la extensión .frx. Visual Basic lee el archivo de datos binario cuando carga el formulario, por lo que el archivo de datos binario (.frx) debe estar disponible para el formulario cuando Visual Basic lo carga. Si comparte formularios con otras personas que usan un archivo de datos binario, asegúrese de proporcionar el archivo de datos binario (.frx), además del formulario (.frm).

Las propiedades que tengan datos binarios como valores aparecen en el formulario como una referencia a un desplazamiento de byte en el archivo de datos binario. Por ejemplo, el valor de una propiedad **Picture** aparece de la siguiente manera en la descripción de un formulario:

```
Begin VB.Image imgDemo
  Picture = "Miform.frx":02EB
End
```

La enumeración de la propiedad significa que los datos binarios que definen la propiedad **Picture** de este control empiezan en el byte 2EB (hex) del archivo Miform.frx.

### Propiedad Icon

El valor de la propiedad **Icon** de un formulario depende de qué icono se utilice para el formulario. La siguiente tabla muestra los valores de la propiedad **Icon** y la manera en que esas propiedades aparecen en un formulario.

Valor de propiedad Icon	Contenido del formulario ASCII
El icono predeterminado	No hace referencia a la propiedad <b>Icon</b>
(Ninguno)	Icon = 0
Cualquier icono distinto del icono predeterminado	Referencia de desplazamiento de byte al archivo de datos binario. Por ejemplo: Icon = "Miform.frx":0000

### Propiedad TablIndex

Si no se especifica la propiedad **TablIndex**, Visual Basic asigna el control a la primera ubicación posible en el orden de tabulación una vez que estén cargados todos los demás controles.

### Unidades de medida

El tamaño de los controles, las coordenadas x e y, así como otros valores de propiedades que utilicen unidades de medida se expresan en twips. Cuando un control usa un modo de escala diferente a los twips y Visual Basic carga el formulario, convierte los valores en twips del formulario ASCII a las unidades de medida especificadas por la propiedad **ScaleMode**.

### Valores de colores

Los valores de colores aparecen como valores RGB (rojo, verde y azul). Por ejemplo, la propiedad **ForeColor** aparece de esta manera:

```
ForeColor = &H00FF0000&
```

Visual Basic también puede leer valores de **QBColor**, convirtiéndolos a RGB cuando carga el formulario. Los formularios ASCII que utilicen valores de **QBColor** deben usar esta sintaxis:

```
ForeColor = QBColor(qbcolor)
```

donde *qbcolor* es un valor de 0 a 15.

Observe que el argumento *qbcolor* corresponde a los valores de color utilizados por las instrucciones gráficas en otras versiones de Basic, como Visual Basic para MS-DOS, Microsoft QuickBasic y el Sistema de desarrollo profesional Microsoft Basic.

## Objetos de propiedades

Algunos objetos de propiedades, como por ejemplo el objeto **Font**, aparecen como un bloque distinto y muestran todos los valores de las diferentes propiedades del objeto. Estos bloques están delimitados por las instrucciones **BeginProperty** y **EndProperty** de la siguiente forma:

**BeginProperty** *nombre\_propiedad*

*propiedad1 = valor1*

*propiedad2 = valor2*

.

.

.

**EndProperty**

## Código de Basic

El código de Basic aparece en el formulario inmediatamente después de la sección de atributos, a continuación de la última instrucción **End** en la descripción del formulario. Las instrucciones en la sección Declaraciones de un formulario aparecen primero, seguidas de los procedimientos de evento, los procedimientos generales y las funciones.

## Errores al cargar archivos de formulario

Cuando Visual Basic carga un formulario en memoria, primero convierte el formulario al formato binario. Al realizar cambios en el formulario y guardar los cambios, Visual Basic vuelve a escribir el archivo en formato ASCII.

Cuando Visual Basic se encuentra con un error mientras está cargando un formulario, crea un archivo de registro e informa del error en el archivo de registro. Visual Basic agrega mensajes de error al archivo de registro cada vez que se encuentra con un error en el formulario. Cuando se ha terminado de cargar el formulario, Visual Basic presenta un mensaje que indica que se creó un archivo de registro de errores.

El archivo de registro tiene el mismo nombre que el archivo de formulario, pero con la extensión .log. Por ejemplo, si se producen errores mientras se carga Miform.frm, Visual Basic creará un archivo de registro llamado Miform.log. Si posteriormente vuelve a cargar Miform.frm y continúan produciéndose errores mientras se carga el formulario, Visual Basic reemplazará el archivo Miform.log anterior.

## Mensajes de registro de error en la carga de un formulario

Los siguientes mensajes de error pueden aparecer en un archivo de registro de errores. Observe que estos mensajes de error sólo tratan problemas que pueden ocurrir cuando Visual Basic carga la descripción del formulario. No indican ningún problema que pueda existir en procedimientos de evento, en procedimientos generales o en cualquier otra parte del código de Basic.

### **Imposible cargar el menú *nombre\_menú*.**

Este mensaje aparece si Visual Basic encuentra un control de menú cuyo menú primario esté definido como un separador de menú. Los controles de menú que actúan como primarios para los controles de menú en un submenú no pueden ser separadores de menú. Visual Basic no carga el control de menú.

Este mensaje también aparece si Visual Basic encuentra un control de menú cuyo menú primario tiene su propiedad **Checked** establecida a **True**. Los controles de menú que actúan como primarios para los controles de menú en un submenú no pueden tener esta propiedad activada. Visual Basic no carga el control de menú.

**Imposible establecer la propiedad Checked en el menú *nombre\_menú*.**

Este mensaje aparece si Visual Basic encuentra un control de menú de nivel superior con su propiedad **Checked** establecida a **True**. Los menús de nivel superior no pueden tener una marca de verificación. Visual Basic carga el control del menú, pero no establece su propiedad **Checked**.

**Imposible establecer la propiedad Shortcut en *nombre\_menú*.**

Este mensaje aparece si Visual Basic encuentra un control de menú de nivel superior con una tecla de método abreviado definida. Los menús de nivel superior no pueden tener una tecla de método abreviado. Visual Basic carga el control del menú, pero no establece la propiedad **Shortcut**.

**La clase *nombre\_clase* del control *nombre\_control* no es una clase de control cargada.**

Este mensaje aparece si Visual Basic encuentra un nombre de clase que no reconoce.

**Imposible cargar el control *nombre\_control*.**

Este mensaje aparece si Visual Basic encuentra un tipo de control desconocido en la descripción del formulario. Visual Basic crea un cuadro de imagen para representar el control desconocido, dando a ese cuadro de imagen todas las propiedades válidas de la descripción del control desconocido. Cuando aparece este mensaje, es probable que le sigan varios errores de propiedades no válidas.

**El control *nombre\_control* tiene una cadena entre comillas donde debería estar el nombre de la propiedad.**

Este mensaje aparece si Visual Basic encuentra texto entre comillas en vez de un nombre de propiedad que no se coloca entre comillas. Por ejemplo:

```
"Caption" = "Comenzar la demostración"
```

En este caso, el nombre de propiedad **Caption** no debería estar entre comillas. Visual Basic pasa por alto la línea de la descripción del formulario que produce este error.

**El nombre de control *nombre\_control* no es válido.**

Este mensaje aparece si el nombre de un control no es una cadena válida en Visual Basic. Visual Basic no cargará el control.

**El nombre de control es demasiado largo; truncado a *nombre\_control*.**

Este mensaje aparece si Visual Basic encuentra un nombre de control con más de 40 caracteres. Visual Basic carga el control, truncando el nombre.

**No se encontró una propiedad de índice y el control *nombre\_control* ya existe. Imposible crear este control.**

Este mensaje aparece si Visual Basic encuentra un control sin un índice que tiene el mismo nombre que un control cargado anteriormente. Visual Basic no carga el control.

**Imposible cargar el formulario *nombre\_formulario*.**

Este mensaje aparece si Visual Basic encuentra inesperadamente el final del archivo o si falta la primera instrucción **Begin**.

**El nombre del formulario o de MDIForm *nombre\_formulario* no es válido; imposible cargar este formulario.**

Este mensaje aparece si el nombre de un formulario no es una cadena válida en Visual Basic. Visual Basic no cargará el formulario.

Las cadenas válidas deben empezar con una letra, sólo pueden incluir letras, números y signos de subrayado, y deben tener como máximo 40 caracteres.

**El nombre *nombre\_propiedad* de la propiedad del control *nombre\_control* no es válido.**

Este mensaje aparece si el nombre de una propiedad no es una cadena válida en Visual Basic o tiene más de 30 caracteres. Visual Basic no establecerá la propiedad.

**Imposible cargar la propiedad *nombre\_propiedad* del control *nombre\_control*.**

Este mensaje aparece si Visual Basic encuentra una propiedad desconocida. Visual Basic pasa por alto esta propiedad cuando carga el formulario.

**Imposible establecer la propiedad nombre\_propiedad del control nombre\_control.**

Este mensaje aparece si Visual Basic no puede establecer la propiedad del control especificado como se indica en la descripción del formulario.

**La propiedad nombre\_propiedad del control nombre\_control tiene un valor no válido.**

Este mensaje aparece si Visual Basic encuentra un valor no válido para una propiedad. Visual Basic cambia el valor de la propiedad al valor predeterminado para esa propiedad.

**La propiedad nombre\_propiedad del control nombre\_control tiene una referencia de archivo no válida.**

Este mensaje aparece si Visual Basic no pudo usar una referencia de nombre de archivo. Esto ocurrirá si el archivo al que se refiere (probablemente un archivo de datos binario para el formulario) no se encuentra en el directorio especificado.

**La propiedad nombre\_propiedad del control nombre\_control tiene un índice de propiedad no válido.**

Este mensaje aparece si Visual Basic encuentra un nombre de propiedad con un índice de propiedad mayor de 255. Por ejemplo:

Prop300 = 5436

Visual Basic pasa por alto la línea de la descripción del formulario que produjo este error.

**La propiedad nombre\_propiedad del control nombre\_control tiene un valor no válido.**

Este mensaje aparece si Visual Basic encuentra una propiedad con un valor que no es correcto para ese control. Por ejemplo:

Top = Cahr(22) ' En realidad se deseaba Char(22).

Visual Basic establece la propiedad a su valor predeterminado.

**La propiedad nombre\_propiedad del control nombre\_control debe ser una cadena entre comillas.**

Este mensaje aparece si Visual Basic encuentra un valor de propiedad sin comillas que debería aparecer entre comillas. Por ejemplo:

Caption = Comenzar la demostración

Visual Basic pasa por alto la línea de la descripción del formulario que produjo este error.

**Error de sintaxis: la propiedad nombre\_propiedad del control nombre\_control no tiene un '='.**

Este mensaje aparece si Visual Basic encuentra un nombre y un valor de propiedad sin un signo igual entre ellos. Por ejemplo:

Text "Comenzar la demostración"

Visual Basic no carga la propiedad.

**Formato del archivo de proyecto (.vbp)**

Visual Basic siempre guarda los archivos de proyecto (.vbp) en formato ASCII. El archivo de proyecto contiene entradas que reflejan los valores de su proyecto. Entre estos se incluyen los formularios y módulos del proyecto, referencias, opciones varias que ha elegido para controlar la compilación, etc.

Esta es la apariencia que debe tener un archivo .vbp. Este proyecto incluye módulos guardados con los nombres de clase y de archivo que se muestran en la siguiente tabla.

Tipo de módulo	Nombre de clase	Nombre de archivo
Formulario MDI	AForm	A_Form.frm
Formulario	BForm	B_Form.frm
Módulo estándar	CModule	C_Module.bas
Módulo de clase	DClass	D_Class.cls

```
Type=Exe
Form=B_Form.frm
Reference=*G{00020430-0000-0000-C000-000000000046}#2.0#0#..\..\WINDOWS\SYSTEM\STDOLE2.TLB#OLE Automation
Form=A_Form.frm
Module=CModule; C_Module.bas
Class=DClass; D_Class.cls
```

```
Startup="BForm"  
Command32=""  
Name="Project1"  
HelpContextID="0"  
CompatibleMode="0"  
MajorVer=1  
MinorVer=0  
RevisionVer=0  
AutoIncrementVer=0  
ServerSupportFiles=0  
VersionCompanyName="Microsoft"  
CompilationType=0  
OptimizationType=0  
FavorPentiumPro(tm)=0  
CodeViewDebugInfo=0  
NoAliasing=0  
BoundsCheck=0  
OverflowCheck=0  
FIPointCheck=0  
FDIVCheck=0  
UnroundedFP=0  
StartMode=0  
Unattended=0  
ThreadPerObject=0  
MaxNumberOfThreads=1
```

Se agregan entradas al archivo .vbp cuando agrega formularios, módulos, componentes, etc. al proyecto. También se agregan entradas cuando establece opciones para el proyecto. Muchas de estas opciones se establecen mediante el cuadro de diálogo **Propiedades del proyecto**.

## Apéndice B: Convenciones de codificación

Este apéndice presenta un conjunto de convenciones de codificación que sugerimos para los programas de Visual Basic.

Las convenciones de codificación son pautas de programación que no están enfocadas a la lógica del programa, sino a su estructura y apariencia física. Facilitan la lectura, comprensión y mantenimiento del código. Las convenciones de codificación pueden incluir:

- Convenciones de nombres para objetos, variables y procedimientos.
- Formatos estandarizados para etiquetar y comentar el código.
- Instrucciones de espaciado, formato y sangría.

En las secciones siguientes se explica cada una de estas áreas y se dan ejemplos de su uso correcto.

### **¿Por qué existen las convenciones de codificación?**

La razón principal de usar un conjunto coherente de convenciones de código es estandarizar la estructura y el estilo de codificación de una aplicación de forma que el autor y otras personas puedan leer y entender el código fácilmente.

Las convenciones de codificación correctas dan como resultado un código fuente preciso, legible y sin ambigüedad, que es coherente con otras convenciones del lenguaje y lo más intuitivo posible.

### **Convenciones de codificación mínimas**

Un conjunto de convenciones de codificación de propósito general debe definir los requisitos mínimos necesarios para conseguir los objetivos explicados anteriormente, dejando libertad al programador para crear la lógica y el flujo funcional del programa.

El objetivo es hacer que el programa sea fácil de leer y de entender sin obstruir la creatividad natural del programador con imposiciones excesivas y restricciones arbitrarias.

Por tanto, las convenciones sugeridas en este apéndice son breves y sugerentes. No muestran todos los objetos y controles posibles, ni especifican todos los tipos de comentarios informativos que podrían ser útiles. Dependiendo del proyecto y de las necesidades específicas de la organización, quizás desee ampliar estas instrucciones para que incluyan elementos adicionales como:

- Convenciones para objetos específicos y componentes desarrollados internamente o comprados a otros proveedores.
- Variables que describan las actividades comerciales o instalaciones de la organización.
- Cualquier otro elemento que el proyecto o la empresa considere importante para conseguir mayor claridad y legibilidad.

### **Convenciones de nombres de objetos**

Los objetos deben llevar nombres con un prefijo coherente que facilite la identificación del tipo de objeto. A continuación se ofrece una lista de convenciones recomendadas para algunos de los objetos permitidos por Visual Basic.

**Prefijos sugeridos para controles**

Tipo de control	Prefijo	Ejemplo
Panel 3D	pnl	pnlGrupo
Botón animado	ani	aniBuzón
Casilla de verificación	chk	chkSóloLectura
Cuadro combinado, cuadro de lista desplegable	cbo	cboInglés
Botón de comando	cmd	cmdSalir
Diálogo común	dlg	dlgArchivoAbrir
Comunicaciones	com	comFax
Control (dentro de procedimientos cuando no se conoce el tipo específico)	ctr	ctrActivo
Control de datos	dat	datBiblio
Cuadro combinado enlazado a datos	dbcbo	dbcboLenguaje
Cuadrícula enlazada a datos	dbgrd	dbgrdResultadoConsulta
Cuadro de lista enlazado a datos	dblst	dblstTipoTarea
Cuadro de lista de directorios	dir	dirOrigen
Cuadro de lista de unidades	drv	drvDestino
Cuadro de lista de archivos	fil	filOrigen
Formulario	frm	frmEntrada
Marco	fra	fraLenguaje
Medidor	gau	gauEstado
Gráfico	gra	graIngresos
Cuadrícula	grd	grdPrecios
Barra de desplazamiento horizontal	hsb	hsbVolumen
Imagen (Image)	img	imgIcono
Estado de tecla	key	keyMayúsculas
Etiqueta	lbl	lblMsjAyuda
Línea	lin	linVertical
Cuadro de lista	lst	lstCódigosDePolítica
Mensaje MAPI	mpm	mpmEnviarMsj
Sesión MAPI	mps	mpsSesión
MCI	mci	mciVideo
Formulario MDI secundario	mdi	mdiNota
Menú	mnu	mnuArchivoAbrir
MS Flex Grid	msg	msgClientes
MS Tab	mst	mstPrimero
ActiveX	ole	oleHojaDeTrabajo
Esquema	out	outDiagramaDeOrg
Pen BEdit	bed	bedNombre
Pen Hedit	hed	hedFirma
Trazo de pluma	ink	inkMapa
Imagen (Picture)	pic	picVGA
Clip de imagen	clp	clpBarraDeHerramientas
Informe	rpt	rptGananciasTrimestre1
Forma	shp	shpCírculo
Cuadro de número	spn	spnPáginas

Cuadro de texto	txt	txtApellido
Cronómetro	tmr	tmrAlarma
Arriba-abajo	upd	updDirección
Barra de desplazamiento vertical	vsb	vsbVelocidad
Control deslizante	sld	sldEscala
Lista de imágenes	ils	ilsTodosLosIconos
Vista de árbol	tre	treOrganización
Barra de herramientas	tlb	tlbAcciones
TabStrip	tab	tabOpciones
Barra de estado	sta	staFechaHora
Lista	lvw	lvwEncabezados
Barra de progreso	prg	prgCargarArchivo
RichTextBox	rtf	rtfInforme

**Prefijos sugeridos para los objetos de acceso a datos (DAO)**

Use los prefijos siguientes para indicar Objetos de acceso a datos (DAO).

<b>Objeto de base de datos</b>	<b>Prefijo</b>	<b>Ejemplo</b>
Contenedor	con	conInformes
Base de datos	db	dbCuentas
Motor de base de datos	dbe	dbeJet
Documento	doc	docInformeVentas
Campo	fld	fldDirección
Grupo	grp	grpFinanzas
Índice	idx	idxEdad
Parámetro	prm	prmCódigoTarea
Definición de consulta	qry	qryVentasPorRegión
Conjunto de registros	rec	recPrevisión
Relación	rel	relDeptDeEmpleados
Definición de tabla	tbd	tbdClientes
Usuario	usr	usrNuevo
Espacio de trabajo	wsp	wspMío

Algunos ejemplos:

```
Dim dbBiblio As Database
Dim recEditoresMAD As Recordset, strSQL As String
Const DB_READONLY = 4 ' Establece la constante.
' Abre la base de datos.
Set dbBiblio = OpenDatabase("BIBLIO.MDB")
' Establece el texto para la instrucción SQL.
strSQL = "SELECT * FROM Editores WHERE Estado = 'MAD'"
' Crea el nuevo objeto Recordset.
Set recEditoresMAD = db.OpenRecordset(strSQL, _
    dbSóloLectura)
```

**Prefijos sugeridos para menús**

Las aplicaciones suelen usar muchos controles de menú, lo que hace útil tener un conjunto único de convenciones de nombres para estos controles. Los prefijos de controles de menús se deben extender más allá de la etiqueta inicial "mnu", agregando un prefijo adicional para cada

nivel de anidamiento, con el título del menú final en la última posición de cada nombre. En la tabla siguiente hay algunos ejemplos.

Secuencia del título del menú	Nombre del controlador del menú
Archivo Abrir	mnuArchivoAbrir
Archivo Enviar correo	mnuArchivoEnviarCorreo
Archivo Enviar fax	mnuArchivoEnviarFax
Formato Carácter	mnuFormatoCarácter
Ayuda Contenido	mnuAyudaContenido

Cuando se usa esta convención de nombres, todos los miembros de un grupo de menús determinado se muestran uno junto a otro en la ventana Propiedades de Visual Basic. Además, los nombres del control de menú documentan claramente los elementos de menú a los que están adjuntos.

### **Selección de prefijos para otros controles**

Para los controles no mostrados arriba, debe intentar establecer un estándar de prefijos únicos de dos o tres caracteres que sean coherentes. Solamente se deben usar más de tres caracteres si proporcionan más claridad.

Para controles derivados o modificados, por ejemplo, amplíe los prefijos anteriores para que no haya dudas sobre qué control se está usando realmente. Para los controles de otros proveedores, se debe agregar al prefijo una abreviatura del nombre del fabricante en minúsculas. Por ejemplo, una instancia de control creada a partir del marco 3D incluido en la Edición profesional de Visual Basic podría llevar el prefijo fra3d para evitar confusiones sobre qué control se está usando realmente.

### **Convenciones de nombres de constantes y variables**

Además de los objetos, las constantes y variables también requieren convenciones de nombres bien compuestas. En esta sección se muestran las convenciones recomendadas para las constantes y variables permitidas por Visual Basic. También se explican cuestiones relacionadas con la identificación del tipo de datos y su alcance.

Las variables se deben definir siempre con el menor alcance posible. Las variables globales (públicas) pueden crear máquinas de estado enormemente complejas y hacer la lógica de una aplicación muy difícil de entender. Las variables globales también hacen mucho más difícil mantener y volver a usar el código.

En Visual Basic las variables pueden tener el alcance siguiente:

Alcance	Declaración	Visible en
Nivel de procedimiento	'Private' en procedimiento, subprocedimiento o función	El procedimiento en el que está declarada
Nivel de módulo	'Private' en la sección Declaraciones de un módulo de formulario o de código (.frm, .bas)	Todos los procedimientos del módulo de formulario o de código
Global	'Public' en la sección Declaraciones de un módulo de código (.bas)	En toda la aplicación

En una aplicación de Visual Basic, las variables globales se deben usar sólo cuando no exista ninguna otra forma cómoda de compartir datos entre formularios. Cuando haya que usar variables globales, es conveniente declararlas todas en un único módulo agrupadas por funciones y dar al módulo un nombre significativo que indique su finalidad, como Public.bas.

Una práctica de codificación correcta es escribir código modular siempre que sea posible. Por ejemplo, si la aplicación muestra un cuadro de diálogo, coloque todos los controles y el código necesario para ejecutar la tarea del diálogo en un único formulario. Esto ayuda a tener el código de la aplicación organizado en componentes útiles y minimiza la sobrecarga en tiempo de ejecución.

A excepción de las variables globales (que no se deberían pasar), los procedimientos y funciones deben operar sólo sobre los objetos que se les pasan. Las variables globales que se usan en los procedimientos deben estar identificadas en la sección Declaraciones al principio del procedimiento. Además, los argumentos se deben pasar a los procedimientos Sub y Function mediante **ByVal**, a menos que sea necesario explícitamente cambiar el valor del argumento que se pasa.

## Prefijos de alcance de variables

A medida que aumenta el tamaño del proyecto, también aumenta la utilidad de reconocer rápidamente el alcance de las variables. Esto se consigue escribiendo un prefijo de alcance de una letra delante del tipo de prefijo, sin aumentar demasiado la longitud del nombre de las variables.

Alcance	Prefijo	Ejemplo
Global	g	gstrNombreUsuario
Nivel de módulo	m	mbInProgresoDelCálculo
Local del procedimiento	Ninguno	dblVelocidad

Una variable tiene alcance global si se declara como **Public** en un módulo estándar o en un módulo de formulario. Una variable tiene alcance de *nivel de módulo* si se declara como **Private** en un módulo estándar o en un módulo de formulario, respectivamente.

## Constantes

El cuerpo del nombre de las constantes se debe escribir en mayúsculas y minúsculas, con la letra inicial de cada palabra en mayúsculas. Aunque las constantes estándar de Visual Basic no incluyen información de tipo de datos y el alcance, los prefijos como i, s, g y m pueden ser muy útiles para entender el valor y el alcance de una constante. Para los nombres de constantes, se deben seguir las mismas normas que para las variables. Por ejemplo:

```

mintMáxListaUsuario ' Límite de entradas máximas para
                    ' la lista de usuarios (valor
                    ' entero, local del módulo)
gstrNuevaLínea      ' Carácter de nueva línea
                    ' (cadena, global de la
                    ' aplicación)
    
```

## Variables

Declarar todas las variables ahorra tiempo de programación porque reduce el número de errores debidos a erratas (por ejemplo, aNombreUsuarioTmp frente a sNombreUsuarioTmp frente a sNombreUsuarioTemp). En la ficha **Editor** del cuadro de diálogo **Opciones**, active la opción **Declaración de variables requerida**. La instrucción **Option Explicit** requiere que declare todas las variables del programa de Visual Basic.

Las variables deben llevar un prefijo para indicar su tipo de datos. Opcionalmente, y en especial para programas largos, el prefijo se puede ampliar para indicar el alcance de la variable.

## Tipos de datos de variables

Use los prefijos siguientes para indicar el tipo de datos de una variable.

Tipo de datos	Prefijo	Ejemplo
Boolean	bln	blnEncontrado
Byte	byt	bytDatosImagen
Objeto Collection	col	colWidgets
Currency	cur	curlIngresos
Date (Time)	dtm	dtmInicio
Double	dbl	dblTolerancia
Error	err	errNúmDeOrden
Integer	int	intCantidad
Long	lng	lngDistancia
Object	obj	objActivo
Single	sng	sngMedia
String	str	strNombreF
Tipo definido por el usuario	udt	udtEmpleado
Variant	vnt	vntCheckSum

## Nombres descriptivos de variables y procedimientos

El cuerpo de un nombre de variable o procedimiento se debe escribir en mayúsculas y minúsculas y debe tener la longitud necesaria para describir su funcionalidad. Además, los nombres de funciones deben empezar con un verbo, como IniciarNombreMatriz o CerrarDiálogo.

Para nombres que se usen con frecuencia o para términos largos, se recomienda usar abreviaturas estándar para que los nombres tengan una longitud razonable. En general, los nombres de variables con más de 32 caracteres pueden ser difíciles de leer en pantallas VGA.

Cuando se usen abreviaturas, hay que asegurarse de que sean coherentes en toda la aplicación. Alternar aleatoriamente entre Cnt y Contar dentro de un proyecto provoca una confusión innecesaria.

## Tipos definidos por el usuario

En un proyecto grande con muchos tipos definidos por el usuario, suele ser útil dar a cada uno de estos tipos un prefijo de tres caracteres sólo suyo. Si estos prefijos comienzan con "u", será fácil reconocerlos cuando se esté trabajando con tipos definidos por el usuario. Por ejemplo, "ucli" se podría usar como prefijo para las variables de un tipo Cliente definido por el usuario.

## Convenciones de codificación estructurada

Además de las convenciones de nombres, las convenciones de codificación estructurada, como comentarios al código y sangrías coherentes, pueden mejorar mucho la legibilidad del código.

## Convenciones de comentarios al código

Todos los procedimientos y funciones deben comenzar con un comentario breve que describa las características funcionales del procedimiento (qué hace). Esta descripción no debe describir los detalles de implementación (cómo lo hace), porque a veces cambian con el tiempo, dando como resultado un trabajo innecesario de mantenimiento de los comentarios o, lo que es peor, comentarios erróneos. El propio código y los comentarios de líneas necesarios describirán la implementación.

Los argumentos que se pasan a un procedimiento se deben describir cuando sus funciones no sean obvias y cuando el procedimiento espera que los argumentos estén en un intervalo específico. También hay que describir, al principio de cada procedimiento, los valores de retorno de funciones y las variables globales que modifica el procedimiento, en especial los modificados a través de argumentos de referencia.

Los bloques del comentario de encabezado del procedimiento deben incluir los siguientes encabezados de sección. En la sección siguiente, "Dar formato al código", hay algunos ejemplos.

Encabezado de sección	Descripción del comentario
Finalidad	Lo que hace el procedimiento (no cómo lo hace).
Premisas	Lista de cada variable externa, control, archivo abierto o cualquier otro elemento que no sea obvio.
Efectos	Lista de cada variable externa, control o archivo afectados y el efecto que tiene (sólo si no es obvio).
Entradas	Todos los argumentos que puedan no ser obvios. Los argumentos se escriben en una línea aparte con comentarios de línea.
Resultados	Explicación de los valores devueltos por las funciones.

Recuerde los puntos siguientes:

- Cada declaración de variable importante debe incluir un comentario de línea que describa el uso de la variable que se está declarando.
- Las variables, controles y procedimientos deben tener un nombre bastante claro para que los comentarios de línea sólo sean necesarios en los detalles de implementación complejos.
- Al principio del módulo .bas que contiene las declaraciones de constantes genéricas de Visual Basic del proyecto, debe incluir un resumen que describa la aplicación, enumerando los principales objetos de datos, procedimientos, algoritmos, cuadros de diálogo, bases de datos y dependencias del sistema. Algunas veces puede ser útil un pseudocódigo que describa el algoritmo.

## Dar formato al código

Como muchos programadores usan todavía pantallas VGA, hay que ajustarse al espacio de la pantalla en la medida de lo posible y hacer que el formato del código siga reflejando la estructura lógica y el anidamiento. Estos son algunos indicadores:

- Los bloques anidados estándar, separados por tabuladores, deben llevar una sangría de cuatro espacios (predeterminado).
- El comentario del esquema funcional de un procedimiento debe llevar una sangría de un espacio. Las instrucciones de nivel superior que siguen al comentario del esquema deben llevar una sangría de un tabulador, con cada bloque anidado separado por una sangría de un tabulador adicional. Por ejemplo:

```

*****
' Finalidad: Ubica el primer caso encontrado de un
'            usuario especificado en la matriz
'            ListaUsuario.
' Entradas:
'   strListaUsuario(): lista de usuarios para buscar.
'   strUsuarioDest: nombre del usuario buscado.
' Resultados: Índice del primer caso de rsUsuarioDest
'             encontrado en la matriz rasListaUsuario.
'             Si no se encuentra el usuario de destino, devuelve -1.
*****

```

```

Function intBuscarUsuario (strListaUsuario() As String, strUsuarioDest As _
String) As Integer
    Dim i As Integer          ' Contador de bucle.
    Dim blnEncontrado As Integer ' Indicador de
                             ' destino encontrado.

    intBuscarUsuario = -1
    i = 0
    While i <= Ubound(strListaUsuario) and Not blnEncontrado
        If strListaUsuario(i) = strUsuarioDest Then
            blnEncontrado = True
            intBuscarUsuario = i
        End If
    Wend
End Function

```

## Agrupación de constantes

Las variables y constantes definidas se deben agrupar por funciones en lugar de dividir las en áreas aisladas o archivos especiales. Las constantes genéricas de Visual Basic se deben agrupar en un único módulo para separarlas de las declaraciones específicas de la aplicación.

## Operadores & y +

Use siempre el operador **&** para unir cadenas y el operador **+** cuando trabaje con valores numéricos. El uso del operador **+** para concatenar puede causar problemas cuando se opera sobre dos variables **Variant**. Por ejemplo:

```

vntVar1 = "10.01"
vntVar2 = 11
vntResult = vntVar1 + vntVar2 'vntResult = 21.01
vntResult = vntVar1 & vntVar2 'vntResult = 10.0111

```

## Crear cadenas para MsgBox, InputBox y consultas SQL

Cuando esté creando una cadena larga, use el carácter de subrayado de continuación de línea para crear múltiples líneas de código, de forma que pueda leer o depurar la cadena fácilmente. Esta técnica es especialmente útil cuando se muestra un cuadro de mensaje (**MsgBox**) o un cuadro de entrada (**InputBox**), o cuando se crea una cadena SQL. Por ejemplo:

```

Dim Msj As String
Msj = "Esto es un párrafo que estará en un" _
& " cuadro de mensajes. El texto está separado en" _
& " varias líneas de código en el código de origen, " _
& "lo que facilita al programador la tarea de leer y depurar."
MsgBox Msj

```

```

Dim CTA As String
CTA = "SELECT *" _
& " FROM Título" _
& " WHERE [Fecha de publicación] > 1988"
ConsultaTítulos.SQL = CTA

```