Comprender el alcance y la visibilidad

El alcance se refiere a la disponibilidad de una variable, constante, o procedimiento para ser usado por otro procedimiento. Hay tres niveles de alcance: nivel de procedimiento, nivel de módulo privado y nivel de módulo público.

El alcance de una variable se determina cuando se declara. es aconsejable declarar todas las variables explícitamente para evitar errores de conflicto de nombres entre variables que tiene alcances distintos.

Definir el alcance de un nivel de procedimiento

Una variable o constante definida desde dentro de un procedimiento no es visible fuera de ese procedimiento. Sólo el procedimiento que contiene la declaración de la variable puede usarlos. En el siguiente ejemplo, el primer procedimiento presenta un cuadro de mensaje que contiene una cadena de texto. El segundo procedimiento presenta un cuadro de mensaje en blanco ya que la variable `strMensaje` es local para el primer procedimiento.

```
Sub VariableLocal()
    Dim strMensaje As String
```

```
    strMensaje = "Esta variable no se puede usa
    MsgBox strMensaje
End Sub

Sub FueraAlcance()
    MsgBox strMensaje
End Sub
```

Definir el alcance de nivel de módulo privado

Se pueden definir variables y constantes de nivel de módulo en la sección Declarations de un módulo. Las variables de nivel de módulo pueden ser públicas o privadas. Las variables públicas están disponibles para todos los procedimientos de todos los módulos de un proyecto; las variables privadas sólo están disponibles para los procedimientos de ese módulo. Las variables declaradas con la instrucción **Dim** en la sección Declarations tiene un alcance privado a no ser que se especifique otra cosa. Sin embargo, es aconsejable colocar la palabra clave **Private** delante del nombre de la variable para que el alcance sea evidente al leer el código.

En el siguiente ejemplo, la variable de cadena `strMensaje` está disponible para cualquier procedimiento definido en el módulo. Cuando se llama al segundo procedimiento, presenta el contenido de la variable de cadena `strMensaje` en un cuadro de dialogo.

```
' Añada lo siguiente a la sección de Declaratio
Private strMensaje sAs String

Sub InicializaVariablePrivada()
    strMensaje = "Esta variable no se puede usa
End Sub

Sub UsaVariablePrivada()
    MsgBox strMensaje
```

```
End Sub
```

Nota Los procedimientos públicos de un módulo estándar o módulo de clase están disponibles para cualquier proyecto de referencia al proyecto actual. Para limitar el alcance de todos los procedimientos de un módulo al proyecto actual, añada una instrucción **Option Private Module** a la sección Declarations del módulo. Las variables y procedimientos públicos seguirán estando disponibles para los restantes procedimientos del proyecto actual, pero no lo estarán para los proyectos que a los que puedan hacer referencia.

Definir el alcance de nivel de módulo público

Si se declara una variable de nivel de módulo como pública, estará disponible para todos los procedimientos del mismo proyecto. En el siguiente ejemplo, la variable de cadena `strMensaje` se puede usar en cualquier procedimiento de un módulo del proyecto.

```
' Incluya esto en la sección Declarations del m
Public strMensaje As String
```

Todos los procedimientos son públicos a menos que se especifique lo contrario, excepto en el caso de los procedimientos evento. Cuando Visual Basic crea un procedimiento evento, la palabra clave **Private** se inserta automáticamente antes de la declaración del procedimiento. Para los restantes procedimientos, debe declarar explícitamente con la palabra clave **Private** si no desea que sean públicos.

Se pueden usar procedimientos, variables y constantes públicas definidas en módulos estándar o módulos de clase de proyectos que hagan referencia al actual. Sin embargo, es preciso establecer una referencia al proyecto en que están definidos.

Los procedimientos, variables y constantes públicas definidas en módulos que no sean estándar o de clase, como pueden ser los módulos de formulario o módulos de informe, no están disponibles desde los proyectos que hacen referencia al actual, ya que estos módulos son privados para el proyecto en el

que residen.

Comprender la automatización

La automatización (antes llamada automatización OLE) es una de las características del *Component Object Model* (COM), una tecnología estándar en la industria usada por las aplicaciones para revelar sus objetos a las herramientas de desarrollo, lenguajes de macros y otras aplicaciones que sean compatibles con la automatización. Por ejemplo, una aplicación de hojas de cálculo puede revelar una hoja de cálculo, un diagrama, celda o rango de celdas, cada una como un tipo diferente de objeto. Un procesador de textos puede revelar objetos como una aplicación, un documento, un párrafo, una frase, un marcador o una selección.

Cuando una aplicación es compatible con la automatización, Visual Basic tiene acceso a los objetos revelados por la aplicación. Utilice Visual Basic para manipular esos objetos mediante la aplicación de métodos sobre el objeto o leyendo y dando valor a las propiedades del objeto. Por ejemplo, se puede crear un objeto de Automatización llamado `MiObj` y escribir el siguiente código para tener acceso al objeto:

```
MiObj.Insert "Hola, amigos."                    ' S
MiObj.Bold = True
If Mac = True    ' Compruebe la constante de pl
    MyObj.SaveAs  "HD:\WORDPROC\DOCS\TESTOBJ.DO
Else
    MyObj.SaveAs  "C:\WORDPROC\DOCS\TESTOBJ.DOC
```

Utilice las siguientes funciones para acceder a un objeto de automatización:

Función	Descripción
CreateObject	Crea un nuevo objeto del tipo especificado.
GetObject	Recupera un objeto de un archivo.

Si desea más detalles sobre las propiedades y métodos compatibles con una aplicación, debe consultar la documentación de la aplicación. Los objetos, funciones, propiedades y métodos compatibles con una aplicación están normalmente definidos en la biblioteca de objetos de la aplicación.

Comprender la compilación condicional

La compilación condicional se puede utilizar para ejecutar de forma selectiva determinados bloques de código, por ejemplo, instrucciones de depuración que comparen la velocidad de distintos métodos de enfocar una tarea de programación, o adaptando una aplicación para distintos idiomas.

En el código se declara una constante de compilación condicional mediante la directiva **#Const** y se identifican los bloques de código que se deben compilar condicionalmente mediante la directiva **#If...Then...#Else**. El siguiente ejemplo ejecuta código de depuración o de producción, en función del valor asignado a la variable `conDepurar`.

```
' Declara como public la constante de compilaci
#Const conDepurar = 1

Sub EjecucionSelectiva()
    #If conDepurar = 1 Then
        .                   ' Ejecuta el código co
        .
        .
    #Else
        .                   ' Ejecuta el código no
```

```
        .
        .
    #End If
End Sub
```

Comprender la vida de las variables

El tiempo durante el que una variable conserva su valor se conoce como vida. El valor de la variable puede cambiar durante su vida, pero conserva algún valor. Cuando una variable pierde su alcance, ya no tiene ningún valor.

Cuando un procedimiento comienza a ejecutarse, se inicializan todas las variables. Las variables numéricas se inicializan a cero, las cadenas de longitud variable se inicializan a una cadena vacía de longitud cero ("") y una cadena de longitud fija se llena con los caracteres correspondientes al código ASCII 0, o **Chr(**0**)**. Las variables Variant se inicializan a Empty. Cada uno de los elementos de una variable de tipo definido por el usuario se inicializa como si se tratase de una variable independiente.

Cuando se declara una variable de objeto, se reserva el espacio correspondiente en memoria, pero se le da el valor **Nothing** hasta que se le asigne una referencia a objeto mediante la instrucción **Set**.

Si el valor de una variable no cambia durante la ejecución del código, conserva su valor de inicialización hasta que pierda alcance.

Una variable de nivel de procedimiento declarada mediante la instrucción **Dim** conserva su valor hasta que el procedimiento termina de ejecutarse. Si el procedimiento llama a otros procedimientos, la variable conserva su valor

también mientras se ejecutan esos procedimientos.

Si una variable de nivel de procedimiento de declara con la palabra clave **Static**, la variable conserva su valor mientras haya código en ejecución, sea cual sea el módulo. Cuando todo el código ha terminado de ejecutarse, la variable pierde su alcance y su valor. Su vida es la misma que la de una variable de nivel de módulo.

Una variable de nivel de módulo es distinta a una variable estática. En un módulo estándar o un módulo de clase, conserva su valor hasta que se termina de ejecutar el código. En un módulo de clase conserva su valor mientras exista una definición de la clase. Las variables de nivel de módulo consumen memoria hasta que sus valores se vuelven a inicializa, por eso sólo se deben utilizar cuando sea necesario.

Si se incluye la palabra clave **Static** antes de una instrucción **Sub** o **Function**, los valores de todas las variables de nivel de procedimiento del procedimiento se conservarán entre las sucesivas llamadas.

Comprender la sintaxis de Visual Basic

En la sección de Ayuda de Visual Basic correspondiente a un método, instrucción o procedimiento Function la sintaxis muestra todos los elementos necesarios para utilizar correctamente ese método, instrucción o función. Los ejemplos siguientes explican como deben interpretarse los elementos sintácticos más comunes.

Sintaxis del método Activate

objeto.**Activate**

En la sintaxis del método **Activate**, la palabra "objeto" en cursiva es la posición reservada para la información que introduce el usuario — en este caso, código que devuelve un objeto. Las palabras que se muestran en negrita deben escribirse exactamente tal y como se indica. Por ejemplo, el siguiente procedimiento activa la segunda ventana en el documento activo.

```
Sub Activar()
    Windows(2).Activate
End Sub
```

Sintaxis de la función MsgBox

MsgBox(*texto*[*, botones*] [*, título*] [*, archivoayuda, contexto*]**)**

En la sintaxis de la función **MsgBox**, las palabras en cursiva y negrita son argumentos con nombre de la función. Los argumentos que aparecen entre corchetes son opcionales. (No escriba los corchetes en el código de Visual Basic). En el caso de la función **MsgBox**, el único argumento que se debe incluir es el texto de la pregunta.

Los argumentos para funciones y métodos se pueden especificar en el código mediante su posición o por su nombre. Para especificar argumentos mediante su posición, siga el orden que se indica en la sintaxis, separando los argumentos con una coma, por ejemplo:

```
MsgBox "¡Su respuesta es correcta!",0,"Cuadro d
```

Para especificar un argumento mediante su nombre basta con usar el nombre del argumento seguido de dos puntos y un signo igual (**:=**) y el valor del argumento. Los argumentos con nombre se pueden especificar en cualquier orden, por ejemplo:

```
MsgBox Title:="Cuadro de Respuesta", Prompt:="¡
```

La sintaxis de las funciones y algunos métodos muestran los argumentos entre paréntesis. Estas funciones y métodos devuelven valores, por eso deben encerrarse los argumentos entre paréntesis al asignar un valor a una variable. Si se ignora el valor de retorno o si no se pasan argumentos en forma alguna, no deben incluirse los paréntesis. Los métodos que no devuelven valores no necesitan que sus argumentos aparezcan encerrados entre paréntesis. Estas normas son aplicables tanto si se usan argumentos posicionales o nominativos.

En el siguiente ejemplo, el valor que devuelve la función **MsgBox** es un número que indica el botón seleccionado almacenado en la variable `miVar`. Dado que se utiliza el valor que devuelve la función, es preciso utilizar paréntesis. Otro cuadro de mensaje presenta entonces en pantalla el valor de la variable.

```
Sub Pregunta()
    miVar = MsgBox(Prompt:="Me gusta mi trabajo
        Title:="Cuadro de respuesta", Buttons:=
```

```
    MsgBox miVar
End Sub
```

Sintaxis de la instrucción Option

Option Compare {**Binary** | **Text** | **Database**}

En la sintaxis de la instrucción **Option Compare**, las llaves y la línea vertical indican una elección obligatoria entre tres opciones. (No escriba las llaves en la instrucción de Visual Basic). Por ejemplo, la siguiente instrucción especifica que dentro del módulo, las cadenas se comparan en un criterio de ordenación que no depende del mayúsculas o minúsculas.

```
Option Compare Text
```

Sintaxis de la instrucción Dim

Dim *nombrevariable*[**(**[*subscriptos*]**)**] [**As** *tipo*] [**,**
nombrevariable[**(**[*subscriptos*]**)**] [**As** *tipo*]] **...**

En la sintaxis de la instrucción **Dim**, la palabra **Dim** es una palabra clave exigida. El único elemento necesario es *nombrevariable* (el nombre de la variable). Por ejemplo, la siguiente instrucción crea tres variables: `miVar`, `siguienteVar` y `terceraVar`. Estas variables se declaran automáticamente como **Variant**.

```
Dim miVar, siguienteVar, terceraVar
```

El siguiente ejemplo declara una variable como **String**. Al incluir un tipo de datos se ahorra memoria y se pueden evitar errores en el código.

```
Dim miRespuesta As String
```

Para declarar varias variables en una instrucción, debe incluirse el tipo de datos para cada variable. Las variables declaradas sin un tipo de datos se declaran automáticamente como **Variant**.

```
Dim x As Integer, y As Integer, z As Integer
```

En la siguiente instrucción, a `x` e `y` se les asigna el tipo de datos **Variant**. Sólo a `z` se le asigna el tipo de datos **Integer**.

```
Dim x, y, z As Integer
```

Si se declara una variable matriz, deben incluirse los paréntesis. Los subscriptos son opcionales. La siguiente instrucción define las dimensiones de una matriz dinámica, `miMatriz`.

```
Dim miMatriz()
```

Comprender las matrices de parámetros

Una matriz de parámetros se utiliza para pasar una matriz de argumentos a un procedimiento. No es necesario saber el número de elementos de la matriz cuando se define el procedimiento.

Para identificar una matriz de parámetros se utiliza la palabra clave **ParamArray**. La matriz debe estar definida como una matriz del tipo **Variant** y debe ser el último argumento en la definición del procedimiento.

El siguiente ejemplo muestra de qué forma se puede definir un procedimiento con una matriz de parámetros.

```
Sub CualquierNumeroArgs(strNombre As String, Pa
    Dim intI As Integer

    Debug.Print strNombre; "    Puntuaciones"
```

```
    ' Utiliza la función UBound para encontrar
    For intI = 0 To UBound(intPuntuacion())
        Debug.Print "          "; intPuntuacion
    Next intI
End Sub
```

El siguiente ejemplo muestra de que formas se puede llamar este procedimiento.

```
CualquierNumeroArgs "Juan", 10, 26, 32, 15, 22,

CualquierNumeroArgs "Manuel", "Alto", "Bajo", "
```

Comprender los argumentos con nombre y opcionales

Cuando se hace una llamada a un procedimiento **Sub** o **Function**, se pueden incluir argumentos de forma posicional, es decir, en el orden en que aparecen en la definición del procedimiento, o se pueden incluir los argumentos con su nombre sin respetar ningún orden.

Por ejemplo, el siguiente procedimiento **Sub** acepta tres argumentos:

```
Sub PasaArgs(strNombre As String, intEdad As In
    Debug.Print strNombre, intEdad, dteNacimien
End Sub
```

Al llamar este procedimiento se pueden incluir sus argumentos en la posición correcta, todos separados por comas, tal y como muestra el siguiente ejemplo:

```
PasaArgs "María", 29, #21-2-69#
```

También se puede hacer una llamada a este procedimiento e incluir argumentos con nombre, separados por comas.

```
PasaArgs intEdad:=29, dteNacimiento:=#21/2/69#,
```

Un argumento nominativo está formado por el nombre del argumento seguido por dos puntos y un signo igual (**:=**), todo ello seguido por el valor del argumento.

Los argumentos con nombre son muy útiles cuando se llama a procedimientos con argumentos opcionales. Si se usan argumentos con nombre no será necesario incluir sucesivas comas para sustituir argumentos posicionales ausentes. Al usar argumentos con nombre es más fácil controlar qué argumentos se pasan y cuáles se omiten.

Los argumentos opcionales van precedidos por la palabra clave **Optional** en la definición del procedimiento. En la definición del procedimiento se puede especificar también un valor predefinido para el argumento opcional. Por ejemplo:

```
Sub ArgsOpcionales(strProvincia As String, Opti
. . .
End Sub
```

Cuando se hace una llamada a un procedimiento con un argumento opcional, se puede optar por especificar o no el argumento opcional. Si no se especifica, se usará el valor predefinido si existe. Si no se ha especificado tampoco un valor predefinido, el argumento tendrá el valor de inicialización que le corresponda a una variable del mismo tipo.

El siguiente procedimiento incluye un argumento opcional, la variable `varPais`. La función **IsMissing** determina si se ha pasado o no un argumento opcional al procedimiento.

```
Sub ArgsOpcionales(strProvincia As String, Opti
Optional strPais As String = "MÉXICO")
    If IsMissing(intRegion) AND  IsMissing(strP
```

```
        Debug.Print strProvincia
    ElseIf IsMissing(strPais) Then
        Debug.Print strProvincia, intRegion
    ElseIf IsMissing(intRegion) Then
        Debug.Print strProvincia, strPais
    Else
        Debug.Print strProvincia, intRegion, st
    End If
End Sub
```

Este procedimiento se puede llamar empleando argumentos con nombre tal y como muestran los siguientes ejemplos.

```
ArgsOpcionales strPais:="MÉXICO", strProvincia:

ArgsOpcionales strProvincia:= "SE", intRegion:=
```

Comprender los tipos de datos Variant

El tipo de datos **Variant** se especifica automáticamente si no se especifica otro tipo de datos al declarar una constante, variable, o argumento. Las variables declaradas como del tipo de datos **Variant** pueden contener valores numéricos, cadenas de texto, fecha, hora o Booleans y pueden convertir los valores que contienen de forma automática. Los valores numéricos **Variant** ocupan 16 bytes de memoria (lo que sólo es significativo en procedimientos grandes o módulos complejos) y son más lentos a la hora de su acceso que las variables de tipo explícito de los restantes tipos. Es muy raro utilizar el tipo de datos **Variant** para

una constante. Los valores de cadena **Variant** necesitan 22 bytes de memoria.

Las siguientes instrucciones crean variables **Variant**:

```
Dim miVar
Dim tuVar As Variant
laVar = "Esto es un texto."
```

La última instrucción no declara explícitamente la variable `laVar`, sino que la declara implícitamente, o automáticamente. Las variables que se declaran implícitamente se especifican como del tipo de datos **Variant**.

Sugerencia Si se especifica un tipo de datos para una variable o argumento y a continuación se utiliza un tipo erróneo de datos, se producirá un error de tipo de datos. Para evitar errores de tipo de datos, se deben usar sólo variables (del tipo de datos **Variant**) o declarar explícitamente todas las variables y especificar para ellas un tipo de datos. El último método es el preferible.

Comprender objetos, propiedades, métodos y eventos

Un objeto representa un elemento de una aplicación, como una hoja de cálculo, una celda, un diagrama, un formulario o un informe. En código de Visual Basic, un objeto debe identificarse antes de se pueda aplicar uno de los métodos del objeto o cambiar el valor de una de sus propiedades.

Una colección es un objeto que contiene varios objetos que normalmente, pero no siempre, son del mismo tipo. En Microsoft Excel, por ejemplo, el objeto **Workbooks** contiene todos los objetos **Workbook** abiertos. En Visual Basic, la colección **Forms** contiene todos los objetos **Form** existentes en una aplicación.

Los elementos de una colección se pueden identificar mediante su número o su nombre. Por ejemplo, en el siguiente procedimiento, `Libro(1)` identifica al primer objeto **Workbook** abierto.

```
Sub CierraPrimero()
```

```
    Libro(1).Close
End Sub
```

El siguiente procedimiento utiliza un nombre especificado como cadena para identificar un objeto **Form**.

```
Sub CierraForm()
    Forms("MiForm.frm").Close
End Sub
```

También es posible operar al mismo tiempo sobre toda una colección de objetos siempre que los objetos compartan métodos comunes. Por ejemplo, el siguiente procedimiento cierra todos los formularios abiertos.

```
Sub CierraTodos()
    Forms.Close
End Sub
```

Método es toda acción que puede realizar un objeto. Por ejemplo, **Add** es un método del objeto **ComboBox** ya que sirve para añadir un nuevo elemento a un cuadro combinado.

El siguiente procedimiento utiliza el método **Add** para añadir un nuevo elemento a un **ComboBox**.

```
Sub AñadeElemen(nuevoElemento as String)
    Combo1.Add nuevoElemento
End Sub
```

Propiedad es un atributo de un objeto que define una de las características del objeto, tal como su tamaño, color o localización en la pantalla, o un aspecto de su comportamiento, por ejemplo si está visible o activado. Para cambiar las características de un objeto, se cambia el valor de sus propiedades

Para dar valor a una propiedad, hay que colocar un punto detrás de la referencia a un objeto, después el nombre de la propiedad y finalmente el signo igual (=) y el nuevo valor de la propiedad. Por ejemplo, el siguiente procedimiento cambia

el título de un formulario de Visual Basic dando un valor a la propiedad **Caption**.

```
Sub CambiaNombre(nuevoTitulo)
    miForm.Caption = nuevoTitulo
End Sub
```

Hay propiedades a las que no se puede dar valor. El tema de ayuda de cada propiedad indica si es posible leer y dar valores a la propiedad (lectura/escritura), leer sólo el valor de la propiedad (sólo lectura) o sólo dar valor a la propiedad (sólo escritura).

Se puede obtener información sobre un objeto devolviendo el valor de una de sus propiedades. El siguiente procedimiento utiliza un cuadro de diálogo para presentar el título que aparece en la parte superior del formulario activo en ese momento.

```
Sub NombreFormEs()
    formNonmbre = Screen.ActiveForm.Caption
    MsgBox formNombre
End Sub
```

Evento es toda acción que puede ser reconocida por un objeto, como puede ser el clic del *mouse* o la pulsación de una tecla y para la que es posible escribir código como respuesta. Los eventos pueden ocurrir como resultado de una acción del usuario o del código de l programa, también pueden ser originados por el sistema.

Devolver objetos

Cada aplicación tiene una forma de devolver los objetos que contiene. Sin embargo estos procedimientos no son siempre iguales, por ello debe consultar el tema de ayuda correspondiente al objeto o colección que está usando en la aplicación para determinar la forma de devolver el objeto.

Crear bucles mediante código

Mediante el uso de instrucciones condicionales y instrucciones de bucle (también conocidas como estructuras de control) es posible escribir código de

Visual Basic que tome decisiones y repita determinadas acciones. Otra estructura de control útil, la instrucción **With**, permite ejecutar una serie de instrucciones sin necesidad de recalificar un objeto.

Utilizar instrucciones condicionales para tomar decisiones

Las instrucciones condicionales evalúan si una condición es **True** o **False** y a continuación especifican las instrucciones a ejecutar en función del resultado. Normalmente, una condición es una expresión que utiliza un operador de comparación para comparar un valor o variable con otro.

Elegir la instrucción condicional a utilizar

If...Then...Else: Salto a una instrucción cuando una condición es **True** o **False**

Select Case: Selección de la instrucción a ejecutar en función de un conjunto de condiciones

Utilizar bucles para repetir código

Empleando bucles es posible ejecutar un grupo de instrucciones de forma repetida. Algunos bucles repiten las instrucciones hasta que una condición es **False**, otros las repiten hasta que la condición es **True**. Hay también bucles que repiten un conjunto de instrucciones un número determinado de veces o una vez para cada objeto de una colección.

Elegir el bucle a utilizar

Do...Loop: Seguir en el bucle mientras o hasta una condición sea **True**.

For...Next: Utilizar un contador para ejecutar las instrucciones un número determinado de veces.

For Each...Next: Repetición del grupo de instrucciones para cada uno de los objetos de una colección.

Ejecutar varias instrucciones sobre el mismo objeto

Normalmente, en Visual Basic, debe especificarse un objeto antes de poder ejecutar uno de sus métodos o cambiar una de sus propiedades. Se puede usar la instrucción **With** para especificar un objeto una sola vez para una serie completa de instrucciones.

With: Ejecutar una serie de instrucciones sobre el mismo objeto

Crear procedimientos recursivos

Los procedimientos tienen un espacio limitado para almacenar variables. Cada vez que un procedimiento se llama a si mismo, consume más de ese espacio. Un procedimiento que se llama a si mismo es lo que se conoce como un procedimiento recursivo. Un procedimiento recursivo que se llama continuamente a si mismo producirá finalmente un error. Por ejemplo:

```
Function Agotar(Máximo)
    Agotar = Agotar(Máximo)
End Function
```

Este error puede resultar menos evidente cuando dos procedimientos se llaman uno al otro de forma indefinida, o cuando nunca se cumple la condición definida como fin de un bucle. Las funciones recursivos tienen sus usos. Por ejemplo, el siguiente procedimiento utiliza una función recursivo para calcular factoriales:

```
Function Factorial (N)
    If N <= 1 Then         ' Se ha llegado al fi
        Factorial = 1    ' (N = 0) abandona las
    Else                     ' Llama nuevamente
        Factorial = Factorial(N - 1) * N
    End If
End Function
```

Debe probar el procedimiento recursivo para comprobar que no se llama a si mismo tantas veces que agota la memoria disponible. Si se produce un error, compruebe que el procedimiento no se llama a si mismo de forma indefinida. Si no es así, trate de ahorrar memoria mediante:

La eliminación de variables innecesarias.

El uso de tipos de datos distintos a **Variant**.

Un nuevo estudio de la lógica del procedimiento. A menudo es posible sustituir bucles anidados por un procedimiento recursivo.

Crear variables de objeto

Se puede crear una variable de objeto de la misma forma que el objeto al que hace referencia. Se pueden activar o devolver las propiedades del objeto o utilizar cualquiera de sus métodos.

Para crear una variable de objeto:

Declare la variable de objeto.

Asigne la variable de objeto a un objeto.

Declarar variables de objeto

Para declarar una variable de objeto se ha de usar la instrucción **Dim** o una de las restantes instrucciones de declaración (**Public**, **Private**, o **Static**). Una variable que se refiere a un objeto debe ser una **Variant**, un **Object**, o un tipo específico de objeto. Por ejemplo, son válidas las siguientes declaraciones:

```
' Declara MiObjeto como tipo de datos Variant.
Dim MiObjeto
' Declara MiObjeto como un tipo de datos Object
Dim MiObjeto As Object
' Declara MiObjeto como un tipo Font.
Dim MiObjeto As Font
```

Nota Si utiliza una variable de objeto sin haberla declarado previamente, el tipo de datos predefinido de la variable de objeto es **Variant**.

Se puede declarar una variable de objeto con el tipo de datos **Object** cuando el tipo de objeto específico no se conoce hasta que se ejecuta el procedimiento. Utilice el tipo de datos **Object** para crear una referencia genérica a cualquier objeto.

Si conoce el tipo específico de objeto, debe declarar así la variable de objeto. Por ejemplo, si la aplicación contiene un tipo de objeto Ejemplo, se puede declarar una variable de objeto para ese objeto empleando una cualquiera de las dos instrucciones siguientes:

```
Dim MiObjeto As Object    ' Se declara como obj
Dim MiObjeto As Ejemplo    ' Se declara sólo co
```

Al declarar objetos específicos es posible comprobar automáticamente los tipos, el código es más rápido de ejecución y mejora su legibilidad.

Asignar una variable de objeto a un objeto

Para asignar una variable de objeto a un objeto se utiliza la instrucción **Set**. Es posible asignar una expresión de objeto o **Nothing**. Por ejemplo, son válidas las siguientes asignaciones a una variable de objeto:

```
Set MiObjeto = SuObjeto    ' Asigna referencia
Set MiObjeto = Nothing    ' Deshace la relación
```

Se puede efectuar al mismo tiempo la declaración de la variable de objeto con la asignación de un objeto a la misma, para ello se utiliza la palabra clave **New** en la instrucción **Set**. Por ejemplo:

```
Set MiObjeto = Nuevo Objeto    ' Crea y asigna
```

Al asignar a una variable de objeto el valor **Nothing** se deshace la relación que pudiera existir entre la variable de objeto y cualquier objeto específico. Así se evita que, accidentalmente, se pueda cambiar el objeto al cambiar la variable. Una variable de objeto queda definida siempre como **Nothing** al cerrar el objeto asociado, así es posible comprobar si la variable de objeto está asociada a un

objeto válido. Por ejemplo:

```
If Not MiObjeto Is Nothing Then
    ' La variable hace referencia a un objeto v
    . . .
End If
```

Por supuesto, esta prueba nunca podrá determinar con absoluta certeza si un usuario ha cerrado o no la aplicación que contiene al objeto al que se hace referencia por la variable de objeto.

Hacer referencia a la definición actual de un objeto

Utilice la palabra clave **Me** para hacer referencia a la definición actual del objeto donde se está ejecutando el código. Todos los procedimientos asociados con el objeto actual tienen acceso al objeto al que se hace referencia como **Me**. La utilización de **Me** es especialmente útil para pasar información sobre la definición actual de un objeto a un procedimiento de otro módulo. Por ejemplo, suponga que existe el siguiente procedimiento en un módulo:

```
Sub CambiaColorObjeto(MiObjetoNombre As Object)
    MiObjetoNombre.ColorFondo = RGB(Rnd * 256,
End Sub
```

Se puede hacer una llamada al procedimiento y pasarle, como argumento, la definición actual del objeto empleando la siguiente instrucción:

```
CambiaColorObjeto Me
```

Declarar constantes

Al declarar una constante, se puede asignar a un valor un nombre que tenga algún significado apropiado. La instrucción **Const** se utiliza para declarar una constante y darle valor. Una constante no puede modificarse o cambiar de valor una vez que ha sido declarada.

Se puede declarar una constante dentro de un procedimiento o al principio de un módulo, en la sección de Declarations. Las constantes a nivel de módulo son privadas, a menos que se especifique lo contrario. Para declarar una constante pública a nivel de módulo, la instrucción **Const** debe ir precedida por la palabra

clave **Public**. Se puede declarar explícitamente una constante como privada colocando la palabra clave **Private** antes de la instrucción **Const** para facilitar la lectura y comprensión del código. Si desea más información, consulte la sección "Comprender el alcance y la visibilidad" en la Ayuda de Visual Basic.

El siguiente ejemplo declara la constante **Public** `EdadCon` como un **Integer** y le asigna el valor `34`.

```
Public Const EdadCon As Integer = 34
```

Las constantes se pueden declarar de uno de los siguientes tipos de datos: **Boolean**, **Byte**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Date**, **String**, o **Variant**. Dado que ya se conoce el valor de una constante, es muy fácil elegir el tipo de datos en la instrucción **Const**. Si desea más información sobre tipos de datos, consulte la sección "Tipo de datos Summary" en la Ayuda de Visual Basic.

En una sola instrucción se pueden declarar varias constantes. Para especificar un tipo de datos, debe incluirse el tipo de datos para cada constante. En la siguiente instrucción se declaran como **Integer** las constantes `EdadCon` y `SalarioCon`.

```
Const EdadCon As Integer = 34, SalarioCon As Cu
```

Declarar matrices

Las matrices se declaran igual que las restantes variables, utilizando instrucciones **Dim**, **Static**, **Private**, o **Public**. La diferencia entre las variables escalares (aquellas que no son matrices) y las variables matriz es que normalmente se debe especificar el tamaño de la matriz. Una matriz con un tamaño especificado es una matriz de tamaño fijo. Una matriz cuyo tamaño puede cambiar mientras el programa se está ejecutando es una matriz dinámica.

Si una matriz se indexa desde 0 ó desde 1 depende del valor de la instrucción **Option Base**. Si **Option Base 1** no se especifica, todos los índices de matrices

comienzan en cero.

Declarar una matriz fija

En la siguiente línea de código se declara como matriz **Integer** una matriz de tamaño fijo con 11 filas y 11 columnas:

```
Dim MiMatriz(10, 10) As Integer
```

El primer argumento corresponde al número de filas y el segundo al número de columnas.

Como sucede en cualquier otra declaración de variable, a menos que se especifique para la matriz un tipo de datos, los elementos de ésta serán del tipo **Variant**. Cada elemento numérico **Variant** de la matriz utiliza 16 bytes. Cada elemento de cadena **Variant** utiliza 22 bytes. Para escribir código de la forma más compacta posible, debe declarar explícitamente sus matrices con un tipo de datos distinto a **Variant**. Las siguientes líneas de código comparan el tamaño de varias matrices:

```
' Una matriz Integer utiliza 22 bytes (11 eleme
ReDim MiMatrizInteger(10) As Integer

' Una matriz Double-precision utiliza 88 bytes
ReDim MiMatrizDoble(10) As Double

' Una matriz Variant utiliza al menos 176 bytes
ReDim MiMatrizVariant(10)

' La matriz Integer utiliza 100 * 100 * 2 bytes
ReDim MiMatrizInteger(99, 99) As Integer

' La matriz Double-precision utiliza 100 * 100
ReDim MiMatrizDoble (99, 99) As Double
```

```
' La matriz Variant utiliza al menos 160.000 by
ReDim MiMatrizVariant(99, 99)
```

El tamaño máximo de una matriz depende del sistema operativo y de la cantidad de memoria disponible. Es más lento utilizar una matriz que sobrepasa la cantidad de memoria RAM disponible en el sistema ya que los datos tienen que ser leídos y escritos del disco.

Declarar una matriz dinámica

Al declarar una matriz dinámica se puede cambiar el tamaño de una matriz mientras que el código se está ejecutando. Para declarar una matriz dinámica se usan las instrucciones **Static**, **Dim**, **Private**, o **Public**, dejando los paréntesis vacíos, tal y como se muestra en el siguiente ejemplo.

```
Dim MatrizSingle() As Single
```

Nota Se puede usar la instrucción **ReDim** para declarar implícitamente una matriz dentro de un procedimiento. Tenga cuidado para no cambiar el nombre de la matriz cuando use la instrucción **ReDim**, ya que se creará una segunda matriz incluso en el caso de que se haya incluido la instrucción **Option Explicit** en el módulo.

La instrucción **ReDim** se puede utilizar en un procedimiento, dentro del alcance de la matriz, para cambiar el número de dimensiones, definir el número de elementos y para definir los límites superior e inferior para cada dimensión. Se puede usar la instrucción **ReDim** para modificar la matriz dinámica cuantas veces sea necesario. Sin embargo, cada vez que se hace, se pierden los valores almacenados en la matriz. Se puede usar la instrucción **ReDim Preserve** para ampliar una matriz conservando los valores que contiene. Por ejemplo, la siguiente instrucción añade 10 nuevos elementos a la matriz `MatrizVar` sin perder los valores almacenados en los elementos originales.

```
ReDim Preserve MatrizVar(UBound(MatrizVar) + 10
```

Nota Cuando se utiliza la palabra clave **Preserve** con una matriz dinámica, sólo se puede cambiar el límite superior de la última dimensión, no pudiendo modificarse el número de dimensiones.

Declarar variables

Para declarar variables se utiliza normalmente una instrucción **Dim**. La instrucción de declaración puede incluirse en un procedimiento para crear una variable de nivel de procedimiento. O puede colocarse al principio de un módulo, en la sección Declarations, para crear una variable de nivel de módulo.

El siguiente ejemplo crea la variable `NombreTexto` y específicamente le asigna el tipo de datos String.

```
Dim NombreTexto As String
```

Si esta instrucción aparece dentro de un procedimiento, la variable `NombreTexto` se puede usar sólo en ese procedimiento. Si la instrucción aparece en la sección Declarations del módulo, la variable `NombreTexto` estará disponible en todos los procedimientos dentro del módulo, pero para los restantes módulos del proyecto. Para hacer que esta variable esté disponible para todos los procedimientos de un proyecto, basta con comenzar la declaración con la instrucción **Public**, tal y como muestra el siguiente ejemplo:

```
Public NombreTexto As String
```

Si desea más información sobre cómo dar nombre a sus variables, puede consultar la sección "Visual Basic Naming Rules" en la Ayuda de Visual Basic.

Las variables se pueden declarar como de uno de los siguientes tipos de datos: **Boolean**, **Byte**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Date**, **String** (para cadenas de longitud variable), **String *** *longitud* (para cadenas de longitud fija), **Object**, o **Variant**. Si no se especifica el tipo de datos, el tipo de datos **Variant** es el predefinido. También es posible crear un tipo definido por el usuario empleando la instrucción **Type**. Si desea más información sobre tipos de datos puede consultar la sección "Tipo de datos Summary" en la Ayuda de Visual Basic.

Se pueden declarar varias variables en una instrucción. Para especificar el tipo de datos se debe incluir un tipo de datos para cada variable. En la siguiente instrucción se declaran las variables `intX`, `intY`, e `intZ` como del tipo **Integer**.

```
Dim intX As Integer, intY As Integer, intZ As I
```

En la siguiente instrucción, `intX` e `intY` se declaran como del tipo **Variant**; y sólo `intZ` se declara como del tipo **Integer**.

```
Dim intX, intY, intZ As Integer
```

No es necesario especificar el tipo de datos en la instrucción de declaración. Si se omite, la variable será del tipo **Variant**.

Utilizar la instrucción Public

La instrucción **Public** se puede utilizar para declarar variables públicas de nivel de módulo.

```
Public NombreTexto As String
```

Las variables públicas se pueden usar en cualquier procedimiento del proyecto. Si una variable pública se declara en un módulo estándar o en un módulo de clase, también se podrá usar en los proyectos referenciados por el proyecto en que se declara la variable pública.

Utilizar la instrucción Private

La instrucción **Private** se puede usar para declarar variables privadas de nivel de módulo.

```
Private MiNombre As String
```

Las variables Private pueden ser usadas únicamente por procedimientos pertenecientes al mismo módulo.

Nota Cuando se utiliza a nivel de módulo, la instrucción **Dim** es equivalente a la instrucción **Private**. Sería aconsejable usar la instrucción **Private** para facilitar la lectura y comprensión del código.

Utilizar la instrucción Static

Cuando se utiliza la instrucción **Static** en lugar de la instrucción **Dim**, la variable declarada mantendrá su valor entre llamadas sucesivas.

Utilizar la instrucción Option Explicit

En Visual Basic se puede declarar implícitamente una variable usándola en una instrucción de asignación. Todas las variables que se definen implícitamente son del tipo **Variant**. Las variables del tipo **Variant** consumen más recursos de memoria que la mayor parte de las otros tipos de variables. Su aplicación será más eficiente si se declaran explícitamente las variables y se les asigna un tipo de datos específico. Al declararse explícitamente las variables se reduce la posibilidad de errores de nombres y el uso de nombres erróneos.

Si no desea que Visual Basic realice declaraciones implícitas, puede incluir en un módulo la instrucción **Option Explicit** antes de todos los procedimientos. Esta instrucción exige que todas las variables del módulo se declaren explícitamente. Si un módulo incluye la instrucción **Option Explicit**, se producirá un error en tiempo de compilación cuando Visual Basic encuentre un nombre de variable que no ha sido previamente declarado, o cuyo nombre se ha escrito incorrectamente.

Se puede seleccionar una opción del entorno de programación de Visual Basic para incluir automáticamente la instrucción **Option Explicit** en todos los nuevos módulos. Consulte la documentación de su aplicación para encontrar la forma de modificar las opciones de entorno de Visual Basic. Tenga en cuenta que esta opción no tiene ningún efecto sobre el código que se haya escrito con anterioridad.

Nota Las matrices fijas y dinámicas siempre se tiene que declarar explícitamente.

Declarar una variable de objeto para automatización

Cuando se utiliza una aplicación para controlar los objetos de otra aplicación, debe establecerse una referencia a la biblioteca de tipos de la otra aplicación. Una vez que se ha establecido la referencia, se pueden declarar variables de objeto conforme a su tipo más específico. Por ejemplo, si desde Microsoft Word se establece una referencia a la biblioteca de tipos de Microsoft Excel, se puede declarar una variable del tipo **Worksheet** desde Microsoft Word para representar un objeto **Worksheet** de Microsoft Excel.

Si se utiliza otra aplicación para controlar objetos de Microsoft Access, es posible, en la mayor parte de los casos, declarar variables objetos del tipo más específico. Se puede usar también la palabra clave **New** para crear automáticamente una nueva definición de un objeto. Sin embargo, puede ser necesario indicar que se trata de un objeto Microsoft Access. Por ejemplo, cuando se declara una variable de objeto para representar un formulario de Microsoft Access desde Microsoft Visual Basic, debe distinguirse entre el objeto **Form** de Microsoft Access y un objeto **Form** de Visual Basic. Para ello se incluye el nombre de la biblioteca de tipos en la declaración de la variable, como muestra el siguiente ejemplo:

```
Dim frmPedidos As New Access.Form
```

Algunas aplicaciones no reconocen algunos de los tipos de objetos de Microsoft Access. En ese caso, incluso después de establecer una referencia a la biblioteca de tipos de Microsoft Access, será necesario declarar todas las variables objeto

de Microsoft Access como del tipo **Object**. Tampoco puede usarse la palabra clave **New** para crear una nueva definición del objeto. El siguiente ejemplo muestra cómo declarar una variable que represente una nueva definición del objeto **Application** de Microsoft Access desde una aplicación que no reconoce los tipos de objeto de Microsoft Access. La aplicación crea entonces una nueva definición del objeto **Application**.

```
Dim appAccess As Object
Set appAccess = CreateObject("Access.Applicatio
```

Para determinar la sintaxis a utilizar con una aplicación determinada debe consultarse la documentación de la aplicación.

Devolver cadenas desde funciones

Algunas funciones tienen dos versiones: una que devuelve un tipo de datos Variant y otra que devuelve un tipo de datos String. Las versiones **Variant** son más aconsejables ya que las variantes realizan automáticamente la conversión entre tipos de datos distintos. También permiten que Null se propague a través de una expresión. Las versiones **String** son más eficientes ya que consumen menos memoria.

Considere la posibilidad de usar la versión **String** cuando:

El programa sea muy largo y use muchas variables.

Se escriban datos directamente en archivos de acceso directo.

Las siguientes funciones devuelven valores en una variable **String** cuando se añade el signo dólar (**$**) al nombre de función. Estas funciones tienen el mismo uso y sintaxis que las funciones equivalentes **Variant** (sin el signo dólar).

Chr$	ChrB$	*Command$
CurDir$	Date$	Dir$
Error$	Format$	Hex$
Input$	InputB$	LCase$

Left$	LeftB$	LTrim$
Mid$	MidB$	Oct$
Right$	RightB$	RTrim$
Space$	Str$	String$
Time$	Trim$	Ucase$

* Puede no estar disponible en todas las aplicaciones.

Ejecutar código cuando se establecen propiedades

Se pueden crear procedimientos **Property Let**, **Property Set** y **Property Get** que compartan el mismo nombre. Así se puede crear un grupo de procedimientos relacionados que operan conjuntamente. Una vez que se utiliza un nombre para un procedimiento **Property**, ese nombre ya no se puede usar para denominar un procedimiento **Sub** o **Function**, una variable, o un tipo definido por el usuario.

La instrucción **Property Let** permite crear un procedimiento que asigna un valor a la propiedad. Un ejemplo podría ser un procedimiento **Property** que crea una propiedad invertida para un mapa de bits en un documento. La sintaxis utilizada

para efectuar la llamada al procedimiento **Property Let** es la siguiente:

```
Form1.Invertido = True
```

El proceso real de invertir un mapa de bits en el documento se realiza enteramente dentro del procedimiento **Property Let**:

```
Private EstáInvertido As Boolean

Property Let Invertido(X As Boolean)
    EstáInvertido = X
    If EstáInvertido Then
        …
        '(instrucciones)
    Else
        '(instrucciones)
    End If
End Property
```

La variable de nivel de documento `EstáInvertido` almacena el valor asignado a la propiedad. Al declararla como **Private**, el usuario sólo puede cambiarla mediante el procedimiento **Property Let**. Utilice un nombre que permita fácilmente recordar que la variable se utiliza para la propiedad.

Este procedimiento **Property Get** se utiliza para devolver el valor actual de la propiedad `Invertido`:

```
Property Get Invertido() As Boolean
    Invertido = EstáInvertido
End Property
```

Los procedimientos Property hacen más fácil la ejecución de código al tiempo que se asigna un valor a la propiedad. Se pueden usar los procedimientos Property para ejecutar los siguientes procesos:

Antes de asignar un valor a la propiedad, para determinar el valor de la

propiedad.

Después de asignarlo, procedimientos basados en el nuevo valor.

Escribir datos en archivos

Cuando se trabaja con grandes cantidades de datos, es a menudo conveniente escribir o leer datos de un archivo. La instrucción **Open** permite crear y acceder a archivos directamente. **Open** proporciona tres tipos de acceso a archivos:

Acceso secuencial (modos **Input**, **Output** y **Append**) se utiliza para escribir archivos de texto, tales como registros de errores e informes.

Acceso directo (modo **Random**) se utiliza para leer y escribir datos en un archivo sin cerrarlo. Los archivos de acceso directo conservan la información en registros, lo que permite recuperarla de forma rápida.

Acceso binario (modo **Binary**) se utiliza para leer y escribir en cualquier byte de un archivo, sirve para almacenar o presentar una imagen de mapa de bits.

Nota La instrucción **Open** no se debe utilizar para abrir archivos del mismo tipo que la aplicación. Por ejemplo, no use **Open** para abrir un documento de Word, unahoja de cálculo de Microsoft Excel o una base de datos de Microsoft Access. Si se hiciera, se perdería la integridad del archivo se corromperían los datos almacenados.

La siguiente tabla muestra las instrucciones normalmente utilizadas para escribir y leer datos en o de un archivo.

Tipo de acceso	Escribir datos	Leer datos
Secuencial	**Print #**, **Write #**	**Input #**
Directo	**Put**	**Get**
Binario	**Put**	**Get**

Escribir instrucciones de asignación

Las instrucciones de asignación asignan un valor o expresión a una variable o constante. Las instrucciones de asignación incluyen siempre un signo igual (=). El siguiente ejemplo asigna el valor que devuelve la función **InputBox** a la variable `suNombre`.

```
Sub Pregunta()
    Dim suNombre As String
    suNombre = InputBox("¿Cómo se llama?")
    MsgBox "Su nombre es " & suNombre
End Sub
```

La instrucción **Let** es opcional y normalmente se omite. Por ejemplo, la instrucción de asignación anterior podría haberse escrito así:

```
Let suNombre = InputBox("¿Cómo se llama?").
```

La instrucción **Set** se utiliza para asignar un objeto a una variable que ha sido declarada como objeto. La palabra clave **Set** es necesaria. En el siguiente ejemplo, la instrucción **Set** asigna un rango de `Hoja1` a la variable de objeto `miCelda`:

```
Sub DarFormato()
Dim miCelda As Range
Set miCelda = Worksheets("Hoja1").Range("A1")
    With miCelda.Font
        .Bold = True
        .Italic = True
    End With
End Sub
```

Las instrucciones que establecen valores propiedad son también instrucciones de asignación. El siguiente ejemplo asigna la propiedad **Bold** del objeto **Font** para la celda activa:

```
ActiveCell.Font.Bold = True
```

Escribir instrucciones de declaración

Las instrucciones de declaración se usan para dar nombre y definir procedimientos, variables, matrices y constantes. Cuando se declara un procedimiento, variable o constante, también se define su alcance que depende del lugar en que se coloque la declaración y de las palabras clave que se usan para ello.

El siguiente ejemplo contiene tres declaraciones.

```
Sub DarFormato()
```

```
    Const limite As Integer = 33
    Dim miCelda As Range
    ' Mas instrucciones
End Sub
```

La instrucción **Sub** (con la correspondiente instrucción **End Sub**) declara un procedimiento llamado `DarFormato`. Todas las instrucciones que aparecen entre las instrucciones **Sub** y **End Sub** se ejecutan cuando el procedimiento `DarFormato` se ejecuta o se llama.

Escribir un procedimiento Sub

La instrucción **Const** declara la constante `limite,` especificando el tipo de datos **Integer** y un valor de 33.

Declarar constantes

La instrucción **Dim** declara la variable `miCelda`. El tipo de datos es objeto, en este caso, un objeto **Range** de Microsoft Excel. Se puede declarar una variable que sea cualquiera de los objetos que están accesibles a la aplicación que se está usando. Las instrucciones **Dim** son un tipo de instrucción que se utiliza para declarar variables. Otras palabras clave utilizadas en las declaraciones son **ReDim**, **Static**, **Public**, **Private** y **Const**.

Declarar variables

Escribir instrucciones de Visual Basic

Una instrucción de Visual Basic es una instrucción completa. Puede incluir palabras clave, operadores, variables, constantes y expresiones. Todas las instrucciones pertenecen a una de las tres categorías siguientes:

Las instrucciones de declaración, que dan nombre a una variable, constante o procedimiento y pueden también especificar su tipo de datos.

Escribir instrucciones de declaración

Las instrucciones de asignación, que asignan un valor o expresión a una

variable o constante.

Escribir instrucciones de asignación

Las instrucciones ejecutables, que inician acciones. Estas instrucciones pueden ejecutar un método o función y pueden saltar a o evitar bloques de código. Las instrucciones ejecutables a menudo contienen operadores condicionales o matemáticas.

Escribir instrucciones ejecutables

Continuar instrucciones en múltiples líneas

Una instrucción cabe normalmente en una línea, pero puede continuarse en la siguiente utilizando un carácter de continuación de línea. En el siguiente ejemplo, la instrucción ejecutable **MsgBox** se extiende por las tres líneas que le siguen:

```
Sub CuadroDemo()    'Este procedimiento declara
                    ' le asigna el valor Juan y
                    ' presenta en pantalla un m
    Dim miVar As String
    miVar = "Juan"
    MsgBox Prompt:="Hola " & myVar, _
        Title:="Cuadro de saludo", _
        Buttons:=vbExclamation
End Sub
```

Añadir comentarios

Los comentarios pueden explicar un procedimiento o una instrucción en particular a cualquier persona que tenga que leer el código. Visual Basic ignora los comentarios cuando ejecuta los procedimientos. Las líneas de comentario comienzan por un apóstrofe (') o con la palabra clave **Rem** seguida por un espacio y puede colocarse en cualquier lugar del procedimiento. Para añadir un

comentario a la misma línea que ocupa una instrucción, debe insertarse un apóstrofe después de esta, seguido por el comentario. Los comentarios aparecen en pantalla en color verde, color predefinido.

Comprobar errores de sintaxis

Si se presiona la tecla ENTRAR después de escribir una línea de código y la línea aparece en pantalla en color rojo (puede que aparezca también un mensaje de error), debe averiguar cuál es el problema en esa instrucción y corregirlo.

Escribir instrucciones ejecutables

Una instrucción ejecutable inicia una acción. Puede ejecutar un método o función y saltar a bloques de código o no ejecutar otros. Las instrucciones ejecutables incluyen a menudo operadores condicionales o matemáticos.

El siguiente ejemplo utiliza la instrucción **For Each...Next** para pasar por cada una de las celdas de un rango llamado `MiIntervalo` en la `Hoja1` de un libro Microsoft Excel activo. La variable `c` es una celda en la colección de celdas que componen `MiIntervalo`.

```
Sub DarFormato()
Const limite As Integer = 33
For Each c In Worksheets("Hoja1").Range("MiInte
    If c.Value > limite Then
        With c.Font
            .Bold = True
            .Italic = True
        End With
    End If
Next c
MsgBox "¡Fin!"
```

```
End Sub
```

La instrucción **If...Then...Else** del ejemplo comprueba el valor de la celda. Si el valor es mayor de 33, la instrucción **With** establece las propiedades **Bold** e **Italic** del objeto **Font** para esa celda. Las instrucciones **If...Then...Else** acaban con una instrucción **End If**.

La instrucción **With** puede evitar que haya que escribir muchas veces las mismas palabras ya que las instrucciones que contiene se ejecutan automáticamente en el objeto que sigue a la palabra clave **With**.

La instrucción **Next** llama a la siguiente celda de la colección de celdas contenida en `MiIntervalo`.

La función **MsgBox** (que presenta en pantalla un cuadro de diálogo de Visual Basic) presenta un mensaje indicando que el procedimiento **Sub** ha terminado de ejecutarse.

Escribir un procedimiento Function

Un procedimiento **Function** es una serie de instrucciones de Visual Basic encerradas entre dos instrucciones **Function** y **End Function**. Un procedimiento **Function** es similar a un procedimiento **Sub**, aunque una función puede devolver además un valor. Un procedimiento **Function** acepta argumentos, como pueden ser constantes, variables o expresiones que le pasa el procedimiento que efectúa la llamada. Si un procedimiento **Function** no tiene argumentos, la instrucción **Function** debe incluir un par de paréntesis vacíos. Una función devuelve un valor asignándolo a su nombre en una o más instrucciones del procedimiento.

En el siguiente ejemplo, la función **Celsius** calcula grados centígrados a partir de grados Fahrenheit. Cuando se llama a la función desde el procedimiento **Principal**, se le pasa una variable que contiene el valor del argumento. El resultado de los cálculos se devuelve al procedimiento que efectúo la llamada y se presenta en un cuadro de mensaje.

```
Sub Principal()
    temp = Application.InputBox(Texto:= _
        "Por favor, introduzca la temperatura e
    MsgBox "La temperatura es " & Celsius(temp)
End Sub

Function Celsius(GradosF)
    Celsius = (GradosF - 32) * 5 / 9
End Function
```

Escribir un procedimiento Property

Un procedimiento **Property** es una serie de instrucciones Visual Basic que permiten a un programador crear y manipular propiedades personalizadas.

Los procedimientos **Property** se pueden usar para crear propiedades de sólo lectura para formularios, módulos estándar y módulos de clase.

Los procedimientos **Property** deben utilizarse en el código en lugar de las variables **Public** que deben ser ejecutadas cuando se asigna un valor a la propiedad.

A diferencia de las variables **Public**, los procedimientos **Property** pueden tener cadenas de Ayuda asignadas en el Examinador de objetos.

Cuando se crea un procedimiento **Property**, se convierte en una propiedad del módulo que contiene al procedimiento. Visual Basic proporciona los siguientes tres tipos de procedimientos **Property**:

Procedimiento	Descripción
Property Let	Un procedimiento que da valor a una propiedad.
Property Get	Un procedimiento que devuelve el valor de una propiedad.
Property Set	Un procedimiento que establece una referencia a un objeto.

La sintaxis para declarar un procedimiento **Property** es la siguiente:

[**Public** | **Private**] [**Static**] **Property** {**Get** | **Let** | **Set**} *nombrepropiedad_*
[(*argumentos*)] [**As** *tipo*]

instrucciones

End Property

Los procedimientos **Property** se usan normalmente en parejas: **Property Let** con **Property Get** y **Property Set** con **Property Get**. Si se declara un procedimiento **Property Get** sólo es como si se declarara una propiedad sólo de lectura. El uso combinado de los tres procedimientos **Property** sólo es útil en el caso de variables **Variant**, dado que sólo una variable **Variant** puede contener información de un objeto u otro tipo de datos. **Property Set** está pensado para

ser usado con objetos, mientras que **Property Let** no lo está.

Los argumentos necesarios para un procedimiento **Property** son los que aparecen en la tabla siguiente:

Procedimiento	Sintaxis de la declaración
Property Get	**Property Get** *nombreprop*(1, …, *n*) **As** *tipo*
Property Let	**Property Let** *nombreprop*(1, …,,,, *n*, *n*+1)
Property Set	**Property Set** *nombreprop*(1, …, *n*, *n*+1)

Desde el primer argumento hasta el penúltimo (1, …, *n*) deben compartir los mismos nombres y tipo de datos en todos los procedimientos **Property** con el mismo nombre.

La declaración de un procedimiento **Property Get** acepta un argumento menos que las declaraciones **Property Let** y **Property Set**. El tipo de datos del procedimiento **Property Get** debe ser el mismo que el tipo de del último argumento (*n*+1) de las declaraciones **Property Let** y **Property Set** correspondientes. Por ejemplo, si se declara el siguiente procedimiento **Property Let**, la declaración **Property Get** debe usar argumentos con el mismo nombre y tipo de datos que los argumentos del procedimiento **Property Let**.

```
Property Let Nombres(intX As Integer, intY As I
    ' Aquí una instrucción.
End Property

Property Get Nombres(intX As Integer, intY As I
    ' Aquí una instrucción.
End Property
```

El tipo de datos del último argumento en una declaración **Property Set** deber ser del tipo de objeto o **Variant**.

Escribir un procedimiento Sub

Un procedimiento **Sub** es una serie de instrucciones Visual Basic, encerradas entre un par de instrucciones **Sub** y **End Sub,** que realizan acciones específicas pero no devuelven ningún valor. Un procedimiento **Sub** puede aceptar argumentos, como constantes, variables o expresiones que le pasa el procedimiento que ha efectuado la llamada. Si un procedimiento **Sub** no tiene argumentos, la instrucción **Sub** debe incluir un par de paréntesis vacío.

El siguiente procedimiento **Sub** dispone de comentarios explicativos en cada línea.

```
' Declara un procedimiento llamado ObtenInforma
' Este procedimiento Sub no acepta argumentos
Sub ObtenInformacion()
' Declara una variable de cadena llamada respue
Dim respuesta As String
```

```
' Asigna el valor que devuelve la funcion Input
respuesta = InputBox(Prompt:="¿Cómo se llama?")
        ' Instrucción condicional If...Then...E
    If respuesta = Empty Then
        ' Llama a la función MsgBox
        MsgBox Prompt:="No ha escrito su nombre
    Else
        ' Función MsgBox concatenada con la var
        MsgBox Prompt:="Su nombre es " & respue
        ' Fin de la instrucción If...Then...Els
    End If
        ' Fin del procedimiento Sub
End Sub
```

Evitar conflictos de nombres

Un conflicto de nombres Se produce al tratar de crear o utilizar un identificador que ya estaba definido. En algunos casos, los conflictos de nombres generan errores del tipo "`Detectado nombre ambiguo`" o "`Declaración duplicada en el alcance`". Los conflictos de nombres que no se detectan pueden producir fallos de programación en el código y generar resultados erróneos, especialmente, si no se declaran explícitamente todas las variables antes de utilizarlas por primera vez.

La mayor parte de los conflictos de nombres se pueden evitar comprendiendo las características de alcance de los identificadores de datos, objetos y procedimientos. Visual Basic tiene tres niveles de alcance: nivel de procedimiento, nivel de módulo privado y nivel de módulo público.

El conflicto de nombres se puede producir cuando un identificador:

Es visible en más de un nivel de alcance.

Tiene dos significados distintos en el mismo nivel.

Por ejemplo, los procedimientos pertenecientes a módulos distintos pueden compartir el mismo nombre. Es posible, por tanto, definir un procedimiento llamado `MiSub` en los módulos `Mod1` y `Mod2`. No se producirá ningún conflicto si cada uno de estos procedimientos es llamado únicamente por otros procedimientos de su mismo módulo. Sin embargo, puede producirse un error si hay una llamada a `MiSub` desde un tercer módulo y no se ha proporcionado ninguna identificación adicional que permita distinguir entre los dos `MiSub` existentes.

La mayor parte de los conflictos de nombres se pueden evitar asignando a cada

identificador un prefijo que consista en el nombre del módulo y, si es necesario, un nombre proyecto. Por ejemplo:

```
MiProyecto.MiMódulo.MiSub MiProyecto.MiMódulo.M
```

El código anterior efectúa una llamada al procedimiento **Sub** `MiSub` y pasa como argumento la variable `MiVariable`. Se puede utilizar cualquier combinación de calificadores para diferenciar identificadores inicialmente idénticos.

Visual Basic hace corresponder cada referencia a un identificador a la declaración de identificador "más parecida" que encuentra. Por ejemplo, si `MiId` se declara como **Public** en dos módulos de un proyecto (`Mod1` y `Mod2`), será posible especificar sin ninguna calificación el identificador `MiId` declarado en `Mod2` desde el mismo módulo `Mod2`, pero deberá calificarse como `Mod2.MyId` para especificarlo en `Mod1`. Lo mismo es también cierto en el caso de que `Mod2` esté en un proyecto al que se hace referencia directamente, aunque sea distinto. Sin embargo, si `Mod2` está en un proyecto al que se hace referencia indirectamente, es decir, en un proyecto al que se hace referencia en el proyecto directamente de referencia, las referencias a la variable `MiId` del módulo `Mod2` deben ir siempre precedidas por el nombre del proyecto como calificador. Si se hace una referencia a `MiId` desde un tercer módulo, al que se hace referencia directamente, la relación se establecerá con la primera declaración que se localice durante la búsqueda:

Proyectos a los que se hace referencia directamente, en el orden en que aparecen en el cuadro de diálogo **Referencias** del menú **Herramientas**.

Los módulos de cada proyecto. Observe que no hay ningún orden inherente a los módulos de un proyecto.

No se pueden utilizar nuevamente nombres de objetos de la aplicación host, por ejemplo R1C1 en Microsoft Excel, a distintos niveles de alcance.

Sugerencia Nombres ambiguos, declaraciones duplicadas, identificadores no declarados y procedimientos no localizables son algunos de los errores más comunes causados por conflictos de nombres. Es posible evitar algunos posibles conflictos de nombres y los errores de programación asociados, iniciando cada

módulo con una instrucción **Option Explicit** que obliga a declarar explícitamente las variables antes de que puedan ser utilizadas.

Los enteros utilizan menos memoria que los tipos de datos Variant y su actualización es algo más rápida. Sin embargo, la diferencia sólo es apreciable si se realizan muchos miles de operaciones. Por ejemplo:

```
Dim CuentaRápido As Integer    ' Primer caso, u
For CuentaRápido = 0 to 32766
Next CuentaRápido

Dim CuentaLento As Variant        ' Segundo cas
For CuentaLento = 0 to 32766
Next CuentaLento
```

El primero de los dos casos consume un tiempo ligeramente menor en su ejecución. Sin embargo, si `CuentaRápido` toma un valor superior a 32.767, se producirá un error. Para corregir el error se puede hacer que `CuentaRápido` pase a

ser del tipo de datos Long, que admite una gama más amplia de valores enteros. En general, cuanto más pequeño sea el tipo de datos, menos tiempo se consumirá en su actualización. Si se utiliza Variant se consumirá algo más de tiempo que si se emplea uno de los tipos de datos equivalentes.

Llamar a procedimientos con el mismo nombre

Es posible efectuar una llamada a un procedimiento ubicado en cualquier módulo del mismo proyecto que el módulo activo de la misma forma en que se haría una llamada a uno de los procedimientos del módulo activo. Sin embargo, si dos o más módulos contienen un procedimiento del mismo nombre, es necesario especificar el nombre del módulo en la instrucción de llamada, tal y como muestra el siguiente ejemplo:

```
Sub Principal()
    Módulo1.MiProcedimiento
End Sub
```

Si se asigna el mismo nombre a dos procedimientos distintos de dos proyectos diferentes, es preciso especificar el nombre de proyecto cuando se haga una llamada al procedimiento. Por ejemplo, el siguiente procedimiento efectúa una llamada al procedimiento `Principal` del módulo `MiMódulo` en el proyecto

`MiProyecto.vbp`.

```
Sub Principal()
    [MiProyecto.vbp].[MiMódulo].Principal
End Sub
```

Nota Las distintas aplicaciones dan diversos nombres a un proyecto. Así por ejemplo, en Microsoft Access, a un proyecto se le conoce como una base de datos (.mdb); en Microsoft Excel, se denomina libro de trabajo (.xls)

Sugerencias para efectuar llamadas a procedimientos

Si cambia el nombre de un módulo o proyecto, asegúrese de cambiar todas las referencias al nombre del módulo o proyectos en todas las instrucciones de llamada, de lo contrario, Visual Basic no será capaz de localizar el procedimiento llamado. Puede utilizar el comando **Reemplazar** del menú **Editar** para encontrar y reemplazar el texto correspondiente en un módulo.

Para evitar conflictos de nombres entre proyectos a los que se hace referencia es aconsejable dar a cada procedimiento un nombre único de forma que sea posible hacer llamadas a los procedimientos sin tener que especificar un nombre de módulo o proyecto.

Llamar a procedimientos Property

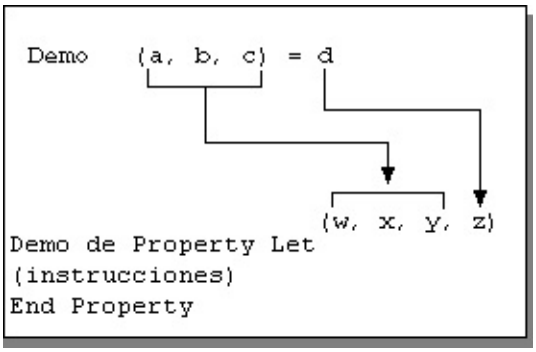
En la siguiente tabla se muestra la sintaxis a emplear en las llamadas a procedimientos Property:

Procedimiento Property	Sintaxis
Property Let	[*objeto*.]*nombreprop*(*argumentos*)] = *argumento*
Property Get	*nombrevar* = [*objeto*.]*nombreprop*(*argumentos*)]
Property Set	**Set** [*objeto*.]*nombreprop*[.(*argumentos*)] = *nombrevar*

Cuando se efectúa una llamada a los procedimientos **Property Let** o **Property Set**, a la derecha del signo igual (**=**) debe siempre aparecer un argumento.

Cuando se declaran un procedimiento **Property Let** o **Property Set** con múltiples argumentos, Visual Basic pasa el argumento situado a la derecha de la instrucción de llamada al último de los argumentos que aparece en la declaración

de **Property Let** o **Property Set**. Por ejemplo, el siguiente diagrama muestra la correspondencia existente entre los argumentos de la llamada al procedimiento **Property** con los argumentos de la declaración **Property Let**:



En la práctica, los procedimientos de propiedades con múltiples argumentos sólo sirven para crear matrices de propiedades.

Llamar a procedimientos Sub y Function

Para efectuar una llamada a un procedimiento **Sub** desde otro procedimiento, escriba el nombre del procedimiento e incluya valores para todos los argumentos requeridos. No es necesaria una instrucción **Call**, pero si la utiliza, los argumentos deben aparecer encerrados entre paréntesis.

Se puede utilizar un procedimiento **Sub** para organizar otros procedimientos de forma que sean más fáciles de entender y depurar. En el siguiente ejemplo, el procedimiento **Sub** `Principal` efectúa una llamada al procedimiento **Sub** `MultiBeep`, pasando como argumento el valor 56. Después de que `MultiBeep` acaba su ejecución, el control vuelve a `Principal` y `Principal` llama al procedimiento **Sub** `Mensaje`. `Mensaje` presenta en pantalla un cuadro de mensaje; cuando el usuario hace clic en **Aceptar**, el control vuelve a `Principal` y `Principal` termina.

```
Sub Principal()
```

```
    MultiBeep 56
    Mensaje
End Sub

Sub MultiBeep(númbips)
    For contador = 1 To númbips
        Beep
    Next contador
End Sub

Sub Mensaje()
    MsgBox "¡Es hora de descansar!"
End Sub
```

Llamar a procedimientos Sub con más de un argumento

El siguiente ejemplo muestra dos formas de llamar a un procedimiento **Sub** con más de un argumento. La segunda vez que se llama a `CalcuCasa`, es necesario utilizar paréntesis a ambos lados de los argumentos ya que se utiliza la instrucción **Call**.

```
Sub Principal()
    CalcuCasa 99800, 43100
    Call CalcuCasa(380950, 49500)
End Sub

Sub CalcuCasa(precio As Single, salario As Sing
    If 2.5 * salario <= 0.8 * precio Then
        MsgBox "No puede permitirse esta casa."
    Else
        MsgBox "Esta casa está a su alcance."
```

```
    End If
End Sub
```

Utilizar paréntesis al efectuar llamadas a procedimientos Function

Para utilizar el valor que devuelve una función debe asignar la función a una variable y encerrar los argumentos entre paréntesis, tal y como muestra el siguiente ejemplo.

```
Respuesta3 = MsgBox("¿Está contento con su sala
```

Si no está interesado en el valor que devuelve una función, puede efectuar la llamada a la función de la misma forma que si llamara a un procedimiento **Sub**. No utilice los paréntesis, incluya una lista de argumentos y no asigne la función a una variable, todo ello como muestra el siguiente ejemplo.

```
MsgBox "¡Tarea concluida!", 0, "Cuadro de tarea
```

Precaución Si en el ejemplo anterior se incluyen paréntesis, la instrucción puede producir un error de sintaxis.

Transferir argumentos con nombre

Una instrucción de un procedimiento **Sub** o **Function** puede pasar valores a los procedimientos que llama mediante argumentos con nombre. Los argumentos con nombre pueden aparecer en cualquier orden. Un argumento con nombre se compone del nombre del argumento seguido por dos puntos y un signo igual (**:=**) y el valor asignado al argumento.

El siguiente ejemplo efectúa una llamada a la función **MsgBox** utilizando argumentos con nombre que no devuelven ningún valor.

```
MsgBox Titulo:="Cuadro de tarea", Mensaje:="¡Ta
```

El siguiente ejemplo llama a la función **MsgBox** empleando argumentos con nombre. El valor que devuelve la función se asigna a la variable `respuesta3`.

```
respuesta3 = MsgBox(Titulo:="Pregunta 3", _
Mensaje:="¿Está satisfecho con su salario?", Bo
```

Reglas de asignación de nombres en Visual Basic

Para dar nombre a procedimientos, constantes, variables y argumentos en un módulo de Visual Basic han de seguirse las siguientes reglas:

El primer carácter debe ser una letra.

En el nombre no se pueden utilizar espacios, puntos (**.**), signos de interjección (**!**), ni los caracteres **@**, **&**, **$,** #.

El nombre no puede tener más de 255 caracteres de longitud.

Como regla general, no se deben usar nombres iguales a los de los procedimientos Function, instrucciones y métodos de Visual Basic. Al final puede terminar usando las mismas palabras clave que utiliza el lenguaje. Para utilizar una función intrínseca del lenguaje, o una instrucción o método, cuyo nombre coincide con uno de los nombres asignados, es preciso identificarlos explícitamente. Para ello se sitúa delante del nombre de la función intrínseca, instrucción o método, el nombre de la biblioteca de tipos asociada. Por ejemplo, si utiliza una variable llamada `Left`, la única forma de utilizar la función **Left** es escribiendo `VBA.Left`.

Los nombres no se pueden repetir dentro del mismo nivel de alcance. Por ejemplo, no se pueden declarar dos variables con el nombre `edad` dentro del mismo procedimiento. Sin embargo, se puede declarar una variable privada `edad` y una variable de nivel de procedimiento llamada `edad` dentro del mismo módulo.

Nota Visual Basic no diferencia entre mayúsculas y minúsculas, pero respeta la forma en que se escriben las instrucciones de declaración de nombres.

Trabajar con otras aplicaciones

Visual Basic puede crear nuevos objetos y recuperar otros ya existentes en muchas aplicaciones Microsoft. Otras aplicaciones pueden proporcionar también objetos que se pueden crear usando Visual Basic. Consulte la documentación de la aplicación para más detalles.

Para crear un nuevo objeto u obtener uno ya existente de otra aplicación, se utilizan las funciones **CreateObject** o **GetObject**, respectivamente:

```
' Arranca Microsoft Excel y crear un nuevo obje
Set ExcelWorksheet = CreateObject("Excel.Sheet"

' Arranca Microsoft Excel y abre un objeto Work
Set ExcelWorksheet = GetObject("Hoja1.XLS")
```

```
' Arrancar Microsoft Word.
Set WordBasic = CreateObject("Word.Basic")
```

La mayor parte de las aplicaciones disponen de un método **Exit** o **Quit** que permite cerrar la aplicación incluso cuando no está visible. Si desea más información sobre objetos, métodos y propiedades que proporciona una determinada aplicación, consulte su documentación.

Algunas aplicaciones permiten usar la palabra clave **New** para crear un objeto de cualquiera de las clases que existen en su biblioteca de tipos. Por ejemplo:

```
Dim X As New Field
```

En este caso, `Field` es un ejemplo de una de las clases existentes en la biblioteca tipo de acceso de datos. Se crea así, con esta sintaxis, una nueva definición del objeto **Field**. Consulte la documentación de la aplicación correspondiente para determinar las clases de objeto que se pueden crear de esta forma.

Transferir argumentos eficientemente

Todos los argumentos se pasan a los procedimientos por referencia, a menos que se especifique lo contrario. Esta forma de actuar es eficiente ya que se consume el mismo tiempo y el mismo espacio de almacenamiento dentro del procedimiento (4 bytes) para pasar todos los argumentos, sea cual sea su tipo de datos.

Se puede pasar un argumento por valor si se incluye la palabra clave **ByVal** en la declaración del procedimiento. Los argumentos que se pasan por valor consumen entre 2 y 16 bytes de almacenamiento del procedimiento, dependiendo del tipo

de datos del argumento. Los tipos de datos más largos consumen más tiempo al pasar por valor que los tipos más pequeños. Por esta razón, los tipos de datos **String** y **Variant** no deberían pasarse normalmente por valor.

Al pasar un argumento por valor se copia la variable original. Los cambios que pueda sufrir el argumento dentro del procedimiento no tienen ningún efecto sobre la variable original. Por ejemplo:

```
Function Factorial (ByVal MiVar As Integer)
    MiVar = MiVar - 1
    If MiVar = 0 Then
        Factorial = 1
        Exit Function
    End If
    Factorial = Factorial(MiVar) * (MiVar + 1)
End Function

' Llama a Factorial con una variable S.
S = 5
Print Factorial(S)    ' Muestra en pantalla 120
Print S                   ' Muestra en pantall
```

Si no se incluye **ByVal** en la declaración de la función, las instrucciones **Print** presentarían en pantalla los valores 1 y 0. La causa es que `MiVar` haría referencia entonces a la variable `S`, que baja de valor de 1 en 1 hasta que vale 0.

Dado que **ByVal** hace una copia del argumento, permite pasar una variante a la función **Factorial** del ejemplo anterior. No se puede pasar una variante por referencia si el procedimiento que declara el argumento es de un tipo de datos distinto.

Utilizar constantes

El código puede contener valores constantes de uso frecuente, o puede depender de ciertos números difíciles de recordar o con un significado oscuro. Puede hacer

que su código sea de más fácil lectura y mantenimiento empleando constantes. Una constante es un nombre con significado que reemplaza a un número o cadena de texto que no va a sufrir cambios. No es posible modificar una constante, ni asignarle un nuevo valor, como a una variable.

Hay tres tipos de constantes:

Constantes intrínsecas o constantes definidas por el sistema, que son proporcionadas por las aplicaciones y controles. Otras aplicaciones que proporcionan bibliotecas de objetos, como Microsoft Access, Microsoft Excel, Microsoft Project , y Microsoft Word, también proporcionan una lista de constantes que se pueden utilizar con sus objetos, métodos y propiedades. En el Examinador de objetos es posible obtener una lista de las constantes proporcionadas por cada biblioteca de objetos individual

Las constantes de Visual Basic aparecen relacionadas en la biblioteca de Visual Basic for Applications y en la biblioteca Data Access Object (DAO).

Nota Visual Basic sigue reconociendo constantes de aplicaciones creadas con versiones anteriores de Visual Basic o Visual Basic for Applications. Es posible actualizar las constantes para utilizar sólo las que aparecen en el **Examinador de objetos**. Las constantes que aparecen en el **Examinador de objetos** no tiene que declararse en su aplicación.

Constantes simbólicas o definidas por el usuario, se declaran mediante la instrucción **Const**.

Constantes de compilación condicional que se declaran empleando la instrucción **#Const**.

En versiones anteriores de Visual Basic, los nombres de constantes se representaban normalmente en mayúsculas y con guiones de subrayado. Por ejemplo:

```
MOSAICO_HORIZONTAL
```

Las constantes intrínsecas cuentan ahora con un calificador para evitar las

confusiones cuando hay constantes con el mismo nombre en más de una biblioteca de objetos, o constantes que pueden tener asignados valores distintos. Hay dos formas de cualificar los nombres de las constantes:

Con un prefijo

Con una referencia a la biblioteca

Calificar constantes por prefijo

Las constantes intrínsecas proporcionadas por todos los objetos tienen nombres en mayúsculas y minúsculas, con un prefijo de 2 caracteres que indica la biblioteca de objetos que define la constante. Las constantes de la biblioteca de objetos Visual Basic for Applications tienen el prefijo "vb" y las constantes de la biblioteca de objetos Microsoft Excel llevan el prefijo "xl". Los siguientes ejemplos muestran las variaciones entre los prefijos de los controles, en función de la biblioteca de tipos.

vbTileHorizontal

xlDialogBorder

Calificar constantes por referencia a biblioteca

También es posible cualificar la referencia a una constante empleando la siguiente sintaxis:

[*nombrebiblioteca*.] [*nombremodulo*.]*nombreconstante*

La sintaxis para cualificar constantes consta de estas partes:

Parte	Descripción
nombrebiblioteca	Opcional. El nombre de la biblioteca tipo que define a la

	constante. Para la mayor parte de los controles (no está disponible en Macintosh), se trata también del nombre de clase del control. Si no recuerda el nombre de clase del control, sitúe el puntero del *mouse* (ratón) sobre el control en la caja de herramientas. El nombre de clase aparecerá en el **ToolTip**.
nombremodulo	Opcional. El nombre del módulo, dentro de la biblioteca tipo que define a la constante. Se puede conocer el nombre del módulo empleando el **Examinador de objetos**.
nombreconstante	El nombre definido para la constante en la biblioteca tipo..

Por ejemplo:

```
Threed. LeftJustify
```

Se pueden usar instrucciones **Do...Loop** para ejecutar un bloque de instrucciones un número indefinido de veces. Las instrucciones se repiten mientras una condición sea **True** o hasta que llegue a ser **True**.

Repetir instrucciones mientras una condición es True

Hay dos formas de utilizar la palabra clave **While** para comprobar el estado de una condición en una instrucción **Do...Loop**. Se puede comprobar la condición antes de entrar en el bucle, o después de que el bucle se haya ejecutado al menos una vez.

En el siguiente procedimiento `ComPrimeroWhile`, la condición se comprueba antes de entrar en el bucle. Si `miNum` vale 9 en vez de 20, las instrucciones contenidas en el bucle no se ejecutarán nunca. En el procedimiento `ComFinalWhile`, las instrucciones contenidas en el bucle sólo se ejecutarán una vez antes de que la condición llegue a ser **False**.

```
Sub ComPrimeroWhile()
    contador = 0
    miNum = 20
```

```
    Do While miNum > 10
        miNum = miNum - 1
        contador = contador + 1
    Loop
    MsgBox "El bucle se ha repetido " & contado
End Sub

Sub ComFinalWhile()
    contador = 0
    miNum = 9
    Do
        miNum = miNum - 1
        contador = contador + 1
    Loop While miNum > 10
    MsgBox "El bucle se ha repetido " & contado
End Sub
```

Repetir instrucciones hasta que una condición llegue a ser True

Hay dos formas de utilizar la palabra clave **Until** para comprobar el estado de una condición en una instrucción **Do...Loop**. Se puede comprobar la condición antes de entrar en el bucle (como muestra el procedimiento `ComPrimeroUntil`) o se pueden comprobar después de que el bucle se haya ejecutado al menos una vez (como muestra el procedimiento `ComFinalUntil`). El bucle sigue ejecutándose mientras la condición siga siendo **False**.

```
Sub ComPrimeroUntil()
    contador = 0
    miNum = 20
    Do Until miNum = 10
        miNum = miNum - 1
        contador = contador + 1
```

```
    Loop
    MsgBox "El bucle se ha repetido " & contado
End Sub

Sub ComFinalUntil()
    contador = 0
    miNum = 1
    Do
        miNum = miNum + 1
        contador = contador + 1
    Loop Until miNum = 10
    MsgBox "El bucle se ha repetido " & counter
End Sub
```

Instrucción de salida de Do...Loop desde dentro del bucle

Es posible salir de **Do...Loop** usando la instrucción **Exit Do**. Por ejemplo, para salir de un bucle sin fin, se puede usar la instrucción **Exit Do** en el bloque de instrucciones **True** de una instrucción **If...Then...Else** o **Select Case**. Si la condición es **False**, el bucle seguirá ejecutándose normalmente.

En el siguiente ejemplo, se asigna a `miNum` un valor que crea un bucle sin fin. La instrucción **If...Then...Else** comprueba esa condición y ejecuta entonces la salida, evitando así el bucle sin fin.

```
Sub EjemploSalida()
    contador = 0
    miNum = 9
    Do Until miNum = 10
        miNum = miNum - 1
        contador = contador + 1
        If miNum < 10 Then Exit Do
```

```
    Loop
    MsgBox "El bucle se ha repetido " & contado
End Sub
```

Nota Para detener la ejecución de un bucle sin fin, presione la tecla ESC o CTRL+PAUSE.

Utilizar instrucciones For Each...Next

Las instrucciones **For Each...Next** repiten un bloque de instrucciones para cada uno de los objetos de una colección o para cada elemento de una matriz. Visual Basic asigna valor automáticamente a una variable cada vez que se ejecuta el bucle. Por ejemplo, el siguiente procedimiento cierra todos los formularios excepto el que contiene al procedimiento que se está ejecutando.

```
Sub CierraFormul()
    For Each frm In Application.Forms
```

```
        If frm.Caption <> Screen.ActiveForm.Cap
    Next
End Sub
```

El siguiente código recorre todos los elementos de una matriz e introduce en cada uno de ellos el valor de la variable índice I.

```
Dim PruebaMatriz(10) As Integer, I As Variant
For Each I In PruebaMatriz
    PruebaMatriz(I) = I
Next I
```

Recorrer un conjunto de celdas

Se puede usar el bucle **For Each...Next** para recorrer las celdas pertenecientes a un rango determinado. El siguiente procedimiento recorre las celdas del rango A1:D10 de la Página1 y convierte cualquier valor absoluto menor de 0,01 en 0 (cero).

```
Sub RedondeoACero()
    For Each miObjeto in miColeccion
        If Abs(miObjeto.Value) < 0.01 Then miOb
    Next
End Sub
```

Salir de un bucle For Each...Next antes de que finalice

Se puede salir de un bucle **For Each...Next** mediante la instrucción **Exit For**. Por ejemplo, cuando se produce un error se puede usar la instrucción **Exit For** en el bloque de instrucciones **True** de una instrucción **If...Then...Else** o **Select Case** que detecte específicamente el error. Si el error no se produce, la instrucción **If…Then…Else** es **False** y el bucle se seguirá ejecutando normalmente.

El siguiente ejemplo detecta la primera celda del rango A1:B5 que no contiene un número. Si se encuentra una celda en esas condiciones, se presenta un mensaje en pantalla y **Exit For** abandona el bucle.

```
Sub BuscaNumeros()
    For Each miObjeto In MiColeccion
        If IsNumeric(miObjeto.Value) = False Th
            MsgBox "El objeto contiene un valor
            Exit For
        End If
    Next c
End Sub
```

Utilizar instrucciones For...Next

Las instrucciones **For...Next** se pueden utilizar para repetir un bloque de instrucciones un número determinado de veces. Los bucles **For** usan una variable contador cuyo valor se aumenta o disminuye cada vez que se ejecuta el bucle.

El siguiente procedimiento hace que el equipo emita un sonido 50 veces. La instrucción **For** determina la variable contador `x` y sus valores inicial y final. La instrucción **Next** incrementa el valor de la variable contador en 1.

```
Sub Bips()
    For x = 1 To 50
        Beep
    Next x
End Sub
```

Mediante la palabra clave **Step**, se puede aumentar o disminuir la variable contador en el valor que se desee. En el siguiente ejemplo, la variable contador `j` se incrementa en 2 cada vez que se repite la ejecución del bucle. Cuando el bucle deja de ejecutarse, `total` representa la suma de 2, 4, 6, 8 y 10.

```
Sub DosTotal()
    For j = 2 To 10 Step 2
        total = total + j
    Next j
    MsgBox "El total es " & total
End Sub
```

Para disminuir la variable contador utilice un valor negativo en **Step**. Para disminuir la variable contador es preciso especificar un valor final que sea menor que el valor inicial. En el siguiente ejemplo, la variable contador `miNum` se disminuye en 2 cada vez que se repite el bucle. Cuando termina la ejecución del bucle, `total` representa la suma de 16, 14, 12, 10, 8, 6, 4 y 2.

```
Sub NuevoTotal()
    For miNum = 16 To 2 Step -2
        total = total + miNum
    Next miNum
    MsgBox "El total es " & total
End Sub
```

Nota No es necesario incluir el nombre de la variable contador después de la instrucción **Next**. En los ejemplos anteriores, el nombre de la variable contador se ha incluido para facilitar la lectura del código.

Se puede abandonar una instrucción **For...Next** antes de que el contador alcance su valor final, para ello se utiliza la instrucción **Exit For**. Por ejemplo, si se produce un error se puede usar la instrucción **Exit For** en el bloque de instrucciones **True** de una instrucción **If...Then...Else** o **Select Case** que detecte específicamente ese error. Si el error no se produce, la instrucción **If…Then…Else** es **False** y el bucle continuará ejecutándose normalmente.

Utilizar instrucciones If...Then...Else

Se puede usar la instrucción **If...Then...Else** para ejecutar una instrucción o bloque de instrucciones determinadas, dependiendo del valor de una condición. Las instrucciones **If...Then...Else** se pueden anidar en tantos niveles como sea necesario. Sin embargo, para hacer más legible el código es aconsejable utilizar una instrucción **Select Case** en vez de recurrir a múltiples niveles de instrucciones **If...Then...Else** anidadas.

Ejecutar una sola instrucción cuando una condición es True

Para ejecutar una sola instrucción cuando una condición es **True**, se puede usar la sintaxis de línea única de la instrucción **If...Then...Else**. El siguiente ejemplo muestra la sintaxis de línea única, en la que se omite el uso de la palabra clave **Else**:

```
Sub FijarFecha()
    miFecha = #13/2/95#
    If miFecha < Now Then miFecha = Now
End Sub
```

Para ejecutar más de una línea de código, es preciso utilizar la sintaxis de múltiples líneas. Esta sintaxis incluye la instrucción **End If**, tal y como muestra el siguiente ejemplo:

```
Sub AvisoUsuario(valor as Long)
    If valor = 0 Then
        Aviso.ForeColor = "Red"
        Aviso.Font.Bold = True
        Aviso.Font.Italic = True
    End If
End Sub
```

Ejecutar unas instrucciones determinadas si una condición es True y ejecutar otras si es False

Use una instrucción **If...Then...Else** para definir dos bloques de instrucciones ejecutables: un bloque que se ejecutará cuando la condición es **True** y el otro que se ejecutará si la condición es **False**.

```
Sub AvisoUsuario(valor as Long)
    If valor = 0 Then
        Aviso.ForeColor = vbRed
        Aviso.Font.Bold = True
        Aviso.Font.Italic = True
    Else
        Aviso.Forecolor = vbBlack
        Aviso.Font.Bold = False
        Aviso.Font.Italic = False
    End If
End Sub
```

Comprobar una segunda condición si la primera

condición es False

Se pueden añadir instrucciones **ElseIf** a una instrucción **If...Then...Else** para comprobar una segunda condición si la primera es **False**. Por ejemplo, el siguiente procedimiento función calcula una bonificación salarial dependiendo de la clasificación del trabajador. La instrucción que sigue a la instrucción **Else** sólo se ejecuta cuando las condiciones de todas las restantes instrucciones **If** y **ElseIf** son **False**.

```
Function Bonificación(rendimiento, salario)
    If rendimiento = 1 Then
        Bonificación = salario * 0.1
    ElseIf rendimiento = 2 Then
        Bonificación= salario * 0.09
    ElseIf rendimiento = 3 Then
        Bonificación = salario * 0.07
    Else
        Bonificación = 0
    End If
End Function
```

Utilizar instrucciones Select Case

La instrucción **Select Case** se utiliza como alternativa a las instrucciones **ElseIf** en instrucciones **If...Then...Else** cuando se compara una expresión con varios valores diferentes. Mientras que las instrucciones **If...Then...Else** pueden comparar una expresión distinta para cada instrucción **ElseIf**, la instrucción **Select Case** compara únicamente la expresión que evalúa al comienzo de la estructura de control.

En el siguiente ejemplo, la instrucción **Select Case** evalúa el argumento `rendimiento` que se pasa al procedimiento. Observe que cada instrucción **Case** puede contener más de un valor, una gama de valores, o una combinación de valores y operadores de comparación. La instrucción opcional **Case Else** se ejecuta si la instrucción **Select Case** no encuentra ninguna igualdad con los valores de la instrucciones **Case**.

```
Function Bonificación(rendimiento, salario)
    Select Case rendimiento
        Case 1
            Bonificación = salario * 0.1
        Case 2, 3
            Bonificación = salario * 0.09
        Case 4 To 6
```

```
            Bonificación = salario * 0.07
        Case Is > 8
            Bonificación = 100
        Case Else
            Bonificación = 0
    End Select
End Function
```

Utilizar instrucciones With

La instrucción **With** permite especificar una vez un objeto o tipo definido por el usuario en una serie entera de instrucciones. Las instrucciones **With** aceleran la ejecución de los procedimientos y ayudan a evitar el tener que escribir repetidas veces las mismas palabras.

El siguiente ejemplo introduce en un rango de celdas el número 30, aplica a esas celdas un formato en negrita y hace que su color de fondo sea el amarillo.

```
Sub RangoFormato()
    With Worksheets("Hoja1").Range("A1:C10")
        .Value = 30
        .Font.Bold = True
        .Interior.Color = RGB(255, 255, 0)
    End With
End Sub
```

Las instrucciones **With** se pueden anidar para aumentar su eficiencia. El siguiente ejemplo inserta una formula en la celda A1 y selecciona a continuación el tipo de letra.

```
Sub MiEntrada()
    With Workbooks("Libro1").Worksheets("Hoja1"
        .Formula = "=SQRT(50)"
        With .Font
            .Name = "Arial"
            .Bold = True
            .Size = 8
        End With
    End With
End Sub
```

Utilizar los tipos de datos eficientemente

A manos que se especifique lo contrario, a las variables no declaradas se les asigna el tipo de datos Variant. Este tipo de datos facilita la escritura de programas, pero no siempre es el tipo de datos más eficiente en su utilización.

Es aconsejable pensar en usar otros tipos de datos si:

El programa es muy grande y utiliza muchas variables.

El programa debe ejecutarse con la máxima rapidez.

Se escriben datos en archivos de acceso directo.

Además de **Variant**, se pueden utilizar los siguientes tipos de datos **Byte**, **Boolean**, **Integer**, **Long**, **Single**, **Double**, **Currency**, **Decimal**, **Date**, **Object** y **String**. Use la instrucción **Dim** para declarar una variable de un tipo determinado, por ejemplo:

```
Dim X As Integer
```

Esta instrucción declara que la variable `x` es un entero — un número no decimal

comprendido entre 32.768 y 32.767. Si se trata de asignar a x un número fuera de ese margen, se producirá un error. Si se trata de asignar a x una fracción, el número se redondeará. Por ejemplo:

```
X = 32768          ' Causa un error.
X = 5.9              ' Asigna a x el valor 6.
```

Utilizar matrices

Se puede declarar una matriz para operar con un conjunto de valores del mismo tipo de datos. Una matriz es una única variable con muchos elementos en que se pueden almacenar valores, mientras que una variable normal tiene sólo un área de almacenamiento en el que sólo se puede archivar un valor. Puede referirse a la matriz como un todo cuando se trata de hacer referencia a todos los valores que contiene, o bien hacer referencia a sus elementos individuales.

Por ejemplo, para almacenar los gastos diarios de todos los días del año se puede declarar una variable matriz con 365 elementos, mejor que declarar 365

variables. Cada elemento de una matriz contiene un valor. La siguiente instrucción declara la variable matriz `curGastos` con 365 elementos. Si no se especifica lo contrario, el índice de una matriz comienza por el cero, con lo que el límite superior de la matriz es 364 en vez de 365.

```
Dim curGastos(364) As Currency
```

Para dar valor a un elemento individual, es preciso especificar el índice del elemento. El siguiente ejemplo asigna un valor inicial de 20 a todos los elementos de la matriz.

```
Sub LlenarMatriz()
    Dim curGastos(364) As Currency
    Dim intI As Integer
    For intI = 0 to 364
        curGastos(intI) = 20
    Next
End Sub
```

Cambiar el límite inferior

Se puede usar la instrucción **Option Base** al principio de un módulo para cambiar el índice predefinido del primer elemento del 0 al 1. En el siguiente ejemplo, la instrucción **Option Base** cambia el índice del primer elemento y la instrucción **Dim** declara la variable matriz `curGastos` con 365 elementos.

```
Option Base 1
Dim curGastos(365) As Currency
```

También se puede fijar de forma explícita el límite inferior de una matriz mediante el uso de la cláusula **To** tal y como muestra el siguiente ejemplo.

```
Dim curGastos(1 To 365) As Currency
Dim strDiaSemana(7 To 13) As String
```

Almacenar valores Variant en matrices

Hay dos formas de crear matrices de valores **Variant**. Una forma consiste en declarar una matriz como del tipo de datos Variant, tal y como muestra el siguiente ejemplo:

```
Dim varDatos(3) As Variant
varDatos(0) = "Cristina Martínez"
varDatos(1) = "C/ Don Ramón de la Cruz, 73"
varDatos(2) = 38
varDatos(3) = Format("06-09-1952", "Fecha gener
```

La otra forma consiste en asignar la matriz que devuelve la función **Matriz** a una variable **Variant**, tal y como muestra el siguiente ejemplo.

```
Dim varDatos As Variant
varDatos = Array("Cristina Martínez", "C/Don Ra
Format("06-09-1952", "Fecha general"))
```

Los elementos de una matriz de valores **Variant** se identifican mediante su índice, sea cual sea la técnica que se haya usado para crear la matriz. Por ejemplo, la siguiente instrucción podría añadirse a cualquiera de los ejemplos anteriores.

```
MsgBox "Los datos de " & varDatos(0) & " se han
```

Utilizar matrices con múltiples dimensiones

En Visual Basic se pueden declarar matrices con hasta 60 dimensiones. Por ejemplo, la siguiente instrucción declara una matriz de dos dimensiones, de 5 por 10.

```
Dim sngMulti(1 To 5, 1 To 10) As Single
```

Si considera a la matriz como una tabla de dos entradas, el primer argumento

representaría a las filas y el segundo a las columnas.

Utilice instrucciones **For...Next** para operar con matrices de dimensiones múltiples. El siguiente procedimiento llena una matriz bidimensional con valores **Single**.

```
Sub LlenaMatrizMulti()
    Dim intI As Integer, intJ As Integer
    Dim sngMulti(1 To 5, 1 To 10) As Single

    ' Llena matriz con valores.
    For intI = 1 To 5
        For intJ = 1 To 10
            sngMulti(intI, intJ) = intI * intJ
            Debug.Print sngMulti(intI, intJ)
        Next intJ
    Next intI
End Sub
```

Utilizar paréntesis en el código

Los procedimientos **Sub**, las instrucciones integradas y algunos métodos no devuelven valor alguno, por lo que los argumentos no aparecen entre paréntesis. Por ejemplo:

```
MiSub "stringArgumento", integerArgumento
```

Los procedimientos **Function**, las funciones integradas y algunos métodos devuelven algún valor, que puede ser ignorado. Si se va a ignorar el valor devuelto, no es necesario incluir paréntesis. La llamada a la función se hará igual que si se estuviera llamando a un procedimiento **Sub**. Omitiendo los paréntesis, incluyendo una lista de argumentos (si los hay) y no asignando la función a una variable. Por ejemplo:

```
MsgBox "¡Tarea concluida!", 0, "Cuadro de tarea
```

Para utilizar el valor que devuelve una función, los argumentos deben encerrarse entre paréntesis tal y como muestra el siguiente ejemplo.

```
Respuesta3 = MsgBox("¿Está satisfecho con su sa
```

Una instrucción de un procedimiento **Sub** o **Function** puede pasar valores al procedimiento al que llama mediante argumentos con nombre. Las normas para el uso de paréntesis se aplican tanto si se usan argumentos con nombre como si no. Cuando se usan argumentos con nombre se pueden colocar en cualquier orden y se pueden omitir los argumentos opcionales. Los argumentos con nombre van siempre seguidos por dos puntos y un signo igual (**:=**) y finalmente el valor del argumento.

El siguiente ejemplo efectúa una llamada a la función **MsgBox** utilizando argumentos con nombre, al tiempo que ignora el valor que devuelve la función:

```
MsgBox Title:="Cuadro de tarea", Prompt:="¡Tare
```

El siguiente ejemplo efectúa una llamada a la función **MsgBox** utilizando argumentos con nombre y asigna el valor devuelto a la variable `respuesta3`:

```
respuesta3 = MsgBox(Title:="Pregunta 3", _
    Prompt:="¿Está contento con su salario?", B
```