

Convenciones de nombres de objetos

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

Prefijos sugeridos para controles

Tipo de control	Prefijo	Ejemplo
Barra de desplazamiento horizontal	hsb	hsbVolumen
Barra de desplazamiento horizontal ligera	lwhsb	lwhsbVolumen
Barra de desplazamiento lisa	fsb	fsbMover
Barra de desplazamiento vertical	vsb	vsbIndice
Barra de desplazamiento vertical ligera	lwvsb	lwvsbAño
Barra de estado	sta	staFechaHora
Barra de herramientas	tlb	tlbAcciones
Barra de progreso	prg	prgCargarArchivo
Botón animado	ani	aniBuzon
Botón de comando ligero	lwcmd	lwcmdQuitar
Botón de número	spn	spnPaginas
Botón de opción	opt	optGenero
Botón de opción ligero	lwopt	lwoptNivelEntrada
Botones de comando	cmd	cmdSalir
Casilla de verificación	chk	chkSoloLectura
Casilla de verificación ligera	lwchk	lwchkGuardar
Comunicaciones	com	comFax
Contenedor OLE	ole	oleHojaCalculo
Control (se usa en procedimientos cuando el tipo específico es desconocido)	ctr	ctrActual

Control deslizante	sld	sldEscala
Cronómetro	tmr	tmrAlarma
Cuadrícula	grd	grdPrecios
Cuadrícula de datos	dgd	dgdTitulos
Cuadrícula enlazada a datos	dbgrd	dbgrdResultadosConsulta
Cuadrícula MS Flex	msg	msgClientes
Cuadro combinado de datos	dbc	dbcAutor
Cuadro combinado de imagen	imgcbo	imgcboProducto
Cuadro combinado enlazado a datos	dbcbo	dbcboIdioma
Cuadro combinado ligero	lwcbo	lwcboAleman
Cuadro combinado, cuadro de lista desplegable	cbo	cboIngles
Cuadro de imagen	pic	picVGA
Cuadro de lista	lst	lstCodigos
Cuadro de lista de archivos	fil	filOrigen
Cuadro de lista de directorios	dir	dirSource
Cuadro de lista de unidades	drv	drvDestino
Cuadro de lista enlazada a datos	dblst	dblstTipoTrabajo
Cuadro de lista ligero	lwlst	lwlstCentroCostos
Cuadro de texto	txt	txtApellido
Cuadro de texto ligero	lwtxt	lwoptCalle
Datos	dat	datBiblio
Datos ADO	ado	adoBiblio
Datos remotos	rd	rdTitulos
Diálogo común	dlg	dlgAbrirArchivo
Etiqueta	lbl	lblMensajeAyuda
Fichas	tab	tabOpciones

Hierarchical Flexgrid	flex	flexPedidos
Forma	shp	shpCirculo
Formulario	frm	frmEntrada
Gauge	gau	gauEstado
Gráfico	gra	graGanancias
Imagen	img	imgIcono
Información del sistema	sys	sysMonitor
Línea	lin	linVertical
Lista de datos	dbl	dblEditor
Lista de imágenes	ils	ilsTodosIconos
Marco	fra	fraIdioma
Marco ligero	lwfra	lwfraOpcionesGuardar
MCI	mci	mciVideo
Mensaje MAPI	mpm	mpmEnviarMensaje
Menú	mnu	mnuAbrirArchivo
MS Chart	ch	chVentasPorRegion
MS Tab	mst	mstPrimero
Panel 3D	pnl	pnlGrupo
Picture clip	clp	clpBarraHerramientas
Repetidor de datos	drp	drpUbicacion
RichTextBox	rtf	rtfInforme
Selector de fecha	dtp	dtpEditado
Sesión MAPI	mps	mpsSesión
UpDown	upd	updDirección
Visor de árbol	tre	treOrganización
Visor de lista	lvw	lvwEncabezados

Vista de mes	mvw	mvwPeriodo
--------------	-----	------------

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

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

Objeto de base de datos	Prefijo	Ejemplo
Base de datos	db	dbCuentas
Campo	fld	fldDireccion
Conjunto de registros	rec	recPrevision
Contenedor	con	conInformes
Definición de consulta	qry	qryVentasPorRegion
Definición de tabla	tbd	tbdClientes
Documento	doc	docInformeVentas
Espacio de trabajo	wsp	wspMio
Grupo	grp	grpFinanzas
Índice	ix	idxEdad
Motor de base de datos	dbe	dbeJet
Parámetro	prm	prmCodigoTarea
Relación	rel	relDeptDeEmpleados
Usuario	usr	usrNuevo

Algunos ejemplos:

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

Prefijos sugeridos para menús

Las aplicaciones suelen usar muchos controles de menú, lo que hace útil tener un conjunto único de convenciones de nombres para estos controles. Los prefijos de controles de menús se deben extender más allá de la etiqueta inicial "mnu", agregando un prefijo adicional para cada nivel de anidamiento, con el título del menú final en la última posición de cada nombre. En la tabla siguiente hay algunos ejemplos.

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

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

Selección de prefijos para otros controles

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

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

Dim (Instrucción)

Declara [variables](#) y les asigna espacio de almacenamiento.

Sintaxis

Dim [**WithEvents**] *nombre_variable*[(*subíndices*)] [**As** [**New**] *tipo*] [, [**WithEvents**] *nombre_variable*[(*subíndices*)] [**As** [**New**] *tipo*]] ...

La sintaxis de la instrucción **Dim** consta de las siguientes partes:

Parte	Descripción
WithEvents	Opcional. Palabra clave que especifica que <i>nombre_variable</i> es una variable de objeto utilizada para responder a eventos desencadenados por un objeto ActiveX . WithEvents solamente es válido en módulos de clase . Puede declarar tantas variables individuales como desee mediante WithEvents , pero no puede crear matrices con WithEvents . No puede utilizar New con WithEvents .
<i>nombre_variable</i>	Requerido. Nombre de la variable; sigue las convenciones estándar de nombre de variable.
<i>subíndices</i>	Opcional. Dimensiones de la variable de matriz; se pueden declarar hasta 60 dimensiones múltiples. El argumento <i>subíndices</i> utiliza la sintaxis siguiente: <i>[inferior To] superior</i> [, <i>[inferior To] superior</i>] ... Cuando no se declara específicamente en <i>inferior</i> , el límite inferior de una matriz se controla mediante la instrucción Option Base . Este límite inferior es cero si no hay ninguna instrucción Option Base .
New	Opcional. Palabra clave que habilita la creación implícita de un objeto. Si utiliza New cuando declara la variable de objeto, se crea una nueva instancia del objeto como primera referencia, de forma que no tiene que utilizar la instrucción Set para asignar la referencia del objeto. La palabra clave New no se puede utilizar para declarar variables de cualquier tipo de datos , intrínseco, para declarar instancias de objetos dependientes ni con WithEvents .
<i>tipo</i>	Opcional. Tipo de datos de la variable; puede ser Byte , Boolean , Integer , Long , Currency , Single , Double , Decimal (actualmente no admitida), Date , String (para cadenas de longitud variable), String * <i>length</i> (para

	cadenas de longitud fija), Object , Variant , un tipo definido por el usuario , o un tipo de objeto . Utilice una cláusula distinta <i>As tipo</i> para cada variable que defina.
--	---

Comentarios

Las variables declaradas con **Dim** en el [nivel de módulo](#) están disponibles para todos los procedimientos disponibles sólo dentro de ese [módulo](#). En el [nivel de procedimiento](#), las variables sólo están disponibles dentro de ese procedimiento.

Utilice la instrucción **Dim** en el nivel de módulo o de procedimiento para declarar el tipo de datos de una variable. Por ejemplo, la siguiente instrucción declara una variable como **Integer**.

```
Dim NúmeroDeEmpleados As Integer
```

También puede utilizar una instrucción **Dim** para declarar el tipo de objeto de una variable. La siguiente línea declara una variable para una nueva instancia de una hoja de cálculo.

```
Dim X As New Worksheet
```

Si no utiliza la palabra clave **New** al declarar una variable de objeto, la variable que se refiere al objeto debe asignarse a un objeto existente mediante la instrucción **Set** antes de su uso. Hasta que se le asigne un objeto, la variable de objeto declarada tiene el valor especial **Nothing**, el cual indica que no se refiere a ninguna instancia en particular de un objeto.

También puede utilizar la instrucción **Dim** con paréntesis vacíos para declarar matrices dinámicas. Después de declarar una matriz dinámica, use la instrucción **ReDim** dentro de un procedimiento para definir el número de dimensiones y elementos de la matriz. Si intenta volver a declarar una dimensión para una variable de matriz cuyo tamaño se ha especificado explícitamente en una instrucción **Private**, **Public** o **Dim**, ocurrirá un error.

Si no especifica un tipo de datos o un tipo de objeto y no existe ninguna instrucción **DefTipo** en el módulo, la variable predeterminada será **Variant**.

Cuando se inicializan variables, una variable numérica se inicializa con 0, una cadena de longitud variable se inicializa con una cadena de longitud 0 ("") y una cadena de longitud fija se llena con ceros. Las variables **Variant** se inicializan con [Empty](#). Cada elemento de una variable de un tipo definido por el usuario se inicializa como si fuera una variable distinta.

Nota Cuando utiliza la instrucción **Dim** en un procedimiento, generalmente pone la instrucción **Dim** al principio del mismo.

Ejemplo de la instrucción Dim

En este ejemplo se muestra cómo utilizar la instrucción **Dim** para declarar variables. La instrucción **Dim** también se utiliza para declarar matrices. El límite inferior predeterminado para los subíndices de matrices es 0 y se puede eludir en el nivel de módulo mediante la instrucción **Option Base**.

```
' AlgúnValor y MiValor se declaran como Variant de
' manera predeterminada, con los valores Empty.
Dim AlgúnValor, MiValor

' Se declara explícitamente una variable del tipo Integer.
Dim Número As Integer

' Declaraciones múltiples en una línea. OtraVar es del tipo Variant
' porque se omite su tipo.
Dim OtraVar, Elegir As Boolean, FechaNacimiento As Date

' MatrizDías es una matriz de Variants con un índice de 51 elementos, que
va de
' 0 hasta 50, suponiendo que Option Base está configurada como 0
(predeterminada)
' en el módulo actual.
Dim MatrizDías(50)

' Matriz es una matriz bidimensional de enteros.
Dim Matriz(3, 4) As Integer

' MiMatriz es una matriz tridimensional de dobles con
' límites explícitos.
Dim MiMatriz(1 To 5, 4 To 9, 3 To 5) As Double

' FechaNacimiento es una matriz de fechas con índices del 1 al 10.
Dim FechaNacimiento(1 To 10) As Date

' MiMatriz es una matriz dinámica de variantes.
Dim MiMatriz()
```

variable

Un lugar de almacenamiento con nombre que puede contener cierto tipo de datos que puede ser modificado durante la ejecución del programa. Cada variable tiene un nombre único que la identifica dentro de su nivel de ámbito. Puede especificar un tipo de datos o no.

Nombres de variable deben comenzar con un carácter alfabético, deben ser únicos dentro del mismo ámbito, no deben contener más de 255 caracteres y no pueden contener un punto o carácter de declaración de tipo.

palabra clave

Una palabra o un símbolo reconocido como parte del lenguaje de programación; por ejemplo, una instrucción, un nombre de función o un operador.

variable de objeto

Variable que contiene una referencia a un objeto.

objeto ActiveX

Objeto que se expone a otras aplicaciones o herramientas de programación mediante interfaces de Automatización

módulo de clase

Módulo que contiene la definición de una clase (sus definiciones de propiedad y método).

matriz

Conjunto de elementos que tienen el mismo tipo de datos y que están ordenados secuencialmente. Cada elemento de una matriz posee un número de índice único que lo identifica. Los cambios efectuados a un elemento de una matriz no afectan a los demás elementos.

tipo de datos

Característica de una variable que determina qué tipo de datos puede tener. Los tipos de datos incluyen **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Date**, **String**, **Object**, **Variant** (predeterminado) y tipos definidos por el usuario, así como tipos específicos de objetos.

tipo de datos Byte

Tipo de datos utilizado para contener números enteros positivos en el intervalo de 0 a 255. Las variables de tipo Byte se almacenan como Single, números sin signo de 8 bits (1 byte).

tipo de datos Boolean

Tipo de datos que sólo tiene dos valores posibles, **True** (-1) o **False** (0). Las variables **Boolean** se almacenan como números de 16 bits (2 bytes).

tipo de datos Integer (Entero)

tipo de datos Long (Entero largo)**tipo de datos Currency (Moneda)****tipo de datos Single (Sencillo)****tipo de datos Double (Doble)**

tipos de datos decimales

Observe que en este momento el tipo de datos **Decimal** sólo se puede utilizar dentro de un tipo **Variant**. No puede declarar una variable de tipo **Decimal**. Sin embargo, puede crear un tipo **Variant** cuyo subtipo es **Decimal** utilizando la función **CDec**.

tipo de datos **Date** (Fecha)

Tipo de datos utilizado para almacenar fechas y horas como un número real. Las variables de tipo **Date** se almacenan como números de 64 bits (8 bytes). El valor de la izquierda de los decimales representa la fecha y valor de la derecha representa una hora

tipo de datos **String** (Texto)

Tipo de datos que consiste en una secuencia de caracteres que representa a los caracteres por sí mismos en vez de sus valores numéricos. Un tipo **String** puede incluir letras, números, espacios en blanco y signos de puntuación. El tipo de datos **String** puede almacenar cadenas de longitud fija en un intervalo de 0 a aproximadamente 63000 caracteres y cadenas dinámicas en un intervalo de longitud de 0 a aproximadamente 2 mil millones de caracteres. El carácter de declaración de tipo es el signo de dólar (\$) que representa el tipo **String** en Visual Basic.

tipo de datos **Object**

Tipo de datos que representa cualquier referencia a un tipo **Object**. Las variables **Object** se almacenan como direcciones de 32 bits (4 bytes) que hacen referencia a objetos.

tipo de datos **Variant**

Un tipo de datos especial que contiene datos numéricos, de cadena o de fecha así como tipos definidos por el usuario y los valores especiales **Empty** y **Null**. El tipo de datos **Variant** tiene un tamaño de almacenamiento numérico de 16 bytes y puede contener datos hasta el intervalo de un tipo **Decimal** o un tamaño de almacenamiento de caracteres de 22 bytes (más la longitud de cadena) y puede almacenar cualquier texto. La función **VarType** define el tratamiento que reciben los datos de un **Variant**. Todas las variables son del tipo **Variant** a menos que se declaren explícitamente como de cualquier otro tipo.

tipo definido por el usuario

Cualquier tipo de datos definido usando la instrucción **Type**. Los tipos de datos definidos por el usuario pueden contener uno o más elementos de cualquier tipo de datos. Las matrices de tipos definidos por el usuario y de otros tipos de datos se crean con la instrucción **Dim**. Se pueden incluir matrices de cualquier tipo dentro de tipos definidos por el usuario

Tipos de datos

Las variables son marcadores de posición que se utilizan para almacenar valores; tienen nombre y un tipo de dato. El tipo de dato de la variable determina cómo se almacenan los bits que representan esos valores en la memoria del equipo. Cuando declare una variable, también puede proporcionarle un tipo de dato. Todas las variables tienen un tipo de dato que determina la clase de datos que pueden almacenar.

De forma predeterminada, si no proporciona un tipo de dato, la variable toma el tipo de dato **Variant**. El tipo de dato **Variant** es como un camaleón; puede representar diferentes tipos de datos en distintas situaciones. No tiene que convertir estos tipos de datos cuando los asigne a una variable **Variant**: Visual Basic realiza automáticamente cualquier conversión necesaria.

Sin embargo, si sabe que una variable almacenará siempre un tipo de dato determinado, Visual Basic tratará de forma más eficiente los datos si declara la variable con ese tipo. Por ejemplo, se representa mejor una variable que va a almacenar nombres de personas como el tipo de dato **String**, ya que un nombre está siempre compuesto de caracteres.

Los tipos de datos se aplican a otras cosas además de a las variables. Cuando asigna un valor a una propiedad, dicho valor tiene un tipo de dato; los argumentos de las funciones tienen también tipos de datos. De hecho, todo lo relacionado con datos en Visual Basic tiene un tipo de dato.

También puede declarar matrices de cualquiera de los tipos fundamentales.

Para obtener más información Para obtener más información, vea la sección "Matrices", más adelante en este capítulo. La selección de tipos de datos para mejorar el rendimiento de la aplicación se explica en "Diseñar buscando el rendimiento y la compatibilidad".

Declarar variables con tipos de datos

Antes de usar una variable que no sea **Variant** debe usar las instrucciones **Private**, **Public**, **Dim** o **Static** para declararla *As tipo*. Por ejemplo, la siguiente instrucción declara un tipo **Integer**, **Double**, **String** y **Currency**, respectivamente:

```
Private I As Integer
Dim Cantidad As Double
Static SuNombre As String
Public PagadoPorJuan As Currency
```

La instrucción de declaración puede combinar varias declaraciones, como en las instrucciones siguientes:

```
Private I As Integer, Amt As Double
Private SuNombre As String, PagadoPorJuan As Currency
Private Prueba, Cantidad, J As Integer
```

Nota Si no proporciona un tipo de dato, se asigna a la variable el tipo predeterminado. En el ejemplo anterior, las variables *Prueba* y *Cantidad* tienen un tipo de dato **Variant**. Esto puede sorprenderle si su experiencia con otros lenguajes de programación le lleva a esperar que todas las variables contenidas en la misma instrucción de declaración tengan el mismo tipo que ha especificado (en este caso, **Integer**).

Tipos de datos numéricos

Visual Basic proporciona varios tipos de datos numéricos: **Integer**, **Long** (entero largo), **Single** (signo flotante de simple precisión), **Double** (signo flotante de doble precisión) y **Currency**. Usar un tipo de dato numérico emplea normalmente menos espacio de almacenamiento que un tipo **Variant**.

Si sabe que una variable siempre va a almacenar números enteros (como 12) en vez de números fraccionarios (como 3,57), declárela como un tipo **Integer** o **Long**. Las operaciones con enteros son más rápidas y estos tipos consumen menos memoria que otros tipos de datos. Resultan especialmente útiles como variables de contador en bucles **For...Next**.

Para obtener más información Para obtener más información acerca de las estructuras de control, vea "Introducción a las estructuras de control", más adelante en este mismo capítulo.

Si la variable contiene una fracción, declárela como variable **Single**, **Double** o **Currency**. El tipo de dato **Currency** acepta hasta cuatro dígitos a la derecha del separador decimal y hasta quince dígitos a la izquierda; es un tipo de dato de signo fijo adecuado para cálculos monetarios. Los números de signo flotante (**Single** y **Double**) tienen más intervalo que **Currency**, pero pueden estar sujetos a pequeños errores de redondeo.

Nota Los valores de signo flotante se pueden expresar como *mmmEeee* o *mmmDeee*, donde *mmm* es la mantisa y *eee* el exponente (potencia de 10). El valor positivo más alto de un tipo de dato **Single** es 3,402823E+38 ó 3,4 veces 10 a la 38ª potencia; el valor positivo más alto de un tipo de dato **Double** es 1,79769313486232D+308 o alrededor de 1,8 veces 10 a la 308ª potencia. Si utiliza **D** para separar la mantisa y el exponente en un literal numérico, el valor se tratará como un tipo de dato **Double**. Igualmente, usar **E** de la misma manera hace que el valor se trate como un tipo de dato **Single**.

El tipo de dato Byte

Si la variable contiene datos binarios, declárela como matriz de tipo **Byte**. (Las matrices se describen en "Matrices", más adelante en este mismo capítulo). Usar variables **Byte** para almacenar datos binarios los preserva durante las conversiones de formato. Cuando se convierten las variables **String** entre los formatos ANSI y Unicode, los datos binarios de la variable resultan dañados. Visual Basic puede convertir automáticamente entre ANSI y Unicode al:

- Leer archivos
- Escribir archivos
- Llamar a archivos DLL
- Llamar a métodos y propiedades en objetos

Todos los operadores que funcionan con enteros funcionan con el tipo de dato **Byte** excepto el de resta unaria. Puesto que **Byte** es un tipo sin signo con el intervalo 0-255, no puede

representar un valor negativo. Así, para la resta unaria, Visual Basic convierte antes el tipo **Byte** en un entero con signo.

Es posible asignar todas las variables numéricas entre sí y a variables del tipo **Variant**. Visual Basic redondea en vez de truncar la parte fraccionaria de un número de signo flotante antes de asignarlo a un entero.

Para obtener más información Para obtener más detalles acerca de las conversiones Unicode y ANSI, vea el capítulo 16 "Aspectos internacionales".

El tipo de dato String

Si tiene una variable que siempre contendrá una cadena y nunca un valor numérico, puede declararla del tipo **String**:

```
Private S As String
```

Así podrá asignar cadenas a esta variable y manipularla mediante funciones de cadena:

```
S = "Base de datos"
```

```
S = Left(S, 4)
```

De forma predeterminada, una variable o argumento de cadena es una *cadena de longitud variable*; la cadena crece o disminuye según le asigne nuevos datos. También puede declarar cadenas de longitud fija. Especifique una *cadena de longitud fija* con esta sintaxis:

String * tamaño

Por ejemplo, para declarar una cadena que tiene siempre 50 caracteres de longitud, utilice un código como este:

```
Dim NombreEmp As String * 50
```

Si asigna una cadena con menos de 50 caracteres, `NombreEmp` se rellenará con espacios en blanco hasta el total de 50 caracteres. Si asigna una cadena demasiado larga a una cadena de longitud fija, Visual Basic simplemente truncará los caracteres.

Puesto que las cadenas de longitud fija se rellenan con espacios al final, verá que las funciones **Trim** y **RTrim**, que quitan los espacios en blanco, resultan muy útiles cuando se trabaja con ellas.

Las cadenas de longitud fija se pueden declarar en módulos estándar como **Public** o **Private**. En módulos de clase y formulario, las cadenas de longitud fija deben declararse como **Private**.

Para obtener más información Vea "Funciones LTrim, RTrim y Trim" en la *Referencia del lenguaje*.

Intercambiar cadenas y números

Puede asignar una cadena a una variable numérica si la cadena representa un valor numérico. También es posible asignar un valor numérico a una variable de cadena. Por ejemplo, coloque un botón de comando, un cuadro de texto y un cuadro de lista en un formulario. Escriba el código siguiente en el evento Click del botón de comando. Ejecute la aplicación y haga clic en el botón de comando.

```
Private Sub Command1_Click()  
    Dim intX As Integer  
    Dim strY As String  
    strY = "100.23"  
    intX = strY ' Pasa la cadena a una variable numérica.
```

```

List1.AddItem Cos(strY)    ' Agrega el coseno del número de la
                           ' cadena al cuadro de lista.
strY = Cos(strY)           ' Pasa el coseno a la variable de cadena.
Text1.Text = strY         ' Se imprime la variable de cadena en el
                           ' cuadro de texto.
End Sub

```

Visual Basic convertirá automáticamente las variables al tipo de dato apropiado. Debe tener cuidado cuando intercambie números y cadenas, ya que pasar un valor no numérico a una cadena producirá un error de tiempo de ejecución.

El tipo de dato Boolean

Si tiene una variable que siempre contendrá solamente información del tipo verdadero y falso, sí y no o activado o desactivado, puede declararla del tipo **Boolean**. El valor predeterminado de **Boolean** es **False**. En el siguiente ejemplo, `blnEjecutando` es una variable **Boolean** que almacena un simple sí o no.

```

Dim blnEjecutando As Boolean
    ' Comprueba si la cinta está en marcha.
    If Recorder.Direction = 1 Then
        blnEjecutando = True
    End If

```

El tipo de dato Date

Los valores de fecha y hora pueden almacenarse en el tipo de dato específico **Date** en variables **Variant**. En ambos tipos se aplican las mismas características generales a las fechas.

Para obtener más información Vea la sección "Valores de fecha y hora almacenados en tipos Variant" en "Temas avanzados sobre Variant".

Cuando se convierten otros tipos de datos numéricos en **Date**, los valores que hay a la izquierda del signo decimal representan la información de fecha, mientras que los valores que hay a la derecha del signo decimal representan la hora. Medianoche es 0 y mediodía es 0.5. Los números enteros negativos representan fechas anteriores al 30 de diciembre de 1899.

El tipo de dato Object

Las variables **Object** se almacenan como direcciones de 32 bits (4 bytes) que hacen referencia a objetos dentro de una aplicación o de cualquier otra aplicación. Una variable declarada como **Object** es una variable que puede asignarse subsiguientemente (mediante la instrucción **Set**) para referirse a cualquier objeto real reconocido por la aplicación.

```

Dim objDb As Object
Set objDb = OpenDatabase("c:\Vb5\biblio.mdb")

```

Cuando declare variables de objeto, intente usar clases específicas (como `TextBox` en vez de `Control` o, en el caso anterior, `Database` en vez de `Object`) mejor que el tipo genérico **Object**. Visual Basic puede resolver referencias a las propiedades y métodos de objetos con tipos específicos antes de que ejecute una aplicación. Esto permite a la aplicación funcionar

más rápido en tiempo de ejecución. En el Examinador de objetos se muestran las clases específicas.

Cuando trabaje con objetos de otras aplicaciones, en vez de usar **Variant** o el tipo genérico **Object**, declare los objetos como se muestran en la lista **Clases** en el Examinador de objetos. Esto asegura que Visual Basic reconozca el tipo específico de objeto al que está haciendo referencia, lo que permite resolver la referencia en tiempo de ejecución.

Para obtener más información Para obtener más información acerca de la creación y asignación de objetos y variables de objetos, vea "Crear objetos" más adelante en este mismo capítulo.

Convertir tipos de datos

Visual Basic proporciona varias funciones de conversión que puede usar para convertir valores en tipos de datos específicos. Por ejemplo, para convertir un valor a **Currency**, utilice la función **CCur**:

```
PagoPorSemana = CCur(horas * PagoPorHora)
```

Funciones de conversión	Convierten una expresión en
Cbool	Boolean
Cbyte	Byte
Ccur	Currency
Cdate	Date
CDbl	Double
Cint	Integer
CLng	Long
CSng	Single
CStr	String
Cvar	Variant
CVErr	Error

Nota Los valores que se pasan a una función de conversión deben ser válidos para el tipo de dato de destino o se producirá un error. Por ejemplo, si intenta convertir un tipo **Long** en un **Integer**, el tipo **Long** debe estar en el intervalo válido del tipo de dato **Integer**.

Para obtener más información Busque las funciones específicas de conversión en la *Referencia del lenguaje*.

El tipo de dato Variant

Una variable **Variant** es capaz de almacenar todos los tipos de datos definidos en el sistema. No tiene que convertir entre esos tipos de datos si los asigna a una variable **Variant**; Visual Basic realiza automáticamente cualquier conversión necesaria. Por ejemplo:

```
Dim AlgúnValor      ' De forma predeterminada es un tipo Variant.
AlgúnValor = "17"    ' AlgúnValor contiene "17" (cadena de dos
                    ' caracteres).
AlgúnValor = AlgúnValor - 15    ' AlgúnValor ahora contiene
                                ' el valor numérico 2.
AlgúnValor = "U" & AlgúnValor    ' AlgúnValor ahora contiene
                                ' "U2" (una cadena de dos
                                ' caracteres).
```

Si bien puede realizar operaciones con variables **Variant** sin saber exactamente el tipo de dato que contienen, hay algunas trampas que debe evitar.

- Si realiza operaciones aritméticas o funciones sobre un **Variant**, el **Variant** debe contener un número. Para obtener más detalles, vea la sección "Valores numéricos almacenados en tipos Variant" en "Temas avanzados sobre Variant".
- Si está concatenando cadenas, utilice el operador **&** en vez del operador **+**. Para obtener más detalles, vea la sección "Cadenas almacenadas en tipos Variant" en "Temas avanzados sobre Variant".

Además de poder actuar como otros tipos de datos estándar, los **Variant** también pueden contener tres valores especiales: **Empty**, **Null** y **Error**.

El valor Empty

A veces necesitará saber si se ha asignado un valor a una variable existente. Una variable **Variant** tiene el valor **Empty** antes de asignarle un valor. El valor **Empty** es un valor especial distinto de 0, una cadena de longitud cero ("") o el valor **Null**. Puede probar el valor **Empty** con la función **IsEmpty**:

```
If IsEmpty(Z) Then Z = 0
```

Cuando un **Variant** contiene el valor **Empty**, puede usarlo en expresiones; se trata como un 0 o una cadena de longitud cero, dependiendo de la expresión.

El valor **Empty** desaparece tan pronto como se asigna cualquier valor (incluyendo 0, una cadena de longitud cero o **Null**) a una variable **Variant**. Puede establecer una variable **Variant** de nuevo como **Empty** si asigna la palabra clave **Empty** al **Variant**.

El valor Null

El tipo de dato **Variant** puede contener otro valor especial: **Null**. **Null** se utiliza comúnmente en aplicaciones de bases de datos para indicar datos desconocidos o que faltan. Debido a la forma en que se utiliza en las bases de datos, **Null** tiene algunas características únicas:

- Las expresiones que utilizan **Null** dan como resultado siempre un **Null**. Así, se dice que **Null** se "propaga" a través de expresiones; si cualquier parte de la expresión da como resultado un **Null**, la expresión entera tiene el valor **Null**.
- Al pasar un **Null**, un **Variant** que contenga un **Null** o una expresión que dé como resultado un **Null** como argumento de la mayoría de las funciones hace que la función devuelva un **Null**.
- Los valores **Null** se propagan a través de funciones intrínsecas que devuelven tipos de datos **Variant**.

También puede asignar un **Null** mediante la palabra clave **Null**:

```
Z = Null
```

Puede usar la función **IsNull** para probar si una variable **Variant** contiene un **Null**:

```
If IsNull(X) And IsNull(Y) Then
    Z = Null
Else
    Z = 0
End If
```

Si asigna **Null** a una variable de un tipo que no sea **Variant**, se producirá un error interceptable. Asignar **Null** a una variable **Variant** no provoca ningún error y el **Null** se propagará a través de expresiones que contengan variables **Variant** (**Null** no se propaga a través de determinadas funciones). Puede devolver **Null** desde cualquier procedimiento **Function** con un valor de devolución de tipo **Variant**.

Null no se asigna a las variables a menos que se haga explícitamente, por lo que si no utiliza **Null** en su aplicación, no tendrá que escribir código que compruebe su existencia y lo trate.

Para obtener más información Para obtener más información acerca de cómo usar **Null** en expresiones, vea "Null" en la *Referencia del lenguaje*.

El valor Error

En un **Variant**, **Error** es un valor especial que se utiliza para indicar que se ha producido una condición de error en un procedimiento. Sin embargo, a diferencia de otros tipos de error, no se produce el tratamiento de errores a nivel normal de aplicación. Esto le permite a usted o a la propia aplicación elegir alternativas basadas en el valor del error. Los valores de error se crean convirtiendo números reales en valores de error mediante la función **CVErr**.

Para obtener más información Para obtener más información acerca de cómo usar el valor **Error** en expresiones, vea "Función CVErr" en la *Referencia del lenguaje*. Para obtener más información acerca del tratamiento de errores, vea el capítulo 13 "Depurar el código y tratar errores". Para obtener más información acerca del tipo de dato **Variant**, vea "Temas avanzados sobre Variant".

Val (Función)

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

Sintaxis

Val(*cadena*)

El [argumento](#) obligatorio *cadena* es cualquier [expresión de cadena](#) válida.

Comentarios

La función **Val** deja de leer la cadena en el primer carácter que no puede reconocer como parte de un número. Los símbolos y caracteres que se consideran a menudo parte de valores numéricos, como signos de moneda y comas, no se reconocen. Sin embargo, la función reconoce los prefijos de base &O (para octal) y &H (para hexadecimal). Los espacios en blanco, los tabuladores y los avances de línea se eliminan del argumento.

Lo siguiente devuelve el valor 1615198:

```
val ("      1615 198 Calle N.E.")
```

En el código que se muestra a continuación, **Val** devuelve el valor decimal -1 correspondiente al valor hexadecimal entre paréntesis:

```
val ("&HFFFF")
```

Nota La función **Val** sólo reconoce el punto (.) como separador decimal válido. Cuando se utilizan separadores decimales diferentes, como en aplicaciones internacionales, debe utilizar **Cdbl** para convertir una cadena a un número.

Option Base (Instrucción)

Se usa en el [nivel de módulo](#) para declarar el límite inferior predeterminado para subíndices de [matriz](#).

Sintaxis

Option Base {0 | 1}

Comentarios

Como la base predeterminada es **0**, la instrucción **Option Base** nunca se requiere. Sin embargo, si usa la [instrucción](#) debe aparecer en un [módulo](#) antes de cualquier [procedimiento](#). **Option Base** sólo puede aparecer una vez en un módulo y debe preceder a las [declaraciones](#) de matriz que incluyen las dimensiones.

Nota La cláusula **To** en las instrucciones **Dim**, **Private**, **Public**, **ReDim** y **Static** proporciona una forma más flexible de controlar el intervalo de los subíndices de una matriz. Sin embargo, si no establece explícitamente el límite inferior con una cláusula **To**, puede usar **Option Base** para cambiar el límite inferior predeterminado a 1. La base de una matriz creada con la función **Array** o la palabra clave **ParamArray** es cero; **Option Base** no afecta a **ParamArray** (o la función **Array**, cuando se califica con el nombre de su biblioteca de tipo, por ejemplo **VBA.Array**).

La instrucción **Option Base** sólo afecta el límite inferior de las matrices en el módulo donde se ubica la instrucción.

Ejemplo de la instrucción Option Base

En este ejemplo se utiliza la instrucción **Option Base** para eludir el valor predeterminado 0 del subíndice de la matriz de base. La función **LBound** devuelve el menor subíndice disponible de la dimensión indicada de la matriz. La instrucción **Option Base** sólo se utiliza en el nivel de módulo.

```
Option Base 1      ' Establece los subíndices de matriz predeterminados a 1.
```

```
Dim Menor
Dim MiMatriz(20), Matriz2D(3, 4)      ' Declara las variables de la matriz.
Dim MatrizCero(0 To 5)                ' Elude el subíndice base predeterminado.
' Utilice la función LBound para determinar los límites inferiores de las
matrices.
Menor = LBound(MiMatriz)              ' Devuelve 1.
Menor = LBound(Matriz2D, 2)          ' Devuelve 1.
Menor = LBound(MatrizCero)           ' Devuelve 0.
```

Option Explicit (Instrucción)

Se usa en el [nivel de módulo](#) para forzar declaraciones explícitas de todas las [variables](#) en dicho [módulo](#).

Sintaxis

Option Explicit

Comentarios

Si se usa, la instrucción **Option Explicit** debe aparecer en un módulo antes de cualquier [procedimiento](#).

Cuando **Option Explicit** aparece en un módulo, debe declarar explícitamente todas las variables mediante las instrucciones **Dim**, **Private**, **Public**, **ReDim** o **Static**. Si intenta usar un nombre de variable no declarado, ocurrirá un error en [tiempo de compilación](#).

Si no usa la instrucción **Option Explicit** todas las variables no declaradas son **Variant**, a menos que el tipo predeterminado esté especificado de otra manera con una instrucción **Deftipo**.

Nota Utilice **Option Explicit** para evitar escribir incorrectamente el nombre de una variable existente o para evitar confusiones en el código, donde el [alcance](#) de la variable no está claro.

Ejemplo de la instrucción Option Explicit

En este ejemplo se utiliza la instrucción **Option Explicit** para forzar la declaración explícita de todas las variables. Si se intenta utilizar una variable no declarada se obtiene un error en el tiempo de compilación. La instrucción **Option Explicit** sólo se utiliza en el nivel de módulo.

```
Option Explicit    ' Fuerza la declaración explícita de variables.  
Dim MiVar          ' Declara la variable.  
MiEnt = 10          ' La variable no declarada genera un error.  
MiVar = 10          ' La variable declarada no generará error.
```

If...Then...Else (Instrucción)

Ejecuta condicionalmente un grupo de [instrucciones](#), dependiendo del valor de una [expresión](#).

Sintaxis

If *condición* **Then** [*instrucciones*]-[**Else** *instrucciones_else*]

Puede utilizar la siguiente sintaxis en formato de bloque:

If *condición* **Then**
[*instrucciones*]

[**ElseIf** *condición-n* **Then**
[*instrucciones_elseif*] . . .

[**Else**
[*instrucciones_else*]]

End If

La sintaxis de la instrucción **If...Then...Else** consta de tres partes:

Parte	Descripción
<i>condición</i>	Requerido. Uno o más de los siguientes dos tipos de expresiones:
	Una expresión numérica o expresión de cadena que puede ser evaluada como True o False . Si <i>condición</i> es Null , <i>condición</i> se considera False .
	Una expresión del formulario TypeOf nombre_objeto Is tipo_objeto . El <i>nombre_objeto</i> es cualquier referencia al objeto y <i>tipo_objeto</i> es cualquier tipo de objeto válido. La expresión es True si <i>nombre_objeto</i> es del tipo de objeto especificado por <i>tipo_objeto</i> ; en caso contrario es False .
<i>instrucciones</i>	Opcional en formato de bloque; se requiere en formato de línea sencilla que no tenga una cláusula Else . Una o más instrucciones separadas por dos puntos ejecutados si la <i>condición</i> es True .
<i>condición-n</i>	Opcional. Igual que <i>condición</i> .

<i>instrucciones_elseif</i>	Opcional. Una o más instrucciones ejecutadas si la <i>condición-n</i> asociada es True .
<i>instrucciones_else</i>	Opcional. Una o más instrucciones ejecutadas si ninguna de las expresiones anteriores <i>condición</i> o <i>condición-n</i> es True .

Comentarios

Puede utilizar la forma de una sola línea (Sintaxis 1) para pruebas cortas y sencillas. Sin embargo, el formato de bloque (Sintaxis 2) proporciona más estructura y flexibilidad que la forma de línea simple y, generalmente, es más fácil de leer, de mantener y de depurar.

Nota Con la sintaxis es posible ejecutar múltiples *instrucciones* como resultado de una decisión **If...Then**, pero todas deben estar en la misma línea y separadas por dos puntos, como en la instrucción siguiente:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

Una instrucción con formato de bloque **If** debe ser la primera de la línea. Las partes **Else**, **ElseIf** y **End If**, de la instrucción, solamente pueden ir precedidas de un [número de línea](#) o una [etiqueta de línea](#). El bloque **If** debe terminar con una instrucción **End If**.

Para determinar si una instrucción **If** es un bloque, examine lo que sigue a la [palabra Then](#). Si lo que aparece detrás de **Then** en la misma línea no es un [comentario](#), la instrucción se considera como una instrucción **If** de una sola línea.

Las cláusulas **Else** y **ElseIf** son opcionales. Puede tener en un bloque **ElseIf**, tantas cláusulas **If** como desee, pero ninguna puede aparecer después de una cláusula **Else**. Las instrucciones de bloque **If** se pueden anidar; es decir, unas pueden contener a otras.

Cuando se ejecuta un bloque **If** (Sintaxis 2), se prueba *condición*. Si *condición* es **True**, se ejecutan las instrucciones que están a continuación de **Then**. Si *condición* es **False**, se evalúan una a una las condiciones **ElseIf** (si existen). Cuando se encuentra una condición **True** se ejecutan las instrucciones que siguen inmediatamente a la instrucción **Then** asociada. Si ninguna de las condiciones **ElseIf** es **True** (o si no hay cláusulas **ElseIf**), se ejecutan las instrucciones que siguen a **Else**. Después de la ejecución de las instrucciones que siguen a **Then** o **Else**, la ejecución continúa con la instrucción que sigue a **End If**.

Sugerencia **Select Case** puede ser más útil cuando se evalúa una única expresión que tiene varias acciones posibles. Sin embargo, la cláusula **TypeOf nombre_objeto Is tipo_objeto** no se puede utilizar en una instrucción **Select Case**.

Nota No se puede usar **TypeOf** con tipos de datos predefinidos como Long, Integer y así sucesivamente, excepto en el tipo de datos Object.

Ejemplo de la instrucción If...Then...Else

Este ejemplo muestra los dos posibles usos de **If...Then...Else** como bloque y en una única línea. También muestra el uso de **If TypeOf...Then...Else**.

```
Dim Número, Dígitos, MiCadena
Número = 53      ' Inicializa variable.
If Número < 10 Then
    Dígitos = 1
ElseIf Número < 100 Then
    ' La condición es True, por lo que se ejecuta la siguiente instrucción.
    Dígitos = 2
Else
    Dígitos = 3
End If

' Asigna un valor con la sintaxis de una línea.
If Dígitos = 1 Then MiCadena = "Una" Else MiCadena = "Más de una"
```

Puede utilizar **If TypeOf** para determinar si el control que se pasa a un procedimiento es un cuadro de texto.

```
Sub ControlProcessor(MiControl As Control)
    If TypeOf MiControl Is CommandButton Then
        Debug.Print "Ha pasado un " & TypeName(MiControl)
    ElseIf TypeOf MiControl Is CheckBox Then
        Debug.Print "Ha pasado un " & TypeName(MiControl)
    ElseIf TypeOf MiControl Is TextBox Then
        Debug.Print "Ha pasado un " & TypeName(MiControl)
    End If
End Sub
```

Definiciones que intervienen

instrucción

Una unidad sintácticamente completa que expresa un tipo de acción, declaración o definición. Normalmente una instrucción tiene una sola línea aunque es posible utilizar dos puntos (:) para poner más de una instrucción en una línea. También se puede utilizar un carácter de continuación de línea (__) para continuar una sola línea lógica en una segunda línea física.

expresión

Una combinación de palabras clave, operadores, variables y constantes, que produce una cadena, un número o un objeto. Una expresión puede realizar un cálculo, manipular caracteres o verificar datos.

expresión numérica

Cualquier expresión que puede ser evaluada como un número. Los elementos de una expresión pueden incluir cualquier combinación de palabras clave, variables, constantes y operadores que dan como resultado un número.

expresión de cadena

Cualquier expresión cuyo valor es equivalente a una secuencia de caracteres contiguos. Los elementos de la expresión pueden incluir una función que devuelve una cadena, un literal de cadena, una constante de cadena, una variable de cadena, una cadena **Variant** o una función que devuelve una cadena **Variant** (**VarType** 8).

Null

Un valor que indica que una variable contiene datos no válidos. **Null** es el resultado de una asignación explícita de una variable como **Null** o cualquier operación entre expresiones que contienen **Null**.

tipo de objeto

Un tipo de objeto expuesto por una aplicación por medio de la Automatización. Por ejemplo, **Aplicación**, **Archivo**, **Intervalo** y **Hoja de cálculo**. Utilice el **Examinador de objetos** o consulte la documentación de la aplicación para obtener una lista completa de objetos disponibles.

número de línea

Un número de línea se usa para identificar una sola línea de código. Este número puede ser cualquier combinación de dígitos que sea única dentro del módulo donde se usa. Los números de línea deben comenzar en la primera columna.

etiqueta de línea

Una etiqueta de línea se usa para identificar una sola línea de código. Puede ser cualquier combinación de caracteres que comience con una letra y que termine con dos puntos (:). Las etiquetas de línea no distinguen mayúsculas y minúsculas y deben empezar en la primera columna.

comentario

Texto agregado a un código por un programador, que explica cómo funciona el código. En Visual Basic cada línea de comentario comienza con un apóstrofo (') o con la palabra clave **Rem** seguida por un espacio.

Select Case (Instrucción)

Ejecuta uno de varios grupos de [instrucciones](#), dependiendo del valor de una [expresión](#).

Sintaxis

```
Select Case expresión_prueba  
[Case lista_expresión-n  
[instrucciones-n]] . . .  
[Case Else  
[instrucciones_else]]
```

End Select

La sintaxis de la instrucción **Select Case** consta de las siguientes partes:

Parte	Descripción
<i>expresión_prueba</i>	Requerido. Cualquier expresión numérica o expresión de cadena .
<i>lista_expresión-n</i>	Requerido si aparece la palabra clave Case . Lista delimitada por comas de una o más de las formas siguientes: <i>expresión</i> , <i>expresión To expresión</i> , <i>Is expresión operador_de_comparación</i> . La palabra clave especifica un intervalo de valores. Si se utiliza la palabra clave To , el valor menor debe aparecer antes de To . Utilice la palabra clave Is con operadores de comparación (excepto Is y Like) para especificar un intervalo de valores. Si no se escribe, la palabra clave Is se insertará automáticamente.
<i>instrucciones-n</i>	Opcional. Una o más instrucciones ejecutadas si <i>expresión_prueba</i> coincide con cualquier parte de <i>lista_expresión-n</i> .
<i>instrucciones_else</i>	Opcional. Una o más instrucciones que se ejecuten si <i>expresión_prueba</i> no coincide con nada de la cláusula Case .

Comentarios

Si *expresión_prueba* coincide con cualquier *lista_expresión* asociada con una cláusula **Case**, las instrucciones que siguen a esa cláusula **Case** se ejecutan hasta la siguiente cláusula **Case** o, para la última cláusula, hasta la instrucción **End Select**. El control pasa después a la instrucción que sigue a **End Select**. Si *expresión_prueba* coincide con una

expresión de *lista_expresión* en más de una cláusula **Case**, sólo se ejecutan las instrucciones que siguen a la primera coincidencia.

La cláusula **Case Else** se utiliza para indicar las instrucciones que se van a ejecutar si no se encuentran coincidencias entre *expresión_prueba* y una *lista_expresión* en cualquiera de las otras selecciones de **Case**. Aunque no es necesario, es buena idea tener una instrucción **Case Else** en el bloque **Select Case** para controlar valores imprevistos de *expresión_prueba*. Cuando no hay una instrucción **Case Else** y ninguna expresión de la lista en las cláusulas **Case** coincide con la expresión de prueba, la ejecución continúa en la instrucción que sigue a **End Select**.

Se pueden utilizar expresiones múltiples o intervalos en cada cláusula **Case**. Por ejemplo, la línea siguiente es válida:

```
Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber
```

Nota El operador de comparación **Is** no es lo mismo que la palabra clave **Is** utilizada en la instrucción **Select Case**.

También puede especificar intervalos y expresiones múltiples para cadenas de caracteres. En el siguiente ejemplo, **Case** coincide con las cadenas que son exactamente iguales a todo, cadenas que están entre nueces y sopa en orden alfabético y el valor actual de ElemPrueba:

```
Case "iguales a todo", "nueces" To "sopa", ElemPrueba
```

Las instrucciones **Select Case** se pueden anidar. Cada instrucción **Select Case** debe tener su correspondiente instrucción **End Select**.

Ejemplo de la instrucción Select Case

En este ejemplo se utiliza la instrucción **Select Case** para evaluar el valor de una variable. La segunda cláusula **Case** contiene el valor de la variable que se evalúa y, por tanto, sólo se ejecuta la instrucción asociada con ella.

```
Dim Número
Número = 8 ' Inicializa variable.
Select Case Número ' Evalúa Número.
Case 1 To 5 ' Número entre 1 y 5, inclusive.
    Debug.Print "Entre 1 y 5"
'Es la única cláusula Case cuyo valor es True.
Case 6, 7, 8 ' Número entre 6 y 8.
    Debug.Print "Entre 6 y 8"
Case 9 To 10 ' Número es 9 ó 10.
    Debug.Print "Mayor que 8"
Case Else ' Otros valores.
    Debug.Print "No está entre 1 y 10"
End Select
```

Estructuras de decisión

Los procedimientos de Visual Basic pueden probar condiciones y, dependiendo de los resultados de la prueba, realizar diferentes operaciones. Entre las estructuras de decisión que acepta Visual Basic se incluyen las siguientes:

- If...Then
- If...Then...Else
- Select Case

If...Then

Use la estructura **If...Then** para ejecutar una o más instrucciones basadas en una condición. Puede usar la sintaxis de una línea o un *bloque* de varias líneas:

If condición Then instrucción

If condición Then

instrucciones

End If

Condición normalmente es una comparación, pero puede ser cualquier expresión que dé como resultado un valor numérico. Visual Basic interpreta este valor como **True** o **False**; un valor numérico cero es **False** y se considera **True** cualquier valor numérico distinto de cero. Si *condición* es **True**, Visual Basic ejecuta todas las *instrucciones* que siguen a la palabra clave **Then**. Puede usar la sintaxis de una línea o de varias líneas para ejecutar una instrucción basada en una condición (estos dos ejemplos son equivalentes):

```
If cualquierFecha < Now Then cualquierFecha = Now
```

```
If cualquierFecha < Now Then  
    cualquierFecha = Now  
End If
```

Observe que el formato de una única línea de **If...Then** no utiliza la instrucción **End If**. Si desea ejecutar más de una línea de código cuando *condición* sea **True**, debe usar la sintaxis de bloque de varias líneas **If...Then...End If**.

```
If cualquierFecha < Now Then  
    cualquierFecha = Now  
    Timer1.Enabled = False ' Desactiva el control Timer.  
End If
```

If...Then...Else

Utilice un bloque **If...Then...Else** para definir varios bloques de instrucciones, uno de los cuales se ejecutará:

If condición1 Then

[bloque de instrucciones 1]

[ElseIf condición2 Then
[bloque de instrucciones 2]] ...

[Else
[bloque de instrucciones n]]

End If

Visual Basic evalúa primero *condición1*. Si es **False**, Visual Basic procede a evaluar *condición2* y así sucesivamente, hasta que encuentre una condición **True**. Cuando encuentra una condición **True**, Visual Basic ejecuta el bloque de instrucciones correspondientes y después ejecuta el código que sigue a **End If**. Opcionalmente, puede incluir un bloque de instrucciones **Else**, que Visual Basic ejecutará si ninguna de las condiciones es **True**.

If...Then...ElseIf es un caso especial de **If...Then...Else**. Observe que puede tener cualquier número de cláusulas **ElseIf** o ninguna. Puede incluir una cláusula **Else** sin tener en cuenta si tiene o no cláusulas **ElseIf**.

Por ejemplo, la aplicación podría realizar distintas acciones dependiendo del control en que se haya hecho clic de una matriz de controles de menú:

```
Private Sub mnuCut_Click (Index As Integer)
    If Index = 0 Then          ' Comando Cortar.
        CopyActiveControl      ' Llama a procedimientos generales.
        ClearActiveControl
    ElseIf Index = 1 Then      ' Comando Copiar.
        CopyActiveControl
    ElseIf Index = 2 Then      ' Comando Borrar.
        ClearActiveControl
    Else                        ' Comando Pegar.
        PasteActiveControl
    End If
End Sub
```

Observe que siempre puede agregar más cláusulas **ElseIf** a la estructura **If...Then**. Sin embargo, esta sintaxis puede resultar tediosa de escribir cuando cada **ElseIf** compara la misma expresión con un valor distinto. Para estas situaciones, puede usar la estructura de decisión **Select Case**.

Para obtener más información Vea "If...Then...Else (Instrucción)" en la *Referencia del lenguaje*.

Select Case

Visual Basic proporciona la estructura **Select Case** como alternativa a **If...Then...Else** para ejecutar selectivamente un bloque de instrucciones entre varios bloques de instrucciones. La instrucción **Select Case** ofrece posibilidades similares a la instrucción **If...Then...Else**, pero hace que el código sea más legible cuando hay varias opciones.

La estructura **Select Case** funciona con una única expresión de prueba que se evalúa una vez solamente, al principio de la estructura. Visual Basic compara el resultado de esta expresión con los valores de cada **Case** de la estructura. Si hay una coincidencia, ejecuta el bloque de instrucciones asociado a ese **Case**:

Select Case *expresiónPrueba*

[Case *listaExpresiones1*

[bloque de instrucciones 1]]

[**Case** *listaExpresiones2*
[*bloque de instrucciones 2*]]

.
. .
.

[**Case Else**
[*bloque de instrucciones n*]]

End Select

Cada *listaExpresiones* es una lista de uno o más valores. Si hay más de un valor en una lista, se separan los valores con comas. Cada *bloque de instrucciones* contiene cero o más instrucciones. Si más de un **Case** coincide con la expresión de prueba, sólo se ejecutará el bloque de instrucciones asociado con la primera coincidencia. Visual Basic ejecuta las instrucciones de la cláusula (opcional) **Case Else** si ningún valor de la lista de expresiones coincide con la expresión de prueba.

Por ejemplo, suponga que agrega otro comando al menú **Edición** en el ejemplo

If...Then...Else. Podría agregar otra cláusula **ElseIf** o podría escribir la función con **Select Case**:

```
Private Sub mnuCut_Click (Index As Integer)
    Select Case Index
        Case 0
            CopyActiveControl      ' Comando Cortar.
            ClearActiveControl      ' Llama a procedimientos generales.
        Case 1
            CopyActiveControl      ' Comando Copiar.
        Case 2
            ClearActiveControl      ' Comando Borrar.
        Case 3
            PasteActiveControl      ' Comando Pegar.
        Case Else
            frmFind.Show           ' Muestra el cuadro de diálogo Buscar.
    End Select
End Sub
```

Observe que la estructura **Select Case** evalúa una expresión cada vez al principio de la estructura. Por el contrario, la estructura **If...Then...Else** puede evaluar una expresión diferente en cada instrucción **ElseIf**. Sólo puede sustituir una estructura **If...Then...Else** con una estructura **Select Case** si la instrucción **If** y cada instrucción **ElseIf** evalúa la misma expresión.

Estructuras de bucle

Las estructuras de bucle le permiten ejecutar una o más líneas de código repetidamente. Las estructuras de bucle que acepta Visual Basic son:

- Do...Loop
- For...Next
- For Each...Next

Do...Loop

Utilice el bucle **Do** para ejecutar un bloque de instrucciones un número indefinido de veces. Hay algunas variantes en la instrucción **Do...Loop**, pero cada una evalúa una condición numérica para determinar si continúa la ejecución. Como ocurre con **If...Then**, la *condición* debe ser un valor o una expresión que dé como resultado **False** (cero) o **True** (distinto de cero).

En el ejemplo de **Do...Loop** siguiente, las *instrucciones* se ejecutan siempre y cuando *condición* sea **True**:

Do While *condición*
instrucciones

Loop

Cuando Visual Basic ejecuta este bucle **Do**, primero evalúa *condición*. Si *condición* es **False** (cero), se salta todas las instrucciones. Si es **True** (distinto de cero), Visual Basic ejecuta las instrucciones, vuelve a la instrucción **Do While** y prueba la condición de nuevo. Por tanto, el bucle se puede ejecutar cualquier número de veces, siempre y cuando *condición* sea distinta de cero o **True**. Nunca se ejecutan las instrucciones si *condición* es **False** inicialmente. Por ejemplo, este procedimiento cuenta las veces que se repite una cadena de destino dentro de otra cadena repitiendo el bucle tantas veces como se encuentre la cadena de destino:

```
Function ContarCadenas (cadenalarga, destino)
    Dim posición, contador
    posición = 1
    Do While InStr(posición, cadenalarga, destino)
        posición = InStr(posición, cadenalarga, destino) _
            + 1
        contador = contador + 1
    Loop
    ContarCadenas = contador
End Function
```

Si la cadena de destino no está en la otra cadena, **InStr** devuelve 0 y no se ejecuta el bucle. Otra variante de la instrucción **Do...Loop** ejecuta las instrucciones primero y prueba *condición* después de cada ejecución. Esta variación garantiza al menos una ejecución de *instrucciones*:

Do

instrucciones

Loop While condición

Hay otras dos variantes análogas a las dos anteriores, excepto en que repiten el bucle siempre y cuando *condición* sea **False** en vez de **True**.

Hace el bucle cero o más veces	Hace el bucle al menos una vez
Do Until <i>condición</i> <i>instrucciones</i> Loop	Do <i>instrucciones</i> Loop Until <i>condición</i>

For...Next

Los bucles **Do** funcionan bien cuando no se sabe cuántas veces se necesitará ejecutar las instrucciones del bucle. Sin embargo, cuando se sabe que se van a ejecutar las instrucciones un número determinado de veces, es mejor elegir el bucle **For...Next**. A diferencia del bucle **Do**, el bucle **For** utiliza una variable llamada contador que incrementa o reduce su valor en cada repetición del bucle. La sintaxis es la siguiente:

For *contador* = *iniciar* **To** *finalizar* [**Step** *incremento*]

instrucciones

Next [*contador*]

Los argumentos *contador*, *iniciar*, *finalizar* e *incremento* son todos numéricos.

Nota El argumento *incremento* puede ser positivo o negativo. Si *incremento* es positivo, *iniciar* debe ser menor o igual que *finalizar* o no se ejecutarán las instrucciones del bucle. Si *incremento* es negativo, *iniciar* debe ser mayor o igual que *finalizar* para que se ejecute el cuerpo del bucle. Si no se establece **Step**, el valor predeterminado de *incremento* es 1.

Al ejecutar el bucle **For**, Visual Basic:

1. Establece *contador* al mismo valor que *iniciar*.
2. Comprueba si *contador* es mayor que *finalizar*. Si lo es, Visual Basic sale del bucle.

(Si *incremento* es negativo, Visual Basic comprueba si *contador* es menor que *finalizar*.)

3. Ejecuta *instrucciones*.
4. Incrementa *contador* en 1 o en *instrucciones*, si se especificó.
5. Repite los pasos 2 a 4.

Este código imprime los nombres de todas las fuentes de pantalla disponibles:

```
Private Sub Form_Click ()  
    Dim I As Integer  
    For i = 0 To Screen.FontCount
```



```

        Print Screen.Fonts(i)
    Next
End Sub

```

En la aplicación de ejemplo VCR, el procedimiento `HighlightButton` utiliza un bucle **For...Next** para pasar por la colección de controles del formulario VCR y mostrar el control **Shape** apropiado:

```

Sub HighlightButton(MyControl As Variant)
    Dim i As Integer
    For i = 0 To frmVCR.Controls.Count - 1
        If TypeOf frmVCR.Controls(i) Is Shape Then
            If frmVCR.Controls(i).Name = MyControl Then
                frmVCR.Controls(i).Visible = True
            Else
                frmVCR.Controls(i).Visible = False
            End If
        End If
    Next
End Sub

```

For Each...Next

El bucle **For Each...Next** es similar al bucle **For...Next**, pero repite un grupo de instrucciones por cada elemento de una colección de objetos o de una matriz en vez de repetir las instrucciones un número especificado de veces. Esto resulta especialmente útil si no sabe cuántos elementos hay en la colección.

He aquí la sintaxis del bucle **For Each...Next**:

For Each *elemento* **In** *grupo*
instrucciones

Next *elemento*

Por ejemplo, el siguiente procedimiento **Sub** abre `Biblio.mdb` y agrega el nombre de cada tabla a un cuadro de lista.

```

Sub ListTableDefs()
    Dim objDb As Database
    Dim MyTableDef as TableDef
    Set objDb = OpenDatabase("c:\vb\biblio.mdb", _
        True, False)
    For Each MyTableDef In objDb.TableDefs()
        List1.AddItem MyTableDef.Name
    Next MyTableDef
End Sub

```

Tenga en cuenta las restricciones siguientes cuando utilice **For Each...Next**:

- Para las colecciones, *elemento* sólo puede ser una variable **Variant**, una variable **Object** genérica o un objeto mostrado en el Examinador de objetos.
- Para las matrices, *elemento* sólo puede ser una variable **Variant**.
- No puede usar **For Each...Next** con una matriz de tipos definidos por el usuario porque un **Variant** no puede contener un tipo definido por el usuario.

¿Qué es un objeto?

Un objeto es una combinación de código y datos que se puede tratar como una unidad. Un objeto puede ser una parte de una aplicación, como un control o un formulario. También puede ser un objeto una aplicación entera. La tabla siguiente describe ejemplos de tipos de objetos que puede usar en Visual Basic.

Ejemplo	Descripción
Botón de comando	Son objetos los controles de un formulario, como botones de comandos y marcos.
Formulario	Cada formulario de un proyecto de Visual Basic es un objeto distinto.
Base de datos	Las bases de datos son objetos y contienen otros objetos, como campos e índices.
Gráfico	Un gráfico de Microsoft Excel es un objeto.

¿De dónde vienen los objetos?

Cada objeto de Visual Basic se define mediante una *clase*. Para comprender la relación entre un objeto y su clase, piense en el molde de las galletas y las galletas. El molde es la clase. Define las características de cada galleta, como por ejemplo el tamaño y la forma. Se utiliza la clase para crear objetos. Los objetos son las galletas.

Mediante dos ejemplos se puede aclarar la relación que existe entre las clases y los objetos en Visual Basic.

- Los controles del cuadro de herramientas de Visual Basic representan clases. El objeto conocido como control no existe hasta que lo dibuja en un formulario. Cuando crea un control, está creando una copia o *instancia* de la clase del control. Esta instancia de la clase es el objeto al que hará referencia en la aplicación.
- El formulario en el que trabaja en tiempo de diseño es una clase. En tiempo de ejecución, Visual Basic crea una instancia de esa clase de formulario.

La ventana Propiedades muestra la clase y la propiedad **Name** de los objetos de una aplicación de Visual Basic, como se muestra en la figura 5.8.

Figura 5.8 Los nombres de objetos y clases se muestran en la ventana Propiedades

Se crean todos los objetos como copias idénticas de sus clases. Una vez que existen como objetos individuales, es posible modificar sus propiedades. Por ejemplo, si dibuja tres

botones de comando en un formulario, cada objeto botón de comando es una instancia de la clase `CommandButton`. Cada objeto comparte un conjunto de características y capacidades comunes (propiedades, métodos y eventos), definidos por la clase. Sin embargo, cada uno tiene su propio nombre, se puede ser activar y desactivar por separado y se puede colocar en una ubicación distinta del formulario, etc.

Para simplificar, la mayoría del material que no pertenece a este capítulo no hará muchas referencias a clases de objetos. Sólo recuerde que, por ejemplo, el término "control de cuadro de lista", significa "instancia de la clase `ListBox`".

¿Qué puede hacer con objetos?

Un objeto proporciona código que no tiene que escribir. Por ejemplo, puede crear su propios cuadros de diálogo **Abrir archivo** y **Guardar archivo**, pero no tiene por qué. En vez de eso, puede usar el control de diálogo común (un objeto) que Visual Basic proporciona. Podría escribir su propia agenda y código de administración de recursos, pero no tiene por qué. En su lugar, puede usar los objetos **Calendar**, **Resources** y **Tasks** que proporciona Microsoft Project.

Visual Basic puede combinar objetos de otros orígenes

Visual Basic proporciona las herramientas que le permiten combinar objetos de distintos orígenes. Ahora puede construir soluciones personalizadas mediante la combinación de las características más poderosas de Visual Basic y de las aplicaciones que aceptan Automatización (antes conocido como Automatización OLE). *Automatización* es una característica del *Modelo de objetos componentes* (COM), un estándar de la industria utilizado por las aplicaciones para exponer objetos a las herramientas de desarrollo y otras aplicaciones.

Puede construir aplicaciones si agrupa entre sí controles intrínsecos de Visual Basic y puede usar también objetos que proporcionen otras aplicaciones. Piense que puede colocar estos objetos en un formulario de Visual Basic:

- Un objeto Gráfico de Microsoft Excel
- Un objeto Hoja de cálculo de Microsoft Excel
- Un objeto Documento de Microsoft Word

Podría usar estos objetos para crear una aplicación con un cuadro de comprobación como el que se muestra en la figura 5.9. Le ahorrará tiempo ya que no tendrá que escribir el código que reproduzca la funcionalidad proporcionada por los objetos de Microsoft Excel y Word.

Figura 5.9 Uso de objetos de otras aplicaciones

Conceptos básicos del trabajo con objetos

Los objetos de Visual Basic aceptan propiedades, métodos y eventos. En Visual Basic, los datos de un objeto (configuración o atributos) se llaman propiedades, mientras que los diversos procedimientos que pueden operar sobre el objeto se conocen como sus métodos. Un evento es una acción reconocida por un objeto, como hacer clic con el *mouse* o presionar una tecla, y puede escribir código que responda a ese evento.

Puede cambiar las características de un objeto si modifica sus propiedades. Piense en una radio. Una propiedad de la radio es su volumen. En Visual Basic, se diría que una radio tiene la propiedad "Volumen" que se puede ajustar modificando su valor. Suponga que establece el volumen de la radio de 0 a 10. Si pudiera controlar una radio en Visual Basic, podría escribir el código en un procedimiento que modificara el valor de la propiedad "Volumen" de 3 a 5 para hacer que la radio suene más alto:

```
Radio.Volumen = 5
```

Además de propiedades, los objetos tienen métodos. Los métodos son parte de los objetos del mismo modo que las propiedades. Generalmente, los métodos son acciones que desea realizar, mientras que las propiedades son los atributos que puede establecer o recuperar. Por ejemplo, marca un número de teléfono para hacer una llamada. Se podría decir que el teléfono tiene un método "Marcar" y podría usar esta sintaxis para marcar el número de siete cifras 5551111:

```
Teléfono.Marcar 5551111
```

Los objetos también tienen eventos. Los eventos se desencadenan cuando cambia algún aspecto del objeto. Por ejemplo, una radio podría tener el evento "CambiarVolumen" y el teléfono podría tener el evento "Sonar".

Controlar objetos mediante sus propiedades

Las propiedades individuales varían ya que puede establecer u obtener sus valores. Se pueden establecer algunas propiedades en tiempo de diseño. Puede usar la ventana Propiedades para establecer el valor de dichas propiedades sin tener que escribir código.

Algunas propiedades no están disponibles en tiempo de diseño, por lo que necesitará escribir código para establecer esas propiedades en tiempo de ejecución.

Las propiedades que establece y obtiene en tiempo de ejecución se llaman *propiedades de lectura y escritura*. Las propiedades que sólo puede leer en tiempo de ejecución se llaman *propiedades de sólo lectura*.

Establecer los valores de las propiedades

Establezca el valor de una propiedad cuando desee modificar la apariencia o el comportamiento de un objeto. Por ejemplo, modifique la propiedad **Text** de un cuadro de texto para modificar el contenido del cuadro de texto.

Para establecer el valor de una propiedad, utilice la sintaxis siguiente:

objeto.propiedad = expresión

Las instrucciones siguientes demuestran cómo se establecen las propiedades:

```
Text1.Top = 200           ' Establece la propiedad Top a 200 twips.
Text1.Visible = True     ' Muestra el cuadro de texto.
Text1.Text = "hola"      ' Muestra 'hola' en el cuadro de texto.
```

Obtener los valores de las propiedades

Obtenga el valor de una propiedad cuando desee encontrar el estado de un objeto antes de que el código realice acciones adicionales (como asignar el valor a otro objeto). Por ejemplo, puede devolver la propiedad **Text** a un control de cuadro de texto para determinar el contenido del cuadro de texto antes de ejecutar código que pueda modificar el valor. En la mayoría de los casos, para obtener el valor de una propiedad se utiliza la sintaxis siguiente:

variable = objeto.propiedad

También puede obtener el valor de una propiedad como parte de una expresión más compleja, sin tener que asignar el valor de la propiedad a una variable. En el siguiente código de ejemplo se calcula la propiedad **Top** del nuevo miembro de una matriz de controles como la propiedad **Top** del miembro anterior más 400:

```
Private Sub cmdAdd_Click()
    ' [instrucciones]
    optButton(n).Top = optButton(n-1).Top + 400
    ' [instrucciones]
End Sub
```

Sugerencia Si va a usar el valor de una propiedad más de una vez, el código se ejecutará más rápidamente si almacena el valor en una variable.

Realizar acciones con métodos

Los métodos pueden afectar a los valores de las propiedades. Por ejemplo, en el ejemplo de la radio, el método **EstablecerVolumen** cambia la propiedad **Volumen**. De forma similar, en Visual Basic, los cuadros de lista tienen una propiedad **List** que se puede modificar con los métodos **Clear** y **AddItem**.

Usar métodos en el código

Cuando utiliza un método en el código, la forma en que escribe la instrucción depende de los argumentos que necesite el método y de si el método devuelve o no un valor. Cuando un método no necesita argumentos, escriba el código mediante la sintaxis siguiente:

objeto.método

En este ejemplo, el método **Refresh** vuelve a dibujar el cuadro de imagen:

```
Picture1.Refresh ' Obliga a redibujar el control.
```

Algunos métodos, como el método **Refresh**, no necesitan argumentos y no devuelven valores.

Si el método necesita más de un argumento, separe los argumentos mediante comas. Por ejemplo, el método **Circle** utiliza argumentos que especifican la ubicación, el radio y el color de un círculo en un formulario:

```
' Dibuja un círculo azul con un radio de 1200 twips.
Form1.Circle (1600, 1800), 1200, vbBlue
```

Si conserva el valor de devolución de un método, debe poner los argumentos entre paréntesis. Por ejemplo, el método **GetData** devuelve una imagen del Portapapeles:

```
Picture = Clipboard.GetData (vbCFBitmap)
```

Si no hay valor de devolución, los argumentos se ponen sin paréntesis. Por ejemplo, el método **AddItem** no devuelve ningún valor:

```
List1.AddItem "sunombre" ' Agrega el texto 'sunombre' a un cuadro  
' de lista.
```

Para obtener más información Vea en la *Referencia del lenguaje* la sintaxis y los argumentos de todos los métodos que proporciona Visual Basic.

¿Cómo se relacionan los objetos entre sí?

Cuando pone dos botones de comando en un formulario, son objetos distintos con distinto valor para la propiedad **Name** (*Command1* y *Command2*), pero comparten la misma clase, *CommandButton*.

También comparten la característica de que están en el mismo formulario. Ha visto antes en este capítulo que un control de un formulario también está contenido en el formulario. Esto coloca los controles en una jerarquía. Para hacer referencia a un control debe hacer primero referencia al formulario, de la misma manera que debe marcar el código de un país o el de una provincia antes de poder hablar con un determinado número de teléfono.

Los dos botones de comando comparten también la característica de que son controles.

Todos los controles tienen características comunes que los hacen diferentes de los formularios y otros objetos del entorno Visual Basic. Las secciones siguientes explican cómo utiliza Visual Basic las colecciones para agrupar objetos relacionados.

Jerarquías de objetos

La jerarquía de un objeto proporciona la organización que determina cómo se relacionan los objetos entre sí y cómo se puede tener acceso a ellos. En la mayoría de los casos, no es necesario conocer la jerarquía de objetos de Visual Basic. Sin embargo:

- Cuando manipule objetos de otra aplicación, debe estar familiarizado con la jerarquía de objetos de esa aplicación. Para obtener más información acerca de cómo moverse por las jerarquías de objetos, vea "Programar con componentes".
- Cuando trabaje con objetos de acceso a datos, debe estar familiarizado con la jerarquía de los Objetos de acceso a datos.

Hay algunos casos comunes en Visual Basic en los que un objeto contiene otros objetos. Estos casos se describen en las secciones siguientes.

Trabajar con colecciones de objetos

Los objetos de una colección tienen sus propias características y métodos. Se hace referencia a los objetos de una colección de objetos como *miembros* de la colección. Cada miembro de una colección está numerado secuencialmente empezando por 0; éste es el *número de índice* del miembro. Por ejemplo, la colección **Controls** contiene todos los controles de un formulario dado, como se muestra en la figura 5.10. Puede usar colecciones para simplificar el código si necesita llevar a cabo la misma operación sobre todos los objetos de la colección.

Figura 5.10 La colección Controls

Por ejemplo, el código siguiente se desplaza a través de la colección **Controls** y muestra el nombre de cada miembro en un cuadro de lista.

```
Dim MyControl as Control
For Each MyControl In Form1.Controls
    ' Se agrega el nombre de cada control a un cuadro de lista.
    List1.AddItem MyControl.Name
Next MyControl
```

Aplicar propiedades y métodos a los miembros de una colección

Puede usar dos técnicas generales para dirigirse a un miembro de un objeto colección:

- Especifique el nombre del miembro. Las expresiones siguientes son equivalentes:
 - Controls("List1")
 - Controls!List1
- Utilice el número de índice del miembro:
 - Controls(3)

Una vez que sea capaz de dirigirse a todos los miembros de forma colectiva y a cada miembro de forma individual, puede aplicar propiedades y métodos de la forma siguiente:

```
' Establece la propiedad Top del control de cuadro de lista a 200.
Controls!List1.Top = 200
```

—o bien—

```
Dim MyControl as Control
For Each MyControl In Form1.Controls()
    ' Establece la propiedad Top de cada miembro a 200.
    MyControl.Top = 200
Next MyControl
```

Objetos que contienen otros objetos

Algunos objetos de Visual Basic contienen otros objetos. Por ejemplo, un formulario contiene normalmente uno o más controles. La ventaja de tener objetos como contenedores de otros objetos es que puede hacer referencia al contenedor en el código para que se vea más claramente el objeto que desea usar. Por ejemplo, la figura 5.11 ilustra dos formularios distintos que podría tener en una aplicación; uno para introducir transacciones de cuentas a pagar y otro para introducir transacciones de cuentas a cobrar.

Figura 5.11 Dos formularios distintos pueden contener controles con el mismo nombre

Ambos formularios pueden tener un cuadro de lista llamado `lstAcctNo`. Puede especificar exactamente cuál desea usar si hace referencia al formulario que contiene el cuadro de lista:

```
frmReceivable.lstAcctNo.AddItem 1201
```

—o bien—

```
frmPayable.lstAcctNo.AddItem 1201
```

Colecciones comunes en Visual Basic

Hay algunos casos comunes en Visual Basic en que un objeto contiene otros objetos. La tabla siguiente describe brevemente las colecciones más utilizadas en Visual Basic.

Colección	Descripción
Forms	Contiene los formularios cargados.
Controls	Contiene los controles de un formulario.
Printers	Contiene los objetos Printer disponibles.

También puede implementar el contenedor objetos en Visual Basic.

Para obtener más información Para obtener más información acerca de cómo contener objetos, vea "Usar colecciones como alternativa a las matrices" en "Detalles de programación". Para obtener más información acerca de la colección **Printers**, vea en el capítulo 12 "Trabajar con texto y gráficos". Para obtener detalles acerca de las colecciones **Controls** y **Forms**, vea la *Referencia del lenguaje*.

La propiedad Container

Puede usar la propiedad **Container** para modificar el contenedor de un objeto en un formulario. Los siguientes controles pueden contener otros controles:

- Control **Frame**
- Control de cuadro **Picture**
- Control **Toolbar** (sólo en las ediciones Profesional y Empresarial)

Este ejemplo muestra cómo mover un botón de comando de un contenedor a otro de un formulario. Abra un proyecto nuevo y dibuje en el formulario un control de marco, un control de cuadro con imagen y un botón de comando.

El código siguiente del evento **Click** del formulario incrementa una variable contador y utiliza un bucle **Select Case** para hacer rotar el botón de comando de un contenedor a otro.

```
Private Sub Form_Click()  
    Static intX as Integer
```



```

Select Case intX
    Case 0
        Set Command1.Container = Picture1
        Command1.Top= 0
        Command1.Left= 0

    Case 1
        Set Command1.Container = Frame1
        Command1.Top= 0
        Command1.Left= 0

    Case 2
        Set Command1.Container = Form1
        Command1.Top= 0
        Command1.Left= 0

End Select
intX = intX + 1
End Sub

```

Para obtener más información Vea "Propiedad Container" en la *Referencia del lenguaje*.

Comunicación entre objetos

Además de usar y crear objetos en Visual Basic, puede comunicarse con otras aplicaciones y manipular sus objetos desde su aplicación. La capacidad de compartir datos entre aplicaciones es una de las características clave del sistema operativo Microsoft Windows. En Visual Basic dispone de gran flexibilidad si sabe cómo comunicarse con otras aplicaciones.

Para obtener más información Para obtener más detalles acerca de la utilización y comunicación con objetos de otras aplicaciones, vea "Programar con componentes".

Crear objetos

La forma más sencilla de crear un objeto es hacer doble clic en un control del cuadro de herramientas. Sin embargo, para obtener las mayores ventajas de todos los objetos disponibles en Visual Basic y otras aplicaciones, puede usar las características de programación de Visual Basic para crear objetos en tiempo de ejecución.

- Puede crear referencias a un objeto con variables de objeto.
- Puede crear sus propios objetos "desde cero" mediante módulos de clase.
- Puede crear sus propias colecciones con el objeto **Collection**.

Para obtener más información Otros capítulos muestran cómo tener acceso a objetos. Por ejemplo, las funciones **CreateObject** y **GetObject** se describen en "Programar con componentes".

Usar variables de objetos

Además de almacenar valores, una variable puede hacer referencia a un objeto. Se asigna un objeto a una variable por las mismas razones que se asigna cualquier valor a una variable:

- Los nombres de variable son a menudo más cortos y fáciles de recordar que los valores que contienen (o, en este caso, los objetos a que hacen referencia).
- Es posible modificar las variables para hacer referencia a otros objetos mientras se ejecuta el código.
- Referirse a una variable que contiene un objeto es más eficiente que referirse repetidamente al objeto propiamente dicho.

Usar una variable de objeto es similar a usar una variable convencional, pero con un paso adicional: asignar un objeto a la variable:

- Primero, declárela:

Dim *variable* **As** *clase*

- Luego, asígnele un objeto:

Set *variable* = *objeto*

Declarar variables de objeto

Declare una variable de objeto de la misma forma en que declara las demás variables, con **Dim**, **ReDim**, **Static**, **Private** o **Public**. Las únicas diferencias son la palabra clave opcional **New** y el argumento *clase*; ambos se muestran más adelante en este capítulo. La sintaxis es la siguiente:

{Dim | ReDim | Static | Private | Public} *variable* **As** [**New**] *clase*

Por ejemplo, puede declarar una variable de objeto que haga referencia a un formulario de la aplicación llamado frmMain:

```
Dim FormVar As New frmMain ' Declara una variable de objeto  
                             ' del tipo frmMain.
```

También puede declarar una variable de objeto que pueda hacer referencia a cualquier formulario de la aplicación:

```
Dim anyForm As Form ' Variable genérica de formulario.
```

De forma similar, puede declarar una variable de objeto que haga referencia a cualquier cuadro de texto de la aplicación:

```
Dim anyText As TextBox      ' Puede hacer referencia a cualquier
                             ' cuadro de texto (pero sólo cuadro de
                             ' texto).
```

También puede declarar una variable de objeto que pueda hacer referencia a un control de cualquier tipo:

```
Dim anyControl As Control   ' Variable genérica de control.
```

Observe que puede declarar una variable de formulario que haga referencia a un formulario específico de la aplicación, pero no puede declarar una variable de control que haga referencia a un control en particular. Puede declarar una variable de control que haga referencia a un tipo específico de control (como **TextBox** o **ListBox**), pero no a un control particular de ese tipo (como `txtEntry` o `List1`). Sin embargo, puede asignar un determinado control a una variable de ese tipo. Por ejemplo, para un formulario con un cuadro de lista llamado `lstSample`, podría escribir:

```
Dim objDemo As ListBox
Set objDemo = lstSample
```

Asignar variables de objeto

Asigne un objeto a una variable de objeto con la instrucción **Set**:

Set *variable* = *objeto*

Utilice la instrucción **Set** siempre que desee que una variable de objeto haga referencia a un objeto.

A veces deseará usar variables de objeto y especialmente variables de control, simplemente para simplificar el código que tiene que escribir. Por ejemplo, podría escribir código como éste:

```
If frmAccountDisplay!txtAccountBalance.Text < 0 Then
    frmAccountDisplay!txtAccountBalance.BackColor = 0
frmAccountDisplay!txtAccountBalance.ForeColor = 255
End If
```

Puede acortar considerablemente este código si utiliza una variable de control:

```
Dim Bal As TextBox
Set Bal = frmAccountDisplay!txtAccountBalance
If Bal.Text < 0 Then
    Bal.BackColor = 0
    Bal.ForeColor = 255
End If
```

Tipos de objeto genéricos y específicos

Las variables de objetos específicos deben hacer referencia a un tipo específico de objeto o clase. Una variable de un formulario específico sólo puede hacer referencia a un formulario de la aplicación (aunque se pueda referir a muchas instancias de ese formulario). De forma similar, una variable de control específico sólo puede referirse a ese tipo particular de control en la aplicación, como `TextBox` o `ListBox`. Para ver un ejemplo, abra un proyecto nuevo y coloque un cuadro de texto en un formulario. Agregue el siguiente código al formulario:

```
Private Sub Form_Click()
    Dim anyText As TextBox
    Set anyText = Text1
```

```
anyText.Text = "Hola"
End Sub
```

Ejecute la aplicación y haga clic en el formulario. La propiedad **Text** del cuadro de texto cambiará a "Hola".

Las variables genéricas de objetos se pueden referir a uno de los muchos tipos específicos de objetos. Por ejemplo, una variable genérica de formulario puede hacer referencia a cualquier formulario de la aplicación; una variable genérica de control sólo puede hacer referencia a cualquier control de cualquier formulario de la aplicación. Para ver un ejemplo, abra un proyecto nuevo y coloque varios controles de marco, etiqueta y botón de comando en un formulario, en cualquier orden. Agregue el código siguiente al formulario:

```
Private Sub Form_Click()
    Dim anyControl As Control
    Set anyControl = Form1.Controls(3)
    anyControl.Caption = "Hola"
End Sub
```

Ejecute la aplicación y haga clic en el formulario. El título del control que ha colocado en tercer lugar en la secuencia del formulario cambiará a "Hola".

Hay cuatro tipos genéricos de objetos en Visual Basic:

Tipo genérico de objeto	Objeto al que se hace referencia
Form	Cualquier formulario de la aplicación (incluyendo los MDI secundarios y el formulario MDI).
Control	Cualquier control de la aplicación.
MDIForm	El formulario MDI de la aplicación (si hay alguno en la aplicación).
Object	Cualquier objeto.

Las variables genéricas de objetos resultan muy útiles cuando no sabe el tipo específico del objeto al que una variable hará referencia en tiempo de ejecución. Por ejemplo, si desea escribir código que pueda operar sobre cualquier formulario de la aplicación, debe usar una variable genérica de formulario.

Nota Puesto que sólo puede haber un formulario MDI en la aplicación, no hay necesidad de usar el tipo genérico MDIForm. En su lugar, puede usar el tipo específico MDIForm (MDIForm1 o cualquiera que especifique en la propiedad **Name** del formulario MDI) siempre que necesite declarar una variable de formulario que haga referencia al formulario MDI. De hecho, como Visual Basic puede resolver referencias a propiedades y métodos de tipos de formularios específicos antes de ejecutar la aplicación, siempre debe usar el tipo específico MDIForm.

El tipo genérico MDIForm sólo se proporciona para completar; si las versiones futuras de Visual Basic permiten tener varios formularios MDI en una única aplicación, resultará útil.

Formularios como objetos

Los formularios se utilizan más a menudo para crear la interfaz de una aplicación, pero también se trata de objetos a los que otros módulos de la aplicación pueden llamar. Los formularios están estrechamente relacionados con los módulos de clase. La diferencia principal entre los dos es que los formularios pueden ser objetos visibles, mientras que los módulos de clase no tienen interfaz visible.

Agregar métodos y propiedades personalizados

Puede agregar métodos y propiedades personalizados a formularios y tener acceso a ellos desde otros módulos de la aplicación. Para crear un método nuevo para un formulario, agregue un procedimiento declarado mediante **Public**.

```
' Método personalizado en Form1
Public Sub LateJobsCount ()
.
. ' <instrucciones>
.
End Sub
```

Puede llamar al procedimiento LateJobsCount desde otro módulo con esta instrucción:

```
Form1.LateJobsCount
```

Crear una propiedad nueva para un formulario es tan sencillo como declarar una variable pública en el módulo de formulario:

```
Public IDNumber As Integer
```

Puede establecer y devolver el valor de IDNumber en Form1 desde otro módulo mediante estas dos instrucciones:

```
Form1.IDNumber = 3
Text1.Text = Form1.IDNumber
```

También puede usar procedimientos **Property** para agregar propiedades personalizadas a un formulario.

Para obtener más información Se proporcionan detalles sobre los procedimientos **Property** en "Programar con objetos".

Nota Puede llamar a una variable, un método personalizado o establecer una propiedad personalizada en un formulario sin tener que cargar el formulario. Esto le permite ejecutar código en un formulario sin cargarlo en memoria. Además, hacer referencia a un control sin hacer referencia a una de sus propiedades o métodos no carga el formulario.

Usar la palabra clave New

Utilice la palabra clave **New** para crear un objeto nuevo definido por su clase. Se puede usar **New** para crear instancias de formularios, clases definidas en módulos de clase y colecciones.

Usar la palabra clave New en formularios

Cada formulario que se crea en tiempo de diseño es una clase. Se puede usar la palabra clave **New** para crear nuevas instancias de esa clase. Para ver cómo funciona, dibuje un botón de comando y diversos controles en un formulario. Establezca la propiedad **Name** del formulario como Ejemplo en la ventana Propiedades. Agregue el código siguiente al procedimiento de evento Click del botón de comando:

```
Dim x As New Ejemplo
x.Show
```

Ejecute la aplicación y haga clic en el botón de comando varias veces. Mueva el formulario que esté en primer plano; como un formulario es una clase con una interfaz visible, podrá ver las copias adicionales. Cada formulario tiene los mismos controles y en las mismas posiciones que en el formulario en tiempo de diseño.

Nota Para hacer que una variable de formulario y una instancia del formulario cargado persistan, utilice una variable **Static** o **Public** en vez de una variable local.

También puede usar **New** con la instrucción **Set**. Pruebe el código siguiente en el procedimiento de evento Click de un botón de comando:

```
Dim f As Form1
Set f = New Form1
f.Caption = "hola"
f.Show
```

Usar **New** con la instrucción **Set** es más rápido y es el método que se recomienda.

Usar la palabra clave New con otros objetos

Se puede usar la palabra clave **New** para crear colecciones y objetos desde clases definidas en módulos de clase. Para ver cómo funciona, pruebe el ejemplo siguiente.

Este ejemplo demuestra cómo la palabra clave **New** crea instancias de una clase. Abra un proyecto nuevo y dibuje un botón de comando en Form1. En el menú **Proyecto**, elija **Agregar módulo de clase** para agregar un módulo de clase al proyecto. Establezca la propiedad **Name** del módulo como ShowMe.

El siguiente código en Form1 crea una instancia nueva de la clase ShowMe y llama al procedimiento contenido en el módulo de clase.

```
Public clsNew As ShowMe
Private Sub Command1_Click()
    Set clsNew = New ShowMe
    clsNew.ShowFrm
End Sub
```

El procedimiento ShowFrm del módulo de clase crea una instancia nueva de la clase Form1, muestra el formulario y después lo minimiza.

```
Sub ShowFrm()
    Dim frmNew As Form1
    Set frmNew = New Form1
    frmNew.Show
    frmNew.WindowState = 1
End Sub
```

Para usar el ejemplo, ejecute la aplicación y haga clic en el botón de comando varias veces. Verá que se muestra un icono de formulario minimizado en el escritorio por cada nueva instancia de la clase ShowMe que se cree.

Para obtener más información Para obtener más información acerca de la utilización de **New** para crear objetos, vea "Programar con componentes".

Restricciones de la palabra clave New

La tabla siguiente describe lo que no se puede hacer con la palabra clave **New**.

No puede usar New para crear	Ejemplo de código <i>no</i> permitido
Variables de los tipos de datos fundamentales.	<code>Dim X As New Integer</code>
Variables de cualquier tipo genérico de objeto.	<code>Dim X As New Control</code>
Variables de cualquier tipo de control específico.	<code>Dim X As New ListBox</code>
Variables de cualquier control específico.	<code>Dim X As New lstNames</code>

Liberar referencias a objetos

Cada objeto utiliza memoria y recursos del sistema. Es conveniente liberar los recursos cuando no necesite usar más el objeto.

- Utilice **Unload** para descargar un formulario o un control de la memoria.
- Utilice **Nothing** para liberar los recursos utilizados por una variable de objeto. Asigne **Nothing** a una variable de objeto con la instrucción **Set**.

Para obtener más información Vea "Evento Unload" y "Nothing" en la *Referencia del lenguaje*.

Pasar objetos a procedimientos

Puede pasar objetos a procedimientos en Visual Basic. En el siguiente código de ejemplo, se supone que hay un botón de comando en un formulario:

```
Private Sub Command1_Click()  
    ' Llama al procedimiento Sub Demo y le pasa el formulario.  
    Demo Form1  
End Sub
```

```
Private Sub Demo(x As Form1)  
    ' Centra el formulario en la pantalla.  
    x.Left = (Screen.Width - x.Width) / 2  
End Sub
```

También es posible pasar un objeto a un argumento por referencia y, desde el procedimiento, establecer el argumento al nuevo objeto. Para ver cómo funciona, abra un proyecto e inserte un segundo formulario. Coloque un control de cuadro de imagen en cada formulario. La tabla siguiente muestra los valores de las propiedades que debe cambiar:

Objeto	Propiedad	Valor
Cuadro Picture en Form2	Name Picture	Picture2 c:\vb\icons\arrows\arw01dn.ico

El procedimiento de evento Form1_Click llama al procedimiento GetPicture en Form2 y le pasa el cuadro de imagen vacío.

```
Private Sub Form_Click()
Form2.GetPicture Picture1
End Sub
```

El procedimiento GetPicture en Form2 asigna la propiedad **Picture** al cuadro de imagen de Form2 como cuadro de imagen vacío de Form1.

```
Private objX As PictureBox
Public Sub GetPicture(x As PictureBox)
' Asigna el cuadro de imagen pasado a una variable
' de objeto.
Set objX = x
' Asigna el valor de la propiedad Picture al cuadro de
' imagen de Form1.
objX.Picture = picture2.Picture
End Sub
```

Para usar este ejemplo, ejecute la aplicación y haga clic en Form1. Verá el icono de Form2 en el cuadro de imagen de Form1.

Para obtener más información Se pretende que los temas anteriores sirvan como introducción a los objetos. Para obtener más información al respecto, vea "Programar con objetos" y "Programar con componentes".