Visual Basic
Manual del Programador

Qué se puede hacer con Visual Basic

Programar con Objetos

Lo que hay que saber acerca de los objetos en Visual Basic

Visual Basic facilita el uso de los objetos, pero es más importante la facilidad que ofrece para efectuar una transición gradual entre la programación por procedimientos y la programación con objetos.

Naturalmente, haber utilizado objetos en Visual Basic contribuye a un mejor entendimiento de los conceptos siguientes. Los temas siguientes ampliarán su conocimiento de los objetos de Visual Basic y le presentarán algunas nuevas características de gran interés.

- Terminología al minuto La terminología de objetos de Visual Basic resumida. Una breve guía para tener a mano al explorar los objetos.

- Descubrir la clase a la que pertenece un objeto Puede usar la palabra clave **TypeOf** o la función **TypeName** para averiguar la clase de un objeto.

- Llamar a una propiedad o método usando un nombre de cadena La función CallByName le permite llamar a las propiedades y métodos de un objeto en tiempo de ejecución.

- Efectuar varias acciones sobre un objeto Puede ahorrar escritura si utiliza la instrucción **With** para establecer múltiples propiedades de un objeto.

- Usar propiedades predeterminadas Las propiedades predeterminadas permiten tener acceso a la propiedad más importante de un objeto sin escribir el nombre de la propiedad, pero debe tener cuidado cuando utilice variables **Variant**.

- Crear matrices de objetos Puede almacenar referencias de objetos en matrices, igual que puede almacenar otros tipos de datos.

- Crear colecciones de objetos Las colecciones suelen proporcionar un modo más cómodo de mantener un número cambiante de referencias de objetos.

- Objeto Collection de Visual Basic Explica las características del objeto genérico **Collection** y sus propiedades y métodos.

- Colecciones en Visual Basic Visual Basic proporciona muchas colecciones incorporadas, además del objeto **Collection**. Todas ellas tienen su propia idiosincrasia.

Terminología al minuto

Lo siguiente es un repaso de los términos que encontrará en las descripciones de los objetos de Visual Basic y sus capacidades. Si nunca ha utilizado Visual Basic pero sí otros lenguajes de programación o si ha trabajado con la terminología ActiveX (anteriormente OLE), este tema le ayudará a realizar la transición.

Si desconoce los objetos, puede que lo encuentre un poco desconcertante. No importa; con esta guía de los términos que va a encontrar, empezará a formarse una imagen de cómo encajan entre sí. A medida que aprenda más sobre los objetos en el resto de este capítulo, puede volver a consultar este tema para integrar cada parte de la información dentro del todo.

Empezamos

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

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

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

Para usar un objeto tiene que mantener una *referencia* a él en una *variable de objeto.* El tipo de *enlace* determina la velocidad con la que se tiene acceso a los métodos de un objeto mediante la variable de objeto. Una variable de objeto puede ser un *enlace en tiempo de compilación* (más lento) o un *enlace en tiempo de diseño*. Las variables enlazadas en tiempo de diseño pueden ser *enlaces DispID* o *enlaces vtable* (más rápidos).

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

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

Las clases de Visual Basic también pueden ser *de reconocimiento de datos*. Una clase puede actuar como *receptor* de datos, enlazando directamente a un origen externo de datos, o actuar como *origen* de datos para otros objetos proporcionando datos de un origen externo.

En la antesala

Bien, si le parece que todo esto ya está muy visto, el resto de este capítulo será un paseo. Si no, no se preocupe; dispone de explicaciones de todos estos términos situadas estratégicamente a lo largo del texto y presentadas con mucha más calma.

Descubrir la clase a la que pertenece un objeto

Las variables de objeto genéricas (es decir, las variables que declara **As Object**) pueden almacenar objetos de muchas clases diferentes. De la misma forma, las variables declaradas con los tipos Form y Control integrados de Visual Basic pueden contener formularios y controles de distintas clases.

Cuando utilice variables de estos tipos, quizá necesite realizar diversas acciones, dependiendo de la clase de un objeto; por ejemplo, algunos objetos pueden no admitir una determinada propiedad o método. Visual Basic ofrece dos formas de hacerlo: la palabra clave **TypeOf** y la función **TypeName**.

La palabra clave **TypeOf** sólo puede usarse en instrucciones **If ... Then ... Else**. Debe incluir el nombre de la clase directamente en el código. Por ejemplo, `If TypeOf MiControl Is CheckBox Then`.

La función **TypeName** es más flexible. Puede usarla en cualquier lugar del código; además, como devuelve el nombre de clase como una cadena, puede compararla con el valor de una variable de cadena.

Llamar a un método o propiedad usando un nombre de cadena

La mayor parte del tiempo puede descubrir las propiedades y métodos de un objeto en tiempo de diseño, y escribir el código para controlarlas. En algunas ocasiones, sin embargo, es posible que no conozca las propiedades y métodos de un objeto por adelantado, o puede que simplemente quiera tener la flexibilidad necesaria para permitir que un usuario final especifique propiedades o ejecute métodos en tiempo de ejecución.

Considere, por ejemplo, una aplicación cliente que evalúe expresiones introducidas por el usuario pasando un operador a una aplicación de servidor. Ahora, suponga que está agregando constantemente nuevas funciones al servidor que requiere nuevos operadores. Lamentablemente, necesitaría volver a compilar y distribuir la aplicación cliente antes de que pudiese usar los nuevos operadores. Para evitar esto, puede usar la función **CallByName** para pasar los nuevos operadores como cadenas, sin cambiar la aplicación.

La función **CallByName** le permite usar una cadena para especificar una propiedad o método en tiempo de ejecución. La signatura para la función **CallByName** tiene una apariencia así:

Result = CallByName(*Object*, *ProcedureName*, *CallType*, *Arguments()*)

El primer argumento para **CallByName** toma el nombre del objeto sobre el que quiere actuar. El segundo argumento, *ProcedureName,* toma una cadena que contiene el nombre del procedimiento del método o propiedad que se va a invocar. El argumento *CallType* toma una constante que representa el tipo de procedimiento que se va a invocar: un método (vbMethod), un property let (vbLet), un property get (vbGet) o un property set (vbSet). El argumento final es opcional, toma una matriz de tipo **Variant** que contiene cualquier argumento para el procedimiento.

Suponga que tiene una aplicación de servidor, MathServer, con una nueva función **SquareRoot**. Su aplicación tiene dos controles TextBox: Text1 contiene la expresión sobre la que se va a hacer la evaluación; Text2 se usa para introducir el nombre de la función. Podría usar el siguiente código en el evento Click de un botón de comando para invocar la función **SquareRoot** sobre la expresión de Text1:

```
Private Sub Command1_Click()
   Text1.Text = CallByName(MathServer, Text2.Text, vbMethod,
Text1.Text)
End Sub
```

Si el usuario introduce "64 / 4" en Text1 y "SquareRoot" en Text 2, el código anterior invocaría a la función **SquareRoot** (la cual toma una cadena que contiene la expresión que se va a evaluar como argumento requerido) y devuelve "4" en Text1 (la raíz cuadrada de 16 ó 64 dividido por 4). Por supuesto, si el usuario introdujera una cadena no válida en Text2 o si la cadena contuviera el nombre de una propiedad en vez de un método, o si el método tuviera un argumento adicional requerido, se produciría un error en tiempo de ejecución. Como puede suponer, necesitará agregar código de tratamiento de errores eficaz cuando use **CallByName** para anticiparse a éstos u otros errores.

Aunque la función **CallByName** puede ser útil en algunas situaciones, deberá sopesar su utilidad frente a las consecuencias que implica en el rendimiento, ya que usar **CallByName** para invocar un procedimiento resulta algo más lento que las llamadas en tiempo de ejecución. Si está invocando a una función que va a ser llamada repetidamente, como dentro de un bucle, **CallByName** podría afectar seriamente al rendimiento.

Efectuar varias acciones sobre un objeto

A menudo necesita realizar varias acciones diferentes sobre el mismo objeto. Por ejemplo, puede que necesite establecer varias propiedades de un mismo objeto. Una manera de hacerlo es usar varias instrucciones.

```
Private Sub Form_Load()
   Command1.Caption = "Aceptar"
   Command1.Visible = True
   Command1.Top = 200
   Command1.Left = 5000
   Command1.Enabled = True
End Sub
```

Observe que todas estas instrucciones utilizan la misma variable de objeto, Command1. Puede hacer que este código sea más fácil de escribir, más legible y más eficaz si utiliza la instrucción **With...End With**.

```
Private Sub Form_Load()
   With Command1
      .Caption = "Aceptar"
      .Visible = True
      .Top = 200
      .Left = 5000
      .Enabled = True
   End With
End Sub
```

También puede anidar instrucciones **With** si coloca una instrucción **With...End With** dentro de otra.

Usar propiedades predeterminadas

Muchos objetos tienen *propiedades predeterminadas*. Puede usar las propiedades predeterminadas para simplificar el código, ya que no tendrá que hacer referencia explícita a la propiedad al establecer su valor. Para un objeto en el que **Value** es la propiedad predeterminada, estas dos instrucciones son equivalentes:

```
object = 20
```

y

```
object.Value = 20
```

Para verlo en práctica, coloque un botón de comando y un cuadro de texto en un formulario. Agregue la siguiente instrucción al evento Click del botón de comando:

```
Text1 = "hola"
```

Ejecute la aplicación y haga clic en el botón de comando. Como **Text** es la propiedad predeterminada del cuadro de texto, éste presentará el texto "hola".

Usar las propiedades predeterminadas con variables de objeto

Cuando en una variable de objeto se almacena una referencia a un objeto, puede seguir utilizando la propiedad predeterminada. Esto se muestra en el siguiente fragmento de código.

```
Private Sub Command1_Click ()
   Dim obj As Object
   ' Poner una referencia a Text1 en la variable de
   ' objeto.
   Set obj = Text1
   ' Establecer el valor de la propiedad
   ' predeterminada(Text).
   obj = "hola"
End Sub
```

En el código anterior, escribir `obj = "hola"` equivale a escribir `obj.Text = "hola"`.

Usar propiedades predeterminadas con variables Variant

El acceso a las propiedades predeterminadas es diferente cuando la referencia de objeto está almacenada en una variable de tipo **Variant** y no en una variable de objeto. Esto se debe a que una variable **Variant** puede contener datos de muchos tipos diferentes.

Por ejemplo, puede leer la propiedad predeterminada de Text1 si utiliza una referencia de una **Variant**, pero si intenta asignar la cadena "adiós" a la propiedad predeterminada, no lo conseguirá. En vez de esto, sustituye la referencia de objeto por la cadena y cambia el tipo **Variant**.

Para ver cómo funciona, escriba el código siguiente en el evento Click del botón de comando del ejemplo anterior:

```
Private Sub Command1_Click()
   Dim vnt As Variant
   ' Establecer la propiedad predeterminada (Text) a
   ' "hola".
   Text1 = "hola"
   ' Poner una referencia a Text1 en la Variant.
   Set vnt = Text1
   ' Presentar la propiedad predeterminada de Text1 y
   ' mostrar que la Variant contiene una referencia
   ' de objeto.
   MsgBox vnt, , "¿Es un objeto? " & IsObject(vnt)
   ' Intento de establecer la propiedad predeterminada
   ' de Text1.
   vnt = "adiós"
   MsgBox vnt, , "¿Es un objeto? " & IsObject(vnt)
End Sub
```

Cuando ejecute la aplicación y haga clic en el botón de comando, primero obtendrá un cuadro de mensajes que presenta el valor actual de la propiedad predeterminada de Text1, "hola", que puede comprobar si ve Text1. El título del cuadro de mensaje confirma que la variable **Variant** contiene una referencia de objeto, es decir, una referencia a Text1.

Cuando haga clic en el botón **Aceptar** del cuadro de mensajes, se asigna el valor "adiós" a la **Variant** y se destruye la referencia a Text1. Después se presenta otro cuadro de mensajes, que muestra el contenido de la **Variant** que, como puede ver, no coincide con el valor actual de Text1.Text.

El título del cuadro de mensajes confirma que la **Variant** ya no contiene una referencia de objeto: ahora contiene la cadena "adiós".

Para obtener más información Para obtener detalles sobre los datos de tipo **Variant** y otros tipos de datos, vea "Introducción a las variables, constantes y tipos de datos" en "Fundamentos de programación".

Otros aspectos del uso de objetos con variables **Variant** se describen en "Objeto Collection de Visual Basic".

Crear matrices de objetos

Puede declarar y usar matrices de un tipo de objeto de la misma manera en que declara y utiliza matrices de cualquier tipo de datos. Dichas matrices pueden ser de tamaño fijo o dinámicas.

Matrices de variables de formulario

Puede declarar una matriz de formularios con **Private**, **Dim**, **ReDim**, **Static** o **Public** del mismo modo en que declara una matriz de cualquier otro tipo. Si declara la matriz con la palabra clave **New**, Visual Basic crea automáticamente una nueva instancia del formulario por cada elemento de la matriz, a medida que utilice los elementos de la matriz.

```
Private Sub Command1_Click ()
   Dim intX As Integer
   Dim frmNew(1 To 5) As New Form1
   For intX = 1 To 5
      frmNew(intX).Show
      frmNew(intX).WindowState = vbMinimized
      ' Para crear formularios minimizados sin hacer
      ' que aparezcan primero con su tamaño normal,
      ' invierta el orden de las dos líneas
      ' anteriores.
   Next
End Sub
```

Si hace clic en el botón de comando para ejecutar el código anterior, creará cinco instancias minimizadas de Form1.

Nota Si observa la barra de tareas, verá Form1 *seis* veces. La instancia adicional de Form1 no está minimizada porque es con la que comenzó.

Matrices de variables de control

Puede declarar una matriz de controles con **Private**, **Dim**, **ReDim**, **Static** o **Public** del mismo modo en que declara una matriz de cualquier otro tipo. Sin embargo, al contrario que las matrices de formularios, no puede declarar las matrices de controles con la palabra clave **New**. Por ejemplo, puede declarar una matriz de un tipo específico de control:

```
ReDim ActiveImages(10) As Image
```

Cuando declara una matriz de un tipo de control concreto, sólo puede asignar a la matriz controles de dicho tipo. Por ejemplo, en el caso de la declaración anterior, sólo puede asignar a la matriz controles de imagen, aunque dichos controles de imagen pueden provenir de formularios diferentes.

Contraste esto con la colección **Controls** incorporada, que puede contener muchos tipos diferentes de controles, todos ellos en el mismo formulario.

Como alternativa puede declarar una matriz de variables genéricas de control. Por ejemplo, puede que desee efectuar un seguimiento de todos los controles que se han colocado en un

control concreto y no permitir que dichos controles se coloquen más de una vez. Puede hacer esto si mantiene una matriz dinámica de variables de control que contenga referencias a cada control que se ha colocado:

```
Private Sub List1_DragDrop(Source As VB.Control, _
      X As Single, Y As Single)
   Dim intX As Integer
   Static intSize As Integer
   Static ctlDropped() As Control
   For intX = 1 To intSize
      ' Si el control colocado está en la matriz, ya
      ' se ha colocado aquí una vez.
      If ctlDropped(intX) Is Source Then
         Beep
         Exit Sub
      End If
   Next
   ' Amplía la matriz.
   intSize = intSize + 1
   ReDim Preserve ctlDropped(intSize)
   ' Guarda una referencia al control colocado.
   Set ctlDropped(intSize) = Source
   ' Agrega el nombre del control al cuadro de lista.
   List1.AddItem Source.Name
End Sub
```

En este ejemplo se utiliza el operador **Is** para comparar las variables de la matriz de controles con el argumento de control. Puede usar el operador **Is** para comprobar la identidad de las referencias de objeto de Visual Basic: si compara dos referencias diferentes al mismo objeto, el operador **Is** devuelve **True**.

En este ejemplo también se utiliza la instrucción **Set** para asignar la referencia de objeto del argumento Source a un elemento de la matriz.

Para obtener más información Vea "Is (Operador)" en la *Referencia del lenguaje*.

Las matrices se describen en "Matrices" y "Matrices dinámicas", en "Fundamentos de programación".

Para obtener una manera más sencilla de hacer un seguimiento de objetos, vea "Crear colecciones de objetos", más adelante en este mismo capítulo.

Crear colecciones de objetos

Las colecciones proporcionan una forma útil de hacer un seguimiento de objetos. Al contrario que las matrices, no tiene que volver a dimensionar los objetos **Collection** a medida que agrega y quita miembros.

Por ejemplo, puede que desee efectuar el seguimiento de todos los controles que se han colocado en un control concreto y no permitir que dichos controles se coloquen más de una vez. Puede hacer esto si mantiene un objeto **Collection** que contenga referencias a los controles que se han colocado:

```
Private Sub List1_DragDrop(Source As VB.Control, _
X As Single, Y As Single)
   Dim vnt As Variant
   Static colDroppedControls As New Collection
   For Each vnt In colDroppedControls
      ' Si el control colocado está en la colección,
      ' ya se ha colocado aquí una vez.
```

```
      If vnt Is Source Then
         Beep
         Exit Sub
      End If
   Next
   ' Guarda una referencia al control colocado.
   colDroppedControls.Add Source
   ' Agrega el nombre del control al cuadro de lista.
   List1.AddItem Source.Name
End Sub
```

En este ejemplo se utiliza el operador **Is** para comparar las referencias de objeto de la colección `colDroppedControls` con el argumento de evento que contiene la referencia al control colocado. Puede usar el operador **Is** para comprobar la identidad de las referencias de objeto de Visual Basic: si se comparan dos referencias diferentes al mismo objeto, el operador **Is** devuelve **True**.

En este ejemplo también se utiliza el método **Add** del objeto **Collection** para colocar en la colección una referencia al control colocado.

Al contrario que las matrices, las colecciones son objetos en sí mismas. La variable `colDroppedControls` se declara **As New**, por lo que se creará una instancia de la clase **Collection** la primera vez que se haga referencia a dicha variable desde el código. La variable también se declara **Static**, por lo que el objeto **Collection** no se destruirá al terminar el procedimiento de evento.

Para obtener más información Vea "Is (Operador)" en la *Referencia del lenguaje*.

Las propiedades y los métodos del objeto **Collection** se describen en "Objeto Collection de Visual Basic", más adelante en este capítulo.

Para comparar el código anterior con el código requerido para las matrices, vea "Crear matrices de objetos", anteriormente en este capítulo.

Para conocer más detalles acerca de cómo puede crear colecciones más robustas mediante la inclusión del objeto **Collection** dentro de una clase colección definida por usted, vea "Crear clases de colección propias", más adelante en este capítulo.

En "Lo que hay que saber acerca de los objetos en Visual Basic", anteriormente en este capítulo, se describe cómo se crean y destruyen los objetos.

Objeto Collection de Visual Basic

Una colección es una forma de agrupar elementos relacionados. En Visual Basic, las colecciones se utilizan para hacer un seguimiento de muchas cosas, como los formularios cargados en un programa (la colección **Forms**) o todos los controles de un formulario (la colección **Controls**).

Visual Basic proporciona la clase genérica **Collection** para ofrecerle la posibilidad de definir sus propias colecciones. Puede crear tantos objetos **Collection**, es decir, instancias de la clase **Collection**, como necesite. Puede usar objetos **Collection** como base de sus propias clases de colección y modelos de objeto, como se describe en "Crear clases de colección propias" y "Modelos de objetos", más adelante en este mismo capítulo.

Por ejemplo, las colecciones son una manera de hacer un seguimiento de múltiples formularios. En "Aplicaciones con interfaz de documentos múltiples (MDI)", en "Crear una interfaz de usuario", se describen las aplicaciones en las que el usuario puede abrir cualquier número de ventanas de documentos. El siguiente fragmento de código muestra cómo se puede usar el método **Add** de un objeto de colección para mantener una lista de

ventanas secundarias MDI creadas por el usuario. En este código se supone que existe un formulario llamado mdiDocument, cuya propiedad **MDIChild** es **True**.

```
' Colección de nivel de módulo en el MDIForm primario.
Public colDocuments As New Collection

' Código para crear un nuevo formulario secundario MDI.
Private Sub mnuFileNew()
   Dim f As New mdiDocument
   Static intDocumentNumber As Integer
   intDocumentNumber = intDocumentNumber + 1
   ' La línea siguiente crea el formulario.
   f.Caption = "Documento" & intDocumentNumber
   ' Agrega la referencia de objeto a la colección.
   colDocuments.Add f
   f.Show
End Sub
```

La colección `colDocuments` actúa como un subconjunto de la colección **Forms** incorporada y sólo contiene instancias del formulario mdiDocument. El tamaño de la colección se ajusta automáticamente al agregar un nuevo formulario. Puede usar **For Each ... Next** para recorrer la colección. Si desea asignar una clave a cada formulario para poder recuperarlos, puede proporcionar una cadena de texto como segundo parámetro del método **Add**, como se describe más adelante en esta sección.

La palabra clave **New** de la declaración de la variable `colDocuments` provoca la creación de un objeto **Collection** la primera vez que se hace referencia a dicha variable desde el código. Como **Collection** es una clase, no un tipo de datos, tiene que crear una instancia y mantener una referencia a dicha instancia (objeto) en una variable.

Al igual que cualquier otro objeto, el objeto **Collection** se destruye cuando se establece a **Nothing** la última variable que contiene una referencia a él o cuando pierde su alcance. De esta forma, se liberarán todas las referencias de objeto que contenga. Por este motivo, la variable `colDocuments` se declara en el formulario MDI primario, de forma que exista durante toda la ejecución del programa.

Nota Si se utiliza una colección para hacer un seguimiento de los formularios, utilice el método **Remove** de la colección para eliminar la referencia de objeto de la colección después de descargar el formulario. No se puede liberar la memoria utilizada por el formulario mientras exista una referencia al formulario y la referencia mantenida por el objeto **Collection** es una referencia como otra cualquiera.

¿De qué se compone un objeto Collection?

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

Las variables **Variant** siempre ocupan 16 bytes, independientemente de lo que contengan, por lo que el uso de objetos **Collection** no es tan eficaz como el uso de matrices. Sin embargo, nunca tiene que aplicar **ReDim** a un objeto **Collection**, lo que produce un código más limpio y más fácil de mantener. Además, los objetos **Collection** ofrecen un acceso por clave extremadamente rápido, que no ofrecen las matrices.

Nota Más exactamente, una variable **Variant** ocupa siempre 16 bytes *incluso si los datos se encuentran almacenados en cualquier otra parte.* Por ejemplo, si asigna una cadena o una matriz a una **Variant**, ésta contiene un puntero a una copia de la

cadena o de los datos de la matriz. En los sistemas de 32 bits sólo se utilizan como puntero 4 bytes de la **Variant** y realmente no hay ningún dato en la **Variant**.

Si almacena un objeto, la **Variant** contiene la referencia de objeto, como si fuera una variable de objeto. Igual que ocurre con las cadenas y las matrices, solamente se utilizan 4 bytes de la **Variant**.

Los tipos de datos numéricos se almacenan dentro de la **Variant**. Independientemente del tipo de datos, la **Variant** sigue ocupando 16 bytes.

A pesar del tamaño de las variables **Variant**, habrá muchas ocasiones en las que tenga más sentido usar un objeto **Collection** para almacenar todos los tipos de datos enumerados anteriormente. Tenga en cuenta su efecto: los objetos **Collection** le permitirán escribir un código limpio y fácil de mantener, al precio de almacenar los elementos en variables **Variant**.

Propiedades y métodos del objeto Collection

Los objetos **Collection** disponen de propiedades y métodos que se pueden usar para insertar, eliminar y recuperar elementos de una colección.

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

Estas propiedades y métodos sólo proporcionan los servicios más básicos de las colecciones. Por ejemplo, el método **Add** no puede comprobar el tipo de objeto que se agrega a la colección para asegurar que la colección contenga un único tipo de objeto. Puede proporcionar una funcionalidad más robusta y propiedades, métodos y eventos adicionales, si crea su propia clase de colección, como se describe en "Crear clases de colección propias", más adelante en este capítulo.

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

Un *índice* es un dato de tipo **Long** entre uno (1) y el número de elementos de la colección. Puede controlar el valor inicial del índice de un elemento mediante los parámetros con nombre ***before*** y ***after***, pero su valor puede cambiar si agrega o elimina otros elementos.

Nota Una colección cuyo índice empieza por 1 se conoce como *basado en uno*, como se explica en "Colecciones en Visual Basic".

Puede usar el índice para recorrer los elementos de una colección. Por ejemplo, el código siguiente muestra dos maneras de aumentar un 10 por ciento el sueldo de los empleados de una colección de objetos Employee, siempre y cuando la variable `colEmployees` contenga una referencia a un objeto **Collection**.

```
Dim lngCt As Long
For lngCt = 1 To colEmployees.Count
   colEmployees (lngCt).Rate = _
      colEmployees (lngCt).Rate * 1.1
Next

Dim emp As Employee
For Each emp In colEmployees
   emp.Rate = emp.Rate * 1.1
Next
```

Sugerencia Para obtener un mayor rendimiento, utilice **For Each** para recorrer los elementos de un objeto **Collection**. En este caso **For Each** es notablemente más rápido que la iteración mediante índice, aunque esto no es cierto en todas las implementaciones de colecciones: depende de cómo almacena la colección los datos internamente.

Agregar elementos a una colección

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

Sub Add (*elemento* **As Variant [,** *clave* **As Variant] [,** ***before*** **As Variant]**

[, ***after*** **As Variant])**

Por ejemplo, para agregar un objeto WorkOrder a una colección WorkOrders mediante la propiedad ID del objeto WorkOrder como *clave*, puede escribir:

```
colWorkOrders.Add woNew, woNew.ID
```

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

```
colWorkOrders.Add woNew, CStr(woNew.ID)
```

El método **Add** acepta argumentos con nombre. Para agregar un elemento como tercer elemento, puede escribir:

```
colWorkOrders.Add woNew, woNew.ID, after:=2
```

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

Eliminar elementos de una colección

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

objeto.**Remove** *índice*

El argumento *índice* puede ser la posición del elemento que desea eliminar o su clave. Si la clave del tercer elemento de una colección es "W017493", puede usar cualquiera de estas dos instrucciones para eliminarlo:

```
colWorkOrders.Remove 3
```

–o bien–

```
colWorkOrders.Remove "W017493"
```

Recuperar elementos de una colección

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

[Set] *variable* **=** *objeto*.**Item(***índice***)**

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

```
Set woCurrent = colWorkOrders.Item(3)
```

-o bien-

```
Set woCurrent = colWorkOrders.Item("W017493")
```

Si utiliza números enteros como claves, debe emplear la función **CStr** para convertirlos a cadenas antes de pasarlos a los métodos **Item** o **Remove**. Los objetos **Collection** siempre asumen que un número entero es un índice.

Sugerencia No permita que los objetos **Collection** decidan si el valor que están recibiendo es un índice o una clave. Si desea que un valor se interprete como clave y la variable que contiene el valor no es de tipo **String**, utilice **CStr** para convertirla. Si desea que un valor se interprete como índice y la variable que contiene el valor no es de un tipo de datos numéricos, utilice **CLng** para convertirla.

Item es el método predeterminado

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

```
Set woCurrent = colWorkOrders(3)
```

–o bien–

```
Set woCurrent = colWorkOrders("W017493")
```

Importante Los objetos **Collection** mantienen sus números de índice automáticamente al agregar y eliminar elementos. El índice numérico de un elemento dado puede variar a lo largo del tiempo. No guarde un valor numérico de índice si espera recuperar el mismo elemento más tarde. Utilice claves para este propósito.

Usar el método Item para invocar propiedades y métodos

No es necesario recuperar una referencia de objeto desde una colección y colocarla en una variable de objeto para poder usarla. Puede usar la referencia mientras está todavía en la colección.

Por ejemplo, suponga que el objeto WorkOrder del código anterior tiene una propiedad **Priority**. Las dos instrucciones siguientes establecerán la prioridad de una orden de trabajo:

```
colWorkOrders.Item("W017493").Priority = 3
colWorkOrders("W017493").Priority = 3
```

Esto funciona porque Visual Basic evalúa la expresión de izquierda a derecha. Cuando llega al método **Item**, explícito o implícito, Visual Basic obtiene una referencia al elemento indicado (en este caso, el objeto WorkOrder cuya clave es `W017493`) y utiliza dicha referencia para evaluar el resto de la línea.

Sugerencia Si va a invocar más de una propiedad o método de un objeto de una colección, copie antes la referencia de objeto a una variable de objeto correctamente definida. Usar una referencia de objeto almacenada en una colección es más lento que usarla después de almacenarla en una variable de objeto correctamente definida (por ejemplo, `Dim woCurrent As WorkOrder`), ya que los objetos **Collection** almacenan los elementos en variables **Variant**. Las referencias de objeto de las **Variant** siempre se enlazan en tiempo de compilación.

Para obtener más información El objeto **Collection** también es una útil alternativa a las matrices en muchas tareas de programación habituales. Vea "Usar colecciones como alternativa a las matrices" en "Detalles de programación".

Visual Basic proporciona varias colecciones incorporadas. Para compararlas con el objeto **Collection**, vea "Colecciones en Visual Basic".

Colecciones en Visual Basic

¿Qué es una colección? En "Objeto Collection de Visual Basic" se definió una colección como una agrupación de objetos relacionados, lo que da pie a varias interpretaciones; es más un concepto que una definición.

De hecho, como verá cuando empiece a comparar colecciones, hay muchas diferencias entre los tipos de colecciones proporcionadas por Visual Basic. Por ejemplo, el código siguiente produce un error:

```
Dim col As Collection
Set col = Forms      ' Error.
```

¿Qué ocurre? **Forms** es una colección; la variable `col` se declara **As Collection**; ¿por qué no puede asignar a la variable `col` una referencia a **Forms**?

La razón es que la clase **Collection** y la colección **Forms** no son *polimórficas*; es decir, no son intercambiables porque se han desarrollado en código diferente. No tienen los mismos métodos, ni almacenan las referencias de objetos de la misma manera ni utilizan los mismos tipos de valores de índice.

Esto hace que el nombre de la clase **Collection** pueda parecer una elección poco afortunada, porque realmente sólo representa una de las muchas posibles implementaciones de las colecciones. En este tema se exploran algunas de las diferencias de implementación que se pueden encontrar.

Colecciones con base cero y con base uno

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

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

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

Valores de índices y claves

Muchas colecciones de Visual Basic permiten tener acceso a un elemento mediante un índice numérico o una clave de cadena, como el objeto **Collection** de Visual Basic. Sin embargo, el objeto **Collection** de Visual Basic permite agregar elementos sin especificar una clave.

Por el contrario, la colección **Forms** sólo permite un índice numérico. Esto se debe a que no hay un valor de cadena único asociado con un formulario. Por ejemplo, puede tener múltiples formularios con el mismo título o múltiples formularios cargados con la misma propiedad **Name**.

Agregar y quitar elementos

Las colecciones también difieren en la posibilidad de agregar elementos y, si esto es posible, en cómo se agregan dichos elementos. Por ejemplo, no puede agregar una impresora a la colección **Printers** desde código de Visual Basic.

Como el objeto **Collection** es una herramienta de programación de propósito general, es más flexible que otras colecciones. Dispone del método **Add**, que puede usar para insertar elementos en la colección, y del método **Remove**, para quitar elementos.

Por el contrario, la única manera de insertar un formulario en la colección **Forms** es cargar el formulario. Si crea un formulario con el operador **New** o mediante una referencia a una variable declarada **As New**, no se agregará a la colección **Forms** hasta que utilice la instrucción **Load** para cargarlo.

Las colecciones **Forms** y **Controls** no disponen de métodos **Remove**. Los formularios y controles se agregan y quitan de estas colecciones de forma indirecta, con las instrucciones **Load** y **Unload**.

¿Qué es lo que contiene realmente?

Como se ha mencionado anteriormente, un formulario no se agrega a la colección **Forms** hasta que se carga. Por tanto, la definición más precisa de la colección **Forms** es que contiene *todos los formularios cargados actualmente en el programa.*

Incluso ésta no es totalmente precisa. Si el proyecto utiliza Microsoft Forms (incluido para compatibilidad con Microsoft Office), encontrará estos formularios en una colección distinta llamada **UserForms**. Así, la colección **Forms** contiene *todos los formularios de Visual Basic cargados actualmente en el programa.*

El contenido de la clase **Collection** se puede definir con más precisión: *cualquier cosa que se pueda almacenar en una* **Variant**. Por tanto, el objeto **Collection** puede contener un objeto o un entero, pero no un tipo definido por el usuario.

Desgraciadamente, esta definición es muy amplia: una instancia dada de la clase **Collection** podría almacenar cualquier combinación de tipos de datos, matrices y objetos.

Sugerencia Una de las razones más importantes para crear clases de colección propias, como se describe en "Crear clases de colección propias", es que se puede controlar el contenido de las colecciones mediante la aplicación práctica del concepto de *seguridad de tipos*.

Enumerar una colección

Puede usar **For Each … Next** para enumerar los elementos de una colección sin preocuparse de si la colección tiene base cero o base uno. Naturalmente, ésta no es una

característica definitoria de las colecciones, ya que Visual Basic también le permite usar **For Each … Next** para enumerar los elementos de una matriz.

For Each … Next funciona gracias a un pequeño objeto llamado *enumerador*. El enumerador efectúa el seguimiento de su ubicación dentro de la colección y devuelve el siguiente elemento cuando se solicita.

Cuando enumera una matriz, Visual Basic crea un objeto enumerador de matriz al momento. Las colecciones tienen sus propios objetos enumeradores, que también se crean cuando son necesarios.

Los enumeradores no saltan elementos

Los enumeradores de las colecciones de Visual Basic no saltan elementos. Por ejemplo, suponga que enumera una colección que contiene "A", "B" y "C", y que mientras lo hace, quita "B". Las colecciones de Visual Basic no saltan a "C" cuando se produce esta situación.

Los enumeradores pueden no obtener los elementos agregados

Si agrega elementos a una colección mientras la enumera, algunos enumeradores incluirán los elementos agregados, mientras que otros no. Por ejemplo, la colección **Forms** no enumerará los formularios cargados durante la enumeración.

El objeto **Collection** enumerará los elementos que agregue durante la enumeración, si permite agregarlos al final de la colección. De esta forma, el bucle siguiente es infinito (es decir, se ejecuta hasta que presiona Ctrl-Inter):

```
Dim col As New Collection
Dim vnt As Variant
col.Add "Infinito"
col.Add "Infinito"
For Each vnt In col
   MsgBox vnt
   col.Add "Infinito"
Next
```

Por otra parte, los elementos agregados al principio de la colección no se incluirán en la enumeración:

```
Dim col As New Collection
Dim vnt As Variant
col.Add "Se enumerará"
For Each vnt In col
   MsgBox vnt
   ' Agrega el elemento al principio.
   col.Add "No se enumerará", Before:=1
Next
```

¿Por qué se utilizan enumeradores?

Al crear un nuevo enumerador cada vez que se inicia un bucle **For Each … Next**, las colecciones permiten enumeraciones anidadas. Por ejemplo, suponga que tiene en la variable `mcolStrings` una referencia a un objeto **Collection** y que la colección sólo contiene cadenas. El código siguiente imprime todas las combinaciones de dos cadenas diferentes:

```
Dim vnt1 As Variant
Dim vnt2 As Variant
For Each vnt1 In mcolStrings
```

```
   For Each vnt2 In mcolStrings
      If vnt1 <> vnt2 Then
         Debug.Print vnt1 & " " & vnt2
      End If
   Next
Next
```

Para obtener más información Vea "Crear clases de colección propias", más adelante en este capítulo.

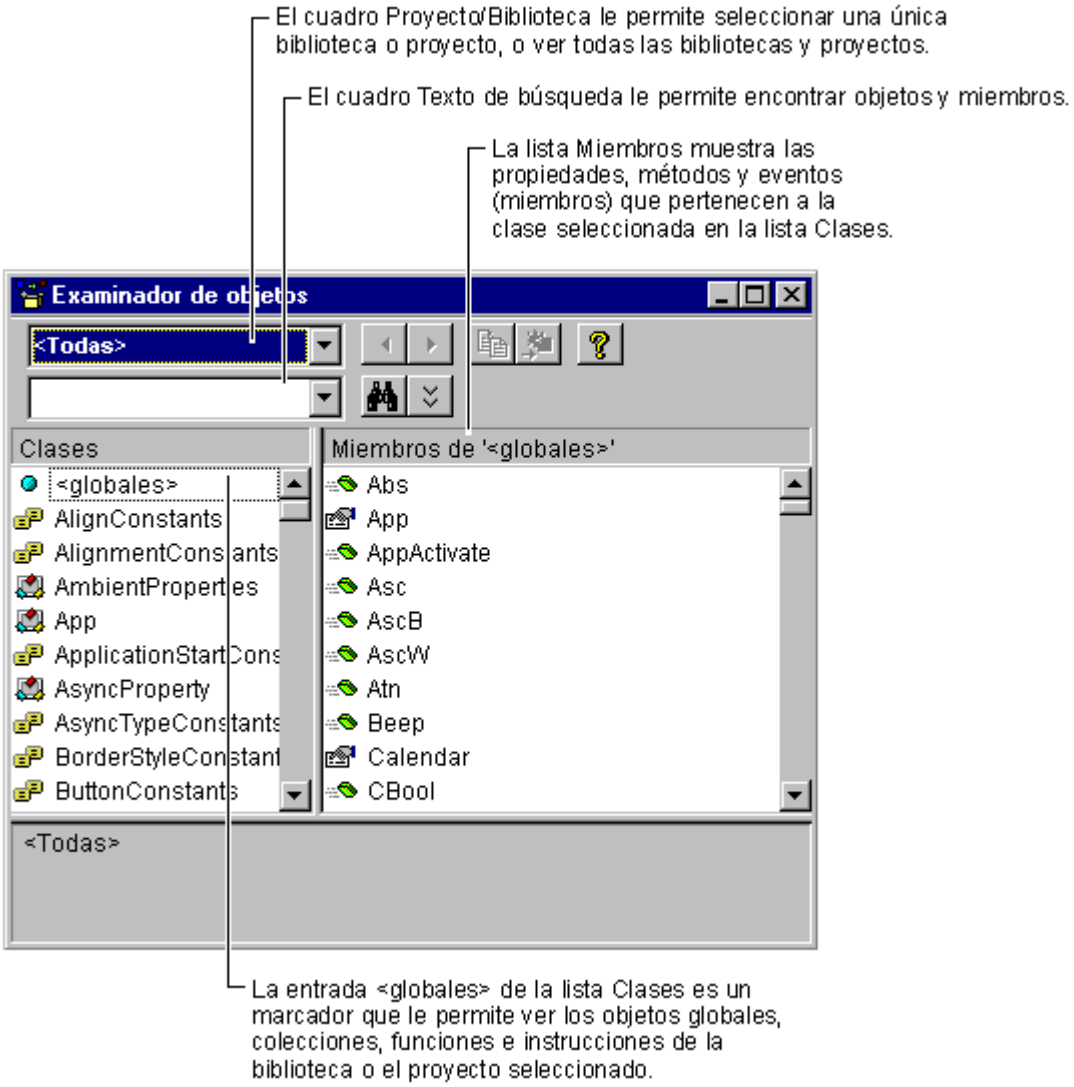
Descripción de objetos

El Examinador de objetos se basa en las *bibliotecas de tipos*, recursos que contienen descripciones detalladas de las clases, incluidos propiedades, métodos, eventos, constantes con nombre, etc.

Visual Basic crea la información acerca de la biblioteca de tipos para las clases que se creen, proporciona bibliotecas de tipos para los objetos que incluye y permite tener acceso a las bibliotecas de tipos proporcionadas por otras aplicaciones.

Puede usar el Examinador de objetos para presentar las clases disponibles en proyectos y bibliotecas, incluidas las clases que se han definido. Los objetos que se creen a partir de dichas clases tendrán los mismos miembros (propiedades, métodos y eventos) que ve en el Examinador de objetos.

Figura 9.1 Examinador de objetos



Para ver el Examinador de objetos

- En el menú **Ver**, elija **Examinador de objetos**.

  –o bien–

Presione F2.

–o bien–

Haga clic en el botón **Examinador de objetos** de la barra de herramientas.

De forma predeterminada, el Examinador de objetos no se puede acoplar a otras ventanas. Esto permite ir del Examinador de objetos a las ventanas de código mediante Ctrl-Tab. Puede modificar este comportamiento si hace clic con el botón secundario del *mouse* (ratón) en el Examinador de objetos, para abrir el menú contextual, y hace clic en **Acoplable**.

Nota Cuando el Examinador de objetos es acoplable, no puede usar Ctrl-Tab para ir desde él a las ventanas de código.

Los temas siguientes detallan las características del Examinador de objetos.

- Contenido del Examinador de objetos El Examinador de objetos muestra las clases y sus miembros. Las cadenas de descripción le recuerdan para qué son los objetos y muestran plantillas de funciones.

- Buscar y examinar objetos Puede buscar los objetos y sus miembros, aunque sólo recuerde una parte de su nombre.

- Agregar descripciones de los objetos Las descripciones que agrega a las clases y los miembros pueden ayudarle a usar las clases.

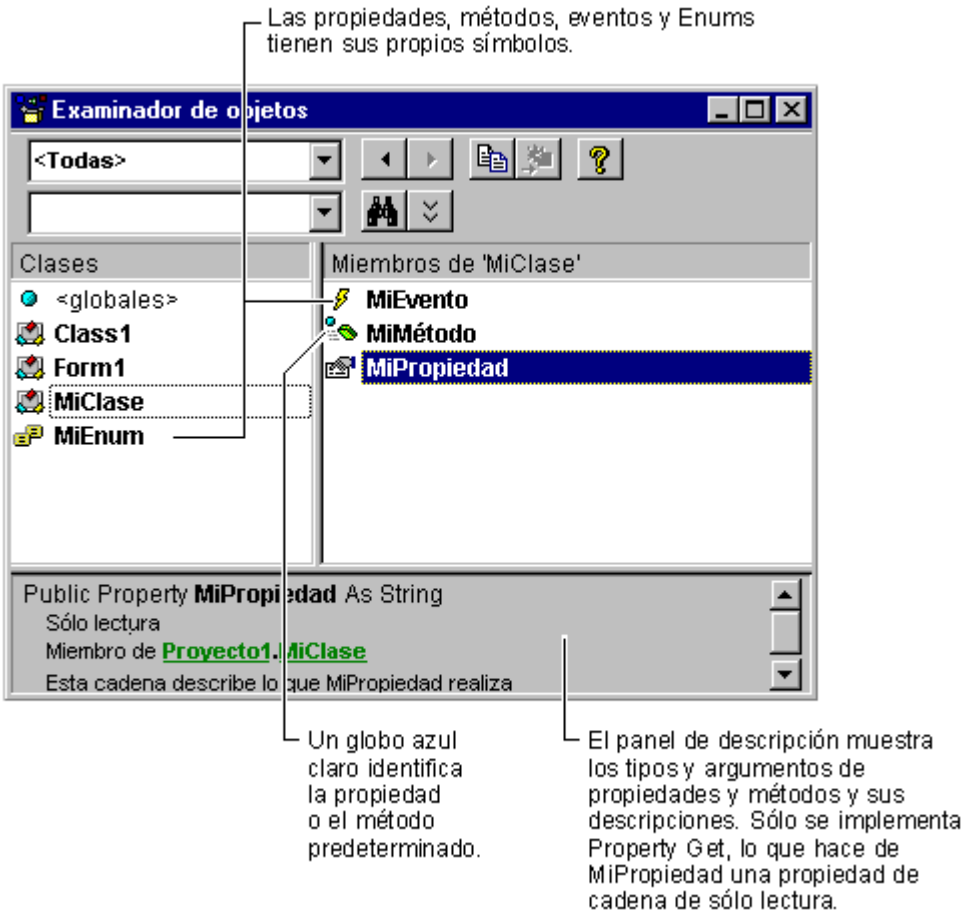
- Moverse entre los procedimientos Puede buscar módulos y procedimientos de los proyectos y saltar a su código.

- Examinar objetos de otras aplicaciones Puede usar objetos de otras aplicaciones en su programa; el Examinador de objetos le muestra los objetos de Automatización proporcionados por otras aplicaciones.

Contenido del Examinador de objetos

El Examinador de objetos presenta la información en una jerarquía de tres niveles, como se muestra en la figura 9.2. Empezando por la parte superior, puede elegir los proyectos y bibliotecas disponibles, incluidos sus propios proyectos de Visual Basic, mediante el cuadro **Proyecto/Biblioteca**.

Figura 9.2 Vista de los miembros de una clase en el Examinador de objetos



- Haga clic en una clase de la lista **Clases** para ver su descripción en el panel de descripción de la parte inferior. Las propiedades, los métodos, los eventos y las constantes de la clase aparecerán en la lista **Miembros** de la derecha. Las clases disponibles se obtienen del proyecto o la biblioteca seleccionados en el cuadro **Proyecto/Biblioteca** o desde todos los proyectos y bibliotecas si selecciona **<Todas>**.

- Puede ver los argumentos y los valores devueltos de un miembro de la clase seleccionada si hace clic en el miembro dentro de la lista **Miembros**. El panel de descripción de la parte inferior del Examinador de objetos muestra dicha información.

- Puede saltar a la biblioteca o al objeto que incluye un miembro si hace clic en el nombre de la biblioteca o del objeto dentro del panel de descripción. Para volver, haga clic en el botón **Volver** de la parte superior del Examinador de objetos.

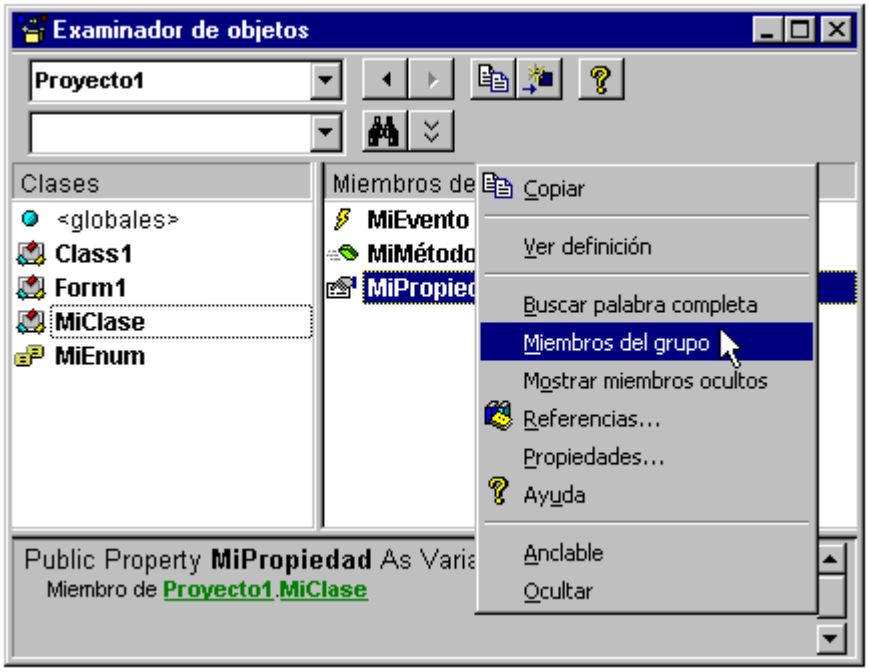
Sugerencia Cuando se encuentra dentro de las listas **Clases** o **Miembros**, si escribe el primer carácter de un nombre, irá al siguiente nombre que empiece por ese carácter.

Controlar el contenido del Examinador de objetos

El menú contextual, mostrado en la figura 9.3, proporciona una alternativa a los botones **Copiar** y **Ver definición** del Examinador de objetos. También permite abrir el cuadro de diálogo **Referencias** y, si está seleccionada una clase o un miembro, se pueden ver las propiedades del elemento seleccionado. Puede establecer descripciones de sus propios

objetos con este elemento del menú, como se describe en "Agregar descripciones de los objetos".

Figura 9.3 Menú contextual del Examinador de objetos



Al hacer clic con el botón secundario del *mouse* en el Examinador de objetos aparece el menú contextual. Además de las funciones ya mencionadas, el menú contextual controla el contenido de las listas **Clases** y **Miembros**.

- Cuando está activada la opción **Miembros del grupo**, todas las propiedades de un objeto aparecen agrupadas, así como todos los métodos, etc. Cuando no está activada **Miembros del grupo**, la lista **Miembros** es alfabética.

- Cuando está activada la opción **Mostrar miembros ocultos**, las listas **Clases** y **Miembros** presentan la información marcada como oculta en la biblioteca de tipos. Normalmente no es necesario ver esta información. Los miembros ocultos se muestran en color gris claro.

Sugerencia Cuando está activada la opción **Miembros del grupo**, al escribir el primer carácter de un nombre saltará al siguiente nombre que empiece por ese carácter, incluso si el nombre pertenece a otro grupo.

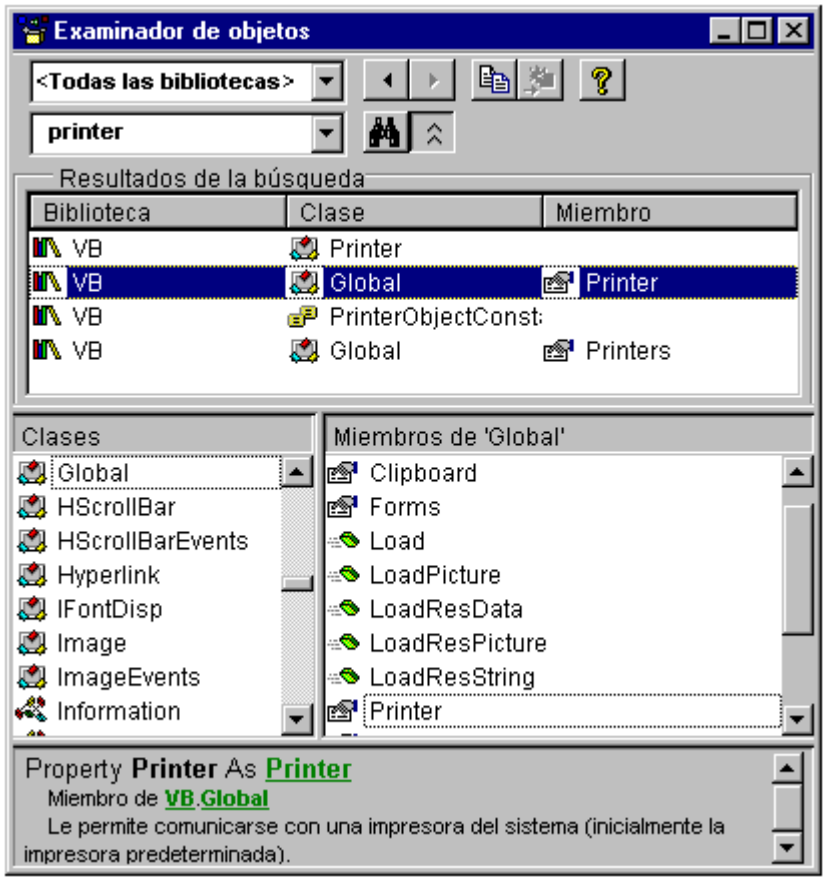
Buscar y examinar objetos

Puede usar el Examinador de objetos para buscar objetos y sus miembros, y para identificar los proyectos o las bibliotecas a los que pertenecen.

Escriba texto en el cuadro **Texto de búsqueda** y después haga clic en el botón **Buscar** (o presione ENTRAR). Las clases y los miembros cuyos nombres incluyan el texto especificado aparecerán en la lista **Resultados de la búsqueda**.

Por ejemplo, la figura 9.4 muestra los resultados de escribir "printer" en el cuadro **Texto de búsqueda** y hacer clic en el botón **Buscar**.

Figura 9.4 Usar el botón Buscar



Puede seleccionar un elemento de la lista **Resultados de la búsqueda** y ver su descripción en el panel de descripción de la parte inferior del Examinador de objetos. Si hace clic en los saltos subrayados del panel de descripción, se seleccionará la biblioteca indicada o se desplazará al objeto o al miembro.

Puede restringir la búsqueda a los elementos que coincidan exactamente con la cadena del cuadro **Búsqueda** si activa **Sólo palabras completas** en el menú contextual.

Agregar descripciones de los objetos

Puede usar el Examinador de objetos para agregar descripciones e identificadores de Ayuda contextual a sus propios procedimientos, módulos, clases, propiedades y métodos. Puede encontrar útiles estas descripciones para trabajar con sus clases.

Nota También puede escribir descripciones de propiedades, métodos y eventos mediante el cuadro de diálogo **Atributos del procedimiento**, al que se tiene acceso desde el menú **Herramientas**.

Para escribir descripciones y vincular clases y miembros con los temas de Ayuda

1. Presione F2 para abrir el **Examinador de objetos**. En el cuadro **Proyecto/Biblioteca**, seleccione el proyecto.

2. En la lista **Clases**, haga clic con el botón secundario del *mouse* en el nombre de una clase para presentar el menú contextual y haga clic en **Propiedades** para abrir el cuadro de diálogo **Opciones de miembro**.

De forma alternativa, en la lista **Miembros** puede hacer clic con el botón secundario del *mouse* en el nombre de una propiedad, método o evento agregado a la clase. En el menú contextual, haga clic en **Propiedades**. Si el miembro es **Private** o **Friend**, se abrirá el cuadro de diálogo **Opciones de miembro**. Si el miembro es **Public**, es decir, si forma parte de la interfaz de la clase, se abrirá el cuadro de diálogo **Atributos del procedimiento**.

Nota La diferencia entre estos dos cuadros de diálogo es que el cuadro de diálogo **Atributos del procedimiento** tiene un botón **Avanzado** que puede usar para hacer que un miembro sea el predeterminado de la clase, como se describe en "Crear una propiedad o un método predeterminados", más adelante en este mismo capítulo.

3. En el cuadro **Id. de contexto de Ayuda**, escriba el identificador de contexto del tema de Ayuda que se va a mostrar al hacer clic en el botón "?" cuando estén seleccionados esta clase o este miembro en el Examinador de objetos.

   Nota Puede crear un archivo de Ayuda para su propio uso y vincular temas con las clases y sus miembros. Para especificar un archivo de Ayuda para su proyecto, utilice la ficha **General** del cuadro de diálogo **Propiedades del proyecto**, al que se tiene acceso desde el menú **Proyecto**.

4. En el cuadro **Descripción**, escriba una breve descripción de la clase o del miembro.

5. Haga clic en **Aceptar** para volver al Examinador de objetos. La cadena de descripción escrita debe aparecer en el panel de descripción de la parte inferior del examinador.

6. Repita los pasos 2 a 5 por cada clase y cada miembro de cada clase.

Nota No puede proporcionar cadenas de búsqueda o temas de Ayuda para las enumeraciones.

Para obtener más información Las enumeraciones se presentan en el tema "Usar enumeraciones para trabajar con conjuntos de constantes" de "Detalles de programación".

Moverse entre los procedimientos

Puede usar el Examinador de objetos para ir rápidamente al código de una clase, un módulo o un procedimiento del proyecto.

Para pasar a una clase, un módulo o un procedimiento

1. (Opcional) Seleccione el proyecto en el cuadro **Proyecto/Biblioteca**

   El paso 1 es opcional si ha seleccionado **<Todas>** en el cuadro **Proyecto/Biblioteca**, ya que están incluidos todos los proyectos.

2. Los nombres de las clases, módulos y miembros pertenecientes a sus proyectos se muestran en negrita. Haga doble clic en cualquier nombre mostrado en negrita para ir a esa clase, módulo o miembro. También puede hacer clic con el botón secundario del *mouse* en un nombre y después seleccione **Ver definición** en la ventana contextual.

   El elemento seleccionado se muestra en la ventana **Código**.

Examinar objetos de otras aplicaciones

Desde Visual Basic puede tener acceso a objetos proporcionados por otras aplicaciones y controlarlos. Por ejemplo, si tiene Microsoft Project y Microsoft Excel en su sistema, podría usar un objeto Graph de Microsoft Excel y un objeto Calendar de Microsoft Project como parte de su aplicación.

Puede usar el Examinador de objetos para explorar las bibliotecas de tipos de otras aplicaciones. Una *biblioteca de tipos* ofrece información acerca de los objetos proporcionados por otras aplicaciones.

Nota En la lista **Proyecto/Biblioteca** hay entradas distintas para Visual Basic (VB) y Visual Basic para aplicaciones (VBA). Aunque hablamos de "objetos proporcionados por Visual Basic", observará que el objeto **Collection** lo proporciona VBA.

Puede agregar bibliotecas al proyecto si selecciona **Referencias** en el menú contextual del Examinador de objetos, para abrir el cuadro de diálogo **Referencias**.

Para obtener más información Para obtener más detalles acerca del uso de Automatización para combinar y manipular objetos de otras aplicaciones, vea "Programar con componentes".

Crear clases propias

Si es un programador experimentado ya dispondrá de una biblioteca de funciones escritas a lo largo de los años. Los objetos no sustituyen a las funciones (seguirá escribiendo y utilizando funciones), sino que proporcionan una forma lógica de organizar procedimientos y datos.

En particular, las clases a partir de las cuales crea objetos combinan datos y procedimientos en una misma unidad. En los temas siguientes se explica cómo aumenta esto la coherencia del código y cómo le puede conducir a nuevas maneras de programar.

- Clases: combinar tipos definidos por el usuario y procedimientos Las clases son tipos definidos por el usuario con una determinada actitud. El encapsulamiento coloca el código y los datos dentro del mismo módulo para producir objetos capaces de proteger y validar los datos propios.

- Personalizar las clases Form Ha estado creando clases durante años: cada vez que ha diseñado un formulario. Esto tiene consecuencias interesantes para el diseño de formularios.

- Módulos de clase paso a paso Breve introducción a los módulos de clase, que incluye la creación de la clase, la creación de objetos a partir de las clases y las reglas de la duración de los objetos.

- Depurar los módulos de clase Describe la opción de interceptación de errores Interrupción en módulos de clases, además de las teclas Alt-F8 y Alt-F5 para ir paso a paso por los errores o saltarlos.

- Ciclo de vida de los formularios de Visual Basic Los ciclos de vida de los formularios y de los controles que contienen siguen unas reglas ligeramente diferentes de las que se aplican a otros objetos.

- Módulos de clase y módulos estándar Hay diferencias importantes entre los módulos de clase y los módulos estándar. Su descripción le ayudará a escribir un código mejor para los objetos.

Clases: combinar tipos definidos por el usuario y procedimientos

Los tipos definidos por el usuario son una poderosa herramienta para agrupar elementos relacionados de datos. Considere por ejemplo el tipo definido por el usuario llamado `udtAccount` definido a continuación:

```
Public Type udtAccount
   Number As Long
   Type As Byte
   CustomerName As String
   Balance As Double
End Type
```

Puede declarar una variable del tipo `udtAccount`, establecer los valores de sus campos individualmente y después pasar todo el registro a procedimientos que lo impriman, lo guarden en una base de datos, realicen cálculos con él, validen sus campos, etc.

Al ser tan poderosos, los tipos definidos por el usuario presentan algunos problemas al programador. Puede crear un procedimiento Withdrawal que produce un error si el cobro

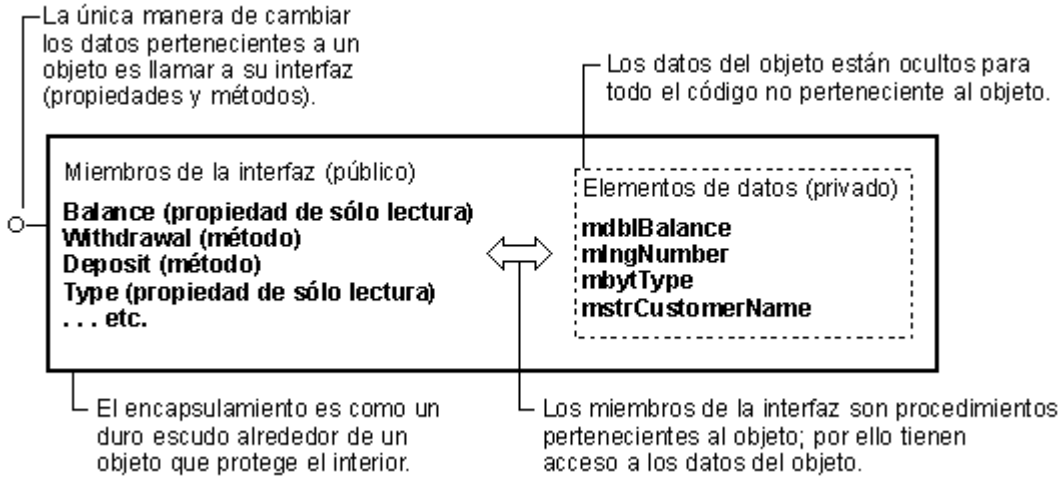
sobrepasa el saldo de la cuenta, pero no puede hacer nada para evitar que el campo Balance se modifique desde otra parte del código del programa.

Es decir, la conexión entre los procedimientos y los tipos definidos por el usuario depende de la disciplina, la memoria y el conocimiento del programador que mantiene el código.

Objetos: tipos definidos por el usuario con una determinada actitud

La programación orientada a objetos resuelve este problema mediante la combinación de datos y procedimientos en una única entidad, como se muestra en la figura 9.5.

Figura 9.5 Los objetos combinan datos y procedimientos



Cuando el tipo definido por el usuario `udtAccount` se convierte en la clase Account, sus datos pasan a ser privados y los procedimientos que tienen acceso a ellos pasan a formar parte de la clase como propiedades y métodos. Esto es lo que significa el término *encapsulamiento*; es decir, un objeto es una unidad (una *cápsula*, si quiere) que contiene código y datos.

Cuando crea un objeto Account a partir de la clase, la única manera de tener acceso a sus datos es mediante las propiedades y los métodos que forman su interfaz. En el siguiente fragmento de código se muestra cómo implementan el encapsulamiento los procedimientos de la clase Account:

```
' El saldo de la cuenta no se ve fuera del código.
Private mdblBalance As Double

' La propiedad de sólo lectura Balance permite que el
' código externo lea el saldo de la cuenta.
Public Property Get Balance() As Double
   Balance = mdblBalance
End Property

' El método Withdrawal cambia el saldo de la cuenta,
' pero sólo si no queda en números rojos.
Public Sub Withdrawal(ByVal Amount As Double)
   If Amount > Balance Then
      Err.Raise Number:=vbObjectError + 2081, _
Description:="Números rojos"
   End If
   mdblBalance = mdblBalance - Amount
End Sub
```

Por el momento, no se preocupe de cómo se incluyen los procedimientos en la clase ni de comprender la sintaxis de los procedimientos de propiedad y de las variables privadas. Lo importante es recordar que puede definir un objeto que encapsula y valida sus propios datos.

Con el objeto Account nunca tiene que preocuparse de si ha utilizado los procedimientos correctos para actualizar la cuenta, porque los únicos procedimientos que puede usar están incorporados al objeto.

Para obtener más información "Personalizar las clases Form" ubica la creación de propiedades y métodos en un marco con el que ya está familiarizado. Más adelante, "Agregar propiedades y métodos a una clase" le explicará la sintaxis.

Para ver más detalles acerca de los tipos definidos por el usuario en vea "Crear tipos de datos propios", en "Detalles de programación".

Para obtener detalles acerca de los procedimientos **Sub** y **Function**, vea "Introducción a los procedimientos" en "Fundamentos de programación".

Personalizar las clases Form

Puede que le sorprenda descubrir que ya ha estado creando clases durante todo el tiempo que ha estado programando en Visual Basic. Es cierto: Form1, ese nombre tan familiar que aparece en cada proyecto que ha empezado, es realmente una clase.

Para comprobarlo, abra un nuevo proyecto Exe estándar. Agregue un botón a Form1 y coloque el siguiente código en el evento Click:

```
Private Sub Command1.Click()
   Dim f As New Form1
   f.Show
End Sub
```

Presione F5 para ejecutar el proyecto y haga clic en el botón. No se sorprenda; hay otra instancia de Form1. Haga clic en el botón: hay otra. Todas las instancias que crea parecen iguales y tienen el mismo comportamiento, puesto que todas son instancias de la clase Form1.

¿Qué está pasando?

Si ha visto "Trabajar con objetos", en "Fundamentos de programación", sabrá que una variable de objeto declarada **As New** contiene **Nothing** hasta la primera vez que hace referencia a ella desde el código. La primera vez que utiliza la variable, Visual Basic observa que contiene el valor especial **Nothing** y crea una instancia de la clase. Y hace bien, porque si no, `f.Show` provocaría un error.

Yo y mi variable global oculta

Puede que se pregunte cómo puede hacer referencia a Form1 en el código, como si fuera una variable de objeto. No hay magia por ninguna parte. Visual Basic crea una variable oculta de objeto global por cada clase de formulario. Es como si Visual Basic hubiera agregado la siguiente declaración al proyecto:

```
Public Form1 As New Form1
```

Cuando selecciona Form1 como objeto inicial o cuando escribe `Form1.Show` en el código está haciendo referencia a esta variable oculta de objeto global. Como está declarada **As New**, se crea una instancia de la clase Form1 la primera vez que utiliza esta variable declarada previamente en el código.

La razón por la que esta declaración está oculta es porque Visual Basic la modifica cada vez que se modifica la propiedad **Name** de un formulario. De esta forma, la variable oculta siempre tiene el mismo nombre que la clase formulario.

Una pregunta

¿Cuáles de las instancias de Form1 creadas en el ejercicio anterior estaban asociadas con la variable global oculta? Si contesta que la primera, tiene razón. Form1 es el objeto inicial predeterminado del proyecto y para Visual Basic eso es como si usara la variable global `Form1` declarada previamente en el código.

Sugerencia Después de descargar un formulario, siempre debe establecer todas las referencias del formulario a **Nothing** para liberar la memoria y los recursos utilizados por el formulario. La referencia que más se suele pasar por alto es la variable oculta de formulario global.

¿Qué ocurre con las demás instancias de Form1?

En "Fundamentos de programación" ha visto que para hacer referencia a un objeto necesita una variable de objeto y que un objeto sólo existe mientras haya al menos una variable de objeto que contenga una referencia al mismo. Entonces, ¿por qué se mantienen todas las demás instancias?

La segunda instancia de Form1, y todas las demás, tenían una variable de objeto mientras se utilizaban los métodos **Show**. Después, esa variable perdió su alcance y se estableció a **Nothing**. Pero Visual Basic mantiene una colección especial llamada **Forms** sobre la que puede profundizar en "Más acerca de los formularios", en "Crear una interfaz de usuario". La colección **Forms** contiene una referencia a cada uno de los formularios cargados en el proyecto, de modo que siempre puede encontrarlos y controlarlos.

Nota Como irá observando, esto no es cierto para todas las clases. Por ejemplo, las clases que diseña no tendrán variables globales ocultas ni colecciones globales para administrarlas: éstas son características especiales de las clases de formulario. Sin embargo, puede declarar sus propias variables globales y sus propias colecciones como se describe en "Crear clases de colección propias".

Propiedades, métodos y eventos de las clases Form

La primera vez que agregó una propiedad a una clase de formulario probablemente lo hizo de forma visual, mediante la colocación de un botón de comando (o algún otro control) en Form1. Al hacerlo, agregó una propiedad Command1 de sólo lectura a la clase de formulario. A continuación invocó esta propiedad de Form1 cuando tenía que invocar un método o una propiedad del botón de comando:

```
Command1.Caption = "Haga clic aquí"
```

Cuando modificaba la propiedad **Name** de cualquier control de un formulario, Visual Basic modificaba ocultamente el nombre de la propiedad de sólo lectura para que siempre coincidieran.

Si todavía tiene abierto el proyecto del ejercicio anterior, puede ver esta propiedad Command1 presionando F2 para abrir el Examinador de objetos. En el cuadro **Proyecto/Biblioteca**, seleccione Project1. Verá Form1 en el panel **Clases**. En el panel **Miembros**, desplácese hasta encontrar Command1 y selecciónelo.

Command1 tiene un símbolo de propiedad al lado y si observa el panel de descripción, verá que es una propiedad **WithEvents**. Como verá en "Agregar eventos a las clases", esto quiere decir que la propiedad (o la variable de objeto) tiene un procedimiento de evento

asociado con ella. Puede que uno de dichos procedimientos de evento, Command1_Click (), contenga el primer código de Visual Basic escrito por usted.

Espere, aún hay más

Colocar controles en un formulario no es la única manera de agregar nuevos miembros a la clase de formulario. Puede agregar sus propias propiedades, métodos y eventos personalizados con la misma facilidad con la que crea nuevas variables y procedimientos.

Para comprobarlo, agregue el código siguiente a la sección Declaraciones de Form1:

```
' Propiedad Comment de la clase Form1.
Public Comment As String
```

Agregue el código siguiente al evento Click de Form1:

```
Private Sub Form_Click()
   MsgBox Comment, , "Mi comentario es:"
End Sub
```

Por último, modifique el código del procedimiento de evento Command1_Click () agregando un línea, de la siguiente manera:

```
Private Sub Command1.Click()
   Dim f As New Form1
   f.Comment = InputBox("¿Cuál es mi comentario?")
   f.Show
End Sub
```

Presione F5 para ejecutar el proyecto. Haga clic en Command1 y, cuando aparezca el cuadro de entrada, escriba cualquier cosa y haga clic en **Aceptar**. Cuando aparezca la nueva instancia de Form1, haga clic en ella para reproducir la propiedad Comment.

Haga clic en la primera instancia de Form1 y observe que la propiedad Comment está vacía. Puesto que Visual Basic creó esta instancia como objeto inicial, nunca tuvo oportunidad de establecer la propiedad Comment.

Los formularios pueden invocar los métodos de otros formularios

Si se ha fijado bien, puede que haya advertido que el código agregado a la clase Form1 no establecía la propiedad Comment del objeto, sino que establecía la propiedad Comment de la *nueva* instancia de Form1 que creaba.

Esta capacidad de los formularios para establecer las propiedades y usar los métodos de otros formularios es una técnica muy útil. Por ejemplo, cuando un formulario MDI abre una nueva ventana secundaria, puede inicializar la nueva ventana mediante el establecimiento de sus propiedades y llamadas a sus métodos.

También puede usar esta técnica para pasar información de un formulario a otro.

Sugerencia Puede crear eventos personalizados en los formularios. "Agregar un evento a un formulario", más adelante en este capítulo, le proporciona el procedimiento que debe seguir.

Otros tipos de módulos

Puede agregar propiedades, métodos y eventos a las clases de formulario mediante la escritura de código en sus módulos de código. De la misma forma, puede agregar propiedades, métodos y eventos a los módulos de clase y, si tiene la Edición Profesional o la Edición Empresarial de Visual Basic, a los módulos de código del control de usuario y del documento de usuario.

Cuando lea "Agregar propiedades y métodos a una clase" y "Agregar eventos a una clase", recuerde que todo lo que lea se aplica a las clases de formulario así como a los módulos de clase.

Para obtener más información ¿Qué es un módulo de clase? En "Módulos de clase paso a paso" se muestra cómo se puede definir una clase y se ilustra el ciclo de vida de los objetos creados a partir de dicha clase.

Módulos de clase paso a paso

En este ejemplo se muestra cómo se pueden usar los módulos de clase para definir clases a partir de las cuales se puede crear después objetos. También explica cómo se pueden crear propiedades y métodos en la nueva clase y muestra cómo se crean y se destruyen los objetos.

Abra un nuevo proyecto Exe estándar e inserte un módulo de clase con **Agregar módulo de clase** del menú **Proyecto**. Coloque cuatro botones de comando en el formulario. En la tabla siguiente se enumeran los valores de propiedades que tiene que establecer para los objetos de este ejemplo.

Objeto	**Propiedad**	**Valor**
Módulo de clase	**Name**	Thing
Command1	**Caption**	Mostrar Thing
Command2	**Caption**	Invertir el nombre de Thing
Command3	**Caption**	Crear nuevo Thing
Command4	**Caption**	Thing temporal

Nota Los módulos de clase se guardan en archivos con la extensión .cls.

Agregue lo siguiente a la sección Declaraciones del módulo de clase:

```
Option Explicit
Public Name As String
Private mdtmCreated As Date
```

La variable `Name` será una propiedad del objeto Thing, puesto que se declara **Public**.

Nota No confunda esta propiedad Name con la propiedad **Name** del módulo de clase, establecida según las indicaciones de la tabla anterior. La propiedad **Name** del módulo de clase asigna el nombre a la clase Thing. ¿Por qué se da a la clase Thing una propiedad **Name**? O mejor aún, ¿por qué no? Puede que desee asignar una propiedad **Name** a la clase Thing porque Things debe tener nombres. Recuerde que no hay restricciones especiales en cuanto a los nombres de las propiedades y los métodos utilizados en Visual Basic. Puede usar los mismos nombres de propiedades y métodos en las clases.

La variable `mdtmCreated` es un miembro privado de datos que se utiliza para almacenar el valor de la propiedad de sólo lectura **Created**. La propiedad **Created** devuelve la fecha y la hora en que se creó el objeto **Thing**. Para implementar la propiedad **Created**, agregue el siguiente procedimiento **Property Get** a la sección Declaraciones del módulo de clase:

```
Property Get Created() As Date
   Created = mdtmCreated
```

```
End Property
```

Nota Si agrega el procedimiento de propiedad mediante el cuadro de diálogo **Agregar procedimiento** del menú **Herramientas**, asegúrese de eliminar la declaración **Property Let** que se agrega automáticamente con este cuadro de diálogo. **Property Let** sólo se requiere en las propiedades de lectura y escritura, como se explica en "Funcionamiento de los procedimientos de propiedad".

El objeto **Thing** tiene un método, **ReverseName**, que simplemente invierte el orden de las letras de la propiedad Name. No devuelve ningún valor, por lo que está implementado como procedimiento **Sub**. Agregue el siguiente procedimiento **Sub** al módulo de clase.

```
Public Sub ReverseName()
   Dim intCt As Integer
   Dim strNew As String
   For intCt = 1 To Len(Name)
      strNew = Mid$(Name, intCt, 1) & strNew
   Next
   Name = strNew
End Sub
```

Los módulos de clase tienen dos eventos, Initialize y Terminate. En la lista desplegable **Objeto** del módulo de clase, seleccione **Class**. La lista desplegable **Procedimiento** mostrará los eventos. Coloque el código siguiente en los procedimientos de evento:

```
Private Sub Class_Initialize()
   ' Establece la fecha y hora de creación del objeto,
   ' que devolverá la propiedad Created.
   mdtmCreated = Now
   ' Muestra las propiedades del objeto.
   MsgBox "Nombre: " & Name & vbCrLf & "Creado: " _
   & Created, , "Inicialización de Thing"
End Sub

Private Sub Class_Terminate()
   ' Muestra las propiedades del objeto.
   MsgBox "Nombre: " & Name & vbCrLf & "Creado: " _
   & Created, , "Terminación de Thing"
End Sub
```

Normalmente, el procedimiento de evento Initialize contiene el código que se ejecuta en el momento de la creación del objeto, por ejemplo la fecha y la hora para la propiedad **Created**. El evento Terminate contiene el código necesario para liberar el objeto cuando éste se destruye.

En este ejemplo, los dos eventos se utilizan principalmente para ofrecer una indicación visual de la creación y destrucción de un objeto **Thing**.

Usar el objeto Thing

Agregue esta declaración a la sección Declaraciones del módulo de formulario:

```
Option Explicit
Private mth As Thing
```

La variable `mth` almacenará una referencia a un objeto **Thing**, que se creará en el evento Load del formulario. Coloque el código siguiente en el procedimiento de evento Form_Load y en los procedimientos de evento Click de los cuatro botones.

```
Private Sub Form_Load()
   Set mth = New Thing
```

```
   mth.Name = InputBox("Escriba un nombre para Thing")
End Sub

' Botón "Mostrar Thing"
Private Sub Command1_Click()
   MsgBox "Nombre: " & mth.Name & vbCrLf _
   & "Creado: " & mth.Created, , "Formulario Thing"
End Sub

' Botón "Invertir el nombre de Thing"
Private Sub Command2_Click()
   mth.ReverseName
   ' Clic en "Mostrar Thing"
   Command1.Value = True
End Sub

' Botón "Crear nuevo Thing"
Private Sub Command3_Click()
   Set mth = New Thing
   mth.Name = InputBox( _
   "Escriba un nombre para el nuevo Thing")
End Sub

' Botón "Thing temporal".
Private Sub Command4_Click()
   Dim thTemp As New Thing
   thTemp.Name = InputBox( _
   "Escriba un nombre para el Thing temporal")
End Sub
```

Ejecutar el proyecto

Presione F5 para ejecutar el proyecto. Si observa el código del procedimiento de evento Form_Load, podrá ver que se utiliza el operador **New** para crear un objeto **Thing**. A la variable `mth` se le asigna una referencia a este objeto **Thing**.

Verá un cuadro de entrada que le pide un nombre para el objeto **Thing**. Cuando escribe un nombre y presiona ENTRAR, el valor devuelto se asigna a la propiedad **Name** del objeto **Thing**.

Mostrar el formulario Thing

Puede comprobar que se ha asignado la propiedad **Name** si hace clic en el primer botón, "Mostrar Thing", que presenta un cuadro de mensajes con todas las propiedades del objeto **Thing**.

Invertir el nombre de Thing

Haga clic en el segundo botón, "Invertir el nombre de **Thing**". Este botón llama al método **ReverseName** para invertir el orden de las letras del nombre del objeto **Thing** y después activa el primer botón para presentar los valores actualizados de las propiedades.

Crear nuevo Thing

Haga clic en el botón "Crear nuevo Thing" para destruir el objeto **Thing** existente y crear uno nuevo. O bien, como escribió realmente, para crear un nuevo **Thing** y después destruir el antiguo.

El operador **New** hace que se cree un nuevo objeto **Thing**, de modo que verá el MsgBox presentado por el evento Initialize del nuevo objeto **Thing**. Cuando hace clic en **Aceptar** se coloca una referencia al nuevo objeto **Thing** en la variable `mth del` nivel de formulario.

Así se elimina la referencia del objeto **Thing** antiguo. Como no hay más referencias a él, se destruye y aparece el cuadro de mensajes de su evento Terminate. Cuando hace clic en **Aceptar**, la instrucción **InputBox** solicita un nombre para el nuevo objeto **Thing**.

Nota Si quiere destruir el objeto **Thing** antiguo antes de crear el nuevo, puede agregar la línea de código `Set mth = Nothing` al principio del procedimiento de evento.

Thing temporal

El cuarto botón demuestra otro aspecto de la duración del objeto. Cuando haga clic en él, se le pedirá un nombre para el objeto **Thing** temporal.

Pero espere, porque todavía no existe un objeto **Thing** temporal. Si no ha visto el cuadro de mensajes del evento Initialize, ¿cómo puede asignarle un nombre?

Como la variable `thTemp` se declaró **As New**, se creará un objeto **Thing** en el momento en que se invoca una de sus propiedades o de sus métodos. Esto ocurrirá cuando se asigne el valor devuelto por **InputBox** a la propiedad **Name**. Escriba un nombre y haga clic en **Aceptar** en el cuadro de entrada.

A continuación verá el cuadro de mensajes Iniciar **Thing**, que muestra que la propiedad **Name** está vacía. Cuando hace clic en **Aceptar** para cerrar el cuadro de mensajes, se asigna el valor de la instrucción **InputBox** a la propiedad **Name**. Esto es mucha actividad para una única línea de código.

Naturalmente, en cuanto haya hecho esto termina el procedimiento de evento Click y la variable `thTemp` pierde su alcance. Se libera la referencia del objeto **Thing** temporal, por lo que verá el cuadro de mensajes Terminar **Thing**. Observe que contiene el nombre que escribió.

Cada vez que haga clic en este botón se creará otro objeto **Thing** temporal, se le asignará un nombre y se destruirá.

Cierre del programa

Cierre el programa con un clic en el botón de cierre del formulario. *No utilice el botón Terminar de la barra de herramientas.* Cuando el programa se cierra, se destruye Form1. La variable `mth` pierde su alcance y Visual Basic libera la referencia al objeto **Thing**. No quedan referencias al objeto **Thing**, por lo que se destruye y se presenta el cuadro de mensajes del evento Terminate.

Vuelva a ejecutar el programa, pero esta vez termínelo mediante el botón **Terminar** de la *barra de herramientas*. Observe que *no* se presenta el cuadro de mensajes Terminar del objeto **Thing**.

Es importante recordar que si termina un programa con el botón **Terminar** o con una instrucción **End** desde el código, detiene el programa *inmediatamente* sin ejecutar los eventos Terminate de los objetos. Siempre es mejor cerrar el programa con todos los formularios descargados.

Puede encontrar útil ejecutar el ejemplo con la tecla F8 para recorrer el código de línea en línea. Ésta es una buena manera de entender el orden de los eventos que tienen lugar en la creación y destrucción de un objeto.

Importante En una aplicación real, los eventos Initialize y Terminate no deben contener cuadros de mensaje ni ningún código que permita procesar los mensajes en Windows. En general, es mejor usar instrucciones **Debug.Print** cuando depure la duración de los objetos.

Para obtener más información Los formularios y los controles son diferentes del resto de los objetos, como se describe en "Ciclo de vida de los formularios de Visual Basic".

Puede encontrar más detalles acerca de las clases y los módulos de clase en "Agregar propiedades y métodos a una clase" y "Agregar eventos a una clase".

Depurar los módulos de clase

La depuración de los módulos de clase difiere ligeramente de la depuración de los programas normales, ya que un error en una propiedad o un método de un módulo de clase siempre actúa como un error controlado. Es decir, siempre hay un procedimiento en la pila de llamadas que puede controlar el error, que es el procedimiento que invocó la propiedad o el método del módulo de clase.

Visual Basic compensa esta diferencia con la opción de interceptación de errores **Interrupción en módulos de clase**, además de las opciones antiguas **Interrupción en errores no controlados** y **Interrupción en todos los errores**.

Nota Puede establecer el estado Interceptación de errores predeterminado en la ficha **General** del cuadro de diálogo **Opciones**, disponible en el menú **Herramientas**. La opción seleccionada afecta a la sesión actual y se convierte en la predeterminada para todas las instancias posteriores de Visual Basic. Para cambiar el valor sólo para la sesión actual, sin afectar a la opción predeterminada, seleccione **Alternar** en el menú contextual de la ventana **Código**, que aparece al hacer clic con el botón secundario del *mouse* en la ventana **Código**.

Por ejemplo, suponga que el módulo de clase Class1 contiene el código siguiente:

```
Public Sub Oops()
   Dim intOops As Integer
   intOops = intOops / 0
End Sub
```

Ahora suponga que un procedimiento de otro módulo de clase, formulario o módulo estándar invoca al miembro Oops:

```
Private Sub Command1_Click()
   Dim c1 As New Class1
   c1.Oops
End Sub
```

Si la opción de interceptación de errores está establecida a **Interrupción en errores no controlados**, la ejecución no se detendrá en la división por cero. En su lugar, el error se producirá en el procedimiento que llama, Command1_Click. La ejecución se detendrá en la llamada al método Oops.

Podría usar **Interrupción en todos los errores** para detenerse en la división por cero, pero esta opción no es la más conveniente en la mayoría de las situaciones, ya que se detiene en todos los errores, incluso en los errores para los que ha escrito código de control.

Es más lógico activar **Interrupción en módulos de clase**:

- La ejecución no se detendrá en el código del módulo de clase para el que ha escrito un controlador de errores.

- La ejecución sólo se detiene en los errores no controlados en el módulo de clase y que, por tanto, se devolverían al que llama al método.

- Cuando se inicia el entorno de desarrollo de Visual Basic, la opción predeterminada es **Interrupción en módulos de clase**.

- Si no hay módulos de clase, **Interrupción en módulos de clase** es equivalente a **Interrupción en errores no controlados**.

Sugerencia Cuando llega a un punto de interrupción con las opciones **Interrupción en módulos de clase** o **Interrupción en errores no controlados**, puede recorrer el módulo o saltar el error si presiona Alt-F8 o Alt-F5 dentro del código de control de errores o dentro del código que llamó al procedimiento en que se produjo el error.

Para obtener más información La depuración se describe en detalle en "Depurar el código y controlar los errores".

Ciclo de vida de los formularios de Visual Basic

Puesto que son visibles para el usuario, los formularios y los controles tienen un ciclo de vida diferente del resto de los objetos. Por ejemplo, un formulario no se cierra sólo por liberar todas las referencias al mismo. Visual Basic mantiene una colección global con todos los formularios de un proyecto y sólo quita un formulario de dicha colección cuando lo descarga.

De forma parecida, Visual Basic mantiene una colección con todos los controles de cada formulario. Puede cargar y descargar controles desde las matrices de controles, pero no basta con liberar todas las referencias a un control para destruirlo.

Para obtener más información Las colecciones **Forms** y **Controls** se describen en "Colecciones en Visual Basic", anteriormente en este capítulo.

Estados por los que pasa un formulario de Visual Basic

Un formulario de Visual Basic pasa normalmente por cuatro estados en su ciclo de vida:

1. Creado, pero no cargado.

2. Cargado, pero no mostrado.

3. Mostrado.

4. Liberación completa de memoria y recursos.

Hay un quinto estado por el que un formulario puede pasar en algunas circunstancias: descargado y sin referencias, mientras se sigue haciendo referencia a un control.

En este tema se describen estos estados y las transiciones entre ellos.

Creado, pero no cargado

El comienzo de este estado viene marcado por el evento Initialize. El código del procedimiento de evento Form_Initialize es, por tanto, el primer código que se ejecuta cuando se crea un formulario.

En este estado, el formulario existe como objeto, pero no tiene ventana. No existe todavía ninguno de sus controles. *Los formularios siempre pasan por este estado*, aunque por poco tiempo.

Por ejemplo, si ejecuta `Form1.Show`, se creará el formulario y se ejecutará Form_Initialize; en cuanto termine Form_Initialize, se cargará el formulario, que es el estado siguiente.

Lo mismo ocurre si especifica un formulario como **Objeto inicial**, en la ficha **General** del cuadro de diálogo **Propiedades del proyecto**, que se encuentra disponible en el menú **Proyecto**. El formulario especificado como objeto inicial se crea al iniciar el proyecto y se carga y se muestra de forma inmediata.

Nota Puede forzar que el formulario se cargue desde Form_Initialize si invoca el método **Show** o las propiedades y los métodos integrados, como se describe a continuación.

Creado, pero no cargado

Por el contrario, el código siguiente crea una instancia de Form1 sin avanzar hasta el estado de cargado:

```
Dim frm As Form1
Set frm = New Form1
```

En cuanto termina Form_Initialize, los únicos procedimientos que puede ejecutar sin forzar la carga del formulario son los procedimientos **Sub**, **Function** y **Property** que agregó en la ventana de código del formulario. Por ejemplo, puede agregar a Form1 el método siguiente:

```
Public Sub ANewMethod ()
   Debug.Print "Ejecutando ANewMethod"
End Sub
```

Puede llamar a este método mediante la variable `frm` (es decir, `frm.ANewMethod`) sin forzar el paso del formulario al siguiente estado. De forma similar, podría llamar a ANewMethod para crear el formulario:

```
Dim frm As New Form1
frm.ANewMethod
```

Puesto que `frm` se declara **As New**, el formulario no se crea hasta la primera vez que utilice la variable en el código: en este caso, cuando se invoca ANewMethod. Después de ejecutar el código anterior, el formulario se mantiene creado, pero no cargado.

Nota Ejecutar `Form1.ANewMethod` sin declarar una variable formulario tiene el mismo efecto que el ejemplo anterior. Como se explica en "Personalizar las clases Form", Visual Basic crea una variable global oculta por cada clase de formulario. Dicha variable tiene el mismo nombre que la clase; es como si Visual Basic declarara `Public Form1 As New Form1`.

Puede ejecutar tantas propiedades y métodos personalizados como desee sin forzar la carga del formulario. Sin embargo, en el momento en que tenga acceso a una de las propiedades integradas del formulario o a cualquier control del formulario, el formulario pasa al siguiente estado.

Nota Puede que le ayude pensar que un formulario se compone de dos partes: una parte de código y una parte visual. Antes de cargar el formulario, sólo está en memoria la parte de código. Puede invocar tantos procedimientos como desee en la parte de código sin cargar la parte visual del formulario.

El único estado por el que pasan todos los formularios

Creado, pero no cargado es el único estado por el que pasan *todos* los formularios. Si la variable `frm` de los ejemplos anteriores se establece a **Nothing**, como se muestra a continuación, el formulario se destruirá antes de pasar a su siguiente estado:

```
Dim frm As New Form1
frm.ANewMethod
Set frm = Nothing       ' Se destruye el formulario.
```

Este uso de los formularios no es mejor que los módulos de clase, ya que la gran mayoría de los formularios pasan al siguiente estado.

Cargado, pero no mostrado

El evento que marca el comienzo de este estado es el evento Load. El código del procedimiento de evento Form_Load se ejecuta en cuanto el formulario entra en el estado cargado.

Cuando se inicia el procedimiento de evento Form_Load, se han creado y cargado todos los controles del formulario y el formulario tiene una ventana completa, con controlador de ventana (hWnd) y contexto de dispositivo (hDC), aunque dicha ventana no se muestre todavía.

Todo formulario que pueda verse tiene que cargarse antes.

Muchos formularios pasan automáticamente del estado Creado pero no cargado, al estado Cargado pero no mostrado. Un formulario se carga automáticamente si:

- El formulario se ha especificado como **Objeto inicial** en la ficha **General** del cuadro de diálogo **Propiedades del proyecto**.

- El método **Show** es la primera propiedad o método del formulario a la que se invoca, por ejemplo `Form1.Show`.

- La primera propiedad o método del formulario que se va a invocar es uno de sus miembros integrados, por ejemplo el método **Move**.

  Nota En este caso se incluyen cualquiera de los controles del formulario, puesto que cada control define una propiedad del formulario; es decir, para tener acceso a la propiedad **Caption** de Command1 tiene que pasar primero por la propiedad Command1 del formulario: `Command1.Caption`.

- Se utiliza la instrucción **Load** para cargar el formulario, sin usar antes **New** o **As New** para crear el formulario, como se ha descrito anteriormente.

Formularios que nunca se muestran

En los dos primeros casos enumerados antes, el formulario pasará directamente al estado visible en cuanto termine Form_Load. En los dos últimos casos, el formulario seguirá cargado, pero no mostrado.

Durante mucho tiempo ha sido una práctica común de programación en Visual Basic cargar un formulario pero no mostrarlo nunca. Esto se puede hacer por varios motivos:

- Para usar el control **Timer** con el fin de generar eventos periódicos.

- Para usar los controles por su funcionalidad, no por su interfaz de usuario. Por ejemplo, en comunicaciones serie o en el acceso al sistema de archivos.

- Para ejecutar transacciones DDE.

Nota Con la Edición Profesional o la Edición Empresarial, puede crear componentes ActiveX (anteriormente llamados servidores OLE), que suelen ofrecer una mejor funcionalidad que los controles para usar código sin parte visual. Vea "Crear componentes ActiveX" en la *Guía de herramientas componentes*.

Vuelta a casa

Los formularios vuelven del estado visible al estado cargado cuando se les oculta. Sin embargo, la vuelta al estado cargado no vuelve a ejecutar el evento Load. Form_Load se ejecuta sólo una vez dentro del ciclo de vida de un formulario.

Mostrado

En cuanto un formulario se hace visible, el usuario puede interactuar con él. Después, se puede ocultar y mostrar el formulario tantas veces como desee antes de descargarlo finalmente.

Interludio: preparación para la descarga

Un formulario puede estar oculto o visible cuando se descarga. Si no está explícitamente oculto, permanece visible hasta que se descarga.

El último evento que el formulario recibe antes de descargarse es el evento **Unload**. Sin embargo, antes de que esto ocurra, recibe un evento muy importante llamado **QueryUnload**. **QueryUnload** le ofrece la oportunidad de detener la descarga del formulario. Si hay datos que quizá desee guardar el usuario, éste es el momento de pedir al usuario que guarde o descarte sus modificaciones.

Importante Si establece el argumento **Cancel** de QueryUnload a **True**, detendrá la descarga del formulario, lo que deniega la instrucción **Unload**.

Una de las características más eficaces de este evento es que le informa de *cómo* se ha provocado la descarga: por un clic del usuario en el botón de cierre, por la ejecución de una instrucción **Unload,** por el cierre de la aplicación o por el cierre de Windows. De esta forma, QueryUnload le permite ofrecer al usuario la posibilidad de cancelar el cierre del formulario, al tiempo que le permite cerrar el formulario desde el código cuando sea necesario.

Importante En algunas circunstancias, los formularios no reciben el evento QueryUnload: si utiliza la instrucción **End** para terminar el programa o si hace clic en el botón **Terminar** (o selecciona **Terminar** en el menú **Ejecutar**) dentro del entorno de desarrollo.

Para obtener más información Vea "QueryUnload (Evento)" en la *Referencia del lenguaje*.

Vuelta al estado Creado, pero no cargado

Cuando se descarga el formulario, Visual Basic lo quita de la colección **Forms**. A menos que mantenga activada alguna variable con una referencia a dicho formulario, el formulario se destruirá y se liberarán su memoria y recursos.

Si mantiene una referencia del formulario en alguna variable, como en la variable global oculta descrita en "Personar las clases Form", el formulario vuelve al estado Creado, pero no cargado. El formulario ya no tiene ventana y sus controles no existen.

El objeto sigue utilizando recursos y memoria. Siguen existiendo todos los datos de las variables de nivel de módulo contenidos en la parte de código del formulario. Sin embargo, desaparecen las variables **Static** de los procedimientos de evento.

Puede usar esta referencia que ha guardado para llamar a los métodos y las propiedades que agregó al formulario, pero si invoca los miembros incorporados del formulario o si tiene acceso a sus controles, el formulario volverá a cargarse y se ejecutará el evento Form_Load.

Liberar completamente la memoria y recursos

La única manera de liberar toda la memoria y los recursos es descargar el formulario y establecer todas sus referencias a **Nothing**. La referencia que más se suele pasar por alto al realizar esta tarea es la variable global oculta mencionada anteriormente. Si en algún momento ha hecho referencia al formulario por su nombre de clase, como aparece en la propiedad **Name** de la ventana Propiedades, ha utilizado la variable global oculta. Para liberar memoria del formulario, tiene que establecer esta variable a **Nothing**. Por ejemplo:

```
Set Form1 = Nothing
```

El formulario recibirá un evento Terminate justo antes de destruirse.

Sugerencia Muchos programadores profesionales evitan el uso de la variable global oculta y prefieren declarar sus propias variables de formulario para administrar la duración de los formularios (por ejemplo, `Dim dlgAbout As New frmAboutBox`).

Nota Ejecutar la instrucción **End** descarga todos los formularios y establece todas las variables de objeto del programa a **Nothing**. Sin embargo, es una manera muy brusca de terminar los programas. Ninguno de los formularios recibirá los eventos QueryUnload, Unload o Terminate, y los objetos que creó no recibirán los eventos Terminate.

Descargado y sin referencias, pero un control tiene referencias

Para llegar a este estado, tiene que descargar y liberar el formulario al tiempo que mantiene una referencia a alguno de sus controles. Si esto le parece una tontería, no dude que lo es.

```
Dim frm As New Form1
Dim obj As Object
frm.Show vbModal
' Cuando se descarta el formulario modal, guarda una
' referencia de uno de sus controles.
Set obj = frm.Command1
Unload frm
Set frm = Nothing
```

El formulario se descarga y se liberan todas las referencias a él. Sin embargo, todavía queda una referencia a uno de sus controles y esto impide que la parte de código del formulario libere la memoria utilizada. Si invoca cualquiera de las propiedades o métodos de este control, se volverá a cargar el formulario:

```
obj.Caption = "Vuelta a la vida"
```

Los valores de las variables de nivel de módulo serán los mismos de antes, pero los valores de las propiedades de todos los controles recuperarán sus valores predeterminados, como si el formulario se cargara por primera vez. Se ejecutará el evento Form_Load.

Nota En algunas versiones anteriores de Visual Basic, el formulario no se volvía a iniciar completamente y Form_Load no se ejecutaba otra vez.

Nota No todos los formularios se comportan como los formularios de Visual Basic. Por ejemplo, Microsoft Forms, proporcionado con Microsoft Office, no tiene eventos Load y Unload; cuando estos formularios reciben los eventos Initialize, todos sus controles existen y están listos para su uso.

Para obtener más información Los formularios se describen en el tema "Diseñar un formulario" de "Formularios, controles y menús" y en el tema "Más acerca de los formularios" de "Crear una interfaz de usuario".

Módulos de clase y módulos estándar

Las clases difieren de los módulos estándar en la forma de almacenar los datos. Nunca hay más de una copia de los datos de un módulo estándar. Esto significa que cuando una parte del programa modifica una variable pública de un módulo estándar y otra parte del programa la lee después, obtendrá el mismo valor.

Por otra parte, los datos de un módulo de clase existen por separado para cada instancia de la clase, es decir, por cada objeto creado a partir de la clase.

Según el mismo razonamiento, los datos de un módulo estándar tienen el alcance del programa; es decir, existen durante toda la ejecución del programa, mientras que los datos de cada instancia de un módulo de clase sólo existen durante la duración del objeto; se crean cuando se crea el objeto y se destruyen cuando se destruye el objeto.

Por último, las variables declaradas **Public** en los módulos estándar son visibles desde cualquier parte del proyecto, mientras que sólo puede tener acceso a las variables **Public** de los módulos de clase si dispone de una variable de objeto que contenga una referencia a una instancia concreta de una clase.

Todo lo anterior también se aplica a los procedimientos públicos de los módulos estándar y de los módulos de clase. Todo esto se ilustra en el ejemplo siguiente. Puede ejecutar este código si abre un proyecto Exe estándar y utiliza el menú **Proyecto** para agregar un módulo y un módulo de clase.

Coloque el código siguiente en Class1:

```
' Lo siguiente es una propiedad de los objetos Class1.
Public Comment As String

' Esto es un método de los objetos Class1.
Public Sub ShowComment()
   MsgBox Comment, , gstrVisibleEverywhere
End Sub
```

Coloque el código siguiente en Module1:

```
' El código del módulo estándar es global.
Public gstrVisibleEverywhere As String

Public Sub CallableAnywhere(ByVal c1 As Class1)
   ' La línea siguiente modifica una variable global
   ' (propiedad) de una instancia de Class1. Sólo
   ' afecta al objeto específico pasado a este
   ' procedimiento.
   c1.Comment = "Tocado por una función global".
End Sub
```

Coloque dos botones de comando en Form1 y agregue el código siguiente a Form1:

```
Private mc1First As Class1
Private mc1Second As Class1
```

```
Private Sub Form_Load()
   ' Crea dos instancias de Class1.
   Set mc1First = New Class1
   Set mc1Second = New Class1
   gstrVisibleEverywhere = "Datos de cadena global"
End Sub

Private Sub Command1_Click()
   Call CallableAnywhere(mc1First)
   mc1First.ShowComment
End Sub

Private Sub Command2_Click()
   mc1Second.ShowComment
End Sub
```

Presione F5 para ejecutar el proyecto. Cuando se carga, Form1 crea dos instancias de Class1, cada una con sus propios datos. Form1 establece también el valor de la variable global `gstrVisibleEverywhere`.

Presione Command1, que llama al procedimiento global y pasa una referencia al primer objeto Class1. El procedimiento global establece la propiedad Comment y Command1 llama al método ShowComment para presentar los datos del objeto.

Como se puede ver en la figura 9.6, el cuadro de mensajes resultante muestra que el procedimiento global CallableAnywhere establece la propiedad Comment del objeto que se le pasa y que la cadena global es visible desde dentro de Class1.

Figura 9.6 Cuadro de mensajes del primer objeto Class1

Presione Command2, que simplemente llama al método ShowComment de la segunda instancia de Class1.

Como se muestra en la figura 9.7, los dos objetos tienen acceso a la variable de cadena global; pero la propiedad Comment del segundo objeto está vacía, porque llamar al procedimiento global CallableAnywhere solamente modificó la propiedad Comment del primer objeto.

Figura 9.7 Cuadro de mensajes del segundo objeto Class1

Importante Evite que el código de sus clases dependa de datos globales, es decir, de variables públicas de módulos estándar. Pueden existir simultáneamente varias

instancias de una clase y todos estos objetos comparten los datos globales del programa.

Además, el empleo de variables globales en el código de los módulos de clase, no cumple con el concepto de encapsulamiento de la programación orientada a objetos, puesto que los objetos creados a partir de dichas clases no contienen todos sus datos.

Datos estáticos de clase

Puede haber ocasiones en las que quiera que todos los objetos creados a partir de un módulo de clase compartan un único elemento de datos. A esto se le suele llamar *datos estáticos de clase*.

No puede implementar datos de clase realmente estáticos en los módulos de clase de Visual Basic. Sin embargo, puede simularlo mediante procedimientos **Property** que establezcan y devuelvan el valor de un miembro de datos **Public** de un módulo estándar, como en el siguiente fragmento de código:

```
' Propiedad de sólo lectura que devuelve el
' nombre de la aplicación.
Property Get CommonString() As String
   ' La variable gstrVisibleEverywhere se almacena en
   ' un módulo estándar y se declara Public.
   CommonString = gstrVisibleEverywhere
End Property
```

Nota No puede usar la palabra clave **Static** para las variables de nivel de módulo en los módulos de clase. La palabra clave **Static** sólo se puede usar dentro de procedimientos.

Es posible simular datos estáticos de clase que no sean de sólo lectura mediante un procedimiento **Property Let** (o **Property Set** si la propiedad contiene una referencia de objeto) que asigne un nuevo valor al miembro de datos del módulo estándar. Sin embargo, usar variables globales de esta forma no cumple con el concepto de encapsulamiento y no se recomienda.

Por ejemplo, la variable `gstrVisibleEverywhere` puede establecerse desde cualquier parte del proyecto, incluso desde código no perteneciente a la clase que tiene la propiedad CommonString. Esto puede conducir a situaciones de error en el programa.

Para obtener más información Los datos globales de los componentes ActiveX requieren un tratamiento diferente al de los programas normales. Si tiene la Edición Profesional o la Edición Empresarial de Visual Basic, vea el tema "Módulos estándar y módulos de clase" en "Principios generales del diseño de componentes", de la sección *Crear componentes ActiveX de* la *Guía de herramientas componentes.*

Agregar propiedades y métodos a una clase

Las propiedades y los métodos de una clase forman la interfaz predeterminada de dicha clase. La interfaz predeterminada es la manera más habitual de manipular los objetos.

En general, las propiedades representan *datos de un objeto*, mientras que los métodos representan *acciones que un objeto puede realizar*. Para expresarlo de otro modo, las propiedades proporcionan la descripción de un objeto y los métodos su comportamiento.

Los temas siguientes describen los mecanismos para agregar propiedades y métodos a las clases y tratan algunos de los problemas que se pueden encontrar.

- Agregar propiedades a una clase Describe las dos maneras de agregar propiedades a las clases: la manera fácil y la correcta.

- Procedimientos Property frente a variables públicas Aunque los procedimientos de propiedad son la mejor manera de implementar la mayoría de las propiedades, las variables públicas tienen su utilidad.

- Poner en funcionamiento los procedimientos de propiedad Proporciona detalles completos acerca del funcionamiento de los procedimientos de propiedad y describe la implementación de los tipos de propiedad más comunes. Por ejemplo, sólo lectura, lectura y escritura, objeto y **Variant**.

- Agregar métodos a una clase Los métodos (el comportamiento de un objeto) son simples procedimientos públicos que se pueden agregar a un módulo de clase.

- Decidir entre propiedad o método Algunas veces no es obvio si la funcionalidad deseada se debe implementar como propiedad o como método. Este tema proporciona algunas recomendaciones.

- Crear una propiedad o un método predeterminados Puede usar el cuadro de diálogo **Atributos del procedimiento** para asignar a una clase un miembro predeterminado.

- Propiedades y métodos Friend Los procedimientos **Friend** se encuentran a mitad de camino entre los **Public** y los **Private**. Al igual que estos últimos, no forman parte de la interfaz de una clase y, al igual que los **Public**, son visibles desde cualquier parte del proyecto.

  Nota Los eventos no forman parte de la interfaz predeterminada. Los eventos son interfaces *salientes,* es decir, interfaces que afectan a otros objetos, mientras que las propiedades y los métodos pertenecen a las *interfaces entrantes,* es decir, las interfaces cuyos miembros se invocan desde otros objetos. La interfaz predeterminada de un objeto de Visual Basic es una interfaz entrante.

Importante Los siguientes nombres no pueden usarse como nombres de propiedades o de métodos, ya que pertenecen a las interfaces subyacentes **IUnknown** e **IDispatch**: QueryInterface, AddRef, Release, GetTypeInfoCount, GetTypeInfo, GetIDsOfNames e Invoke. Estos nombres producirán un error de compilación.

Para obtener más información Los eventos se describen en "Agregar eventos a una clase", más adelante en este capítulo.

Agregar propiedades a una clase

La forma más sencilla de definir las propiedades de una clase es agregar variables públicas al módulo de clase. Por ejemplo, puede crear de forma sencilla una clase Account si declara dos variables públicas en un módulo de clase llamado Account:

```
Public Balance As Double
Public Name As String
```

Es muy fácil. Tanto como crear los datos privados de una clase; simplemente declare una variable **Private** y sólo será accesible desde el código del módulo de clase:

```
Private mstrMothersMaidenName As String
Private mintWithdrawalsMonthToDate As Integer
```

Ocultar datos

La capacidad de proteger parte de los datos de un objeto a la vez que se expone el resto como propiedades se llama *ocultación de los datos.* Éste es un aspecto del principio de encapsulamiento orientado a objetos, como se explica en "Clases: combinar tipos definidos por el usuario y procedimientos".

Ocultar los datos significa que puede efectuar modificaciones en la implementación de una clase sin afectar al código existente que utilice el objeto Account, por ejemplo mediante el cambio de **Integer** a **Long** de la variable privada de la clase Account `mintWithdrawalsMonthToDate`.

Ocultar los datos también permite definir propiedades de sólo lectura. Por ejemplo, se puede usar un procedimiento **Property Get** para obtener el valor de la variable privada que contiene el número de retiradas de efectivo del mes e incrementar sólo la variable desde el código del objeto Account. Lo que nos lleva a los procedimientos de propiedad.

Procedimientos de propiedad

Ocultar los datos no tendría utilidad si la única forma de crear propiedades fuera la declaración de variables públicas. ¿De qué sirve dar una propiedad Type a la clase Account, si cualquier código que tenga una referencia de un objeto Account puede establecer alegremente el tipo de cuenta a cualquier valor?

Los procedimientos de propiedad le permiten ejecutar código cuando se establece o se obtiene el valor de una propiedad. Por ejemplo, podría implementar la propiedad Type del objeto Account como una pareja de procedimientos de propiedad:

```
Public Enum AccountTypes
   atSavings = 1
   atChecking
   atLineOfCredit
End Enum

' Almacenamiento de datos privado de la propiedad Type.
Private mbytType As AccountTypes

Public Property Get Type() As AccountTypes
   Type = mbytType
End Property

Public Property Let Type(ByVal NewType As AccountTypes)
   Select Case NewType
      Case atChecking, atSavings, atLineOfCredit
         ' No hace nada si NewType es válido.
```

```
      Case Else
         Err.Raise Number:=vbObjectError + 32112, _
         Description:="Tipo de cuenta no válido"
   End Select
   If mbytType > NewType Then
      Err.Raise Number:=vbObjectError + 32113, _
      Description:="Imposible actualizar el tipo de
      cuenta"
   Else
      mbytType = NewType
   End If
End Property
```

Ahora suponga que tiene una variable llamada `acct` que contiene una referencia a un objeto Account. Cuando se ejecuta el código `x = acct.Type`, se invoca el procedimiento **Property Get** para devolver el valor almacenado en el miembro de datos privado `mbytType` del módulo de clase.

Cuando se ejecuta el código `acct.Type = atChecking` se invoca el procedimiento **Property Let**. Si el objeto Account es nuevo, `mbytType` será cero y se le puede asignar cualquier tipo válido de cuenta. Si el tipo actual de cuenta es atSavings, se actualizará la cuenta.

Sin embargo, si el tipo actual de cuenta es atLineOfCredit, **Property Let** producirá un error que evita la actualización. De igual forma, si se ejecuta el código `acct.Type = 0`, la instrucción **Select** del procedimiento **Property Let** detectará un tipo de cuenta no válido y producirá un error.

En resumen, *los procedimientos de propiedad permiten que un objeto proteja y valide sus propios datos*.

Para obtener más información Entonces, ¿para qué sirven las variables públicas? En "Procedimientos Property frente a variables públicas" se explican los usos apropiados de cada uno.

Las posibilidades de los procedimientos de propiedad se exploran con más detalle en "Poner en funcionamiento los procedimientos de propiedad".

Procedimientos Property frente a variables públicas

Los procedimientos de propiedad son un medio tan poderoso de encapsulamientos que quizá se esté preguntando si alguna vez va a trabajar con variables públicas. La respuesta, como siempre en programación, es "Por supuesto, algunas veces". He aquí algunas reglas básicas:

Utilice procedimientos de propiedad cuando:

- La propiedad es de sólo lectura o no se puede modificar después de establecerla.
- La propiedad tiene un conjunto de valores bien definido que es necesario validar.
- Los valores fuera de un determinado intervalo (por ejemplo, números negativos) son válidos para el tipo de datos de la propiedad, pero provocarían errores de programa si se permite que la propiedad asuma dichos valores.
- Establecer la propiedad provoca algún cambio perceptible en el estado del objeto, por ejemplo una propiedad **Visible**.

- Establecer la propiedad provoca cambios en otras variables internas o en los valores de otras propiedades.

Utilice variables públicas en las propiedades de lectura y escritura cuando:

- La propiedad es de un tipo autovalidable. Por ejemplo, si a una variable booleana se le asigna un valor distinto de **True** o **False**, se produce un error o una conversión automática de datos.

- Es válido cualquier valor del intervalo aceptado por el tipo de datos. Esto se dará en muchas propiedades de tipo **Single** o **Double**.

- La propiedad es del tipo de datos **String** y no hay limitaciones en el tamaño o el valor de la cadena.

Nota No implemente una propiedad como variable pública solamente por evitar una llamada de función. Internamente, Visual Basic implementará de todos modos las variables públicas de los módulos de clase como parejas de procedimientos de propiedad, ya que esto es lo requerido por la biblioteca de tipos.

Para obtener más información Las capacidades de los procedimientos de propiedad se explican con más detalle en "Poner en funcionamiento los procedimientos de propiedad".

Poner en funcionamiento los procedimientos de propiedad

Visual Basic proporciona tres tipos de procedimientos de propiedad, que se describen en la tabla siguiente.

Procedimiento	Propósito
Property Get	Devuelve el valor de una propiedad.
Property Let	Establece el valor de una propiedad.
Property Set	Establece el valor de una propiedad de objeto, es decir, una propiedad que contiene una referencia a un objeto.

Como puede ver en la tabla, cada uno de estos procedimientos de propiedad desempeña un papel concreto en la definición de una propiedad. La propiedad típica estará compuesta por una pareja de procedimientos de propiedad: **Property Get** para obtener el valor de la propiedad y **Property Let** o **Property Set** para asignar un nuevo valor.

Estos papeles se pueden superponer en algunos casos. La razón por la que hay dos tipos de procedimientos de propiedad para asignar un valor es que Visual Basic tiene una sintaxis especial para la asignación de referencias de objeto a las variables de objeto:

```
Dim wdg As Widget
Set wdg = New Widget
```

La regla es sencilla: Visual Basic llama a **Property Set** si se utiliza la instrucción **Set** y a **Property Let** en caso contrario.

Sugerencia Para distinguir entre **Property Let** y **Property Set**, recuerde los lenguajes Basic antiguos, cuando en vez de `x = 4` tenía que escribir `Let x = 4`, sintaxis aceptada por Visual Basic hasta hoy en día. Visual Basic siempre llama al procedimiento de propiedad que corresponde al tipo de asignación: **Property Let**

para `Let x = 4` y **Property Set** para `Set c1 = New Class1`, es decir, para las propiedades de objeto.

Para obtener más información En el tema "Trabajar con objetos" de "Fundamentos de programación" se explica el uso de la instrucción **Set** con las variables de objeto.

Propiedades de lectura y escritura

El siguiente fragmento de código muestra una propiedad típica de lectura y escritura:

```
' Almacenamiento privado del valor de la propiedad.
Private mintNumberOfTeeth As Integer

Public Property Get NumberOfTeeth() As Integer
   NumberOfTeeth = mintNumberOfTeeth
End Property

Public Property Let NumberOfTeeth(ByVal NewValue _
   As Integer)
   ' (Se omite el código para validar la propiedad.)
   mintNumberOfTeeth = NewValue
End Property
```

El nombre de la variable privada que almacena el valor de la propiedad está compuesto por un prefijo de alcance (`m`) que la identifica como variable de nivel de módulo, un prefijo de tipo (`int`) y un nombre (`NumberOfTeeth`). Usar el mismo nombre que para la propiedad sirve para recordar que la variable y la propiedad están relacionadas.

Como sin duda habrá observado, en este ejemplo y en los anteriores los nombres de los procedimientos de propiedad que forman una propiedad de lectura y escritura tienen que ser iguales.

Nota Los procedimientos de propiedad son públicos de forma predeterminada, de modo que si se omite la palabra clave **Public**, seguirán siendo públicos. Si por alguna razón quiere que una propiedad sea privada, es decir, sólo accesible desde dentro del objeto, tiene que declararla con la palabra clave **Private**. Lo correcto es usar la palabra clave **Public**, aunque no se requiera, ya que deja claras sus intenciones.

Procedimientos de propiedad como trabajo y como diversión

Es instructivo recorrer el código de algún procedimiento de propiedad. Abra un nuevo proyecto Exe estándar y agregue un módulo de clase mediante el menú **Proyecto**. Copie el código de la propiedad NumberOfTeeth, mostrado anteriormente, a Class1.

Cambie a Form1 y agregue el siguiente código al evento Load:

```
Private Sub Form_Load()
   Dim c1 As Class1
   Set c1 = New Class1
   ' Asigna un nuevo valor a la propiedad.
   c1.NumberOfTeeth = 42
   ' Presenta el valor de la propiedad.
   MsgBox c1.NumberOfTeeth
End Sub
```

Presione F8 para recorrer el código de línea en línea. Observe que cuando se asigna el valor a la propiedad, pasa al procedimiento **Property Let** y que cuando lo obtiene, pasa a

Property Get. Puede que encuentre útil repetir este ejercicio con otras combinaciones de procedimientos de propiedad.

Los argumentos de las parejas de procedimientos de propiedad tienen que coincidir

Los ejemplos de procedimientos de propiedad vistos hasta ahora eran sencillos, como lo son para la mayoría de las propiedades. Sin embargo, los procedimientos de propiedad pueden tener múltiples argumentos e incluso argumentos opcionales. Los múltiples argumentos son útiles en las propiedades que actúan como matrices, como se describe a continuación.

Cuando utilice argumentos, tienen que coincidir los argumentos de cada pareja de procedimientos de propiedad. La tabla siguiente muestra los argumentos requeridos en las declaraciones de los procedimientos de propiedad.

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

Desde el primer argumento al último (1,..., *n*), todos deben compartir los mismos nombres y tipos de datos en los procedimientos de propiedad con el mismo nombre. Como en otros tipos de procedimientos, todos los parámetros requeridos de esta lista tienen que preceder al primer parámetro opcional.

Probablemente habrá advertido que la declaración del procedimiento **Property Get** tiene un argumento menos que los de **Property Let** o **Property Set** relacionados. El tipo de datos del procedimiento **Property Get** tiene que ser igual que el tipo de datos del último argumento (*n*+1) de los **Property Let** o **Property Set** relacionados.

Por ejemplo, considere esta declaración de **Property Let** para una propiedad que actúa como matriz de dos dimensiones:

```
Public Property Let Things(ByVal X As Integer, _
ByVal Y As Integer, ByVal Thing As Variant)
   ' (Se omite el código para asignar el elemento
   '  de la matriz.)
End Property
```

La declaración de **Property Get** tiene que usar argumentos con el mismo nombre y el mismo tipo de datos que los argumentos del procedimiento **Property Let**:

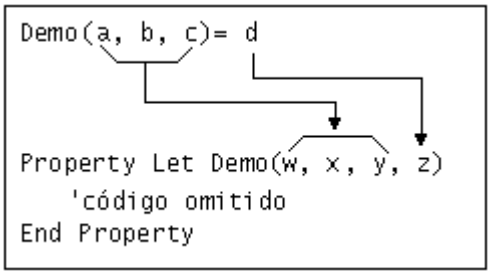
```
Public Property Let Things(ByVal X As Integer, _
ByVal Y As Integer) As Variant
   ' (Se omite el código para leer la matriz.)
End Property
```

El tipo de datos del último argumento de una declaración **Property Set** tiene que ser un tipo de objeto o una variable **Variant**.

Coincidencia de los argumentos

La razón de estas reglas de coincidencia de los argumentos se ilustra en la figura 9.8, que muestra cómo Visual Basic asocia las partes de la instrucción de asignación con los argumentos de un procedimiento **Property Let**.

Figura 9.8 Llamada a un procedimiento Property Let



El uso más común de los procedimientos de propiedad con varios argumentos es el de crear matrices de propiedades.

Propiedades de sólo lectura

Para crear una propiedad de sólo lectura, simplemente omita el procedimiento **Property Let** o, para las propiedades de objeto, el procedimiento **Property Set**.

Propiedades de objeto

Si va a crear una propiedad de objeto de lectura y escritura, utilice una **Property Get** y una **Property Set**, como se muestra a continuación:

```
Private mwdgWidget As Widget

Public Property Get Widget() As Widget
   ' La instrucción Set se debe usar para
   ' obtener una referencia de objeto.
   Set Widget = mwdgWidget
End Property

Public Property Set Widget(ByVal NewWidget As Widget)
   Set mwdgWidget = NewWidget
End Property
```

Propiedades Variant

Las propiedades de lectura y escritura del tipo de datos **Variant** son las más complicadas. Utilizan los tres tipos de procedimientos de propiedad, como se muestra a continuación:

```
Private mvntAnything As Variant

Public Property Get Anything() As Variant
   ' La instrucción Set sólo se utiliza cuando la
   ' propiedad Anything contiene una referencia de
   ' objeto.
   If IsObject(mvntAnything) Then
      Set Anything = mvntAnything
   Else
      Anything = mvntAnything
   End If
End Property

Public Property Let Anything(ByVal NewValue As Variant)
   ' (Se omite el código de validación.)
   mvntAnything = NewWidget
End Property
```

```
Public Property Set Anything(ByVal NewValue As Variant)
   ' (Se omite el código de validación.)
   Set mvntAnything = NewWidget
End Property
```

Los procedimientos **Property Set** y **Property Let** no tienen dificultad, puesto que siempre se les llama en las circunstancias correctas. Sin embargo, **Property Get** tiene que controlar los dos casos siguientes:

```
strSomeString = objvar1.Anything
Set objvar2 = objvar1.Anything
```

En el primer caso, la propiedad **Anything** contiene una cadena, que se asigna a una variable **String**. En el segundo caso, **Anything** contiene una referencia de objeto, que se asigna a una variable de objeto.

Puede escribir código para el procedimiento **Property Get** de forma que controle estos dos casos, mediante el uso de la función **IsObject**, que comprueba la variable **Variant** privada antes de devolver el valor.

Naturalmente, si se llama a la primera línea de código cuando **Anything** contiene una referencia de objeto, se produce un error, pero eso no es un problema de **Property Get**, sino un problema derivado del uso de las propiedades **Variant**.

Propiedades de una única escritura

Hay muchas combinaciones posibles de procedimientos de propiedad. Todas ellas son válidas, pero algunas son relativamente poco frecuentes, como las propiedades de sólo escritura (sólo **Property Let**, sin un **Property Get**). Además, algunas dependen de factores distintos a los tipos de procedimientos de propiedad que combine.

Por ejemplo, cuando organiza los objetos del programa con la creación de un modelo de objetos, como se describe en "Modelos de objetos", más adelante en este capítulo, puede desear que un objeto sea capaz de devolver una referencia al objeto que lo contiene. Puede hacer esto si implementa una propiedad **Parent**.

Tiene que establecer esta propiedad **Parent** cuando se crea el objeto, pero a partir de ese momento quizás desee impedir su modificación, ya sea de forma accidental o intencionada. En el ejemplo siguiente se muestra cómo el objeto Account puede implementar una propiedad **Parent** que apunte al objeto **Department** que contiene la cuenta.

```
' Almacenamiento de datos privado de la propiedad
' Parent.
Private mdeptParent As Department

Property Get Parent() As Department
   ' Instrucción Set para devolver
   ' referencias de objeto.
   Set Parent = mdeptParent
End Property

' El valor de la propiedad sólo se puede
' establecer una vez.
Public Property Set Parent(ByVal NewParent _
As Department)
   If deptParent Is Nothing Then
      ' Asigna el valor inicial.
      Set mdeptParent = NewParent
   Else
```

```
      Err.Raise Number:=vbObjectError + 32144, _
      Description:="La propiedad Parent es de sólo lectura"
   End If
End Property
```

Cuando tiene acceso al objeto primario de un objeto **Account**, por ejemplo con el código `strX = acctNew.Parent.Name` para obtener el nombre del departamento, se invoca **Property Get** para devolver la referencia al objeto primario.

El procedimiento **Property Set** de este ejemplo se ha escrito para que la propiedad **Parent** sólo se establezca una vez. Por ejemplo, cuando el objeto **Department** crea una nueva cuenta, puede ejecutar el código `Set acctNew.Parent = Me` para establecer la propiedad. Posteriormente, la propiedad es de sólo lectura.

Para obtener más información Puesto que los formularios de Visual Basic son clases, puede agregar propiedades personalizadas a los formularios. Vea "Personalizar las clases Form", anteriormente en este capítulo.

Agregar métodos a una clase

Los métodos de una clase son como los procedimientos **Sub** o **Function** públicos que suele declarar. Como los procedimientos **Sub** y **Function** son públicos de forma predeterminada, ni siquiera tiene que especificar explícitamente la palabra clave **Public** al crear un método.

Por ejemplo, para crear un método Withdrawal para la clase Account puede agregar este procedimiento **Public Function** al módulo de clase:

```
Public Function WithDrawal(ByVal Amount As Currency, _
ByVal TransactionCode As Byte) As Double
   ' (Código para realizar el cobro y devolver el
   ' nuevo saldo o un error.)
End Function
```

Sugerencia Aunque no es necesario escribir la palabra clave **Public**, es conveniente hacerlo puesto que deja claras sus intenciones a otras personas que vayan a mantener el código en el futuro.

Declaración de métodos como procedimientos Public Sub

La devolución del nuevo saldo es opcional, ya que podría llamar fácilmente a la propiedad Balance del objeto Account después de llamar al método **Withdrawal**. Por tanto, podría escribir el método **Withdrawal** como procedimiento **Public Sub**.

Sugerencia Si siempre tiene que llamar a Balance después de llamar a **Withdrawal**, la devolución del nuevo saldo sería más eficiente. Esto se debe a que, como puede ver en "Agregar propiedades a módulos de clase", el acceso a cualquier propiedad, incluso la lectura de una variable pública, significa una llamada a una función; es decir, un **Property Get** explícito o implícito.

Importante Los siguientes nombres no se pueden usar como nombres de propiedades o de métodos ya que pertenecen a las interfaces subyacentes **IUnknown** e **IDispatch**: QueryInterface, AddRef, Release, GetTypeInfoCount, GetTypeInfo, GetIDsOfNames e Invoke. Estos nombres producirán un error de compilación.

Para obtener más información Para obtener más información acerca de los procedimientos **Sub** y **Function**, vea "Introducción a los procedimientos", en "Fundamentos de programación".

Proteger los detalles de implementación

La interfaz pública de una clase se define mediante las declaraciones de las propiedades y los métodos del módulo de clase. Como en la ocultación de los datos, los procedimientos declarados como **Private** no forman parte de la interfaz. Esto quiere decir que puede realizar modificaciones en los procedimientos internos de un módulo de clase sin afectar al código que utiliza los objetos.

Y más importante aún, también puede modificar el código de los procedimientos públicos **Sub** o **Function** que implementa un método sin afectar al código que utiliza el método. Mientras no modifique los tipos de datos de los argumentos del procedimiento o el tipo de datos devuelto por un procedimiento **Function**, la interfaz no se altera.

La ocultación de los detalles de la implementación de un objeto detrás de su interfaz es otra faceta del encapsulamiento. El encapsulamiento le permite mejorar el rendimiento de los métodos o modificar completamente la implementación de un método sin tener que modificar el código que utiliza dicho método.

Nota Las recomendaciones de nomenclatura de los elementos de la interfaz, descritos en "Nombres de propiedades, métodos y eventos", no sólo se aplican a los nombres de las propiedades y de los métodos, sino también a los nombres de los parámetros de los procedimientos **Sub** y **Function** que definen los métodos. Estos nombres de parámetros son visibles cuando ve los métodos en el Examinador de objetos y puede usarlos como parámetros con nombre (es decir, *parámetro:=valor*) al invocar los métodos.

Para obtener más información Los argumentos con nombre se describen en "Pasar argumentos a procedimientos", en "Fundamentos de programación".

Agregar métodos a las clases formulario es una eficaz técnica de programación, que se describe en "Personalizar las clases Form".

Algunas veces no está muy claro si un miembro debe ser una propiedad o un método. "Decidir entre propiedad o método" le ofrece algunas recomendaciones.

Decidir entre propiedad o método

En general, una propiedad es un dato sobre un objeto, mientras que un método es una acción que puede pedir que realice el objeto. Algunos elementos son obviamente propiedades, como **Color** y **Name**, y algunas son obviamente métodos, como **Move** y **Show**.

Sin embargo, como en cualquier otra faceta del comportamiento humano, hay un terreno de nadie en el que un argumento puede ser cualquiera de los dos.

Por ejemplo, ¿por qué el método **Item** de la clase **Collection** de Visual Basic es un método y no una propiedad indizada? ¿No son datos los elementos de una colección? El método **Item** de una supuesta clase de colección **Widgets** podría implementarse de cualquiera de las dos maneras, como se muestra a continuación:

```
' Almacenamiento privado para los objetos de la colección
' Widgets (el mismo para ambas implementaciones).
Private mcol As New Collection

Public Property Get Item(Index As Variant) As Widget
   Set Item = mcol.Item(Index)
End Function
```

```
- o bien -

Public Function Item(Index As Variant) As Widget
   Set Item = mcol.Item(Index)
End Function
```

No hay una gran diferencia entre estas dos implementaciones. Las dos son de sólo lectura, de modo que ambas dependen del método **Add** de la clase **Widgets** para insertar objetos **Widget** en la colección. Las dos delegan todo en un objeto **Collection**, incluso sus errores los genera el objeto **Collection**.

Para obtener más información La delegación se explica en "Múltiples facetas (e interfaces) de la reutilización del código" y en "Crear clases de colección propias", más adelante en este capítulo.

Puede que nunca termine de decidir si un miembro es un dato sobre el objeto o un comportamiento del objeto. Por ejemplo, puede argumentar que **Item** es un método porque la colección realiza una tarea que le ha pedido: busca el **Widget** que desea. Sin embargo, este razonamiento también es válido para explicar lo contrario.

Puede encontrar más útil dejar dar la vuelta al argumento y preguntarse cómo *quiere* definir el miembro. Si quiere que la gente piense que es un dato del objeto, haga que sea una propiedad. Si quiere que piensen que es una acción que puede efectuar el objeto, haga que sea un método.

El argumento de la sintaxis

Una razón de peso para implementar un miembro mediante procedimientos de propiedad depende de la manera en que quiere usar el miembro desde el código. Es decir, ¿podrá el usuario de una colección **Widgets** escribir el código siguiente?

```
Set Widgets.Item(4) = wdgMyNewWidget
```

Si se le permite, implemente el miembro como propiedad de lectura y escritura, mediante **Property Get** y **Property Set**, puesto que los métodos no aceptan esta sintaxis.

Nota En la mayoría de las implementaciones de colecciones con las que encontrará, no se permite esta sintaxis. La implementación de un procedimiento **Property Set** en una colección no es tan fácil como parece.

El argumento de la ventana Propiedades

También puede suponer, por un momento, que el objeto es como un control. ¿Puede imaginarse al miembro visible en la ventana Propiedades o en una página de propiedades? Si no lo ve claro, no implemente el miembro como una propiedad.

El argumento del error sensible

Si olvida que hizo de **Item** una propiedad de sólo lectura e intenta asignarle un valor, entenderá más fácilmente el mensaje de error que Visual Basic provoca en **Property Get** ("Imposible asignar la propiedad de sólo lectura") que el mensaje de error que provoca en un procedimiento **Function** ("La llamada a Function en la parte izquierda de la asignación debe devolver un tipo Variant u Object").

El argumento del último recurso

Como último recurso, tire una moneda al aire. Si ninguno de los argumentos de este tema le parece convincente, es que probablemente no hay una gran diferencia.

Para obtener más información Los procedimientos de propiedad se presentan en "Agregar propiedades a las clases", anteriormente en este capítulo. Los métodos se describen en "Agregar métodos a las clases".

Crear una propiedad o un método predeterminado

Puede asignar propiedades predeterminadas a los objetos creados a partir de las clases, como las propiedades predeterminadas de los objetos proporcionados por Visual Basic. El mejor candidato para ser el miembro predeterminado es el que tenga un uso más frecuente.

Para establecer una propiedad o un método como predeterminado

1. En el menú **Herramientas**, seleccione **Atributos del procedimiento** para abrir el cuadro de diálogo **Atributos del procedimiento**.

2. Haga clic en **Avanzadas** para expandir el cuadro de diálogo **Atributos del procedimiento**.

3. En el cuadro **Nombre**, seleccione la propiedad o el método que es actualmente el predeterminado de la clase. Si la clase no tiene actualmente un miembro predeterminado, pase al punto 5.

   Nota Puede usar el Examinador de objetos para buscar el miembro predeterminado actual de una clase. Cuando selecciona la clase en la lista **Clases**, puede desplazarse por los miembros de la lista **Miembros**; el miembro predeterminado estará marcado con un pequeño círculo azul al lado del icono.

4. En el cuadro **Id. del procedimiento**, seleccione **Ninguno** para quitar el estado predeterminado de la propiedad o del método.

5. En el cuadro **Nombre**, seleccione la propiedad o el método que quiera que sea el nuevo miembro predeterminado.

6. En el cuadro **Id. del procedimiento**, seleccione **(Predeterminado)**; después haga clic en **Aceptar**.

Importante Una clase sólo puede tener un miembro predeterminado. Si una propiedad o un método ya está marcado como predeterminado, tiene que restablecer el **Id. del procedimiento** a **Ninguno** antes de hacer que otra propiedad o método sea el predeterminado. No se producen errores de compilación si dos miembros están marcados como predeterminados, pero es imposible predecir cuál es el que Visual Basic elegirá como predeterminado.

También puede abrir el cuadro de diálogo **Atributos del procedimiento** desde el Examinador de objetos. Esto es muy cómodo cuando va a modificar el miembro predeterminado de una clase, ya que le permite encontrar rápidamente el miembro predeterminado existente.

Para modificar una propiedad predeterminada mediante el Examinador de objetos

1. Presione F2 para abrir el **Examinador de objetos**.

2. En la lista **Clases**, seleccione la clase cuyo miembro predeterminado quiere modificar.

3. En la lista **Miembros**, haga clic con el botón secundario del *mouse* en el miembro que tiene un pequeño círculo azul al lado del icono para abrir el menú contextual. Haga clic en **Propiedades** para mostrar el cuadro de diálogo **Atributos del procedimiento**.

4. Haga clic en **Avanzados** para expandir el cuadro de diálogo **Atributos del procedimiento**.

5. En el cuadro **Id. del procedimiento**, seleccione **(Ninguno)** para quitar el estado predeterminado de la propiedad o del método; después haga clic en **Aceptar**.

6. En la lista **Miembros**, haga clic con el botón secundario en el miembro que quiere que sea el nuevo miembro predeterminado para abrir el menú contextual. Haga clic en **Propiedades** para mostrar el cuadro de diálogo **Atributos del procedimiento**.

7. Haga clic en **Avanzados** para expandir el cuadro de diálogo **Atributos del procedimiento**.

8. En el cuadro **Id. del procedimiento**, seleccione **(Predeterminado)**; después haga clic en **Aceptar**.

Nota No puede usar el cuadro de diálogo **Atributos del procedimiento** para modificar el miembro predeterminado de una clase proporcionada por Visual Basic.

Corregir los predeterminados que ha marcado accidentalmente como Private o Friend

El cuadro de diálogo **Atributos del procedimiento** solamente permite seleccionar propiedades y métodos públicos como predeterminados para una clase. Si marca una propiedad o método públicos como predeterminados para una clase y más tarde cambia la declaración a **Private** o **Friend**, es probable que la propiedad o método sigan actuando igual que si aún estuvieran declarados como **Public**.

Para corregir este problema, deberá volver a marcar la propiedad o método como **Public**, ya que el cuadro de diálogo **Atributos del procedimiento** no mostrará los procedimientos declarados como **Private** y **Friend**. Una vez que haya cambiado la declaración a **Public**, podrá usar el cuadro de diálogo **Atributos del procedimiento** para quitar el atributo **Default**. A continuación, podrá volver a cambiar la declaración a **Friend** o **Private**.

Propiedades y métodos Friend

Además de declarar propiedades y métodos **Public** y **Private**, puede declararlos **Friend**. Los miembros **Friend** son como miembros **Public** para los demás objetos del proyecto. Es decir, parecen parte de la interfaz de una clase. Pero no lo son.

En los componentes ActiveX que puede crear con la Edición Profesional y la Edición Empresarial de Visual Basic, los miembros **Friend** desempeñan un papel importante. Como no forman parte de la interfaz de un objeto, no pueden tener acceso a ellos los programas que utilizan los objetos del componente. Sin embargo, son visibles para todos los demás objetos del componente, por lo que permiten una comunicación interna segura dentro del componente.

Importante Puesto que los miembros **Friend** no forman parte de la interfaz pública de un objeto, no se puede tener acceso a ellos con enlaces en tiempo de compilación, es decir, mediante variables declaradas **As Object**. Para usar

miembros **Friend** tiene que declarar las variables con enlaces en tiempo de diseño, es decir, **As** *clase*.

Los proyectos Exe estándar no pueden ser componentes ActiveX, ya que sus módulos de clase no pueden ser **Public** y, por tanto, no los pueden usar otras aplicaciones. Toda la comunicación entre los objetos de un proyecto Exe estándar es privada y no necesita miembros **Friend**.

Sin embargo, los miembros **Friend** tienen una característica especialmente útil. Como no forman parte de una interfaz ActiveX, se pueden usar para pasar tipos definidos por el usuario de un objeto a otro sin exponerlos públicamente. Por ejemplo, suponga que tiene el siguiente tipo definido por el usuario en un módulo estándar:

```
Public Type udtDemo

   intA As Integer
   lngB As Long
   strC As String
End Type
```

Puede definir la siguiente variable privada y los siguientes miembros **Friend** en Class1:

```
Private mDemo As udtDemo

Friend Property Get Demo() As udtDemo
   Demo = mDemo
End Property

' Observe que udtDemo tiene que pasarse por referencia.
Friend Property Let Demo(NewDemo As udtDemo)
   mDemo = NewDemo
End Property

Friend Sub SetDemoParts(ByVal A As Integer, _
      ByVal B As Long, ByVal C As String)
   mDemo.intA = A
   mDemo.lngB = B
   mDemo.strC = C
End Sub

Public Sub ShowDemo()
   MsgBox mDemo.intA & vbCrLf _
   & mDemo.lngB & vbCrMf & mDemo.strC
End Sub
```

Nota Cuando pasa tipos definidos por el usuario como argumentos de procedimientos **Sub**, **Function** o **Property**, tiene que pasarlos por referencia. (**ByRef** es el paso predeterminado para los argumentos de procedimientos.)

Por tanto, puede escribir el código siguiente para usar Class1:

```
Private Sub Command1_Click()
   Dim c1A As New Class1
   Dim c1B As New Class1
   c1A.SetDemoParts 42, 1138, "Hola"
   c1B.Demo = c1A.Demo
   c1B.ShowDemo
End Sub
```

El cuadro de mensajes presentará 42, 1138 y "Hola".

Nota Puesto que los procedimientos **Friend** no forman parte de la interfaz de una clase, no se incluyen cuando utiliza la instrucción **Implements** para implementar múltiples interfaces, como se describe en "Polimorfismo".

Para obtener más información El uso de miembros **Friend** en los componentes se describe en el tema "Comunicación privada entre los objetos" de "Principios generales del diseño de componentes", en la *Guía de herramientas componentes*.

Agregar eventos a una clase

Bien, suponga que ha creado una simulación de dinosaurios con las clases Estegosaurio, Triceratops y Tiranosaurio. Como toque final, quiere que el Tiranosaurio ruja y que, cuando lo haga, los demás dinosaurios se enteren.

Si la clase Tiranosaurio tiene un evento Rugir, puede controlar dicho evento en todas las demás clases de dinosaurios. En este tema se describe la declaración y el control de eventos en los módulos de clase.

Nota Niños, no intentéis esto en casa, al menos no con muchos dinosaurios. La conexión de cada uno de los dinosaurios con todos los demás dinosaurios mediante eventos podría hacer tan lentos a los dinosaurios que los objetos mamíferos ganarían la simulación.

Las propiedades y los métodos se dice que pertenecen a las *interfaces entrantes* porque se llaman desde fuera del objeto. Por otra parte, los eventos se dice que forman parte de las *interfaces salientes*, puesto que se inician desde dentro del objeto y se controlan fuera del mismo.

En los temas siguientes se describe el proceso de declarar, desencadenar y controlar eventos, con ejemplos.

- Declarar y desencadenar eventos Como los procedimientos, los eventos (y sus argumentos) se tienen que declarar. Para que tenga lugar un evento declarado, el objeto tiene que desencadenarlo.

- Controlar los eventos de un objeto Los eventos que desencadenan los objetos pueden controlarse mediante otros objetos si utiliza variables declaradas con la palabra clave **WithEvents**.

- Comparar WithEvents con los eventos de los controles de los formularios Similitudes y diferencias entre los procedimientos de evento asociados con las variables **WithEvents** y los asociados con los controles de los formularios.

- Agregar un evento a un formulario Un breve ejemplo paso a paso de cómo agregar un evento personalizado a un formulario.

- Resumen de cómo declarar, desencadenar y controlar eventos Un resumen del proceso del uso de eventos en las clases.

Para obtener más información En *Crear componentes ActiveX,* de la *Guía de herramientas componentes* proporcionada con la Edición Profesional y la Edición Empresarial, se describe el uso de los eventos para el diseño de componentes de software propios.

Para obtener una descripción de la mejor manera de administrar los dinosaurios, vea "Polimorfismo", más adelante en este capítulo.

Declarar y desencadenar eventos

Suponga por un momento que tiene una clase Widget. Su clase Widget tiene un método que puede tardar bastante tiempo en ejecutarse y le gustaría que su aplicación pudiera ofrecer algún tipo de indicación de su estado.

Desde luego, puede hacer que el objeto **Widget** muestre un cuadro de diálogo de porcentaje de terminación, pero de esta forma siempre tendría este cuadro de diálogo en todos los proyectos en los que utilice la clase Widget. Una buena práctica de diseño de objetos es permitir que la aplicación que utiliza un objeto controle la interfaz de usuario, a menos que el único propósito del objeto sea administrar un formulario o un cuadro de diálogo.

El propósito de Widget es realizar otras tareas, por lo que es razonable incluir un evento PercentDone y permitir que el procedimiento que llama a los métodos de **Widget** controlen dicho evento. El evento PercentDone también puede ofrecer un mecanismo de cancelación de la tarea.

Puede empezar a crear el ejemplo de código de este tema si abre un proyecto Exe estándar y agrega dos botones y una etiqueta a Form1. En el menú **Proyecto**, seleccione **Agregar módulo de clase** para agregar un módulo de clase al proyecto. Asigne un nombre a los objetos como se muestra en la tabla siguiente.

Objeto	Propiedad	Valor
Módulo de clase	**Name**	Widget
Primer botón	**Caption**	Iniciar tarea
Segundo botón	**Caption**	Cancelar
Etiqueta	**Name** **Caption**	lblPercentDone "0"

La clase Widget

Los eventos se declaran en la sección Declaraciones de los módulos de clase mediante la palabra clave **Event**. Un evento puede tener argumentos **ByVal** y **ByRef**, como se muestra en el evento PercentDone de Widget:

```
Option Explicit
Public Event PercentDone(ByVal Percent As Single, _
ByRef Cancel As Boolean)
```

Cuando el objeto que llama recibe un evento PercentDone, el argumento Percent contiene el porcentaje realizado de la tarea. El argumento **ByRef Cancel** puede establecerse a **True** para cancelar el método que desencadenó el evento.

Nota Puede declarar argumentos de evento como si fueran argumentos de procedimientos, con las excepciones siguientes: los eventos no pueden tener argumentos con nombre, argumentos opcionales ni argumentos ParamArray. Los eventos no tienen valores devueltos.

Desencadenar el evento PercentDone

El evento PercentDone lo desencadena el método **LongTask** de la clase Widget. El método **LongTask** recibe dos argumentos: el periodo de tiempo durante el que el método realizará el trabajo y el mínimo intervalo de tiempo antes de cual **LongTask** se detenga para desencadenar el evento PercentDone.

```
Public Sub LongTask(ByVal Duration As Single, _
ByVal MinimumInterval As Single)
   Dim sngThreshold As Single
   Dim sngStart As Single
   Dim blnCancel As Boolean
```

```
   ' La función Timer devuelve el número fraccionario
   ' de segundos desde medianoche, como Single.
   sngStart = Timer
   sngThreshold = MinimumInterval

   Do While Timer < (sngStart + Duration)
      ' En una aplicación real, una parte del trabajo
      ' se haría aquí en cada paso del bucle.

      If Timer > (sngStart + sngThreshold) Then
         RaiseEvent PercentDone( _
         sngThreshold / Duration, blnCancel)
         ' Comprueba si la operación se canceló.
         If blnCancel Then Exit Sub
         sngThreshold = sngThreshold + MinimumInterval
      End If
   Loop
End Sub
```

Cada segundo de `MinimumInterval` se desencadena el evento PercentDone. Cuando el evento termina, LongTask comprueba si el argumento **Cancel** es **True**.

Nota Por simplificar, **LongTask** asume que se conoce con antelación la duración de la tarea. Pero casi nunca se puede saber. La división de una tarea en partes iguales puede ser difícil y normalmente lo que importa a los usuarios es únicamente el tiempo que pasa hasta que reciben una indicación de que se está realizando alguna tarea.

Para obtener más información Ahora que ha declarado un evento y lo ha desencadenado, ¿cómo puede hacer que lo controle otro objeto? "Controlar los eventos de un objeto" continúa con la historia del objeto Widget.

Controlar los eventos de un objeto

Un objeto que desencadena eventos se conoce como *origen del evento.* Para controlar los eventos desencadenados por un origen de eventos, puede declarar una variable de la clase del objeto con la palabra clave **WithEvents**.

En este tema se continúa el ejemplo del objeto **Widget** empezado en "Declarar y desencadenar eventos". Para controlar el evento PercentDone de un objeto **Widget**, coloque el código siguiente en la sección Declaraciones de Form1:

```
Option Explicit
Private WithEvents mWidget As Widget
Private mblnCancel As Boolean
```

La palabra clave **WithEvents** especifica que la variable `mWidget` se usará para controlar los eventos de un objeto. El tipo de objeto se especifica con el nombre de la clase a partir de la que se creará el objeto.

La variable `mWidget` se declara en la sección Declaraciones de Form1, puesto que las variables **WithEvents** tienen que ser variables de nivel de módulo. Esto es cierto independientemente del tipo de módulo en que las coloque.

La variable `mblnCancel` se usará para cancelar el método LongTask.

Limites de las variables WithEvents

Debe tener en cuenta las siguientes limitaciones del uso de las variables **WithEvents**:

- Una variable **WithEvents** no puede ser una variable genérica de objeto. Es decir, no puede declararla **As Object**: tiene que especificar el nombre de la clase cuando declare la variable.

- No puede declarar una variable **WithEvents** como **As New**. El objeto origen del evento se debe crear y asignar a la variable **WithEvents** de forma explícita.

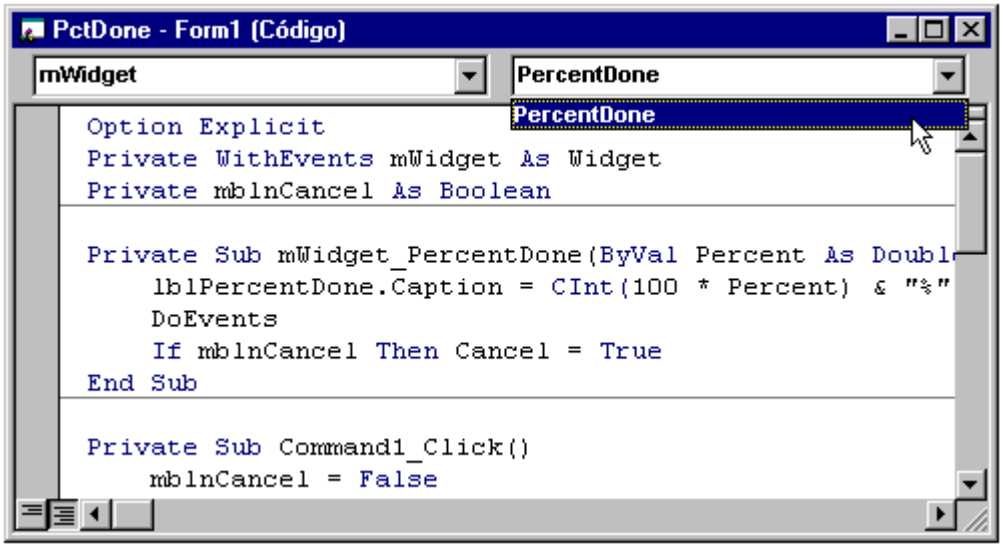
- No puede declarar variables **WithEvents** en un módulo estándar. Sólo puede declararlas en módulos de clase, en módulos de formulario y en otros módulos que definan clases.

- No puede crear matrices de variables **WithEvents**.

Escritura de código para controlar un evento

En cuanto declare una variable **WithEvents**, el nombre de la variable aparece en la lista desplegable de la izquierda de la ventana de código del módulo. Cuando seleccione `mWidget`, los eventos de la clase Widget aparecerán en la lista desplegable de la derecha, como se muestra en la figura 9.9.

Figura 9.9 Evento asociado con una variable WithEvents



La selección de un evento presentará el procedimiento de evento correspondiente, con el prefijo `mWidget_`. Todos los procedimientos de evento asociados con una variable **WithEvents** tendrán el nombre de la variable como prefijo. Agregue el código siguiente al procedimiento de evento mWidget_PercentDone.

```
Private Sub mWidget_PercentDone(ByVal Percent As _
Single, Cancel As Boolean)
   lblPercentDone.Caption = CInt(100 * Percent) & "%"
   DoEvents
   If mblnCancel Then Cancel = True
End Sub
```

Cada vez que se desencadena un evento PercentDone, el procedimiento de evento presenta el porcentaje de terminación en un control **Label**. La instrucción **DoEvents** permite que la etiqueta se vuelva a dibujar y también ofrece al usuario la oportunidad de hacer clic

en el botón **Cancelar**. Agregue el código siguiente al evento Click del botón cuyo título es Cancelar.

```
Private Sub Command2_Click()
   mblnCancel = True
End Sub
```

Si el usuario hace clic en el botón **Cancelar** mientras se ejecuta LongTask, el evento Command2_Click se ejecutará tan pronto como la instrucción **DoEvents** permita procesar el evento. La variable de nivel de módulo `mblnCancel` se establece a **True** y el evento mWidget_PercentDone la comprueba y establece el argumento **ByRef Cancel** a **True**.

Conectar una variable WithEvents con un objeto

Form1 está preparado para controlar los eventos del objeto **Widget**. Todo lo que queda es encontrar un **Widget** en alguna parte.

Cuando declara una variable **WithEvents** en tiempo de diseño, no hay ningún objeto asociado con ella. Una variable **WithEvents** es como cualquier otra variable de objeto. Tiene que crear un objeto y asignar a la variable **WithEvents** una referencia al objeto. Agregue el código siguiente al procedimiento de evento Form_Load para crear el objeto **Widget**.

```
Private Sub Form_Load()
   Set mWidget = New Widget
End Sub
```

Cuando se ejecuta el código anterior, Visual Basic crea un objeto **Widget** y conecta sus eventos con los procedimientos de evento asociados con `mWidget`. A partir de este punto, siempre que **Widget** desencadene el evento PercentDone se ejecutará el procedimiento de evento mWidget_PercentDone.

Para llamar al método LongTask, agregue el código siguiente al evento Click del botón cuyo título es Iniciar tarea.

```
' Botón Iniciar tarea.
Private Sub Command1_Click()
   mblnCancel = False
   lblPercentDone.Caption = "0%"
   lblPercentDone.Refresh

   Call mWidget.LongTask(14.4, 0.66)

   If Not mblnCancel Then lblPercentDone.Caption = 100
End Sub
```

Antes de llamar al método **LongTask** tiene que iniciar la etiqueta que presenta el porcentaje de terminación y tiene que establecer a **False** el indicador booleano de nivel de módulo de la cancelación del método.

LongTask se invoca con una duración de la tarea de 14,4 segundos. El evento PercentDone se desencadenará cada dos tercios de segundo. Cada vez que se desencadena el evento se ejecuta el procedimiento de evento mWidget_PercentDone.

Cuando termina **LongTask** se comprueba `mblnCancel` para ver si **LongTask** ha terminado normalmente o si terminó porque `mblnCancel` se estableció a **True**. El porcentaje de terminación sólo se actualiza en el primer caso.

Ejecutar el programa

Presione F5 para poner el proyecto en modo de ejecución. Haga clic en el botón **Iniciar tarea**. Cada vez que se desencadena el evento PercentDone, la etiqueta se actualiza con el porcentaje terminado de la tarea. Haga clic en el botón **Cancelar** para detener la tarea. Observe que la apariencia del botón **Cancelar** no cambia de forma inmediata cuando hace clic en él. El evento Click no puede producirse mientras la instrucción **DoEvents** no permita el proceso de eventos.

Puede que encuentre instructivo ejecutar el programa con F8 y recorrer el código de línea en línea. Puede ver con claridad cómo la ejecución entra en **LongTask** y después vuelve a pasar brevemente por Form1 cada vez que se desencadena el evento PercentDone.

¿Qué ocurriría si mientras la ejecución vuelve al código de Form1 se llama otra vez al método LongTask? Confusión, caos y, tarde o temprano (si ocurre todas las veces que se desencadena el evento), un desbordamiento de la pila.

Control de eventos de un Widget diferente

Puede hacer que la variable `mWidget` controle los eventos de un objeto **Widget** diferente si asigna a `mWidget` una referencia al nuevo objeto **Widget**. De hecho, puede hacer que el código de Command1 realice esto cada vez que hace clic en el botón si agrega dos líneas de código:

```
Set mWidget = New Widget       '<- Línea nueva.
Call mWidget.LongTask(14.4, 0.66)
Set mWidget = Nothing          '<- Línea nueva.
```

El código anterior crea un nuevo objeto **Widget** cada vez que se hace clic en el botón. Tan pronto como termine el método **LongTask** se libera la referencia al objeto **Widget** mediante el establecimiento de `mWidget` a **Nothing** y se destruye el antiguo objeto **Widget**.

Una variable **WithEvents** sólo puede contener una referencia de objeto al mismo tiempo, de modo que si asigna a `mWidget` un objeto **Widget** diferente, no podrá controlar los eventos del objeto **Widget** anterior. Si `mWidget` es la única variable de objeto que contiene una referencia al **Widget** antiguo, el objeto se destruirá.

Nota Puede declarar tantas variables **WithEvents** como necesite, pero no se aceptan matrices de variables **WithEvents**.

Terminar el control de eventos en una variable WithEvents

Mientras haya un objeto **Widget** asignado a la variable `mWidget`, los procedimientos de evento asociados con `mWidget` se invocarán cada vez que el objeto **Widget** desencadene un evento. Para terminar el control de eventos, puede establecer `mWidget` a **Nothing**, como se muestra en el siguiente fragmento de código.

```
' Termina el control de eventos para mWidget.
Set mWidget = Nothing
```

Cuando una variable **WithEvents** se establece a **Nothing**, Visual Basic desconecta los eventos del objeto desde los procedimientos de evento asociados con la variable.

Importante Una variable **WithEvents** contiene una referencia de objeto, como cualquier otra variable de objeto. Esta referencia de objeto ayuda a mantener vivo un objeto. Cuando establezca todas las referencias de un objeto a **Nothing** para destruirlo, no olvide las variables declaradas **WithEvents**.

Para obtener más información Los procedimientos de evento asociados con las variables **WithEvents** se parecen mucho a los procedimientos de evento de los controles de los formularios. En "Comparar WithEvents con los eventos de los controles de los formularios" se describen las similitudes y las diferencias.

Comparar WithEvents con los eventos de los controles de los formularios

Probablemente habrá observado algunas similitudes entre la manera de usar las variables **WithEvents** y la manera de controlar los eventos desencadenados por los controles de un formulario. En ambos casos, cuando selecciona el evento en la lista desplegable de la derecha de una ventana de código obtiene un procedimiento de evento que contiene los argumentos apropiados del evento.

De hecho, el mecanismo es exactamente el mismo. Un control se trata como una *propiedad* de la clase de formulario y el nombre de dicha propiedad es el valor asignado a la propiedad **Name** del control en la ventana Propiedades.

Es como si hubiera una variable **Public** de nivel de módulo con el mismo nombre que el control y todos los nombres de los procedimientos de evento del control empezaran por dicho nombre de variable, como si fueran variables **WithEvents**.

Puede comprobar fácilmente esto si declara la variable `mWidget` como **Public** en lugar de **Private**. Cuando hace esto, `mWidget` se muestra en el Examinador de objetos como una propiedad de Form1, igual que los controles del formulario.

La diferencia entre los dos casos es que Visual Basic crea automáticamente instancias de todos los controles de un formulario cuando éste se crea, mientras que usted tiene que crear sus propias instancias de las clases cuyos eventos quiere controlar y asignar a las variables **WithEvents** referencias a dichos objetos.

Para obtener más información Puede agregar eventos propios a los formularios, como se describe en "Agregar un evento a un formulario".

Agregar un evento a un formulario

El siguiente procedimiento paso a paso muestra cómo se puede crear eventos personalizados para los formularios. Para ejecutar este ejemplo, abra un nuevo proyecto Exe estándar y haga lo siguiente:

Para agregar un evento a Form1

1. En el menú **Proyecto**, seleccione **Agregar módulo de clase** para agregar un módulo de clase al proyecto. Coloque el código siguiente en la sección **Declaraciones** de Class1:

```
Public Property Get Form1() As Form1
   Set Form1 = mForm1
End Property

Public Property Set Form1(ByVal NewForm1 As Form1)
   Set mForm1 = NewForm1
End Property
```

Si utiliza **Ver procedimiento** los procedimientos de propiedad no se pueden ver al mismo tiempo. Haga clic en el botón **Ver módulo completo** de la esquina inferior izquierda de la ventana código para pasar a Ver módulo completo. Puede volver a Ver procedimiento si hace clic en el botón **Ver procedimiento** que está al lado. Mueva el *mouse* (ratón) sobre los botones para identificarlos.

2. Agregue el código siguiente a la sección **Declaraciones** de Form1:

```
Event Gong
Private mc1 As Class1
```

   Ahora que ha creado Class1, puede crear una variable del tipo Class1. Este procedimiento cambia varias veces entre Form1 y Class1, puesto que un paso en un módulo requiere agregar antes código en el otro.

3. Vuelva a Class1 y agregue el código siguiente a la sección Declaraciones.

```
Private WithEvents mForm1 As Form1
```

   Como se describe en "Agregar eventos a las clases", la palabra clave **WithEvents** significa que esta instancia de Form1 está asociada *con eventos*. Observe que este paso no era posible hasta la creación del evento Gong.

4. En la lista desplegable de la izquierda (**Objeto**) de la ventana **Código** de Class1, seleccione **mForm1** para obtener el procedimiento de evento Gong. Agregue el código siguiente al procedimiento de evento:

```
Private Sub mForm1_Gong()
   MsgBox "¡Gong!"
End Sub
```

5. Vuelva a Form1. En la lista desplegable **Objeto**, seleccione **Form**. En la lista desplegable de la derecha (**Procedimiento**), seleccione **Load**. Agregue el código siguiente al procedimiento de evento:

```
Private Sub Form_Load()
   Set mc1 = New Class1
   Set mc1.Form1 = Me
End Sub
```

   La primera línea crea un objeto Class1 y la segunda asigna a la propiedad Form1 (creada en el paso 1) una referencia a Form1, es decir, **Me**: cuando se encuentra en la ventana **Código** de Form1, **Me** hace referencia a Form1; cuando se encuentra en la ventana **Código** de Class1, **Me** hace referencia a Class1.

6. Coloque tres cuadros de texto en Form1. Utilice las listas desplegables **Objeto** y **Procedimiento** para seleccionar el procedimiento de evento **Change** de cada control y coloque esta línea de código en cada uno de ellos:

```
Private Sub Text1_Change()
   RaiseEvent Gong
End Sub
```

   Cada vez que cambie el contenido de un cuadro de texto, se desencadena el evento Gong del formulario.

7. Presione F5 para ejecutar el proyecto. Cada vez que escriba un carácter en uno de los cuadros de texto, el cuadro de mensajes emitirá un pitido. Es un poco molesto, pero le muestra cómo puede agregar un evento a un formulario y, de esta forma, obtener notificaciones desde varios controles.

Como se ha mostrado en "Declarar y desencadenar eventos", puede agregar argumentos a los eventos. Por ejemplo, puede pasar el nombre del control o, mejor aún, una referencia al control, al receptor del evento.

Resumen de cómo declarar, desencadenar y controlar eventos

Para agregar un evento a una clase y usar después el evento:

- En la sección Declaraciones del módulo de clase que define la clase, utilice la instrucción **Event** para declarar el evento con los argumentos que desee que tenga. Los eventos son siempre **Public**.

  Nota Los eventos no pueden tener argumentos con nombre, argumentos opcionales ni argumentos ParamArray. Los eventos no tienen valores devueltos.

- En los lugares apropiados del código del módulo de clase, utilice la instrucción **RaiseEvent** para desencadenar el evento, junto con los argumentos necesarios.

- En la sección Declaraciones del módulo que va a controlar el evento, agregue una variable del tipo de clase, mediante la palabra clave **WithEvents**. Tiene que ser una variable a nivel de módulo.

- En la lista desplegable de la izquierda de la ventana Código, seleccione la variable declarada **WithEvents**.

- En la lista desplegable de la derecha, seleccione el evento que quiere controlar. Puede declarar múltiples eventos para una clase.

- Agregue código al procedimiento de evento mediante los argumentos suministrados.

Para obtener más información En "Agregar eventos a una clase" se proporcionan detalles y ejemplos de código.

Crear clases de reconocimiento de datos

Si ha leído el material anterior sobre creación de clases, ya sabrá que una clase es un objeto que encapsula datos y código, y que las propiedades de una clase son los datos que describen un objeto. También sabrá que puede usar procedimientos de propiedades o propiedades públicas para exponer los datos representados por dichas propiedades.

Por el momento todos los ejemplos que hemos visto se han referido a datos transitorios, es decir, datos que se crean y consumen en tiempo de ejecución. En el caso de muchos programas, puede que esto sea todo lo necesita, pero ¿qué ocurre si necesita almacenar datos entre sesiones, o usar datos que ya existen fuera de su programa? Para trabajar con orígenes externos de datos, debe hacer una clase de reconocimiento de datos.

Las clases de reconocimiento de datos se pueden dividir en dos categorías: receptoras de datos y orígenes de datos. Los módulos de clase tienen dos propiedades en tiempo de diseño, **DataBindingBehavior** y **DataSourceBehavior**, que determinan cómo interactuará una clase con los datos externos. El objeto **BindingCollection** se usa para enlazar las clases de reconocimiento de datos con controles o entre sí.

Orígenes de datos

Un origen de datos es una clase que proporciona datos de una fuente externa para que los reciban otros objetos. Un control Data es, en realidad, una instancia de una clase que es un origen de datos, aunque las clases que se han establecido para actuar como orígenes de datos pueden ser mucho más potentes que un control Data. A diferencia del control Data, una clase de reconocimiento de datos no necesita tener una representación visual y no está limitada a una interfaz de datos en particular, como Data Access Objects (DAO) o Remote Data Objects (RDO). De hecho, una clase de reconocimiento de datos puede actuar como origen de datos para cualquier tipo de datos, incluyendo los orígenes ODBC tradicionales, ActiveX Data Objects (ADO) o cualquier proveedor de bases de datos (DB) OLE.

La propiedad **DataSourceBehavior** determina si una clase puede actuar como origen de datos. Al establecer **DataSourceBehavior** a 1 (**vbDataSource**), su clase puede actuar como origen de datos para otros objetos.

Receptores de datos

Expuesto de una manera simple, un receptor de datos es una clase que se puede enlazar a un origen externo de datos, parecido a como un control TextBox se puede enlazar a un control Data. En versiones anteriores de Visual Basic, los controles eran los únicos objetos que se podían enlazar a un origen de datos. Las clases de reconocimiento de datos establecidas como receptores de objetos le permiten enlazar cualquier objeto a cualquier otro objeto creado desde una clase que se haya establecido como origen de datos.

La propiedad **DataBindingBehavior** permite a una clase enlazar a datos externos. Estableciendo esta propiedad a 1 (**vbSimpleBound**), un objeto creado desde su clase estará enlazado a un único campo de datos en un origen externo de datos. Estableciendo **DataBindingBehavior** a 2 (**vbComplexBound**), su clase estará enlazada a una fila de datos en un origen externo de datos. Visto de este modo, si sus objetos fueran controles, un control TextBox estaría enlazado de forma sencilla, mientras que un control de cuadrícula lo estaría de forma compleja.

Objeto BindingCollection

Del mismo modo que enlazaría un control a una base de datos mediante un control Data, las clases de reconocimiento de datos necesitan un objeto central que las enlace. Ese objeto es

BindingCollection. Así como suena, **BindingCollection** es una colección de enlaces entre un origen de datos y uno o más receptores de datos.

Para usar el objeto **BindingCollection** debe agregar primero una referencia a Microsoft Data Binding Collection seleccionándola en el cuadro de diálogo Referencias, disponible en el menú Proyecto. Como con cualquier otro objeto, necesitará crear una instancia del objeto **BindingCollection** en tiempo de ejecución.

La propiedad DataSource del objeto **BindingCollection** se usa para especificar el objeto que proporcionará los datos. Este objeto debe ser una clase o **UserControl** con su propiedad **DataSourceBehavior** establecida a **vbDataSource**.

Una vez que se ha instanciado **BindingCollection** y se ha establecido su DataSource, puede usar el método Add para definir las relaciones del enlace. El método Add toma tres argumentos requeridos: el nombre del objeto receptor, la propiedad de ese objeto que se va a enlazar con el origen y el campo del origen que se enlazará a la propiedad. Puede agregar múltiples enlaces a **BindingCollection** repitiendo el método Add; puede usar el método Remove para eliminar un enlace.

Para obtener más información Para ejemplos paso a paso sobre la creación de clases de reconocimiento de datos, vea "Crear un origen de datos " y "Crear un receptor de datos".

Crear un origen de datos

En esta sección, examinaremos paso a paso el proceso de creación de una clase de reconocimiento de datos que actúe como origen de datos. Este ejemplo enlazará un control TextBox a nuestra clase de origen de datos para mostrar los datos. La siguiente sección, "Crear un receptor de datos", muestra cómo enlazar nuestra clase de origen de datos a una clase de receptor de datos.

Los ejemplos de código de esta sección se han tomado del ejemplo Clases de reconocimiento de datos (Dataware.vbp). Encontrará esta aplicación en el directorio **Samples**.

La creación de un origen de datos es un proceso con dos pasos. En el primer paso, crearemos la clase de origen de datos; en el segundo paso, la conectaremos a un control TextBox para mostrar el resultado.

Crear la clase de origen

El primer paso para crear una clase de origen consiste en definir una nueva clase y darle las propiedades y métodos necesarios para proporcionar datos:

1. Abra un nuevo proyecto EXE estándar e inserte un módulo de clase seleccionando **Agregar módulo de clase** del menú **Proyecto**.

2. En la ventana **Propiedades**, establezca las propiedades de la clase del siguiente modo:

Propiedad	Valor
Name	MySource
DataSourceBehavior	VbDataSource

Cuando **DataSourceBehavior** está establecida a vbDataSource, se agrega un nuevo procedimiento Sub GetDataMember al módulo de clase. Puede ver esto seleccionando Clase en la lista Objeto del Editor de código y seleccionando después la lista Evento.

3. Seleccione **Referencias** en el menú **Proyecto** y agregue una referencia a la Biblioteca Microsoft ActiveX Data Objects 2.0.

4. Agregue lo siguiente a la sección **Declaraciones** del módulo de clase:

```
Option Explicit
Private rs As ADODB.Recordset
```

Esto declara una variable de objeto para el objeto ADO Recordset.

5. Agregue el siguiente código al procedimiento de evento Initialize del módulo de clase:

```
Private Sub Class_Initialize()
   Dim strPath As String, strName As String
   Dim i As Integer

   ' Crea una instancia del Recordset.
   Set rs = New ADODB.Recordset

   ' Establece las propiedades del Recordset.
   With rs
      .Fields.Append "DirID", adInteger
      .Fields.Append "Directory", adBSTR, 255
      .CursorType = adOpenStatic
      .LockType = adLockOptimistic
      .Open
   End With

   ' Hace un bucle por los directorios y llena
   ' el Recordset.
   strPath = "C:\"
   strName = Dir(strPath, vbDirectory)
   i = 0
   Do While strName <> ""
      If strName <> "." And strName <> ".." Then
         If (GetAttr(strPath & strName) And _
            vbDirectory) = vbDirectory Then
            i = i + 1
            With rs
               .AddNew
               .Fields.Item("DirID") = i
               .Fields.Item("Directory") = strName
               .Update
            End With
         End If
      End If
      strName = Dir
   Loop

   ' Vuelve al primer registro.
   rs.MoveFirst
End Sub
```

En este ejemplo estamos creando un objeto ADO Recordset sobre la marcha y lo estamos llenando con una lista de directorios. Como alternativa, podría usar un conjunto de registros existente asignando la propiedad Connect del ADO Recordset en el evento Initialize.

6. Seleccione **Clase** de la lista **Objeto** en el **Editor de código**, a continuación seleccione GetDataMember de la lista **Evento**. Agregue el siguiente código al procedimiento GetDataMember Sub:

```
Private Sub Class_GetDataMember(DataMember As String, Data As
Object)
   ' Asigna el Recordset al objeto Data.
   Set Data = rs
End Sub
```

El procedimiento GetDataMember establece el origen de los datos para la clase. Su clase de origen de datos puede proporcionar múltiples orígenes de datos agregando una instrucción Select Case al procedimiento GetDataMember y pasando un nombre de origen en el argumento DataMember.

7. Agregue un nuevo procedimiento Sub para proporcionar un método público para hacer un bucle por el Recordset:

```
Public Sub Cycle()
   ' Se mueve en ciclos por el Recordset.
   rs.MoveNext
   If rs.EOF = True Then
      rs.MoveFirst
   End If
End Sub
```

Para desplazarnos por el conjunto de registros, necesitamos exponer los métodos de navegación para nuestra clase. Por razones de simplicidad, este ejemplo solamente puede hacer el bucle hacia delante a través del conjunto de registros. Para que la clase sea más útil, puede que le interese exponer métodos como MoveFirst, MoveNext, Add, y Delete.

Usar la clase de origen

Una vez definida la clase de origen, podemos hacer algo útil con ella. En este ejemplo, la enlazaremos a un control TextBox para poder ver el resultado; también usaremos un control CommandButton para ejecutar nuestro método Cycle.

1. Seleccione Form1 y agregue un control TextBox y un control CommandButton al formulario.

2. En la ventana **Propiedades**, establezca las propiedades de TextBox del siguiente modo:

Propiedad	Valor
Name	TxtConsumer
Text	(en blanco)

3. En la ventana **Propiedades**, establezca las propiedades de CommandButton del siguiente modo:

Propiedad	Valor
Name	cmdCycle
Caption	Cycle

4.

5. Seleccione **Referencias** en el menú **Proyecto** y agregue una referencia a Microsoft Data Binding Collection.

   El objeto **DataBinding** proporcionado por Data Binding Collection es un "pegamento" que enlaza un origen de datos a un receptor de datos:

6. Agregue lo siguiente a la sección **Declaraciones** del módulo de clase:

```
Option Explicit
Private objSource As MySource
Private objBindingCollection As BindingCollection
```

   Necesitamos declarar nuestra clase de origen (MySource) y el objeto **BindingCollection** usando un enlace anterior.

7. Agregue el siguiente código al procedimiento de evento **Form_Load**:

```
Private Sub Form_Load()
   Set objSource = New MySource
   Set objBindingCollection = New BindingCollection

   ' Asigna la clase de origen a la propiedad
   ' DataSource de Binding Collection.
   Set objBindingCollection.DataSource = objSource
   ' Agrega un enlace.
   ObjBindingCollection.Add txtConsumer, "Text", "Directory"
```

   En el evento Load, creamos instancias de la clase de origen y del objeto **BindingCollection**. A continuación asignamos el objeto de origen a la propiedad DataSource de **BindingCollection**. Por último, agregamos un enlace especificando el nombre del receptor (txtConsumer), la propiedad del receptor al que se va a enlazar (la propiedad **Text**) y la propiedad Field del objeto de origen al que estamos enlazando (Directory).

8. Agregue el siguiente código al procedimiento de evento Click de cmdCycle:

```
Private cmdCycle_Click()
   ' Llama al método Cycle del origen de datos.
   ObjSource.Cycle
End Sub
```

   Esto ejecutará el método **Cycle** de nuestra clase de origen.

9. Presione F5 para ejecutar el proyecto.

   Al presionar el botón Cycle, los nombres de los directorios del conjunto de registros creados en la clase de origen aparecerán en TextBox. Enhorabuena, acaba de enlazar un control a una clase de origen de datos sin usar un control Data.

10. Guarde el proyecto. Cuando le pidan los nombres de archivo, use los siguientes nombres.

    Guarde la clase de origen como "MySource.cls".

Guarde el formulario como "Dataform.frm".

Guarde el proyecto como "Dataware.vbp".

Estos archivos se usarán más adelante en "Crear un receptor de datos".

En la siguiente sección "Crear un receptor de datos", examinaremos el proceso de creación de una clase de reconocimiento de datos que actúe como receptor de datos.

Crear un receptor de datos

En esta sección examinaremos paso a paso el proceso de creación de una clase de reconocimiento de datos que actúe como receptor de datos. La sección anterior, "Crear un origen de datos", mostraba cómo crear un origen de datos al que se pudiera enlazar un receptor de datos. Este ejemplo muestra cómo crear una clase para recepción de datos y enlazarla al origen de datos creado en la sección anterior.

Los ejemplos de código de esta sección se han extraído del ejemplo Clases de reconocimiento de datos (Dataware.vbp). Encontrará esta aplicación en el directorio **Samples**.

Enlazar un objeto receptor de datos a un objeto origen de datos

Este ejemplo muestra cómo crear una clase receptora de datos y cómo enlazarla a una clase origen de datos. El ejemplo usa la clase MySource creada en el tema "Crear un origen de datos".

1. Abra el proyecto Dataware.vbp. Seleccione Abrir proyecto del menú Archivo.

   Nota Si no ha completado el ejemplo "Crear un origen de datos", este proyecto no existirá. También puede encontrar una versión completa del proyecto Dataware.vbp en el directorio **Samples**.

2. Inserte un nuevo módulo de clase seleccionando **Agregar módulo de clase** del menú **Proyecto**.

3. En la ventana **Propiedades**, establezca las propiedades de la nueva clase del siguiente modo:

Propiedad	Valor
Name	MyConsumer
DataBindingBehavior	VbSimpleBound

4. Agregue lo siguiente a la sección **Declaraciones** del módulo de clase:

```
Option Explicit
Private mDirectory As String
```

5. Agregue una pareja de procedimientos Property Get / Property Let para una propiedad pública DirName:

```
Public Property Get DirName() As String
   DirName = mDirectory
End Property

Public Property Let DirName(mNewDir As String)
```

```
   mDirectory = mNewDir
   ' Muestra el nuevo valor en la ventana Inmediato.
   Debug.Print mDirectory
End Property
```

Puesto que MySource es una clase no visual, necesitamos usar una instrucción Debug.Print en el procedimiento Property Let para comprobar que está recuperando nuevos valores del origen de datos.

6. Seleccione Form1 y agregue el siguiente código a la sección **Declaraciones**:

```
Option Explicit
Private objSource As MySource
Private objBindingCollection As BindingCollection
Private objConsumer As MyConsumer
```

La nueva línea de código agrega una referencia a nuestra clase receptora.

7. Agregue el siguiente código al procedimiento de evento Form Load:

```
Private Sub Form_Load()
   Set objSource = New MySource
   Set objBindingCollection = New BindingCollection
   Set objConsumer = New MyConsumer

   ' Asigna la clase de origen a la propiedad
   ' DataSource de la colección Binding.
   Set objBindingCollection.DataSource = objSource
   ' Agrega un enlace.
   objBindingCollection.Add txtConsumer, "Text", "Directory"
   objBindingCollection.Add objConsumer, "DirName", "Directory"
```

El nuevo código crea una instancia de la clase receptora y se la agrega a la colección **Binding**, enlazando la propiedad **DirName** del receptor al campo Directory del origen de datos.

8. Presione F5 para ejecutar el proyecto. Compruebe que la ventana **Inmediato** es visible.

Al presionar el botón Cycle, los nombres de directorio proporcionados por MySource aparecerán tanto en TextBox como en la ventana Inmediato, probando que MyConsumer está ligado a MySource.

Asignar nombres a propiedades, métodos y eventos

Las propiedades, métodos y eventos que se agrega a un módulo de clase definen la interfaz que se va a usar para manipular los objetos creados a partir de la clase. Cuando asigne un nombre a estos elementos y a sus argumentos, puede encontrar útil seguir unas cuantas reglas sencillas.

- Utilice palabras completas siempre que sea posible, por ejemplo SpellCheck. Las abreviaturas pueden provocar confusión. Si las palabras completas son demasiado largas, utilice las primeras sílabas completas.

- Utilice mayúsculas y minúsculas para los identificadores, con una mayúscula en la primera letra de cada palabra o de cada sílaba; por ejemplo, ShortcutMenus o AsyncReadComplete.

- Utilice el plural correcto en los nombres de las clases de colección, por ejemplo Worksheets, Forms o Widgets. Si la colección contiene objetos cuyo nombre termina en "s", agregue la palabra "Collection", como por ejemplo SeriesCollection.

- Utilice de forma coherente el orden verbo/objeto y objeto/verbo en los nombres de los métodos. Es decir, utilice InsertWidget, InsertSprocket, etc. o escriba siempre primero el objeto, como en WidgetInsert y SprocketInsert.

Nota Aunque se puede usar un carácter de subrayado en un nombre de propiedad, en un nombre de evento generaría un error. El carácter de subrayado se utiliza en procedimientos Event para separar el nombre de evento del nombre de objeto. Por este motivo, es mejor evitar el uso del carácter subrayado al dar nombre a propiedades, métodos y eventos.

Una de las mayores ventajas de la programación con objetos es la reutilización del código. Si sigue las reglas anteriores, que forman parte de las recomendaciones de las interfaces ActiveX, le será fácil recordar los nombres y el propósito de las propiedades, los métodos y los eventos.

Para obtener más información Si tiene la Edición Profesional o la Edición Empresarial de Visual Basic, vea la lista expandida de "¿Qué hay en un nombre?" en el tema "Principios generales del diseño de componentes" de la sección *Crear componentes ActiveX* de la *Guía de herramientas componentes.*

Polimorfismo

Polimorfismo significa que muchas clases pueden proporcionar la misma propiedad o el mismo método y que el que llama no tiene por qué saber la clase a la que pertenece el objeto antes de llamar a la propiedad o al método.

Por ejemplo, una clase Pulga y una clase Tiranosaurio podrían tener las dos un método Morder. El polimorfismo significa que puede invocar Morder sin saber si el objeto es una Pulga o un Tiranosaurio, aunque seguramente lo descubrirá después.

Los siguientes temas describen la implementación del polimorfismo en Visual Basic y cómo puede usarlo en sus programas.

- Cómo proporciona Visual Basic el polimorfismo La mayoría de los lenguajes orientados a objetos proporcionan polimorfismo mediante la herencia; Visual Basic utiliza la técnica de interfaz múltiple del Modelo de objetos componentes (COM).

- Crear e implementar una interfaz Un amplio ejemplo de código muestra cómo puede crear una interfaz abstracta Animal e implementarla para las clases Tiranosaurio y Pulga.

- Implementar propiedades Las interfaces que implementa pueden tener tanto propiedades como métodos, aunque hay algunas diferencias en la forma de implementar las propiedades.

- Breve descripción de objetos e interfaces Explica los términos *objeto* e *interfaz*, presenta el concepto de consultar una interfaz y describe otros orígenes de interfaces para implementar.

- Múltiples facetas (e interfaces) de la reutilización del código Además de implementar interfaces abstractas, puede volver a usar el código si implementa la interfaz de una clase normal y después delega de forma selectiva en una instancia oculta de la clase.

Para obtener más información En la Edición Profesional y en la Edición Empresarial de Visual Basic el polimorfismo se convierte en un poderoso mecanismo para la evolución de los sistemas de componentes software. Esto se describe en "Principios generales del diseño de componentes", en la sección *Crear componentes ActiveX* de la *Guía de herramientas componentes*.

Cómo proporciona Visual Basic el polimorfismo

La mayoría de los sistemas de programación orientada a objetos proporcionan polimorfismo mediante la *herencia*. Es decir, las hipotéticas clases Pulga y Tiranosaurio pueden heredar características de una clase Animal. Cada clase invalidaría el método Morder de la clase Animal para aportar sus propias características de mordedura.

El polimorfismo proviene del hecho de que puede llamar al método Morder de un objeto que pertenezca a cualquier clase derivada de Animal sin conocer la clase a la que pertenece el objeto.

Proporcionar polimorfismo mediante interfaces

Visual Basic no utiliza la herencia para proporcionar polimorfismo. Visual Basic proporciona el polimorfismo mediante múltiples *interfaces* ActiveX. En el Modelo de objetos componentes (COM) que forma la infraestructura de la especificación ActiveX, las múltiples interfaces

permiten que los sistemas de componentes software evolucionen sin afectar al código existente.

Una *interfaz* es un conjunto de propiedades y métodos relacionados. Gran parte de la especificación ActiveX se ocupa de la implementación de interfaces estándar para obtener servicios del sistema o para proporcionar funcionalidad a otros programas.

En Visual Basic podría crear una interfaz Animal e implementarla en las clases Pulga y Tiranosaurio. Después podría invocar el método Morder de cualquiera de los dos tipos de objeto sin conocer su tipo.

Polimorfismo y rendimiento

El polimorfismo es importante por razones de rendimiento. Para comprobarlo, observe la función siguiente:

```
Public Sub GetFood(ByVal Critter As Object, _
ByVal Food As Object)
   Dim dblDistance As Double
   ' Código para calcular la distancia a la comida
   '(omitido).
   Critter.Mover dblDistance   ' Enlace en tiempo de
                              ' compilación
   Critter.Morder Food         ' Enlace en tiempo de
                              ' compilación
End Sub
```

Los métodos Mover y Morder son *enlaces en tiempo de compilación* con `Critter`. El enlace en tiempo de compilación tiene lugar cuando Visual Basic no puede determinar en tiempo de compilación el tipo de objeto que contendrá una variable. En este ejemplo, el argumento Critter está declarado como **As Object**, de forma que en tiempo de ejecución pueda contener una referencia a cualquier tipo de objeto, como un Coche o una Roca.

Como no puede saber cuál será el objeto, Visual Basic compila una parte de código adicional para preguntar al objeto si acepta el método llamado. Si el objeto acepta el método, dicho código adicional lo invoca; si no, el código adicional desencadena un error. Todas las llamadas a métodos o a propiedades incurren en esta carga adicional.

Por el contrario, las interfaces permiten los *enlaces en tiempo de diseño*. Cuando Visual Basic conoce en tiempo de compilación la interfaz que se va a llamar, puede comprobar en la biblioteca de tipos si dicha interfaz acepta el método. Después, Visual Basic puede compilar un salto directo al método mediante una tabla virtual de funciones (vtable). Esto es muchas veces más rápido que el enlace en tiempo de compilación.

Ahora suponga que los métodos Mover y Morder pertenecen a la interfaz Animal y que todas las clases de animales implementan esta interfaz. Ahora podría declarar el argumento Critter como As Animal y los métodos Mover y Morder se enlazarán en tiempo de diseño:

```
Public Sub GetFood(ByVal Critter As Animal, _
ByVal Food As Object)
   Dim dblDistance As Double
   ' Código para calcular la distancia a la comida
   '(omitido).
   Critter.Mover dblDistance   ' Enlace en tiempo de
                              ' diseño.
   Critter.Morder Food         ' Enlace en tiempo de
                              ' diseño.
End Sub
```

Para obtener más información En "Crear e implementar una interfaz" se crea una interfaz Animal y se implementa en las clases Pulga y Tiranosaurio.

Crear e implementar una interfaz

Como se ha explicado en "Cómo proporciona Visual Basic el polimorfismo", una interfaz es un conjunto de propiedades y métodos. En el siguiente ejemplo de código se creará una interfaz Animal y se implementará en dos clases, Pulga y Tiranosaurio.

Puede crear la interfaz Animal si agrega un módulo de clase al proyecto, le asigna el nombre Animal e inserta el código siguiente:

```
Public Sub Mover(ByVal Distance As Double)

End Sub

Public Sub Morder(ByVal What As Object)

End Sub
```

Observe que estos métodos no contienen código. Animal es una *clase abstracta,* que no contiene código de implementación. El propósito de una clase abstracta no es crear objetos, sino proporcionar una plantilla de interfaz que puede agregar a otras clases. No obstante, como verá, algunas veces resulta útil implementar la interfaz de una clase no abstracta; esto se describe más adelante en este tema.

Nota Hablando con propiedad, una clase abstracta es una clase a partir de la cual no puede crear objetos. Siempre puede crear objetos a partir de clases de Visual Basic, incluso aunque no contengan código; así que no se pueden considerar realmente abstractas.

Ahora puede agregar dos módulos de clase más. Asigne a uno de ellos el nombre Pulga y al otro Tiranosaurio. Para implementar la interfaz Animal en la clase Pulga, utilice la instrucción **Implements**:

```
Option Explicit
Implements Animal
```

En cuanto agregue esta línea de código, puede hacer clic en la lista desplegable (**Objeto**) de la izquierda de la ventana **Código**. Una de las entradas será Animal. Cuando la seleccione, la lista desplegable (**Procedimiento**) de la derecha mostrará los métodos de la interfaz Animal.

Seleccione los métodos de uno en uno para crear plantillas de procedimiento vacías en todos los métodos. Las plantillas tendrán los argumentos y los tipos de datos correctos, como se han definido en la clase Animal. Cada nombre de procedimiento tendrá el prefijo `Animal_` para identificar la interfaz.

Importante Una interfaz es como un contrato. Al implementar la interfaz, una clase se compromete a responder cuando se invoque cualquier propiedad o método de la interfaz. Por tanto, debe implementar *todas* las propiedades y los métodos de una interfaz.

Ahora puede agregar el código siguiente a la clase Pulga:

```
Private Sub Animal_Mover(ByVal Distance As Double)
   ' (Se omite el código del salto.)
   Debug.Print "Pulga movida"
End Sub
```

```
Private Sub Animal_Morder(ByVal What As Object)
   ' (Se omite el código del picotazo.)
   Debug.Print "La pulga mordió un " & TypeName(What)
End Sub
```

Puede que se pregunte por qué los procedimientos se declaran **Private**. Si fueran **Public**, los procedimientos Animal_Mover y Animal_Morder formarían parte de la interfaz Pulga y tendría el mismo enlace original, al declarar el argumento Critter **As Object** para que pueda contener una Pulga o un Tiranosaurio.

Interfaces múltiples

La clase Pulga tiene ahora dos interfaces: la interfaz Animal recién implementada, que tiene dos miembros, y la interfaz Pulga predeterminada, que no tiene miembros. Más adelante en este ejemplo se agregará un miembro a una de las interfaces predeterminadas.

Puede implementar la interfaz Animal de forma similar para la clase Tiranosaurio:

```
Option Explicit
Implements Animal

Private Sub Animal_Mover(ByVal Distance As Double)
   ' (Se omite el código de la zancada.)
   Debug.Print "Tiranosaurio movido"
End Sub

Private Sub Animal_Morder(ByVal What As Object)
   ' (Se omite el código de la dentellada.)
   Debug.Print "El Tiranosaurio ha mordido un " _
               & TypeName(What)
End Sub
```

Pruebas con el Tiranosaurio y la Pulga

Agregue el código siguiente al evento Load de Form1:

```
Private Sub Form_Load()
   Dim fl As Pulga
   Dim ty As Tiranosaurio
   Dim anim As Animal

   Set fl = New Pulga
   Set ty = New Tiranosaurio
   ' Primero le toca a la pulga.
   Set anim = fl
   Call anim.Morder(ty)   ' La pulga muerde al
                        ' dinosaurio.
   ' Ahora es el turno del Tiranosaurio.
   Set anim = ty
   Call anim.Morder(fl)   ' El dinosaurio muerde a
                        ' la pulga.
End Sub
```

Presione F8 para recorrer el código. Observe los mensajes de la ventana Inmediato. Cuando la variable `anim` contiene una referencia a Pulga, se invoca la implementación de Morder de Pulga y de la misma forma para Tiranosaurio.

La variable `anim` puede contener una referencia a cualquier objeto que implemente la interfaz Animal. De hecho, *sólo* puede contener referencias a dichos objetos. Si intenta asignar un objeto **Form** o **PictureBox** a `anim`, se producirá un error.

El método Morder se enlaza en tiempo de diseño cuando lo llama mediante `anim`, puesto que Visual Basic sabe en tiempo de compilación que cualquier objeto asignado a `anim` tendrá un método Morder.

Paso de Tiranosaurios y Pulgas a procedimientos

Recuerde el procedimiento GetFood (Alimentarse) de "¿Cómo proporciona Visual Basic el polimorfismo?". Puede agregar la *segunda* versión del procedimiento GetFood (el que ilustra el polimorfismo) a Form1 y sustituir el código del evento Load por el siguiente:

```
Private Sub Form_Load()
   Dim fl As Pulga
   Dim ty As Tiranosaurio

   Set fl = New Pulga
   Set ty = New Tiranosaurio
   'La pulga se come al dinosaurio.
   Call GetFood(fl, ty)
   ' Y viceversa.
   Call GetFood(ty, fl)
End Sub
```

El repaso de este código muestra cómo una referencia de objeto pasada a un argumento de otro tipo de interfaz se convierte en referencia a la segunda interfaz (en este caso, Animal). Lo que ocurre es que Visual Basic consulta el objeto para saber si acepta la segunda interfaz. Si el objeto lo acepta, devuelve una referencia a la interfaz y Visual Basic coloca esa referencia en la variable del argumento. Si el objeto no acepta la segunda interfaz, se produce un error.

Implementar métodos que devuelven valores

Suponga que el método Mover devolviera un valor. Después de todo, sabe cuánto quiere que se mueva el Animal, pero un espécimen concreto podría no ser capaz de llegar tan lejos. Podría estar viejo y decrépito o podría haber un muro en su camino. Puede usar el valor devuelto por el método Mover para saber indicarle el movimiento real del Animal.

```
Public Function Mover(ByVal Distance As Double) _
As Double

End Function
```

Cuando implemente este método en la clase Tiranosaurio, asigne el valor devuelto al nombre del procedimiento, como lo haría en cualquier otro procedimiento **Function**:

```
Private Function Animal_Mover(ByVal Distance _
As Double) As Double
   Dim dblDistanceMoved As Double
   ' Código para calcular la distancia (basado en
   ' edad, salud y obstáculos) omitido.
   ' Este ejemplo asume que el resultado se encuentra
   ' en la variable dblDistanceMoved.
   Debug.Print "Tiranosaurio movido ";
   dblDistanceMoved
   Animal_Mover = dblDistanceMoved
End Function
```

Para asignar el valor devuelto, utilice el nombre completo del procedimiento, incluido el prefijo de la interfaz.

Para obtener más información Las interfaces que implemente pueden tener propiedades y métodos. En "Implementar propiedades" se describen algunas diferencias en la forma de implementar las propiedades.

Implementar propiedades

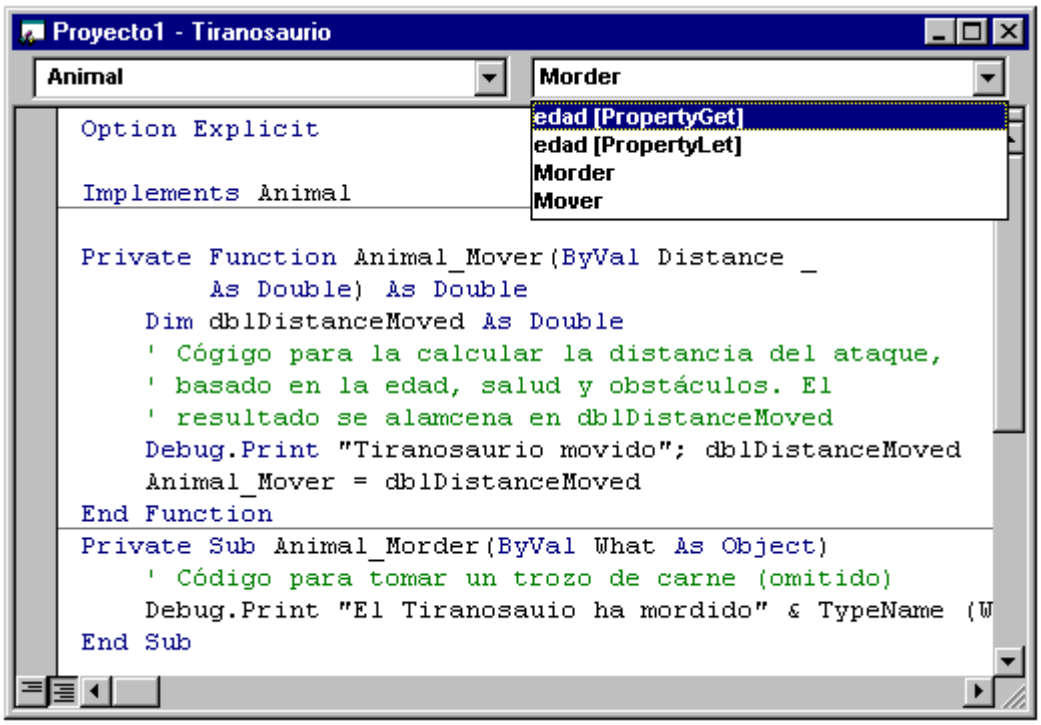
En este tema continúa el ejemplo de código empezado en "Crear e implementar una interfaz" y se agregan propiedades a la interfaz Animal implementada en las clases Pulga y Tiranosaurio. Puede que le interese leer dicho tema antes de empezar éste.

Suponga que incluye una propiedad Edad en la clase Animal mediante una variable **Public** en la sección Declaraciones:

```
Option Explicit
Public Edad As Double
```

Las listas desplegables **Procedimiento** de los módulos de código de las clases Tiranosaurio y Pulga contienen ahora procedimientos de propiedad para implementar la propiedad Edad, como se muestra en la figura 9.10.

Figura 9.10 Implementación de los procedimientos de propiedad



Esta figura ilustra una característica mencionada en "Agregar propiedades a una clase", anteriormente en este capítulo. El empleo de una variable pública para implementar una propiedad es estrictamente una comodidad para el programador. Internamente, Visual Basic implementa las propiedades como parejas de procedimientos de propiedad.

Debe implementar ambos procedimientos. Los procedimientos de propiedad se implementan fácilmente si almacena el valor en un miembro privado de datos, como se puede ver a continuación:

```
Private mdblEdad As Double

Private Property Get Animal_Edad() As Double
```

```
   Animal_Edad = mdblEdad
End Property

Private Property Let Animal_Edad(ByVal RHS As Double)
   mdblEdad = RHS
End Property
```

El miembro privado de datos es un detalle de la implementación, de modo que tiene que agregarlo manualmente.

Nota Cuando **Implements** proporciona la plantilla de un **Property Set** o un **Property Let**, no hay manera de determinar el nombre del último argumento, de forma que sustituye el nombre `RHS`, como se ve en el ejemplo anterior.

No hay validación de datos en una propiedad implementada como miembro público de datos, pero esto no significa que no pueda agregar código de validación en el procedimiento **Property Let** de Animal_Edad. Por ejemplo, puede que desee restringir los valores de las edades apropiadas para un Tiranosaurio o para una Pulga, respectivamente.

De hecho, esto demuestra la independencia de la interfaz y de la implementación. Siempre y cuando la interfaz coincida con la descripción de la biblioteca de tipos, la implementación puede realizarse de cualquier modo.

Antes de ir al paso siguiente, quite la implementación de la propiedad Edad para lectura y escritura de ambos módulos de clase.

Implementar una propiedad de sólo lectura

Por supuesto, la posibilidad de permitir que la edad de un animal pueda establecerse de forma arbitraria es un mal diseño del objeto. El objeto tiene que saber su edad y proporcionarla al usuario como propiedad de sólo lectura. Quite la variable pública `Edad` de la clase Animal y agregue la plantilla de la propiedad de sólo lectura Edad, de esta forma:

```
Public Property Get Edad() As Double

End Property
```

Ahora, las listas desplegables **Procedimiento** de las ventanas de código de las clases Tiranosaurio y Pulga sólo contienen una entrada, Edad [**PropertyGet**]. Podría implementarla para el objeto Tiranosaurio de la manera siguiente:

```
Private mdblBirth As Double

Private Property Get Animal_Edad() As Double
   Animal_Edad = Now - mdblBirth
End Property
```

El código anterior devuelve la edad del Tiranosaurio en días. Podría establecer `mdblBirth` en el evento Initialize de la clase Tiranosaurio, como se muestra a continuación:

```
Private Sub Class_Initialize()
   mdblBirth = Now
End Sub
```

Desde luego, podría devolver el valor de la propiedad en unidades más comunes, como años.

Para obtener más información Se han tratado las interfaces y los objetos como si fueran iguales, con el cuidado de poner las referencias a los objetos en una variable de objeto y las referencias a las interfaces en otra. En "Breve descripción de objetos e interfaces" se aclaran las diferencias.

Breve descripción de objetos e interfaces

En este tema termina el ejemplo de código empezado en "Crear e implementar una interfaz" y continuado en "Implementar propiedades". Puede que le interese ver dichos temas antes de empezar éste.

El ejemplo de código de Tiranosaurio y Pulga parece tratar con comodidad las interfaces y los objetos. Las referencias a objetos se asignan a una variable de objeto y las referencias a interfaces a otra.

De hecho, *todas las referencias son referencias de objeto*. Una referencia a una interfaz también es una referencia al objeto que implementa la interfaz. Además, un objeto puede tener múltiples interfaces y seguir siendo el mismo objeto con todas ellas.

En Visual Basic, cada clase tiene una interfaz predeterminada con el mismo nombre que la clase. Es casi lo mismo. Por convención se escribe un carácter de subrayado como prefijo del nombre de clase. El subrayado indica que esta interfaz se encuentra oculta en la biblioteca de tipos.

De esta forma, la clase Tiranosaurio tiene una interfaz predeterminada llamada _Tiranosaurio. Como esta clase también implementa la interfaz Animal, tiene una segunda interfaz llamada Animal.

Sin embargo, debajo de todas ellas el objeto sigue siendo un Tiranosaurio. Coloque un botón de comando en Form1 y agregue el código siguiente:

```
Private Sub Command1_Click()
   Dim ty As Tiranosaurio
   Dim anim As Animal

   Set ty = New Tiranosaurio
   Set anim = ty
   MsgBox TypeName(anim)
End Sub
```

Puede que espere que el cuadro de mensajes presente "Animal", pero en realidad presenta "Tiranosaurio".

Consultar las interfaces

Cuando asigna un objeto Tiranosaurio a una variable de tipo `Animal`, Visual Basic pregunta al objeto Tiranosaurio si acepta la interfaz Animal. El método utilizado recibe el nombre de QueryInterface o su abreviatura, QI; puede que algunas veces se encuentre QI utilizado como verbo. Si la respuesta es no, se produce un error.

Si la respuesta es sí, el objeto se asigna a la variable. A través de dicha variable sólo se puede tener acceso a los métodos y propiedades de la interfaz Animal.

Variables genéricas de objeto e interfaces

¿Qué ocurre si se asigna una referencia de objeto a una variable genérica de objeto, como en el código siguiente?

```
Private Sub Command1_Click()
   Dim ty As Tiranosaurio
   Dim anim As Animal
   Dim obj As Object

   Set ty = New Tiranosaurio
   Set anim = ty
```

```
      Set obj = anim
      MsgBox TypeName(obj)
   End Sub
```

El resultado vuelve a ser un objeto Tiranosaurio. Y ahora, ¿qué interfaz obtiene cuando llama a las propiedades y los métodos mediante la variable `obj`? Agregue el método siguiente a la clase siguiente Tiranosaurio:

```
   Public Sub Rugir()
      Debug.Print "Rrrrrr"
End Sub
```

El método Rugir pertenece a la interfaz predeterminada del objeto Tiranosaurio. En el código del evento Click del botón de comando, sustituya la instrucción **MsgBox** por las dos líneas de código siguientes:

```
   obj.Mover 42
   obj.Rugir
```

Cuando ejecuta el proyecto y hace clic en el botón, la ejecución se detiene en el método Rugir, con el error "El objeto no acepta esta propiedad o método". Está claro que la interfaz sigue siendo Animal.

Debe tener esto en cuenta cuando utilice variables de tipo **Object** con objetos que tienen múltiples interfaces. La interfaz a la que tiene acceso la variable es la *última interfaz asignada.* Por ejemplo:

```
   Private Sub Command1_Click()
      Dim ty As Tiranosaurio
      Dim anim As Animal
      Dim obj As Object

      Set ty = New Tiranosaurio
      Set anim = ty
      Set obj = anim
      obj.Mover 42       ' Correcto
      obj.Rugir          ' Erróneo

      Set obj = ty
      obj.Mover 42       ' Erróneo
      obj.Rugir          ' Correcto
   End Sub
```

Afortunadamente, no hay muchas razones a favor de usar los tipos de datos **Object** enlazados en tiempo de compilación, que son lentos, con los objetos que tienen múltiples interfaces. Una de las principales razones a favor del empleo de múltiples interfaces es la ventaja de disponer de enlaces en tiempo de diseño mediante el polimorfismo.

Otros orígenes de interfaces

Los módulos de clase de Visual Basic no son el único origen de interfaces implementables. Puede implementar cualquier interfaz contenida en una biblioteca de tipos si dicha interfaz acepta Automatización.

Si tiene la Edición Profesional o la Edición Empresarial de Visual Basic, puede crear sus propias bibliotecas de tipos de clases abstractas. Dichas bibliotecas de tipos se pueden usar en muchos proyectos, como se describe en el tema "Principios generales del diseño de componentes" de *Crear componentes ActiveX*, en la *Guía de herramientas componentes*.

La Edición Profesional y la Edición Empresarial también incluyen el programa MkTypLib (Generar biblioteca de tipos) en el directorio Tools. Si ha utilizado este programa en

Microsoft Visual C++, puede que le parezca una manera más apropiada para crear interfaces.

Usar interfaces en el proyecto

Para usar una interfaz en el proyecto, haga clic en **Referencias** en el menú **Proyecto** para abrir el cuadro de diálogo **Referencias**. Si registró la biblioteca de tipos, aparecerá en la lista de referencias y podrá seleccionarla. Si la biblioteca de tipos no se encuentra en la lista, puede usar el botón **Examinar** para buscarla.

En cuanto tenga una referencia a una biblioteca de tipos, puede usar **Implements** para implementar cualquiera de las interfaces de Automatización contenidas en la biblioteca de tipos.

Para obtener más información No está limitado a implementar interfaces abstractas. En "Múltiples facetas (e interfaces) de la reutilización del código" se describe cómo puede implementar una interfaz y reusar selectivamente las propiedades y los métodos de la clase que proporciona la interfaz.

Múltiples facetas (e interfaces) de la reutilización del código

Existen principalmente dos formas de reutilización del código: binario y origen. La reutilización de código binario se consigue mediante la creación y uso de un objeto, mientras que la reutilización de código fuente se consigue por herencia, lo que no se permite en Visual Basic. La reutilización de código fuente también se puede conseguir mediante la copia y modificación del código fuente, pero esta técnica no es nada nuevo y conlleva muchos problemas ya conocidos.

Visual Basic ha sido uno de los pioneros en la reutilización de código binario; los controles son el ejemplo más clásico. Puede volver a usar el código de un control si coloca una instancia del control en un formulario. Esto se conoce como relación de *contenedor* o de *posesión*; por ejemplo, el formulario *contiene* o *posee un* control **CommandButton**.

Para obtener más información Las relaciones de contenedores se describen en "Modelos de objetos", más adelante en este capítulo.

Delegar en un objeto implementado

Implements proporciona un medio eficaz para volver a usar el código. Puede implementar una clase abstracta, como se describe en "Crear e implementar una interfaz", o puede implementar la interfaz de una clase totalmente funcional. Puede crear el *objeto interno* (es decir, el objeto implementado) en el evento Initialize del *objeto externo* (el que implementa la interfaz del objeto interno).

Como se menciona en "Crear e implementar una interfaz", una interfaz es como un contrato: tiene que implementar todos los miembros de la interfaz del objeto interno en el módulo de clase del objeto externo. Sin embargo, puede ser muy selectivo en la forma en que delega en las propiedades y los métodos del objeto interno. En un método podría delegar directamente en el objeto interno, mediante el paso de argumentos sin modificar, mientras que en otro método podría ejecutar parte del código antes de llamar al objeto interno y en un tercer método podría ejecutar únicamente su propio código y pasar por alto completamente el del objeto interno.

Por ejemplo, suponga que tiene una clase HombreOrquesta y una clase Cacofonía que generan sonidos. Le gustaría agregar la funcionalidad de la clase Cacofonía a la clase HombreOrquesta y reusar parte de la implementación de los métodos de la clase Cacofonía.

```
' HombreOrquesta implementa la interfaz Cacofonía.
Implements Cacofonía
```

```
' Variable de objeto para mantener la referencia.
Private mcac As Cacofonía

Private Sub Class_Initialize()
   ' Crea el objeto.
   Set mcac = New Cacofonía
End Sub
```

Ahora puede ir a la lista desplegable **Objeto**, seleccionar **Cacofonía** y después obtener las plantillas de procedimiento para los métodos de la interfaz Cacofonía. Para implementar dichos métodos, puede delegar en el objeto Cacofonía. Por ejemplo, el método **Beep** podría tener la apariencia siguiente:

```
Private Sub Cacofonía_Beep(ByVal Frequency As Double, _
ByVal Duration As Double)
   ' Delega en el objeto interno Cacofonía.
   Call mcac.Beep(Frequency, Duration)
End Sub
```

La implementación anterior es muy sencilla. El objeto externo (HombreOrquesta) delega directamente en el interno (Cacofonía) y vuelve a usar el método Beep del objeto Cacofonía sin modificarlo. Esto está muy bien, pero es sólo el principio.

La instrucción **Implements** es una herramienta de reutilización de código muy eficaz, puesto que ofrece gran flexibilidad. Puede que decida alterar el efecto del método **Beep** de la clase HombreOrquesta. Para ello, inserte su propio código antes, o después, de la llamada al objeto interno Cacofonía:

```
Private Sub Cacofonía_Beep(ByVal Frequency As Double, _
ByVal Duration As Double)
   ' Eleva las notas una octava.
   Frequency = Frequency * 2
   ' Basado en otra propiedad de la clase
   ' HombreOrquesta, Staccato, acorta la duración de
   ' cada nota.
   If Staccato Then Duration = Duration * 7 / 8
   Call mcac.Beep(Frequency, Duration)
   ' Incluso puede llamar a otros métodos de
   ' HombreOrquesta.
   If Staccato Then Pause(Duration * 1 / 8)
End Sub
```

En algunos de los métodos, la implementación puede delegar directamente en el objeto interno Cacofonía, mientras que en otros puede interponer su propio código antes y después de delegar o incluso omitir la delegación si sólo utiliza su propio código para implementar un método.

Como la clase HombreOrquesta implementa la interfaz Cacofonía, puede usarla con cualquier aplicación musical que llame a dicha interfaz. Los detalles de la implementación están ocultos para la aplicación que llama, pero los sonidos resultantes son los diseñados por usted.

Nota COM proporciona otro mecanismo para la reutilización de código binario, llamado *agregación*. En la agregación se utiliza una interfaz completa sin modificaciones y la implementación la proporciona una instancia de la clase agregada. Visual Basic no acepta esta forma de reutilización de código.

¿No es tedioso?

La escritura de código de delegación puede resultar muy tediosa, especialmente si la mayor parte de las propiedades y los métodos del objeto externo simplemente delegan de forma directa en las propiedades y los métodos correspondientes del objeto interno.

Si tiene la Edición Profesional o la Edición Empresarial de Visual Basic, puede usar el Modelo de extensibilidad de Visual Basic para crear su propio asistente de delegación y automatizar la tarea, que es similar al Asistente de clases incluido en la Edición Profesional y en la Edición Empresarial.

Para obtener más información El uso del polimorfismo y de las interfaces múltiples en los componentes software se describe en "Principios generales del diseño de componentes" de la sección *Crear componentes ActiveX* de la *Guía de herramientas componentes*.

El uso del Modelo de extensibilidad está documentado en *Ampliación del entorno de desarrollo de Visual Basic con complementos,* en la *Guía de herramientas componentes*.

Programar con objetos propios

Puede empezar a usar objetos de forma gradual, en las tareas en que sea conveniente la combinación de código y datos. Puede usar la funcionalidad de estos objetos si declara variables de objeto, les asigna nuevos objetos e invoca las propiedades y los métodos del objeto.

A medida que agrega más objetos a los programas, empezará a ver las relaciones que existen entre ellos. Puede empezar por diseñar programas más dependientes de los objetos y sus relaciones, y puede empezar a usar técnicas más sólidas, como la creación de clases de colección personalizadas, para expresar dichas relaciones en el código.

En algún momento observará que, repentinamente, la forma de relacionar los objetos modifica la naturaleza de los programas y estará en condiciones de empezar a diseñar programas basados en objetos desde los cimientos.

Los temas siguientes le proporcionan una introducción a estos cambios evolutivos de su estilo de escritura de código. Consúltelos ahora para hacerse una idea de dónde se encuentra y vuelva a leerlos cuando empiecen a asentarse sus ideas sobre la programación orientada a objetos.

- Referencias de objeto y recuento de referencias Cuanto más utiliza los objetos más necesita administrar las referencias de objeto, de forma que su componente libere memoria y se cierre de forma ordenada cuando deba hacerlo.

- Modelos de objetos ¿Cómo se relacionan entre sí los objetos que crea a partir de las clases? Los modelos de objetos expresan las relaciones en términos de contenedores; es decir, un tipo de objeto contiene uno o más objetos de otro tipo. Se trata de un eficaz principio de organización de los programas.

- Crear clases de colección propias Expone algunos problemas del objeto **Collection** de Visual Basic y muestra cómo puede crear sus clases colección; es decir, clases que delegan en un objeto **Collection** (con la posibilidad de usar **For Each … Next**) a la vez que proporcionan más coherencia y funcionalidad.

Para obtener más información Los componentes ActiveX abren otra dimensión más en la reutilización de código y en la programación orientada a objetos. Si tiene la Edición Profesional o la Edición Empresarial de Visual Basic, puede empezar a explorar dicha dimensión en *Crear componentes ActiveX* de la *Guía de herramientas componentes*.

Referencias de objeto y recuento de referencias

La regla principal de la duración de un objeto es muy sencilla: un objeto se destruye cuando se libera la última referencia al mismo. Sin embargo, como todo en la vida, lo sencillo no siempre es fácil.

A medida que utiliza más objetos y va manteniendo más variables que contienen referencias a esos objetos, puede que se encuentre en la situación de parecerle imposible librarse de sus objetos cuando quiere.

En algún momento sucederá que Visual Basic tiene que efectuar el seguimiento de las referencias de objetos; de otro modo ¿cómo podría saber cuándo se libera la última referencia a un objeto? Puede que en ese momento piense que la depuración sería mucho más fácil si pudiera tener acceso a los recuentos de referencias de Visual Basic.

Desgraciadamente, no es cierto. Para que el uso de los objetos sea más eficaz, el Modelo de objetos componentes (COM) especifica una serie de accesos directos complejos para las

reglas del recuento de referencias. El resultado es que no podría fiarse del recuento de referencias, incluso si tuviera acceso a él.

De acuerdo con las reglas de COM, la única información en la que puede confiar es *si el recuento de referencias es o no cero*. Puede saber cuándo el recuento de referencias llega a cero porque se produce el evento Terminate del objeto. Aparte de ésta, no hay más información fiable acerca del mismo.

Nota El hecho de no tener que recordar las reglas del recuento de referencias de COM no es poco. Administrar el recuento de referencias es mucho más difícil que realizar el seguimiento de las variables de objeto que contienen referencias a objetos.

Sugerencia Declare sus variables de objeto como tipos de clases, no **As Object**. De este modo, si tiene un objeto **Widget** que no termina, las únicas variables de las que tiene que preocuparse son las declaradas **As Widget**.

En las colecciones de referencias de objeto, no utilice el objeto **Collection** de Visual Basic por sí solo. Las referencias de objeto de un objeto **Collection** de Visual Basic se almacenan en variables **Variant** que, como las variables declaradas **As Object**, pueden contener referencias a objetos de cualquier clase. En su lugar, cree sus propias clases de colección que acepten objetos de una única clase, como se describe en "Crear clases de colección propias". De esta manera, las únicas colecciones en las que tiene que buscar el objeto **Widget** son aquellas del tipo **Widget**.

Organice los objetos de forma jerárquica, como se describe en "Modelos de objetos". Si todos los objetos están conectados entre sí, será fácil escribir un procedimiento que recorra todo el modelo e informe de los objetos existentes.

No declare variables **As New**. Son como las velas que se vuelven a encender después de haberlas apagado: si utiliza una después de haberla establecido a **Nothing**, Visual Basic crea inevitablemente otro objeto.

Para obtener más información Las referencias circulares son las más difíciles de deshacer de forma limpia. Vea "Modelos de objetos".

Modelos de objetos

Después de definir una clase mediante la creación de un módulo de clase y la asignación de propiedades y métodos, puede crear cualquier número de objetos a partir de dicha clase. ¿Cómo puede efectuar el seguimiento de los objetos creados?

La manera más sencilla de efectuar el seguimiento de los objetos es declarar una variable de objeto por cada objeto que piensa crear. Naturalmente, esto supone un límite en cuanto al número de objetos que puede crear.

Puede mantener varias referencias de objeto en una matriz o en una colección, como se describe en "Crear matrices de objetos" y "Crear colecciones de objetos", anteriormente en este capítulo.

Al principio, probablemente colocará las variables, las matrices y las colecciones de objetos en formularios o en módulos estándar, como hace con las variables normales. Pero cuando agregue más clases, descubrirá que los objetos que utiliza tienen claras relaciones entre ellos.

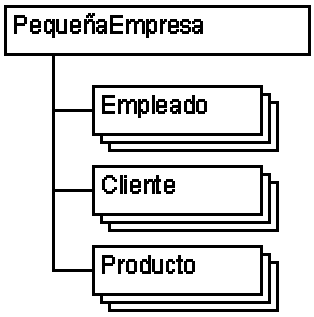
Los modelos de objetos definen relaciones de contenedores

Los modelos de objetos aportan la estructura a los programas basados en objetos. Cuando define las relaciones entre los objetos de un programa, el modelo de objetos organiza los objetos de forma que la programación sea más sencilla.

Normalmente, un modelo de objetos expresa el hecho de que unos objetos son "más grandes" o más importantes que otros. Se puede decir que estos objetos contienen otros objetos o que se componen de otros objetos.

Por ejemplo, puede crear un objeto **PequeñaEmpresa** como núcleo de su programa. Podría querer que el objeto **PequeñaEmpresa** tuviera asociados otros tipos de objetos, como objetos Empleado y objetos Cliente. Probablemente también querrá que contenga un objeto Producto. En la figura 9.11 se muestra un modelo de objetos para este programa.

Figura 9.11 Modelo de objeto



Puede definir cuatro módulos de clase, llamados PequeñaEmpresa, Empleado, Cliente y Producto, y asignar a cada uno de ellos las propiedades y los métodos apropiados, pero ¿cómo conecta los objetos entre sí? Tiene dos herramientas para este propósito: las propiedades de objeto y el objeto **Collection**. El siguiente fragmento de código muestra una manera de implementar la jerarquía de la figura 9.11.

```
' Código de la sección Declaraciones del
' módulo de clase PequeñaEmpresa.
Public Nombre As String
Public Producto As New Producto
Public Empleados As New Collection
Public Clientes As New Collection
```

La primera vez que haga referencia a la propiedad Producto se creará el objeto, puesto que está declarada **As New**. Por ejemplo, con el código siguiente puede crear y establecer el nombre y el precio del objeto Producto del objeto **PequeñaEmpresa**.

```
' Código de un módulo estándar.
Public sbMain As New PequeñaEmpresa
Sub Main
   sbMain.Nombre = "Organización Velociraptor S.A."
   ' La primera vez que utiliza la variable Producto
   ' en el código se crea el objeto Producto.
   sbMain.Producto.Nombre = "Velociraptor hinchable"
   sbMain.Producto.Precio = 1.98
   .
   .   ' Código para inicializar y mostrar el
   .   ' formulario principal.
   .
End Sub
```

Nota Implementar una propiedad de objeto con variables públicas es algo peligroso. Podría destruir el objeto Producto sin darse cuenta si establece la propiedad a **Nothing** en alguna parte del código. Es mejor crear las propiedades de objeto como propiedades de sólo lectura, como se muestra en el siguiente fragmento de código.

```
' Código de una propiedad objeto más robusta. El
' almacenamiento es privado y así no se puede
' establecer a Nothing desde fuera del objeto.
Private mProducto As New Producto

Property Get Producto() As Producto
   ' La primera vez que invoca la propiedad,
   ' mProducto contiene Nothing, por lo que Visual Basic
   ' creará un objeto Producto.
   Set Producto = mProducto
End If
```

Relaciones uno a varios

Las propiedades de objeto funcionan bien cuando la relación entre los objetos es de uno a uno. Sin embargo, ocurre con frecuencia que un objeto de un tipo contiene varios objetos de otro tipo. En el modelo de objetos **PequeñaEmpresa**, la propiedad Empleados está implementada como un objeto **Collection** para que el objeto **PequeñaEmpresa** pueda contener varios objetos Empleado. En el siguiente fragmento de código se muestra cómo puede agregar nuevos objetos Empleado a esta colección.

```
Public Function NuevoEmpleado(Nombre, Salario,
FechaEntrada, ID) As Empleado
   Dim empNew As New Empleado
   empNew.Nombre = Nombre      ' Creación implícita.
   empNew.Salario = Salario
   empNew.FechaEntrada = FechaEntrada
   ' Agrega a la colección, con el Id. como clave.
   sbMain.Empleados.Add empNew, CStr(ID)
   ' Devuelve una referencia al nuevo Empleado.
   Set NuevoEmpleado = empNew
End Function
```

La función **NuevoEmpleado** se puede llamar tantas veces como sea necesario para crear empleados en la empresa representada por el objeto **PequeñaEmpresa**. Los empleados existentes se pueden enumerar en cualquier momento recorriendo la colección Empleados.

Nota Una vez más, ésta no es una implementación muy sólida. Una forma mejor de programar sería crear sus propias clases de colección y exponerlas como propiedades de sólo lectura. Esto se describe en "Crear clases de colección propias".

Sugerencia El programa Generador de clases, incluido en la Edición Profesional y en la Edición Empresarial de Visual Basic, puede generar gran parte del código necesario para implementar un modelo de objetos. El Generador de clases crea robustas propiedades de objeto y clases de colección, y permite reorganizar el modelo con facilidad.

Propiedades Parent

Cuando tiene una referencia a un objeto, puede tener acceso a los objetos que contiene mediante las propiedades de objeto y las colecciones. También es muy útil poder recorrer la jerarquía para llegar al objeto que contiene el objeto al que se hace referencia.

El desplazamiento hacia arriba se suele efectuar con propiedades **Parent**. La propiedad **Parent** devuelve una referencia al contenedor del objeto. Para obtener una descripción del desplazamiento por el modelo de objetos, vea "Explorar los modelos de objetos" en "Programar con componentes".

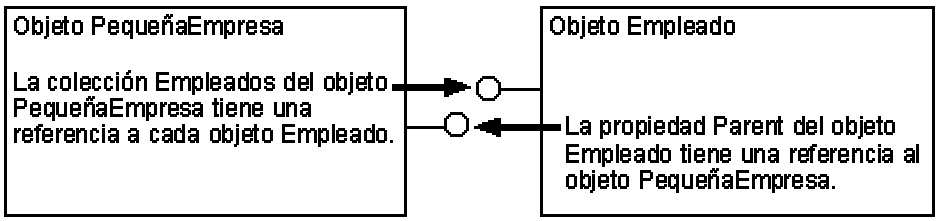
Puede encontrar un ejemplo de la propiedad **Parent** en "Agregar propiedades a una clase", anteriormente en este capítulo.

Sugerencia Cuando asigne una propiedad **Parent** a un objeto de una colección, no utilice una referencia al objeto **Collection**. El objeto primario es realmente el objeto que contiene la colección. Si la propiedad **Parent** apunta a la colección, tendrá que usar dos niveles de indirección para llegar al objeto primario real; es decir, `obj.Parent.Parent` en vez de `obj.Parent`.

Propiedades Parent, referencias circulares y limpieza de objetos

Uno de los mayores problemas de las propiedades **Parent** es que pueden crear referencias circulares. El objeto "mayor" tiene una referencia al objeto que contiene y el objeto contenido tiene una referencia al primero mediante su propiedad **Parent**, lo que crea un bucle como el que se muestra en la figura 9.12.

Figura 9.12 Caso de referencias circulares



¿Dónde está el error en esta imagen? La manera de deshacerse de los objetos cuando ya no son necesarios es liberar todas las referencias a ellos. Si supone que la referencia al objeto **PequeñaEmpresa** se encuentra en una variable llamada `sbMain`, como en la sección anterior de este tema, puede escribir el código siguiente:

```
Set sbMain = Nothing
```

Desgraciadamente, todavía hay una referencia al objeto **PequeñaEmpresa**. De hecho, puede haber muchas referencias, porque cada propiedad **Parent** del objeto Empleado mantiene una referencia al objeto **PequeñaEmpresa**.

Como la colección Empleados del objeto **PequeñaEmpresa** mantiene una referencia a cada objeto Empleado, no se podrá destruir ninguno de los objetos.

Métodos de limpieza

Una solución consiste en incluir un método de limpieza en el objeto **PequeñaEmpresa**. Este método podría establecer todas las propiedades de objeto de **PequeñaEmpresa** a **Nothing** y todos los objetos **Collection** (Empleados, Clientes) a **Nothing**.

Cuando se destruye un objeto **Collection**, Visual Basic establece todas las referencias de objeto que contenía a **Nothing**. Si no hay más referencias a los objetos Empleado y Cliente contenidos en las colecciones Empleados y Clientes, se destruirán.

Desde luego, si el objeto Empleado se compone de objetos más pequeños, tendrá el mismo problema de referencia circular que el objeto primario. En ese caso, debe incluir un método de limpieza en la clase Empleado. En vez de establecer simplemente el objeto de colección Empleados a **Nothing**, el objeto **PequeñaEmpresa** recorrerá antes la colección y llamará al método de limpieza de cada objeto Empleado.

Todavía no hemos terminado

Incluso en esta situación puede que no se destruyan todos los objetos. Si en alguna parte del programa hay variables que siguen manteniendo referencias al objeto **PequeñaEmpresa** o a cualquiera de los objetos que contiene, dichos objetos no se destruirán. Una parte de la limpieza final del programa tiene que asegurarse de que todas las variables de objeto se establezcan a **Nothing**.

Para comprobar si esto ocurre, puede que tenga que agregar algo de código de depuración a los objetos. Por ejemplo, puede agregar el código siguiente a un módulo estándar:

```
' Colección global de depuración
Public gcolDebug As New Collection

' Función global para dar a cada objeto un Id. único.
Public Function DebugSerial() As Long
   Static lngSerial As Long
   lngSerial = lngSerial + 1
   DebugSerial = lngSerial
End Function
```

En cada módulo de clase puede colocar código similar al siguiente. Cada clase proporciona su propio nombre donde aparece "Producto".

```
' Almacenamiento para el Id. de depuración.
Private mlngDebugID As Long

Property Get DebugID() As Long
   DebugID = mlngDebugID
End Property

Private Sub Class_Initialize()
   mlngDebugID = DebugSerial
   ' Agrega una entrada de cadena a la colección
   ' global.
   gcolDebug.Add "Inicio de Producto; DebubID=" _
   & DebugID, CStr(DebugID)
End Sub

Private Sub Class_Terminate()
   ' Quita la entrada de cadena; así sabe que el
```

```
    ' objeto ha dejado de existir.
    gcolDebug.Remove CStr(DebugID)
End Sub
```

Cuando se crea un objeto, se coloca una cadena en la colección global; cuando se destruye, se quita la cadena. En cualquier momento puede recorrer la colección global para ver qué objetos no se han destruido.

Para obtener más información Los modelos de objetos adquieren una nueva importancia y otros problemas diferentes cuando utiliza la Edición Profesional o la Edición Empresarial de Visual Basic para crear componentes ActiveX. Vea "Principios generales del diseño de componentes" en la sección *Crear componentes ActiveX* de la *Guía de herramientas componentes*.

Crear clases de colección propias

Puede usar tres técnicas para implementar objetos contenedores mediante colecciones. Considere la colección Empleados del objeto **PequeñaEmpresa** descrito en "Modelos de objetos". Para implementar esta colección podría:

- En el módulo de clase PequeñaEmpresa, declarar una variable `Empleados` como **As Collection** y hacerla **Public**. Ésta es la solución más sencilla.

- En el módulo de clase PequeñaEmpresa, declarar una variable `mcolEmpleados` como **As Collection** y hacerla **Private**. Incluya en el objeto **PequeñaEmpresa** un conjunto de métodos para agregar y eliminar objetos. De los tres diseños, éste es el menos orientado a objetos.

- Implementar una clase de colección propia mediante la creación de un módulo de clase de colección llamado Empleados, como se describe más adelante en este capítulo. Incluya en el objeto **PequeñaEmpresa** una propiedad de sólo lectura de la clase Empleados.

Estas estrategias se enumeran de menor a mayor robustez. Se podrían comparar con la casa de paja, la casa de madera y la casa de ladrillos.

- Ejemplo de colección pública: la casa de paja La propia flexibilidad del objeto **Collection** le puede traicionar: puede poner cualquier cosa en un objeto **Collection**, incluso el objeto Desagüe.

- Ejemplo de colección privada: la casa de madera Puede mejorar la robustez si crea el objeto **Collection** como privado, pero pierde la posibilidad de usar **For Each … Next** con la colección.

- Crear una clase de colección propia: la casa de ladrillos Crear su propia clase colección le ofrece la solidez del encapsulamiento y, como propina, tiene la posibilidad de usar **For Each … Next**.

- Ventajas del correcto diseño orientado a objetos Las clases de colección hacen que el código sea más limpio y más seguro.

Ejemplo de colección pública: la casa de paja

Para crear el ejemplo, abra un nuevo proyecto e inserte dos módulos de clase. Dibuje cinco botones de comando, un cuadro de lista, dos cuadros de texto y dos etiquetas en el formulario, como se muestra en la figura 9.13.

Figura 9.13 Ejemplo de la colección Empleados

En la tabla siguiente se enumeran los valores de las propiedades que tiene que establecer en este ejemplo.

Objeto	Propiedad	Valor
Módulo de clase	**Name**	Empleado
Módulo de clase	**Name**	PequeñaEmpresa
Formulario	**Caption**	Colección Empleados
Primer botón de comando	**Caption** **Name**	Agregar cmdAddEmployee
Segundo botón de comando	**Caption** **Name**	Eliminar cmdDeleteEmployee
Tercer botón de comando	**Caption** **Name**	Actualizar lista cmdListEmployee
Cuarto botón de comando	**Caption** **Name**	Problema cmdTrouble
Quinto botón de comando	**Caption** **Name**	Cerrar cmdClose
Primera etiqueta	**Caption**	Nombre
Segunda etiqueta	**Caption**	Salario
Primer cuadro de texto	**Name** **Text**	txtName (vacío)
Segundo cuadro de texto	**Name**	txtSalary

Objeto	Propiedad	Valor
	Text	(vacío)
Cuadro de lista	**Name**	lstEmployees

En el módulo de clase Empleado, agregue las siguientes declaraciones y procedimientos de propiedad:

```
Option Explicit
' Propiedades de la clase Empleado.
Public Nombre As String
Public Salario As Long

' Datos privados de la propiedad ID de una única
' escritura.
Private mstrID As String

Property Get ID() As String
   ID = mstrID
End Property

' La primera vez que se establece la propiedad ID,
' también se establece la variable booleana estática.
' Las llamadas siguientes no hacen nada.
' (Sería mejor desencadenar un error.)
Property Let ID(strNew As String)
   Static blnAlreadySet As Boolean
   If Not blnAlreadySet Then
      blnAlreadySet = True
      mstrID = strNew
   End If
End Property
```

La propiedad ID es la clave para obtener o eliminar un objeto Empleado de la colección, de modo que se establece una vez y ya no se modifica. Esto se consigue con la variable **Static Boolean**, que se establece a **True** la primera vez que se establece la propiedad. Siempre se puede leer la propiedad, puesto que hay un procedimiento **Property Get**.

En el módulo de clase PequeñaEmpresa, agregue la siguiente declaración. El objeto de colección se creará la primera vez que se haga referencia a la variable `Empleados` desde el código.

```
Option Explicit
Public Empleados As New Collection
```

El formulario hace todo el trabajo

Todo el resto del código se encuentra en el módulo de formulario. Agregue la siguiente declaración en la sección Declaraciones.

```
Option Explicit
Public sbMain As New PequeñaEmpresa
```

El código del evento cmdAddEmployee_Click agrega un miembro a la colección.

```
Private Sub cmdAddEmployee_Click()
   Dim empNew As New Empleado
   Static intEmpNum As Integer
   ' El uso de With hace que el código sea más
```

```
    ' rápido y conciso (.ID en vez de empNew.ID).
    With empNew
       ' Genera un Id. único para el nuevo empleado.
       intEmpNum = intEmpNum + 1
       .ID = "E" & Format$(intEmpNum, "00000")
       .Nombre = txtName.Text
       .Salario = CDbl(txtSalary.Text)
       ' Agrega la referencia del objeto Empleado a la
       ' colección, con la propiedad ID como clave.
       sbMain.Empleados.Add empNew, .ID
    End With
    txtName.Text = ""
    txtSalary.Text = ""
    ' Clic en el botón Actualizar lista.
    cmdListEmployees.Value = True
End Sub
```

El código del procedimiento de evento cmdListEmployees_Click utiliza una instrucción **For Each ... Next** para agregar toda la información de empleados al control **ListBox**.

```
Private Sub cmdListEmployees_Click()
    Dim emp As Empleado
    lstEmployees.Clear
    For Each emp In sbMain.Empleados
       lstEmpleados.AddItem emp.ID & ", " & _
       emp.Nombre & ", " & emp.Salario
    Next
End Sub
```

El evento cmdDeleteEmployee_Click utiliza el método **Remove** del objeto **Collection** para eliminar el miembro de la colección actualmente seleccionado en el control **ListBox**.

```
Private Sub cmdDeleteEmployee_Click()
    ' Comprueba que hay un empleado seleccionado.
    If lstEmployees.ListIndex > -1 Then
       ' Los seis primeros caracteres son el Id.
       sbMain.Empleados.Remove _
       Left(lstEmployees.Text, 6)
    End If
    ' Clic en el botón Actualizar lista.
    cmdListEmployees.Value = True
End Sub
```

Agregue el código siguiente al botón **Problema**.

```
Private Sub cmdTrouble_Click()
    ' ¿Cómo dice?
    sbMain.Empleados.Add Me
End Sub
```

El evento cmdClose_Click cierra la aplicación. Cuando cierre proyectos que utilizan objetos, descargue todos los formularios para estar seguro de que se ejecutan los procedimientos de evento Terminate de todos los módulos de clase. Por el contrario, el uso de la instrucción **End** termina el programa de forma brusca sin ejecutar los eventos Terminate.

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

Para agregar empleados en el ejemplo, ejecute la aplicación, escriba valores en los dos cuadros de texto y después haga clic en el botón **Agregar**. Agregue unos cuantos empleados y después experimente con los botones **Eliminar** y **Actualizar lista**.

Tan sólida como una casa de paja

Esta sencilla implementación no es muy robusta. Como la propiedad Empleados es un objeto público **Collection**, podría tener acceso a ella de forma inadvertida desde cualquier parte del programa. Además, el método **Add** del objeto **Collection** no comprueba el tipo. Por ejemplo, el código del evento Click del botón **Problema** inserta alegremente una referencia al formulario en la colección Empleados.

Haga clic en el botón **Problema** y observe que no se produce ningún error. Después haga clic en el botón **Actualizar lista**. Cuando el bucle **For Each ... Next** llega al objeto de tipo inesperado, provoca el error 13, "El tipo no coincide".

Éste es un ejemplo del tipo de error al que se expone cuando crea un modelo de objetos con objetos públicos **Collection**. Los objetos se pueden agregar desde cualquier parte del proyecto y no hay garantías de que se inicien correctamente. Si un programador copia el código con el que se agrega un empleado y después se modifica el código original, los errores que se provoquen serán muy difíciles de detectar.

Para obtener más información El ejemplo iniciado en este tema continúa en "Ejemplo de colección privada: la casa de madera".

Ejemplo de colección privada: la casa de madera

Este tema continúa el ejemplo de código iniciado en "Ejemplo de colección pública: la casa de paja". Puede que le interese leer ese tema antes de empezar éste.

Una forma algo más sólida de vincular objetos Empleado con el objeto **PequeñaEmpresa** es hacer que el objeto **Collection** sea privado. En este ejemplo volverá a usar el formulario y la mayor parte del código del ejemplo de la colección pública.

El módulo de clase Empleado sigue igual. Sin embargo, el módulo de clase PequeñaEmpresa cambia de cara totalmente. Sustituya la declaración del objeto público **Collection** por la siguiente declaración y agregue los procedimientos **Sub** y **Function** descritos en los párrafos siguientes.

```
Option Explicit
Private mcolEmpleados As New Collection
```

Como antes, el código con el que se agrega un empleado efectúa casi todo el trabajo. Puede aprovechar el bloque de código entre las líneas de puntos del procedimiento de evento cmdEmployeeAdd_Click del ejemplo anterior.

El cambio más importante es que el método **Add** del objeto **Collection** ya no se puede invocar desde cualquier módulo del programa, puesto que `mcolEmpleados` es **Private**. Sólo puede agregar un objeto **Empleado** mediante el método **AgregarEmpleado**, que inicia correctamente el nuevo objeto:

```
' Método de la clase PequeñaEmpresa.
Public Function AgregarEmpleado(ByVal Name As String, _
ByVal Salary As Double) As Empleado
   ' - - - - - - - - - - - - - - - - -
   Dim empNew As New Empleado
   Static intEmpNum As Integer
   ' El uso de With hace que el código sea más
   ' rápido y conciso (.ID en vez de empNew.ID).
   With empNew
      ' Genera un Id. único para el nuevo empleado.
      intEmpNum = intEmpNum + 1
      .ID = "E" & Format$(intEmpNum, "00000")
      .Nombre = Name
```

```
      .Salario = Salary
      ' Agrega la referencia del objeto Empleado a la
      ' colección, con la propiedad ID como clave.
      ' - - - - - - - - - - - - - - - - -
      mcolEmpleados.Add empNew, .ID
   End With
   ' Devuelve una referencia al nuevo objeto Empleado.
   Set AgregarEmpleado = empNew
End Function
```

El método **AgregarEmpleado** devuelve una referencia al objeto Empleado recién agregado. Ésta es la forma correcta de programar, porque después de crear un objeto se suele querer hacer algo con él.

Los métodos **ContarEmpleados**, **EliminarEmpleado** y Empleados *delegan* en los métodos correspondientes del objeto **Collection**. La delegación significa que el objeto **Collection** realiza todo el trabajo.

```
' Métodos de la clase PequeñaEmpresa.
Public Function ContarEmpleados() As Long
   ContarEmpleados = mcolEmpleados.Count
End Function

Public Sub EliminarEmpleados(ByVal Index As Variant)
   mcolEmpleados.Remove Index
End Sub

Public Function Empleados(ByVal Index As Variant) _
As Empleado
   Set Empleados = mcolEmpleados.Item(Index)
End Function
```

Nota Puede agregar funcionalidad adicional a estos métodos. Por ejemplo, puede desencadenar errores si el índice no es válido.

El último método es Problema. Este método intenta agregar a la colección un objeto **Empleado** no inicializado. ¿Qué puede pasar?

```
' Método de la clase PequeñaEmpresa.
Public Sub Problema()
   Dim x As New Empleado
   mcolEmpleados.Add x
End Sub
```

Modificaciones en el formulario

Tendrá que efectuar algunas modificaciones en el módulo de formulario. Puede usar las declaraciones de nivel de módulo empleadas en el ejemplo anterior y el evento Click del botón **Cerrar** es el mismo, pero se han cambiado otros procedimientos de evento: el código del botón **Agregar** es mucho más breve, mientras que el código de los botones **Eliminar** y **Actualizar lista** ha cambiado poco pero de forma muy significativa:

```
Private Sub cmdAddEmployee_Click()
   sbMain.AgregarEmpleado txtName.Text, txtSalary.Text
   txtName.Text = ""
   txtSalary.Text = ""
   cmdListEmployees.Value = True
End Sub

Private Sub cmdDeleteEmployees_Click()
```

```
   ' Verifica que hay un empleado seleccionado.
   If lstEmployees.ListIndex > -1 Then
      ' Los seis primeros caracteres son el Id.
      sbMain.EliminarEmpleado Left(lstEmployees.Text,
      6)
   End If
   cmdListEmployees.Value = True
End Sub

Private Sub cmdListEmployees_Click()
   Dim lngCt As Long
   lstEmployees.Clear
   For lngCt = 1 To sbMain.ContarEmpleados
      With sbMain.Empleados(lngCt)
         lstEmployees.AddItem .ID & ", " & .Nombre _
         & ", " & .Salario
      End With
   Next
End Sub
```

¿Qué es todo este código extra de cmdListEmployees_Click? Desgraciadamente, en favor de la solidez ha perdido la posibilidad de usar **For Each ... Next** para recorrer los elementos de la colección, ya que el objeto **Collection** está declarado como **Private**. Si intenta escribir lo siguiente, recibirá un error:

```
' No funciona, porque Empleados no es una
' colección.
For Each emp In sbMain.Empleados
```

Afortunadamente, el método **ContarEmpleados** se puede usar para delimitar el intervalo de iteración.

El botón **Problema** también cambia un poco, pero sigue siendo lo que es, un problema.

```
Private Sub cmdTrouble_Click()
   sbMain.Problema
End Sub
```

Ejecute el proyecto y experimente con los botones **Agregar**, **Eliminar** y **Actualizar lista**. Todo funciona igual que antes.

Cuando hace clic en el botón **Problema** tampoco se generan errores. Sin embargo, si hace clic en el botón **Actualizar lista**, puede ver que de alguna manera se ha agregado a la colección el objeto **Empleado** no inicializado.

¿Cómo ha podido ser? Al crear el objeto **Collection** como privado, lo protegió del código del programa que está *fuera* del objeto **PequeñaEmpresa**, pero no del código de *dentro*. El objeto **PequeñaEmpresa** puede ser grande y complejo, con una gran cantidad de código. Por ejemplo, probablemente tiene métodos como **AgregarCliente**, **AgregarProducto**, etc.

Un error de escritura de código o la creación de un duplicado del método **AgregarEmpleado** puede hacer que se inserten en la colección datos incorrectos e incluso objetos no válidos, ya que la variable privada es visible en todo el módulo de clase.

Para obtener más información Este ejemplo continúa en "Crear una clase de colección propia: la casa de ladrillos".

Crear una clase de colección propia: la casa de ladrillos

Este tema continúa el ejemplo de código iniciado en "Ejemplo de colección pública: la casa de paja" y en "Ejemplo de colección privada: la casa de madera". Puede que le interese ver esos temas antes de empezar éste.

La manera más robusta de implementar una colección es convertirla en un módulo de clase. En contraste con los ejemplos anteriores, pasar todo el código de la creación del objeto a la clase de colección cumple los principios del buen diseño de objetos.

En este ejemplo se utiliza el mismo formulario y el mismo módulo de clase Empleado que en los ejemplos anteriores. Inserte un nuevo módulo de clase y establezca su propiedad **Name** a "Empleados". Inserte las declaraciones y el código siguientes en el nuevo módulo de clase.

```
Option Explicit
Private mcolEmpleados As New Collection
```

Los métodos Add, Count y Delete de la clase Empleados son esencialmente los mismos que los de la antigua clase PequeñaEmpresa. Puede quitarlos del módulo de clase PequeñaEmpresa, pegarlos en el módulo de clase Empleados y cambiar sus nombres.

Los nombres pueden cambiar porque ya no es necesario distinguir AgregarEmpleado de, por ejemplo, AgregarCliente. Cada clase colección que implemente tiene su propio método **Add**.

```
' Métodos de la clase de colección Empleados.
Public Function Add(ByVal Name As String, _
ByVal Salary As Double) As Empleado
   Dim empNew As New Empleado
   Static intEmpNum As Integer
   ' El uso de With hace que el código sea más
   ' rápido y conciso (.ID en vez de empNew.ID).
   With empNew
      ' Genera un Id. único para el nuevo empleado.
      intEmpNum = intEmpNum + 1
      .ID = "E" & Format$(intEmpNum, "00000")
      .Nombre = Name
      .Salario = Salary
      ' Agrega la referencia del objeto Empleado a la
      ' colección, con la propiedad ID como clave.
      mcolEmpleados.Add empNew, .ID
   End With
   ' Devuelve una referencia al nuevo objeto Empleado.
   Set Add = empNew
End Function

Public Function Count() As Long
   Count = mcolEmpleados.Count
End Function

Public Sub Delete(ByVal Index As Variant)
   mcolEmpleados.Remove Index
End Sub
```

El método Empleados del objeto **PequeñaEmpresa** se convierte en el método **Item** de la clase de colección. Sigue delegando en el objeto **Collection** para poder obtener los miembros por índice o por clave.

```
' Método de la clase de colección Empleados.
```

```
Public Function Item(ByVal Index As Variant) _
As Empleado
   Set Item = colEmpleados.Item(Index)
End Function
```

Aquí puede agregar un toque de estilo. Si hace que **Item** sea el método predeterminado de la clase Empleados, tendrá la posibilidad de escribir `Empleados("E00001")`, igual que podía hacerlo con el objeto **Collection**.

Para hacer de Item la propiedad predeterminada

1. En el menú **Herramientas**, haga clic en **Atributos del procedimiento** para abrir el cuadro de diálogo **Atributos del procedimiento**. En el cuadro **Nombre**, seleccione el método **Item**.

2. Haga clic en **Avanzados** para mostrar las características avanzadas. En el cuadro **Id. del procedimiento**, seleccione **(Predeterminado)** para que el método **Item** sea el predeterminado. Haga clic en **Aceptar**.

   Nota Una clase sólo puede tener un miembro predeterminado (una propiedad o un método).

Activar For Each … Next

Junto con la solidez, vuelve a disponer de **For Each … Next**. Una vez más, puede delegar todo el trabajo en el objeto **Collection** si agrega el método siguiente:

```
' NewEnum tiene que devolver la interfaz IUnknown del
' enumerador de una colección.
Public Function NewEnum() As IUnknown
   Set NewEnum = mcolEmpleados.[_NewEnum]
End Function
```

Lo más importante que delega en el objeto **Collection** es el *enumerador*. El enumerador es un pequeño objeto que sabe cómo recorrer los elementos de una colección. No puede escribir un objeto enumerador con Visual Basic, pero como la clase Empleados está basada en un objeto **Collection**, puede usar el enumerador del objeto **Collection** que, naturalmente, sabe de sobra cómo enumerar los elementos del objeto **Collection** que mantiene.

Los corchetes que enmarcan el método **_NewEnum** del objeto **Collection** son necesarios a causa del subrayado inicial del nombre del método. Este subrayado inicial es una convención que indica que el método se encuentra oculto dentro de la biblioteca de tipos. No puede asignar un nombre al método **_NewEnum**, pero puede ocultarlo en la biblioteca de tipos y darle el Id. de procedimiento que requiere **For Each … Next**.

Para ocultar el método NewEnum y darle el Id. de procedimiento necesario

1. En el menú **Herramientas**, haga clic en **Atributos del procedimiento** para abrir el cuadro de diálogo **Atributos del procedimiento**. En el cuadro **Nombre**, seleccione el método **NewEnum**.

2. Haga clic en **Avanzados** para mostrar las características avanzadas. Active **Ocultar este miembro** para ocultar **NewEnum** en la biblioteca de tipos.

3. En el cuadro **Id. del procedimiento**, escriba **–4** (menos cuatro) para dar a **NewEnum** el Id. de procedimiento requerido por **For Each … Next**. Haga clic en **Aceptar**.

Importante Para que las clases de colección acepten **For Each … Next**, tiene que proporcionar un método **NewEnum** oculto con el Id. de procedimiento correcto.

No queda mucho de la clase PequeñaEmpresa

La clase PequeñaEmpresa tiene ahora mucho menos código. Para sustituir el objeto **Collection** y todos los métodos quitados hay una nueva declaración y una propiedad de sólo lectura:

```
Option Explicit
Private mEmpleados As New Empleados

Public Property Get Empleados() As Empleados
   Set Empleados = mEmpleados
End If
```

Esto merece una explicación. Suponga por un momento que quita el procedimiento **Property Get** y simplemente declara `Public Empleados As New Empleados`.

Todo funciona bien mientras nadie cometa errores, pero ¿qué pasa si escribe accidentalmente `Set sbMain.Empleados = Nothing`? Eso es, se destruirá la colección Empleados. Si convierte Empleados en una propiedad de sólo lectura, evita dicha posibilidad.

Modificaciones al formulario

El código del módulo de formulario es muy parecido al del ejemplo anterior. Puede usar las mismas declaraciones de nivel de módulo y el evento Click del botón **Cerrar** es el mismo.

El único cambio en la mayoría de los procedimientos de evento es que se reemplazan los antiguos métodos de la clase PequeñaEmpresa por los nuevos métodos del objeto de colección Empleados:

```
Private Sub cmdAddEmployee_Click()
   sbMain.Empleados.Add txtName.Text, txtSalary.Text
   txtName.Text = ""
   txtSalary.Text = ""
   cmdListEmployees.Value = True
End Sub

Private Sub cmdDeleteEmployee_Click()
   ' Comprueba que hay un empleado seleccionado.
   If lstEmpleados.ListIndex > -1 Then
      ' Los seis primeros caracteres son el Id.
      sbMain.Empleados.Delete _
      Left(lstEmployees.Text, 6)
   End If
   cmdListEmployees.Value = True
End Sub

Private Sub cmdListEmployees_Click()
   Dim emp As Empleado
   lstEmployees.Clear
   For Each emp In sbMain.Empleados
      lstEmployees.AddItem emp.ID & ", " & _
      emp.Nombre & ", " & emp.Salario
   Next
End Sub
```

Observe que otra vez puede usar **For Each … Next** para enumerar los empleados.

Ejecute el proyecto y compruebe que todo funciona bien. Esta vez no hay código en el botón **Problema**, puesto que el encapsulamiento impide que haya problemas.

Para obtener más información Para obtener un mayor conocimiento de las colecciones, vea "Objeto Collection de Visual Basic" y "Colecciones en Visual Basic".

El programa Generador de clases incluido en la Edición Profesional y en la Edición Empresarial le ayudará a crear clases de colección.

Las lecciones de los ejemplos de la casa de paja, la casa de madera y la casa de ladrillos se resumen en "Ventajas del correcto diseño orientado a objetos".

Ventajas del correcto diseño orientado a objetos

En este tema se resumen los resultados del ejemplo de código iniciado en "Ejemplo de colección pública: la casa de paja" y continuado en "Ejemplo de colección privada: la casa de madera" y "Crear una clase colección propia: la casa de ladrillos". Puede que le interese ver esos temas antes de empezar este.

La creación de la clase de colección Empleados da como resultado un estilo de escritura de código muy claro y modular. Todo el código de la colección está en la clase de colección (encapsulamiento), lo que reduce el tamaño del módulo de clase PequeñaEmpresa. Si aparecen colecciones de objetos Empleado en más de un lugar de la jerarquía de objetos, volver a usar la clase de colección no requiere la duplicación del código.

Mejorar las clases de colección

Puede implementar métodos y propiedades adicionales para las clases colección. Por ejemplo, puede implementar métodos **Copy** y **Move** o una propiedad **Parent** de sólo lectura que contenga una referencia al objeto PequeñaEmpresa.

También puede agregar un evento. Por ejemplo, cada vez que el método **Add** o el método **Remove** modifique el número de elementos de la colección, puede desencadenar un evento CountChanged.

Solidez, solidez y solidez

No siempre tiene que implementar las colecciones de la manera más sólida posible. Sin embargo, una de las ventajas de programar con objetos es la reutilización del código; es mucho más fácil reusar objetos que copiar código fuente y es mucho más seguro usar código sólido y encapsulado.

Como dijo un sabio, "Si quiere escribir código realmente robusto es porque asume que se le van a presentar problemas".

Clases de colección y software componente

Si utiliza la Edición Profesional o la Edición Empresarial de Visual Basic, puede convertir su proyecto en un componente ActiveX, de modo que otros programadores de su organización puedan usar los objetos que ha creado.

Pasos para implementar una clase de colección

En la lista siguiente se resumen los pasos requeridos para crear una clase colección.

1. Agregue un módulo de clase al proyecto y asígnele un nombre (normalmente el plural del nombre del objeto que va a contener la clase colección). Vea "Asignar nombres a propiedades, métodos y eventos", anteriormente en este capítulo.

2. Agregue una variable privada para incluir una referencia al objeto **Collection** en el que se van a delegar las propiedades y los métodos.

3. En el procedimiento de evento Class_Initialize, cree el objeto **Collection**. Si quiere retrasar la creación de este objeto hasta cuando se vaya a usar, puede declarar la variable privada del paso 2 como **As New Collection**. Esto agrega un poco más de proceso cada vez que se tiene acceso al objeto **Collection**.

4. Agregue al módulo de clase una propiedad **Count** y métodos **Add**, **Item** y **Remove**; en cada caso, delegue en el objeto privado **Collection** mediante una llamada al miembro correspondiente.

5. Cuando implemente el método **Add**, puede eludir el comportamiento no discriminatorio del método **Add** del objeto **Collection** para aceptar solamente objetos de un tipo. Puede incluso impedir que se agreguen a la colección objetos creados externamente, de forma que el método **Add** controle completamente la creación e *inicialización* de los objetos.

6. Utilice el cuadro de diálogo **Atributos del procedimiento** para que el método **Item** sea el predeterminado de la clase de colección.

7. Agregue un método **NewEnum**, como se muestra a continuación. Utilice el cuadro de diálogo **Atributos del procedimiento** para marcarlo como oculto y asignarle un Id. de procedimiento de –4, para poder usar **For Each … Next**.

   ```
   Public Function NewEnum() As IUnknown
      Set NewEnum = mcol.[_NewEnum]
   End Function
   ```

   Nota En este código se asume que la variable privada del paso 2 se llama `mcol`.

8. Agregue propiedades, métodos y eventos personalizados a la clase de colección.

Nota El programa Generador de clases, incluido en la Edición Profesional y en la Edición Empresarial de Visual Basic, le ayudará a crear estas clases de colección. Puede personalizar el código fuente que produce.

Para obtener más información Puede encontrar más información acerca de los componentes software en la sección *Crear componentes ActiveX* de la *Guía de herramientas componentes*.

Diseñadores ActiveX

Un *diseñador* proporciona una ventana de diseño visual dentro del entorno de desarrollo de Visual Basic. Puede usar esta ventana para diseñar nuevas clases de forma visual. Visual Basic tiene diseñadores integrados para los formularios y, en la Edición Profesional y la Edición Empresarial, para los controles y documentos ActiveX.

Los objetos creados a partir de las clases que diseñe de esta manera tienen un comportamiento y apariencia diferentes en tiempo de diseño y en tiempo de ejecución, aunque muchos objetos, como los formularios y los controles, tienen una apariencia muy similar en los dos modos.

Además de los diseñadores integrados, Visual Basic permite que se puedan desarrollar diseñadores para usarlos en el entorno de desarrollo de Visual Basic. Estos *diseñadores ActiveX* funcionan de la misma manera que los diseñadores integrados de Visual Basic y son fáciles de aprender y de usar.

- ¿Qué son los diseñadores ActiveX? Compara y contrasta los diseñadores ActiveX con los diseñadores integrados de Visual Basic.

- Agregar un diseñador ActiveX al menú Proyecto Antes de poder usar un diseñador ActiveX, tiene que agregarlo al menú **Proyecto**.

- Insertar una nueva instancia de un diseñador ActiveX Explica la utilización de un diseñador ActiveX en un proyecto.

¿Qué son los diseñadores ActiveX?

Los diseñadores ActiveX pueden proporcionar interfaces visuales para tareas que, de otro modo, requerirían una gran cantidad de código. Por ejemplo, el diseñador UserConnection incluido en la Edición Empresarial de Visual Basic proporciona herramientas visuales para definir consultas de bases de datos complejas. En tiempo de ejecución, dichas consultas se pueden invocar con muy poco código.

Similitudes entre los diseñadores ActiveX y los diseñadores integrados

Los diseñadores ActiveX son como los diseñadores de formularios en los siguientes aspectos:

- Los diseñadores ActiveX producen clases a partir de las cuales puede crear objetos. Estas clases aparecen en la ventana Proyecto, igual que las clases de formulario.

- Las clases creadas con un diseñador ActiveX tienen sus propios módulos de código, en los que puede escribir código para los procedimientos de evento proporcionados por el diseñador.

- Puede personalizar una clase si agrega propiedades, métodos y eventos a los proporcionados por el diseñador ActiveX.

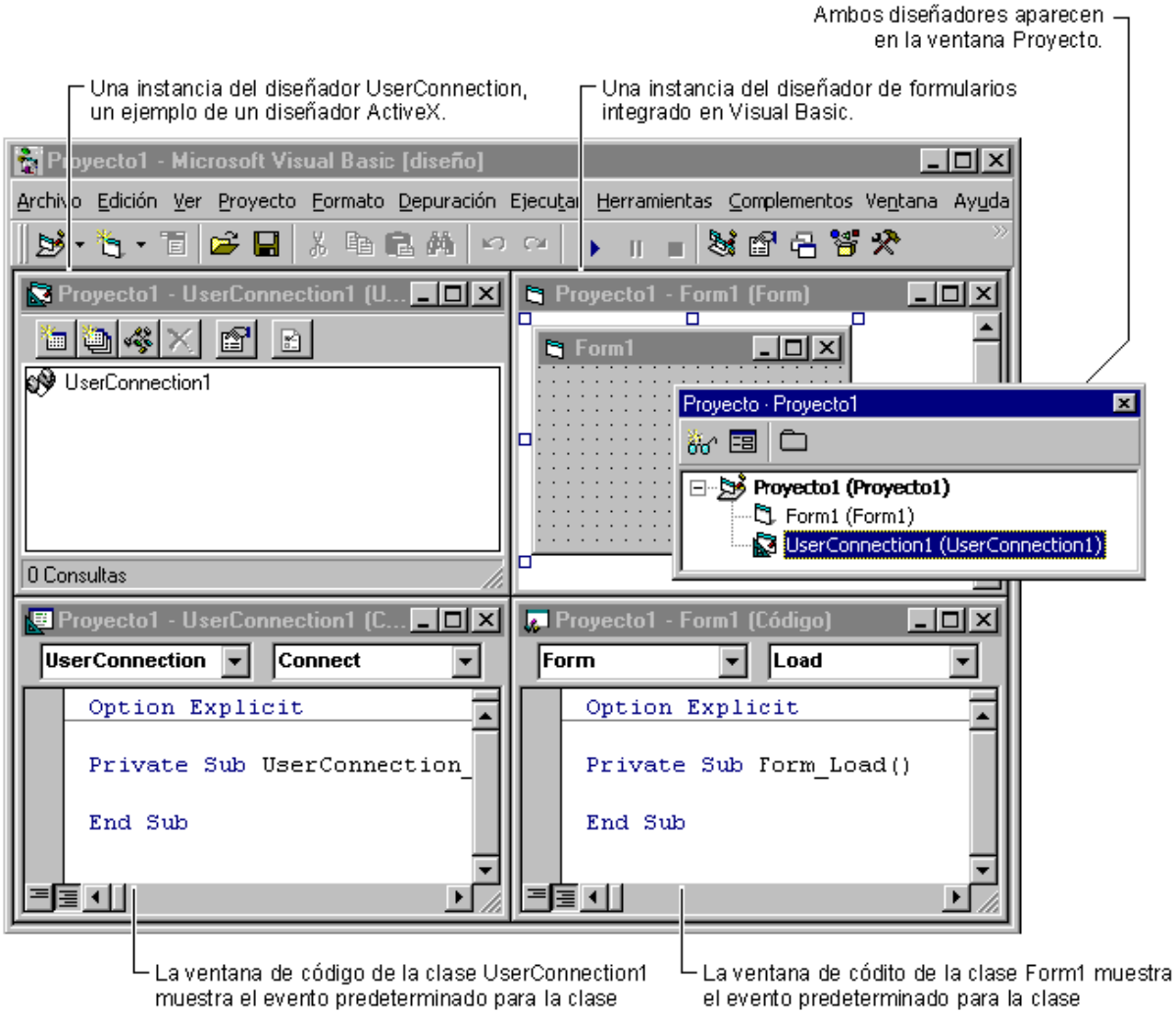
- Los objetos creados a partir de las clases diseñadas pueden tener características diferentes en tiempo de diseño y en tiempo de ejecución.

- La ventana de diseño del diseñador ActiveX está totalmente integrada en el entorno de desarrollo. Puede modificar su tamaño y configurarla de la misma forma que las ventanas de diseño incorporadas.

- Puede agregar al proyecto tantas instancias de un diseñador ActiveX como considere necesario, de la misma forma que puede agregar tantos diseñadores de formularios como quiera.

En la figura 9.14 se compara el diseñador de formulario integrado de Visual Basic con el diseñador UserConnection, uno de los diseñadores ActiveX incluidos en la Edición Empresarial de Visual Basic.

Figura 9.14 Diseñador ActiveX y diseñador integrado de Visual Basic



Comparar las clases de diseñadores ActiveX con otras clases diseñadas de forma visual

Los diseñadores ActiveX son extremadamente flexibles. Algunos, como el diseñador UserConnection, crean clases cuyas instancias de tiempo de ejecución son programables, pero no visibles. Otros, como el diseñador Forms de Microsoft utilizado por Microsoft Office, producen objetos visibles similares a los formularios de Visual Basic.

Los diseñadores ActiveX que tienen componentes visibles en tiempo de ejecución pueden albergar controles ActiveX. De hecho, se convierten en formularios alternativos que se pueden usar además de los formularios originales de Visual Basic.

En la lista siguiente se comparan las clases producidas por los diseñadores ActiveX con las producidas por los diseñadores integrados de Visual Basic.

- Si un objeto creado a partir de una clase creada por un diseñador ActiveX es visible en tiempo de ejecución, tiene su propia ventana. No está contenido dentro de otro formulario como lo están los controles ActiveX.

- Al igual que las clases de formulario, pero a diferencia de los controles ActiveX, las clases producidas por los diseñadores ActiveX son clases privadas. Si utiliza la Edición Profesional o la Edición Empresarial de Visual Basic para crear componentes ActiveX, no puede declarar métodos públicos que utilicen dichas clases como tipos de argumento o como valores devueltos.

  Por ejemplo, las siguientes declaraciones de métodos producen errores de tiempo de compilación si aparecen en una clase pública:

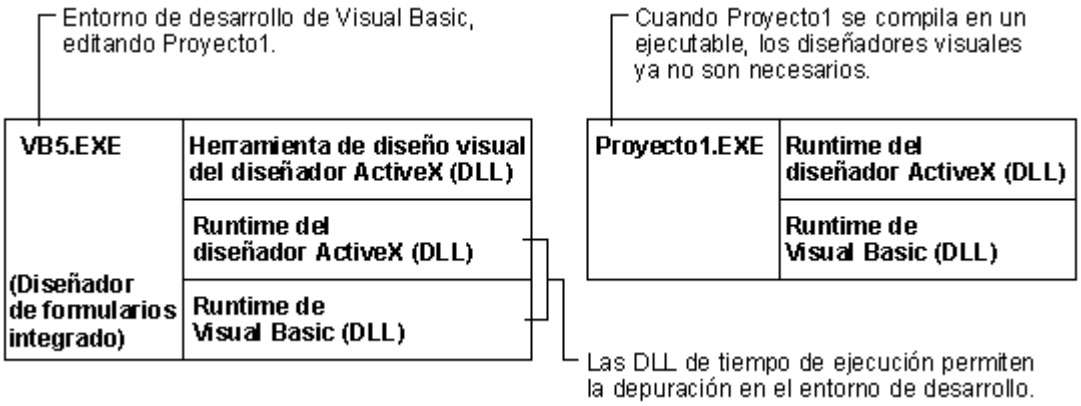
```
Public Function A() As UseConnection1        'Error
Public Sub B(CallBack As UseConnection1)     'Error
```

  Precaución Aunque es posible pasar referencias a objetos privados fuera del proyecto mediante la declaración de valores devueltos **As Object**, es una mala práctica y puede desestabilizar el programa. Para obtener más información, vea *Crear componentes ActiveX* en la *Guía de herramientas componentes.*

Usar los objetos diseñadores ActiveX en tiempo de ejecución

Como el diseñador de formularios integrado, los diseñadores ActiveX solamente se encuentran disponibles en el entorno de desarrollo. En cuanto el proyecto es un ejecutable, sólo utiliza la .dll de ejecución del diseñador ActiveX. Ésta suele ser más pequeña que la .dll de tiempo de diseño, porque no incluye la herramienta de diseño visual. La figura 9.15 ilustra este concepto.

Figura 9.15 Componentes del diseñador en memoria



Como ya se ha mencionado, los diseñadores ActiveX pueden producir clases cuyos objetos no sean visibles en tiempo de ejecución. El diseñador UserConnection mostrado en la figura 9.14 es un ejemplo. Este diseñador produce clases cuyos objetos administran conexiones con bases de datos SQL en tiempo de ejecución. No hay ninguna razón por la que dichos objetos tengan que ser visibles en tiempo de ejecución.

Para usar una clase creada con el diseñador UserConnection, declare una variable del tipo de clase y cree una instancia de la clase. Por ejemplo, si agregó un diseñador UserConnection y estableció su propiedad **Name** a GeneralLedger, puede crear un objeto GeneralLedger como se muestra en el siguiente fragmento de código:

```
' Variable global de un módulo estándar, para mantener
' una referencia al objeto GeneralLedger.
Public gGeneralLedger As GeneralLedger
```

```
' Código de un módulo de formulario para crear el
' objeto GeneralLedger y establecer una conexión
' con una base de datos.
Private Sub Command1_Click()
   Set gGeneralLedger = New gGeneralLedger
   gGeneralLedger.EstablishConnection
   ' (Código que utiliza el objeto.)
End Sub
```

Crear diseñadores ActiveX

Puede usar el Kit de desarrollo de software (SDK) ActiveX Designer para crear nuevos diseñadores ActiveX y usarlos con Visual Basic. El SDK incluye instrucciones completas y código de ejemplo. Puede encontrarlo en Microsoft Development Network (Red de Desarrollo Microsoft) bajo el título "SDK Documentation (Documentación del SDK)".

Nota El ActiveX Designer Software Development Kit (SDK de Diseñadores ActiveX) requiere un compilador C++, como Microsoft Visual C++. No puede escribir diseñadores ActiveX con Visual Basic.

Para obtener más información Los procedimientos para incorporar diseñadores ActiveX a un proyecto se encuentran en "Agregar un diseñador ActiveX al menú Proyecto" y en "Insertar una nueva instancia de un diseñador ActiveX".

Agregar un diseñador ActiveX al menú Proyecto

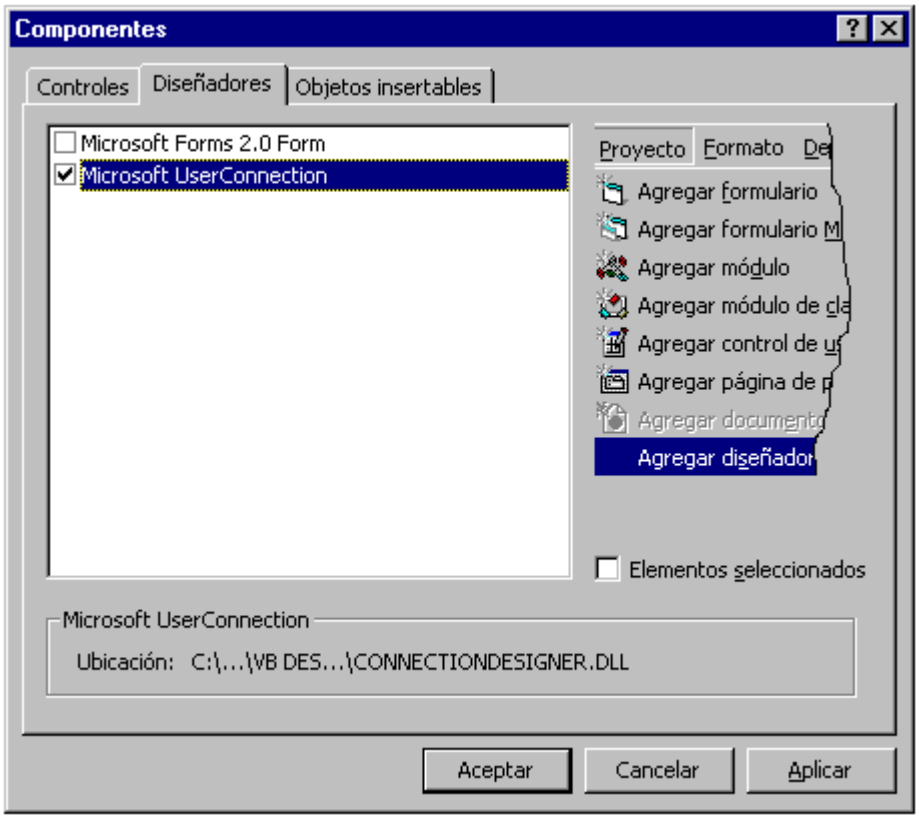
Después de instalar un nuevo diseñador ActiveX, mediante el programa de instalación suministrado por el proveedor, debe poner el diseñador a disposición de los proyectos, agregándolo al menú **Proyecto**.

La instalación del diseñador ActiveX se registrará en el Registro de Windows, dentro de la categoría de componente apropiada. Después, estará disponible en la ficha **Diseñadores** del cuadro de diálogo **Componentes**.

Para agregar un diseñador ActiveX al menú Proyecto

1. En el menú **Proyecto**, haga clic en **Componentes** para abrir el cuadro de diálogo **Componentes**.

2. Haga clic en la ficha **Diseñadores** y seleccione el diseñador que quiere usar, como se muestra en la figura siguiente. Haga clic en **Aceptar**.



Nota El diseñador Forms de Microsoft se incluye en todas las versiones de Visual Basic para permitir portar con facilidad los formularios creados en las aplicaciones de Microsoft Office. Como todos los diseñadores, el diseñador Forms de Microsoft tiene su propia .dll de tiempo de ejecución. Por tanto, el uso de este diseñador en un proyecto de Visual Basic incrementará los requisitos de memoria del ejecutable resultante.

Para obtener más información Para agregar un diseñador al proyecto, vea "Insertar una nueva instancia de un diseñador ActiveX".

Insertar una nueva instancia de un diseñador ActiveX

Después de usar el cuadro de diálogo **Componentes** para agregar un diseñador al menú **Proyecto**, como se muestra en "Agregar un diseñador ActiveX al menú Proyecto", puede insertar tantas instancias del diseñador como desee.

Para insertar una instancia de un diseñador ActiveX

- En el menú **Proyecto**, haga clic en **Agregar diseñador ActiveX** para presentar la lista de los diseñadores instalados. Elija el diseñador deseado en la lista, como se muestra en la figura siguiente:

  Nota Si se cargan más de cuatro diseñadores, los siguientes aparecerán en el submenú **Más diseñadores ActiveX** del menú **Proyecto**.

Después de agregar una instancia de un diseñador ActiveX al proyecto, puede usar la interfaz visual para diseñar una clase. Haga clic en el botón **Código** de la ventana **Proyecto** para abrir el módulo de código del diseñador y agregue código a los procedimientos de evento. Puede seguir personalizando la clase si agrega más propiedades, métodos y eventos, de igual forma que haría en un módulo de clase de Visual Basic.

Diseñadores ActiveX y control de código fuente

Cuando utilice los diseñadores ActiveX con un paquete de control de código fuente como Visual SourceSafe, el comportamiento puede ser diferente entre los diseñadores integrados. Cuando intente editar un archivo creado por un diseñador, puede que no le pregunte acerca de desproteger el archivo antes de editarlo. Adicionalmente, algunos diseñadores escriben archivos fuera del sistema de archivos; el control de código fuente puede no tener en cuenta estos archivos.

Para agregar un archivo creado por un diseñador al control de código fuente:

1. Elija **Agregar archivo** desde el menú **Proyecto**.

2. En el cuadro de diálogo **Agregar archivo**, seleccione el archivo que quiera agregar, active la casilla **Agregar como documento relacionado** y haga clic en **Abrir.**

   Esto podría agregar el archivo a su proyecto como un documento relacionado. El paquete de control de código fuente le preguntará para agregar el archivo (esto podría ocurrir cuando agregue el archivo o cuando guarde el proyecto la próxima vez, dependiendo del paquete de control de código fuente).

Nota Los archivos creados con el Diseñador de páginas HTML de Visual Basic pueden opcionalmente almacenar el HTML junto con el archivo de diseñador (.dsr). Si desea agregar estos archivos al control de código fuente, necesita comprobar la casilla **Guardar HTML como parte del proyecto VB**, en la página de propiedades del Diseñador de páginas.

Para obtener más información Los diseñadores se presentan en "Diseñadores ActiveX". Para agregar un diseñador al menú **Proyecto**, vea "Agregar un diseñador ActiveX al menú Proyecto", anteriormente en este mismo capítulo.