Objects are central to Visual Basic programming. Forms and controls are objects. Databases are objects. There are objects everywhere you look.

If you've used Visual Basic for a while, or if you've worked through the examples in the first five chapters of this book, then you've already programmed with objects — but there's a lot more to objects than what you've seen so far.

In this chapter, user-defined types will take on personalities of their own, and become classes. You'll see how easy it is to create your own objects from the classes you define, and to use objects to simplify your coding and increase code reuse.

**Note** If you are using the Control Creation Edition of Visual Basic, some of the material covered in this document may not be applicable. With the full editions of Visual Basic you have the ability to create applications, ActiveX documents, and other types of components. Although some of the terminology may relate to application specific objects such as forms, in most cases the underlying concepts also apply to ActiveX control creation.

1

# Contents

The following topics introduce the possibilities opened by programming with objects.


- What You Need to Know About Objects in Visual Basic
- Finding Out About Objects
- Creating Your Own Classes
- Adding Properties and Methods to a Class
- Adding Events to a Class
- Naming Properties, Methods, and Events
- Polymorphism
- Programming with Your Own Objects
- Object Models
- Creating Your Own Collection Classes


2

# Sample Application: ProgWOb.vbg

Some of the code examples in this chapter are taken from the Programming with Objects (ProgWOb.vbg) sample. You'll find this application in the \ProgWOb subdirectory of the Visual Basic samples directory (\Vb\Samples\Pguide).




Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: .
Revision #: 7 Page: 1 of 96 Printed: 09/16/96 04:48 PM

# What You Need to Know About Objects in Visual Basic

Visual Basic makes using objects easy, but more importantly it makes possible a gradual transition between procedural coding and programming with objects.

- Of course, it helps that you've been using objects for as long as you've been using Visual Basic.

# The One-Minute Terminologist

The following is a whirlwind tour of terms you'll meet in discussions of Visual Basic objects and their capabilities. If you're coming to Visual Basic from another programming language, or from having worked with ActiveX (formerly OLE) terminology, this topic will help you make the transition.

If you're new to objects, you may find it all a little bewildering. That's okay — by taking a quick tour of the terms you're going to meet, you'll start forming a picture of how they fit together. As you discover more about objects in the rest of this chapter, you can return to this topic to integrate each piece of information into the whole.

### Here Goes

Objects are *encapsulated* — that is, they contain both their code and their data, making them more easier to maintain than traditional ways of writing code.

Visual Basic objects have *properties, methods,* and *events.* Properties are data that describe an object. Methods are things you can tell the object to do. Events are things the object does; you can write code to be executed when events occur.

Objects in Visual Basic are created from *classes;* thus an object is said to be *an instance of a class.* The class defines an object's interfaces, whether the object is public, and under what circumstances it can be created. Descriptions of classes are stored in *type libraries,* and can be viewed with *object browsers.*

To use an object, you must keep a *reference* to it in an *object variable.* The type of *binding* determines the speed with which an object's methods are accessed using the object variable. An object variable can be *late bound* (slowest), or *early bound*. Early-bound variables can be *DispID bound* or *vtable bound* (fastest).

A set of properties and methods is called an *interface.* The default interface of a Visual Basic object is a *dual interface* which supports all three forms of binding. If an object variable is *strongly typed* (that is, Dim … As *classname*), it will use the fastest form of binding.

In addition to their default interface, Visual Basic objects can implement extra interfaces to provide *polymorphism*. Polymorphism lets you manipulate many

different kinds of objects without worrying about what kind each one is. *Multiple interfaces* are a feature of the Component Object Model (COM); they allow you to evolve your programs over time, adding new functionality without breaking old code.

### On to Symphony Hall

Whew! If all of that seemed like old hat to you, you'll cruise through the rest of this chapter. If not, don't worry — there are strategically located explanations of all these terms sprinkled through the text (and presented at a much less frenetic pace).

# Performing Multiple Actions on an Object

You often need to perform several different actions on the same object. For example, you might need to set several properties for the same object. One way to do this is to use several statements.

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

3

Notice that all these statements use the same object variable, Command1. You can make this code easier to write, easier to read, and more efficient to run by using the With...End With statement.

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

4

You can also nest With statements by placing one With...End With statement inside another With...End With statement.

# Using Default Properties

Many objects have *default properties*. You can use default properties to simplify your code, because you don't have to refer explicitly to the property when setting its value. For an object where Value is the default property, these two statements are equivalent:

```
object = 20
```

5

and



Filename: in.doc     Project: A Traditional User's Guide Template
Template:     Author: Ruth Silverio     Last Saved By: .
Revision #: 7     Page: 3 of 96     Printed: 09/16/96 04:48 PM

```
object.Value = 20
```

6

To see how this works, draw a command button and a text box on a form. Add the following statement to the command button's Click event:

```
Text1 = "hello"
```

7

Run the application and click the command button. Because Text is the default property of the text box, the text box will display the text, "hello."

# Using Default Properties with Object Variables

When a reference to an object is stored in an object variable, you can still use the default property. The following code fragment demonstrates this.

```
Private Sub Command1_Click()
    Dim obj As Object
    ' Place a reference to Text1 in the object
    '     variable.
    Set obj = Text1
    ' Set the value of the default property (Text).
    obj = "hello"
End Sub
```

8

In the code above, `obj = "hello"` is exactly the same as typing `obj.Text = "hello"`.

# Using Default Properties with Variants

Accessing default properties is different when an object reference is stored in a variable of type Variant, instead of in an object variable. This is because a Variant can contain data of many different types.

For example, you can read the default property of Text1 using a reference in a Variant, but trying to assign the string "goodbye" to the default property doesn't work. Instead, it replaces the object reference with the string, and changes the Variant type.

To see how this works, enter the following code in the Click event of the command button from the previous example:

```
Private Sub Command1_Click()
    Dim vnt As Variant
    ' Set the default property (Text) to "hello".
    Text1 = "hello"
    ' Place a reference to Text1 in the Variant.
    Set vnt = Text1
    ' Display the default property of Text1, and show
    '     that the Variant contains an object reference.
    MsgBox vnt, , "IsObject? " & IsObject(vnt)
    ' Attempt to set the default property of Text1.
    vnt = "goodbye"
    MsgBox vnt, , "IsObject? " & IsObject(vnt)
End Sub
```

When you run the application and click the command button, you first get a message box displaying the current value of the default property of Text1, “hello,” which you can verify by looking at Text1. The caption of the message box confirms that the Variant contains an object reference — that is, a reference to Text1.

When you click the OK button on the message box, “goodbye” is assigned to the Variant, destroying the reference to Text1. Another message box is then displayed, showing the contents of the Variant — which as you can see doesn’t match the current value of Text1.Text.

The caption of the message box confirms that the Variant no longer contains an object reference — it now contains the string “goodbye.”

**For More Information** For details on Variants and other data types, see “Introduction to Variables, Constants, and Data Types” in “Programming Fundamentals.”

Other aspects of using objects with Variants are discussed in “The Visual Basic Collection Object.”

10

# Creating Arrays of Objects

You can declare and use arrays of an object type just as you declare and use an array of any data type. These arrays can be fixed-size or dynamic.

## Arrays of Form Variables

You can declare an array of forms with Private, Dim, ReDim, Static, or Public in the same way you declare an array of any other type. If you declare the array with the New keyword, Visual Basic automatically creates a new instance of the form for each element in the array as you use the elements in the array.

```
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
        ' To create minimized forms without having them
        ' first appear briefly at normal size, reverse
        ' the order of the two lines above.
    Next
End Sub
```

11

Pressing the command button to execute the code above will create five minimized instances of Form1.

**Note** If you look at the Task Bar, you’ll see Form1 *six* times. The extra instance of Form1 isn’t minimized — it’s the one you started with.

12

# Arrays of Control Variables

You can declare an array of controls with Private, Dim, ReDim, Static, or Public in the same way you declare an array of any other type. Unlike form arrays, however, control arrays cannot be declared with the New keyword. For example, you can declare an array to be a specific control type:

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

13

When you declare an array to be a particular control type, you can assign only controls of that type to the array. In the case of the preceding declaration, for example, you can only assign image controls to the array — but those image controls can come from different forms.

Contrast this with the built-in Controls collection, which can contain many different types of controls — all which must be on the same form.

Alternatively, you can declare an array of generic control variables. For example, you might want to keep track of every control that was dropped onto a particular control, and not allow any control to be dropped more than once. You can do this by maintaining a dynamic array of control variables that contains references to each control that has been dropped:

```
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
        ' If the dropped control is in the array, it's
        ' already been dropped here once.
        If ctlDropped(intX) Is Source Then
            Beep
            Exit Sub
        End If
    Next
    ' Enlarge the array.
    intSize = intSize + 1
    ReDim Preserve ctlDropped(intSize)
    ' Save a reference to the control that was dropped.
    Set ctlDropped(intSize) = Source
    ' Add the name of the control to the list box.
    List1.AddItem Source.Name
End Sub
```

14

This example uses the Is operator to compare the variables in the control array with the control argument. The Is operator can be used to test the identity of Visual Basic object references: If you compare two different references to the same object, the Is operator returns True.

The example also uses the Set statement to assign the object reference in the Source argument to an element in the array.

**For More Information** Arrays are introduced in "Arrays" and "Dynamic Arrays" in "Programming Fundamentals." For an easier way to keep track of objects, see "Creating Collections of Objects" later in this chapter.

15

# Creating Collections of Objects

Collections provide a useful way to keep track of objects. Unlike arrays, Collection objects don't have to be re-dimensioned as you add and remove members.

For example, you might want to keep track of every control that was dropped onto a particular control, and not allow any control to be dropped more than once. You can do this by maintaining a Collection that contains references to each control that has been dropped:

```
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
        ' If the dropped control is in the collection,
        ' it's already been dropped here once.
        If vnt Is Source Then
            Beep
            Exit Sub
        End If
    Next
    ' Save a reference to the control that was dropped.
    colDroppedControls.Add Source
    ' Add the name of the control to the list box.
    List1.AddItem Source.Name
End Sub
```

16

This example uses the Is operator to compare the object references in the colDroppedControls collection with the event argument containing the reference to the dropped control. The Is operator can be used to test the identity of Visual Basic object references: If you compare two different references to the same object, the Is operator returns True.

The example also uses the Add method of the Collection object to place a reference to the dropped control in the collection.

Unlike arrays, Collections are objects themselves. The variable colDroppedControls is declared As New, so that an instance of the Collection class will be created the first time the variable is referred to in code. The variable is also declared Static, so that the Collection object will not be destroyed when the event procedure ends.

**For More Information** Properties and methods of the Collection object are discussed in "The Visual Basic Collection Object" later in this chapter. To compare the code above with the code required to use arrays, see "Creating Arrays of Objects," earlier in this chapter. To learn how to create more robust collections by

wrapping the Collection object in your own collection class, see “Creating Your Own Collection Classes” later in this chapter. “What You Need to Know About Objects in Visual Basic,” earlier in this chapter, describes how objects are created and destroyed.

17

# The Visual Basic Collection Object

A collection is a way of grouping a set of related items. Collections are used in Visual Basic to keep track of many things, such as the loaded forms in your program (the Forms collection), or all the controls on a form (the Controls collection).

Visual Basic provides the generic Collection class to give you the ability to define your own collections. You can create as many Collection objects — that is, instances of the Collection class — as you need. You can use Collection objects as the basis for your own collection classes and object models, as discussed in “Creating Your Own Collection Classes” and “Object Models” later in this chapter.

## What’s a Collection Object Made Of?

A Collection object stores each item in a Variant. Thus the list of things you can add to a Collection object is the same as the list of things that can be stored in a Variant. This include standard data types, objects, and arrays — but not user-defined types.

Variants always take up 16 bytes, no matter what’s stored in them, so using a Collection object is not as efficient as using arrays. However, you never have to ReDim a Collection object, which results in much cleaner, more maintainable code. In addition, Collection objects have extremely fast look-ups by key, which arrays do not.

**Note** To be precise, a Variant always takes up 16 bytes *even if the data are actually stored elsewhere.* For example, if you assign a string or an array to a Variant, the Variant contains a pointer to a copy of the string or array data. Only 4 bytes of the Variant is used for the pointer on 32-bit systems, and none of the data is actually inside the Variant.

If you store an object, the Variant contains the object reference, just as an object variable would. As with strings and arrays, only 4 bytes of the Variant are being used.

Numeric data types are stored inside the Variant. Regardless of the data type, the Variant still takes up 16 bytes.

Despite the size of Variants, there will be many cases where it makes sense to use a Collection object to store all of the data types listed above. Just be aware of the tradeoff you’re making: Collection objects allow you to write very clean, maintainable code — at the cost of storing items in Variants.

18

# Properties and Methods of the Collection Object

Each Collection object comes with properties and methods you can use to insert, delete, and retrieve the items in the collection.

| Property or method | Description |
| --- | --- |
| Add method | Add items to the collection. |
| Count property | Return the number of items in the collection. Read-only. |
| Item method | Return an item, by index or by key. |
| Remove method | Delete an item from the collection, by index or by key. |

19

These properties and methods provide only the most basic services for collections. For example, the Add method cannot check the type of object being added to a collection, to ensure that the collection contains only one kind of object. You can provide more robust functionality — and additional properties, methods, and events — by creating your own collection class, as described in “Creating Your Own Collection Classes” later in this chapter.

The basic services of adding, deleting, and retrieving from a collection depend on keys and indexes. A *key* is String value. It could be a name, a driver’s license number, a social security number, or simply an Integer converted to a String. The Add method allows you to associate a key with an item, as described later in this section.

An *index* is a Long between one (1) and the number of items in the collection. You can control the initial value of an item’s index, using the ***before*** and ***after*** named parameters, but its value may change as other items are added and deleted.

**Note** A collection whose index begins at 1 is called *one-based*, as explained in “Collections in Visual Basic.”

20

You can use the index to iterate over the items in a collection. For example, the following code shows two ways to give all the employees in a collection of Employee objects a 10 percent raise, assuming that the variable colEmployees contains a reference to a Collection object.

```
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
```

21



Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: .
Revision #: 7 Page: 9 of 96 Printed: 09/16/96 04:48 PM

**Tip** For better performance, use For Each to iterate over the items in a Collection object. For Each is significantly faster than iterating with the index. This is not true of all collection implementations — it's dependent on the way the collection stores data internally.

22

# Adding Items to a Collection

Use the Add method to add an item to a collection. The syntax is:

**Sub Add (*item* As Variant [, *key* As Variant] [, *before* As Variant] [, *after* As Variant] )**

23

For example, to add a work order object to a collection of work orders using the work order's ID property as the key, you can write:

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

24

This assumes that the ID property is a String. If the property is a number (for example, a Long), use the CStr function to convert it to the String value required for keys:

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

25

The Add method supports named arguments. To add an item as the third element, you can write:

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

26

You can use the ***before*** and ***after*** named arguments to maintain an ordered collection of objects. For example, `before:=1` inserts an item at the beginning of the collection, because Collection objects are one-based.

# Deleting Items from a Collection

Use the Remove method to delete an item from a collection. The syntax is:

*object*.**Remove** *index*

27

The *index* argument can either be the position of the item you want to delete, or the item's key. If the key of the third element in a collection is "W017493," you can use either of these two statements to delete it:

```
colWorkOrders.Remove 3
```

1– or –

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

28

# Retrieving Items from a Collection

Use the Item method to retrieve specific items from a collection. The syntax is:

**[Set]** *variable* = *object*.**Item(***index***)**

29

As with the Remove method, the index can be either the position in the collection, or the item's key. Using the same example as for the Remove method, either of these statements will retrieve the third element in the collection:

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

2– or –

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

30

If you use whole numbers as keys, you must use the CStr function to convert them to strings before passing them to the Item or Remove methods. A Collection object always assumes that a whole number is an index.

**Tip** Don't let Collection objects decide whether a value you're passing is an index or a key. If you want a value to be interpreted as a key, and the variable that contains the value is anything but String, use CStr to convert it. If you want a value to be interpreted as an index, and the variable that contains the value is not one of the integer data types, use CLng to convert it.

31

## Item Is the Default Method

The Item method is the default method for a Collection object, so you can omit it when you access an item in a collection. Thus the previous code example could also be written:

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

3– or –

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

32

**Important** Collection objects maintain their numeric index numbers automatically as you add and delete elements. The numeric index of a given element will thus change over time. Do not save a numeric index value and expect it to retrieve the same element later in your program. Use keys for this purpose.

33

## Using the Item Method to Invoke Properties and Methods

You don't have to retrieve an object reference from a collection and place it in an object variable in order to use it. You can use the reference while it's still in the collection.

For example, suppose the WorkOrder object in the code above has a Priority property. The following statements will both set the priority of a work order:

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

34

The reason this works is that Visual Basic evaluates the expression from left to right. When it comes to the Item method — explicit or implied — Visual Basic gets a

reference to the indicated item (in this case, the WorkOrder object whose key is W017493), and uses this reference to evaluate the rest of the line.

**Tip** If you're going to invoke more than one property or method of an object in a collection, copy the object reference to a strongly typed object variable first. Using an object reference while it's still in a collection is slower than using it after placing it in a strongly typed object variable (for example, Dim woCurrent As WorkOrder), because the Collection object stores items in Variants. Object references in Variants are always late bound.

35

**For More Information** The Collection object is also a useful alternative to arrays for many ordinary programming tasks. See "Using Collections as an Alternative to Arrays" in "More About Programming." Visual Basic provides a number of built-in collections. To compare them with the Collection object, see "Collections in Visual Basic."

36

# Collections in Visual Basic

What is a collection? In "The Visual Basic Collection Object," a collection was defined as a way of grouping related objects. That leaves a lot of room for interpretation; it's more of a concept than a definition.

In fact, as you'll see when you begin comparing collections, there are a lot of differences even among the kinds of collections provided in Visual Basic. For example, the following code causes an error:

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

37

What's happening here? The Forms collection is a collection; the variable col is declared As Collection; why can't you assign a reference to Forms to the variable col?

The reason for this is that the Collection class and the Forms collection are not *polymorphic*; that is, you can't exchange one for the other, because they were developed from separate code bases. They don't have the same methods, store object references in the same way, or use the same kinds of index values.

This makes the Collection class's name seem like an odd choice, because it really represents only one of many possible collection implementations. This topic explores some of the implementation differences you'll encounter.

## Zero-Based and One-Based Collections

A collection is either *zero-based* or *one-based*, depending on what its starting index is. As you might guess, the former means that the index of the first item in the collection is zero, and the latter means it's one. Examples of zero-based collections

are the Forms and Controls collections. The Collection object is an example of a one-based collection.

Older collections in Visual Basic are more likely to be zero-based, while more recent additions are more likely to be one-based. One-based collections are somewhat more intuitive to use, because the index ranges from one to Count, where Count is the property that returns the number of items in a collection.

The index of a zero-based collection, by contrast, ranges from zero to one less than the Count property.

## Index and Key Values

Many collections in Visual Basic allow you to access an item using either a numeric index or a string key, as the Visual Basic Collection object does. (Visual Basic's Collection object allows you to add items without specifying a key, however.)

The Forms collection, by contrast, allows only a numeric index. This is because there's no unique string value associated with a form. For example, you can have multiple forms with the same caption, or multiple loaded forms with the same Name property.

## Adding and Removing Items

Collections also differ in whether or not you can add items to them, and if so, how those items are added. You can't add a printer to the Printers collection using Visual Basic code, for example.

Because the Collection object is a general-purpose programming tool, it's more flexible than other collections. It has an Add method you can use to put items into the collection, and a Remove method for taking items out.

By contrast, the only way to get a form into the Forms collection is to load the form. If you create a form with the New operator, or by referring to a variable declared As New, it will not be added to the Forms collection until you use the Load statement to load it.

The Forms and Controls collections don't have Remove methods. You add and remove forms and controls from these collections indirectly, by using the Load and Unload statements.

## What Has It Got In Its Pocketses?

As noted above, a form is not added to the Forms collection until it's loaded. Thus the most accurate specification of the Forms collection is that it contains *all of the currently loaded forms in the program.*

Even that's not completely accurate. If your project uses Microsoft Forms (included for compatibility with Microsoft Office), you'll find those forms in a separate



Filename: in.doc     Project: A Traditional User's Guide Template
Template:     Author: Ruth Silverio     Last Saved By: .
Revision #: 7     Page: 13 of 96     Printed: 09/16/96 04:48 PM

collection named UserForms. So the Forms collection contains *all of the currently loaded Visual Basic forms in the program.*

The contents of the Collection class are very precisely specified: *anything that can be stored in a* Variant. Thus the Collection object can contain an object or an integer, but not a user-defined type.

Unfortunately, this specification covers a lot of territory — a given instance of the Collection class could store any mongrel assortment of data types, arrays, and objects.

**Tip** One of the most important reasons for creating your own collection classes, as discussed in “Creating Your Own Collection Classes,” is so you can control the contents of your collections — a concept called *type safety*.

38

# Enumerating a Collection

You can use For Each … Next to enumerate the items in a collection, without worrying about whether the collection is zero-based or one-based. Of course, this is hardly a defining characteristic of collections, because Visual Basic allows you to use For Each … Next to enumerate the items in an array.

What makes For Each … Next work is a tiny object called an *enumerator*. An enumerator keeps track of where you are in a collection, and returns the next item when it’s needed.

When you enumerate an array, Visual Basic creates an array enumerator object on the fly. Collections have their own enumerator objects, which are also created as needed.

## Enumerators Don’t Skip Items

The enumerators of collections in Visual Basic don’t skip items. For example, suppose you enumerate a collection containing “A,” “B,” and “C,” and that while doing so you remove “B.” Visual Basic collections will not skip over “C” when you do this.

## Enumerators May Not Catch Added Items

If you add items to a collection while enumerating it, some enumerators will include the added items, while some will not. The Forms collection, for example, will not enumerate any forms you load while enumerating.

The Collection object will enumerate items you add while enumerating, if you allow them to be added at the end of the collection. Thus the following loop never ends (until you hit CTRL+BREAK, that is):

```
Dim col As New Collection
Dim vnt As Variant
col.Add "Endless"
col.Add "Endless"
```

```
For Each vnt In col
    MsgBox vnt
    col.Add "Endless"
Next
```

39

On the other hand, items you add at the beginning of the collection will not be included in the enumeration:

```
Dim col As New Collection
Dim vnt As Variant
col.Add "Will be enumerated"
For Each vnt In col
    MsgBox vnt
    ' Add the item at the beginning.
    col.Add "Won't be enumerated", Before:=1
Next
```

40

## Why Enumerators?

By emitting a new enumerator each time a For Each … Next begins, a collection allows nested enumerations. For example, suppose you have a reference to a Collection object in the variable mcolStrings, and that the collection contains only strings. The following code prints all the combinations of two different strings:

```
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
```

41

**For More Information** See “Creating Your Own Collection Classes” later in this chapter.

42

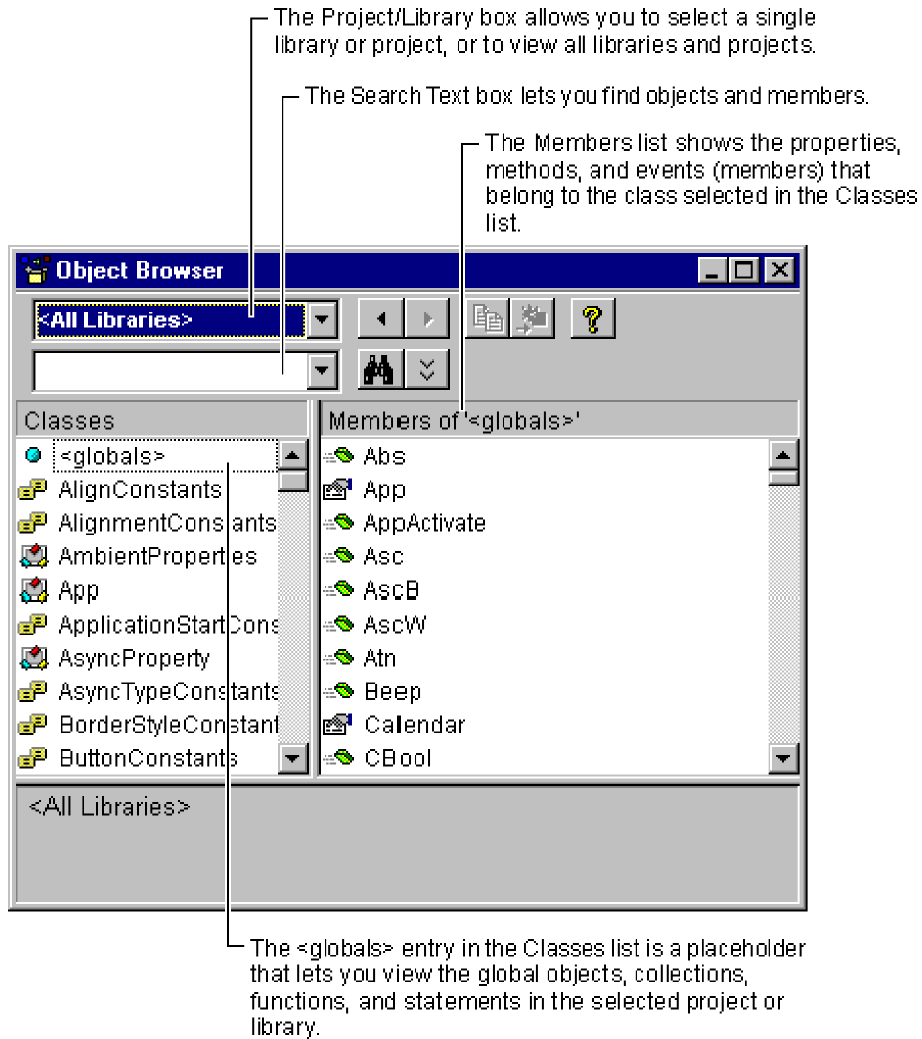
# Finding Out About Objects

The Object Browser is based on *type libraries*, resources that contain detailed descriptions of classes, including properties, methods, events, named constants, and more.

Visual Basic creates type library information for the classes you create, provides type libraries for the objects it includes, and lets you access the type libraries provided by other applications.

You can use the Object Browser to display the classes available in projects and libraries, including the classes you’ve defined. The objects you create from those classes will have the same members — properties, methods, and events — that you see in the Object Browser.

**Figure 9.1 The Object Browser**




1

## To display the Object Browser

- From the **View** menu, choose **Object Browser**.

4– or –

5Press F2.

6– or –

7Click the **Object Browser** button on the toolbar.

2

By default, the Object Browser cannot be docked to other windows. This allows you to move between the Object Browser and code windows using CTRL+TAB. You can change this by right-clicking the Object Browser to open its context menu, and clicking Dockable.

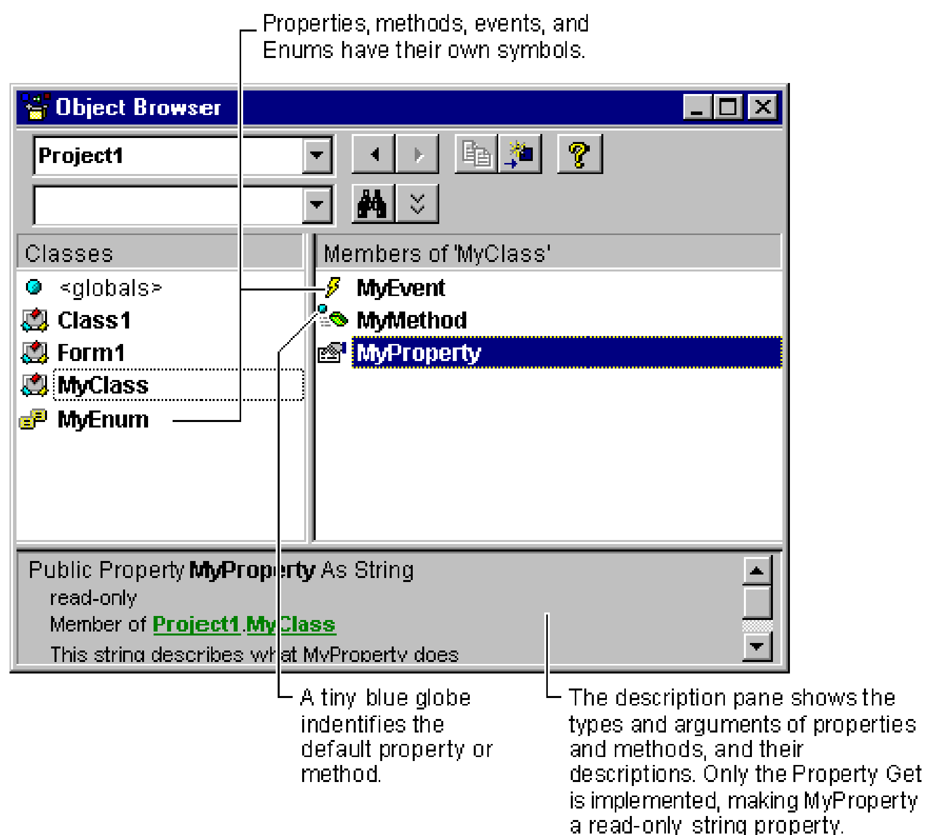
**Note** When the Object Browser is dockable, you cannot use CTRL+TAB to move to it from your code windows.

43

# Contents of the Object Browser

The Object Browser displays information in a three-level hierarchy, as shown in Figure 9.2. Beginning from the top, you can select from available projects and libraries, including your own Visual Basic projects, using the Project/Library box.

**Figure 9.2 Viewing a class's members in the Object Browser**




3

- Click on a class in the Classes list to view its description in the description pane at the bottom. The class's properties, methods, events, and constants will appear in the Members list on the right. The classes available are drawn from

the project or library selected in the Project/Library box, or from all projects and libraries if <All Libraries> is selected.

- You can view the arguments and return values of a member of the selected class, by clicking on the member in the Members list. The description pane at the bottom of the Object Browser shows this information.
- You can jump to the library or object that includes a member by clicking the library or object name in the description pane. You can return by clicking the Go Back button at the top of the Object Browser.

4

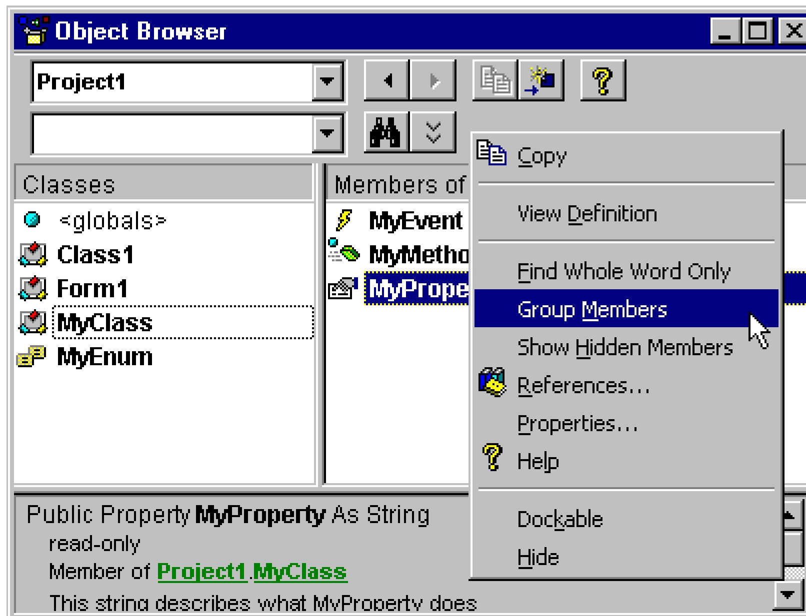
**Tip** When you're in either the Classes list or the Members list, typing the first character of a name will move to the next name that begins with that character.

44

# Controlling the Contents of the Object Browser

The context menu, shown in Figure 9.3, provides an alternative to the Copy and View Definition buttons on the Object Browser. It also allows you to open the References dialog box, and — if a class or member is selected — to view the properties of the selected item. You can set descriptions for your own objects using this menu item, as described in "Adding Descriptions for Your Objects."

**Figure 9.3 The Object Browser's context menu**




5



Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: .
Revision #: 7 Page: 18 of 96 Printed: 09/16/96 04:48 PM

Right-clicking on the Object Browser brings up the context menu. In addition to the functions mentioned above, the context menu controls the contents of the Classes list and the Members list.

- When Group Members is checked, all the properties of an object are grouped together, all the methods are grouped together, and so on. When Group Members is not checked, the Members list is alphabetical.
- When Show Hidden Members is checked, the Class list and Members list display information marked as hidden in the type library. Normally you don't need to see this information. Hidden members are shown in light gray type.

6

**Tip** When Group Members is selected, typing the first letter of a name will jump to the next name that begins with that character, even if the name is in another group.

45

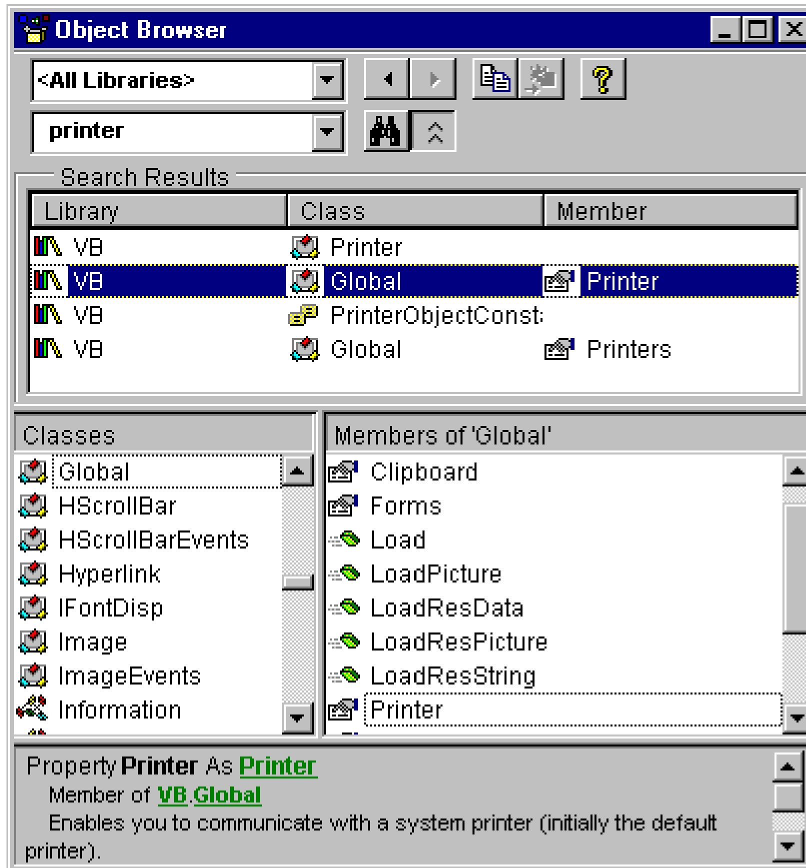
# Finding and Browsing Objects

You can use the Object Browser to find objects and their members, and to identify the projects or libraries they come from.

Enter text in the Search Text box and then click the Search button (or press ENTER). The classes and members whose names include the text you specified will appear in the Search Results list.

For example, Figure 9.4 shows the results of typing "printer" in the Search Text box and clicking the Search button.

**Figure 9.4   Using the Search button**




7

You can select an item in the Search Results list, and view its description in the description pane at the bottom of the Object Browser. Clicking on the underlined jumps in the description pane selects the indicated library or navigates to the object or member.

You can restrict the search to items that exactly match the string in the Search box by checking Find Whole Word Only on the context menu.

# Adding Descriptions for Your Objects

You can use the Object Browser to add descriptions and HelpContextIDs to your own procedures, modules, classes, properties, and methods. You may find these descriptions useful while working with your classes.

**Note** You can also enter descriptions for properties, methods, and events using the Procedure Attributes dialog box, accessed from the Tools menu.

46

### To enter description strings and link your classes and their members to Help topics

1 Press F2 to open the **Object Browser**. In the **Project/Library** box, select your project.

2 In the **Classes** list, right click the name of a class to bring up the context menu, and click **Properties** to open the **Member Options** dialog box.

8Alternatively, in the **Members** list you can right click the name of a property, method, or event you added to the class. On the context menu, click **Properties**. If the member is Private or Friend, this will open the **Member Options** dialog box. If the member is Public — that is, part of the class's interface — it will open the **Procedure Attributes** dialog box instead.

1**Note** The difference between these two dialog boxes is that the **Procedure Attributes** dialog box has an **Advanced** button that can be used to make a member the default for the class, as described in "Making a Property or Method the Default" later in this chapter.

8

3 In the **Help Context ID** box, type the context ID of the Help topic to be shown if you click the "?" button when this class or member is selected in the **Object Browser**.

2**Note** You can create a Help file for your own use, and link topics to your classes and their members. To specify a Help file for your project, use the **General** tab of the **Project Properties** dialog box, accessed from the **Project** menu.

47

4 In the **Description** box, type a brief description of the class or member.

5 Click **OK** to return to the **Object Browser**. The description string you entered should appear in the description pane at the bottom of the browser.

6 Repeat steps 2 through 5 for each class and for each member of each class.

9

**Note** You cannot supply browser strings or Help topics for enumerations.

48

**For More Information** Enumerations are introduced in "Using Enumerations to Work with Sets of Constants" in "More About Programming."

49

## Moving Between Procedures

You can use the Object Browser to move quickly to the code for a class, module, or procedure in your project.

### To move to a class, module, or procedure

7 (Optional) Select your project from the **Project/Library** box.

9Step 1 is optional if you have **<All Libraries>** selected in the **Project/Library** box, because all of your projects are included.

8 Names of classes, modules, and members that belong to your projects are shown in bold type. Double-click any name shown in bold type to move to that class, module, or member. (Or right-click a name and then select **View Definition** from the context window.)

10The selected item is displayed in the **Code** window.

10

## Browsing Objects from Other Applications

From within Visual Basic, you can access and control objects supplied by other applications. For example, if you have Microsoft Project and Microsoft Excel on your system, you could use a Graph object from Microsoft Excel and a Calendar object from Microsoft Project as part of your application.

You can use the Object Browser to explore the type libraries of other applications. An *type library* provides information about the objects provided by other applications.

**Note** In the Project/Library list, there are separate entries for Visual Basic (VB) and Visual Basic for Applications (VBA). Although we speak of “objects provided by Visual Basic,” you’ll notice that the Collection object is provided by VBA.

50

You can add libraries to your project by selecting References from the Object Browser’s context menu, to open the References dialog box.

# Creating Your Own Classes

If you’re an experienced programmer, you already have a library of useful functions you’ve written over the years. Objects don’t replace functions — you’ll still write and use utility functions — but they provide a convenient, logical way to organize procedures and data.

- In particular, the classes from which you create objects combine data and procedures into a unit.

# Classes: Putting User-Defined Types and Procedures Together

User-defined types are a powerful tool for grouping related items of data. Consider, for example, the user-defined type named udtAccount defined here:

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

51

You can declare a variable of type udtAccount, set the values of its fields individually, and then pass the whole record to procedures that print it, save it to a database, perform computations on it, validate its fields, and so on.

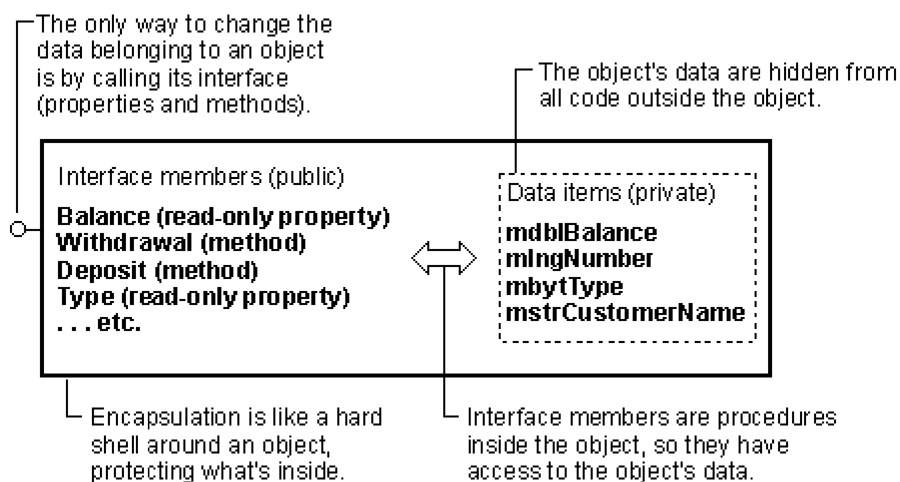
Powerful as they are, user-defined types present the programmer with some problems. You may create a Withdrawal procedure that raises an error if a withdrawal exceeds the balance in the account, but there's nothing to prevent the Balance field from being reduced by other code in your program.

In other words, the connection between procedures and user-defined types depends on the discipline, memory, and knowledge of the programmer maintaining the code.

## Objects: User-Defined Types with an Attitude

Object-oriented programming solves this problem by combining data and procedures in a single entity, as shown in Figure 9.5.

**Figure 9.5 Objects combine data and procedures**



11

When the user-defined type udtAccount becomes the Account class, its data become private, and the procedures that access them move inside the class and become properties and methods. This is what's meant by the term *encapsulation* — that is, an object is a unit (a *capsule*, if you will) containing both code and data.

When you create an Account object from the class, the only way you can access its data is through the properties and methods that make up its interface. The following code fragment shows how the procedures inside the Account class support encapsulation:

```
' The account balance is hidden from outside code.
Private mdblBalance As Double

' The read-only Balance property allows outside code
' to find out the account balance.
Public Property Get Balance() As Double
    Balance = mdblBalance
End Property

' The Withdrawal method changes the account balance,
' but only if an overdraft error doesn't occur.
Public Sub Withdrawal(ByVal Amount As Double)
    If Amount > Balance Then
        Err.Raise Number:=vbObjectError + 2081, _
        Description:="Overdraft"
    End If
    mdblBalance = mdblBalance - Amount
End Sub
```

52

For the moment, don't worry about how you get the procedures inside the class, or about understanding the syntax of property procedures and private variables. The important thing to remember is that you can define an object that encapsulates and validates its own data.

With the Account object, you never have be concerned about whether you've called the right procedures to update the account, because the only procedures you can call are built into the object.

**For More Information** "Customizing Form Classes" puts property and method creation into a framework you're already familiar with. Later, "Adding Properties and Methods to a Class" will explain the syntax. You can read about user-defined types in "Creating Your Own Data Types" in "More About Programming." For details about Sub and Function procedures, see "Introduction to Procedures" in "Programming Fundamentals."

53

# Customizing Form Classes

It may surprise you to learn that you've been creating classes for as long as you've been programming in Visual Basic. It's true: Form1, that familiar denizen of every project you've ever started, is really — a class.

To see this, open a new Standard Exe project. Add a button to Form1, and place the following code in its Click event:

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

54

Press F5 to run the project, and click the button. Holy smokes, there's another instance of Form1! Click its button. There's another! Every instance you create looks the same, and has the same behavior, because they're all instances of the Form1 class.

### What's Going On Here?

If you've read "Working with Objects" in "Programming Fundamentals," you know that an object variable declared As New contains Nothing until the first time you refer to it in code. When you use the variable for the first time, Visual Basic notices that it contains the special value Nothing, and creates an instance of the class. (And a good thing it does, too, or f.Show would cause an error.)

### Me and My Hidden Global Variable

You may be wondering how it is that you can refer to Form1 in code, as if it were an object variable. There's no magic involved. Visual Basic creates a hidden global object variable for every form class. It's as if Visual Basic had added the following declaration to your project:

```
Public Form1 As New Form1
```

55

When you select Form1 as your startup object, or type Form1.Show in code, you're referring to this hidden global object variable. Because it's declared As New, an instance of the Form1 class is created the first time you use this predeclared variable in code.

The reason this declaration is hidden is that Visual Basic changes it every time you change the Name property of a form. In this way, the hidden variable always has the same name as the form class.

### A Very Short Quiz

Which of the instances of Form1 you created in the exercise above was associated with the hidden global variable? If you guessed the first one, you're right. Form1 is the default startup object for the project, and to Visual Basic that's just like using the predeclared global variable Form1 in code.

**Tip** After you unload a form, you should always set any references to the form to Nothing in order to free the memory and resources the form was using. The reference most often overlooked is the hidden global form variable.

56

### What About All Those Other Instances of Form1?

In "Programming Fundamentals," you learned that to refer to an object, you need an object variable, and that an object exists only as long as there's at least one object variable containing a reference to it. So what was keeping all those other instances alive?

The second instance of Form1, and all the ones that followed, had an object variable for just as long as it took to call their Show methods. Then that variable went out of scope, and was set to Nothing. But Visual Basic keeps a special collection named Forms. The Forms collection contains a reference to each of the loaded forms in your project, so that you can always find and control them.

**Note** As you'll learn, this is not true of all classes. For example, the classes you design won't have hidden global variables or global collections to keep track of them — those are special features of form classes. However, you can declare your own global variables, and you can create your own collections — as described in "Creating Your Own Collection Classes."

57

## Properties, Methods, and Events of Form Classes

The first time you added a property to a form class, you probably did it visually, by dropping a command button (or some other control) on Form1. In doing so, you added a read-only Command1 property to the form class. Thereafter, you invoked this property of Form1 whenever you needed to call a method or property of the command button:

```
Command1.Caption = "Click Me"
```

58

When you changed the Name property of any control on a form, Visual Basic quietly changed the name of the read-only property, so they always matched.

If you still have the project open from the earlier exercise, you can see this Command1 property by pressing F2 to open the Object Browser. In the Project/Library box, select Project1. You'll see Form1 in the Classes pane. In the Members pane, scroll down until you find Command1, and select it.

Command1 has a property symbol beside it, and if you look in the description pane, you'll see that it's a WithEvents property. As you'll learn in "Adding Events to Classes," this means that the property (or object variable) has event procedures associated with it. One of those event procedures, Command1_Click(), may have been the first place you ever wrote Visual Basic code.

### But Wait, There's More

Dropping controls on a form is not the only way to add new members to the form class. You can add your own custom properties, methods, and events, as easily as you create new variables and procedures.

To see this, add the following code to the Declarations section of Form1:

```
' The Comment property of the Form1 class.
Public Comment As String
```

59

Add the following code to the Click event of Form1:

```
Private Sub Form_Click()
    MsgBox Comment, , "My comment is:"
End Sub
```

60

Finally, change the code in the Command1_Click() event procedure by adding a line, as follows:

```
Private Sub Command1.Click()
    Dim f As New Form1
    f.Comment = InputBox("What's my comment?")
    f.Show
End Sub
```

61

Press F5 to run the project. Click Command1, and when the input box appears, type in some racy comment and click OK. When the new instance of Form1 appears, click on it to play back its Comment property.

Click on the first instance of Form1, and notice that its Comment property is blank. Because Visual Basic created this instance as the Startup Object, you never got a chance to set its Comment property.

### Forms Can Call Each Other's Methods

If you were watching closely, you may have noticed that the code you added to the Form1 class didn't set the object's own Comment property — it set the Comment property of the *new* instance of Form1 it was creating.

This ability of forms to set each other's properties and call each other's methods is a very useful technique. For example, when an MDIForm is opening a new child window, it can initialize the new window by setting its properties and calling its methods.

You can also use this technique to pass information between forms.

**Tip** You can create custom events for forms. "Adding an Event to a Form" later in this chapter, provides a step by step procedure.

62

### Other Kinds of Modules

You add properties, methods, and events to form classes by putting code in their code modules. In the same way, you can add properties, methods, and events to class modules and — if you have the Professional or Enterprise Edition of Visual Basic — to UserControl and UserDocument code modules.

As you read "Adding Properties and Methods to a Class" and "Adding Events to a Class," remember that everything you read applies to form classes as well as to class modules.

**For More Information** What the heck is a class module? "Class Module Step by Step" shows how to define a class and illustrates the life cycle of the objects you create from that class.

63

# Class Module Step by Step

This example shows how you can use class modules to define classes, from which you can then create objects. It will also show you how to create properties and methods for the new class, and demonstrate how objects are created and destroyed.

Open a new Standard Exe project, and insert a class module by selecting Add Class Module from the Project menu. Draw four command buttons on the form. The following table lists the property values you need to set for the objects in this example.

| Object | Property | Setting |
|---|---|---|
| Class module | Name | Thing |
| Command1 | Caption | Show the Thing |
| Command2 | Caption | Reverse the Thing's Name |
| Command3 | Caption | Create New Thing |
| Command4 | Caption | Temporary Thing |

64

**Note** Class modules are saved in files with the extension .cls.

65

In the class module Declarations section, add the following:

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

66

The variable Name will be a property of the Thing object, because it's declared Public.

**Note** Don't confuse this Name property with the Name property of the class module, which the table above instructed you to set. (The Name property of the class module gives the Thing class its name.) Why would you give the Thing class a Name property? A better question might be, why not? You may want to give the Thing class a Name property because Things should have names! Remember that there's nothing special about the property and method names Visual Basic uses. You can use those same property and method names for your classes.

67

The variable mdatCreated is a private data member that is used to store the value of the read-only Created property. The Created property returns the date and time a Thing object was created. To implement the Created property, add the following Property Get to the Declarations section of the class module:

```
Property Get Created() As Date
    Created = mdatCreated
End Property
```

68

**Note** If you added the property procedure using the Add Procedure dialog box, on the Tools menu, be sure to delete the Property Let declaration that is automatically added by this dialog. Property Let is only required for read-write properties, as explained in “Putting Property Procedures to Work for You.”

69

The Thing object has one method, ReverseName, which simply reverses the order of the letters in the Name property. It doesn’t return a value, so it’s implemented as a Sub procedure. Add the following Sub procedure to the class module.

```
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
```

70

Class modules have two events, Initialize and Terminate. In the Object drop down of the class module, select Class. The Procedure drop down will show the events. Place the following code in the event procedures:

```
Private Sub Class_Initialize()
    ' Set date/time of object creation, to be returned
    '     by the read-only Created property.
    mdatCreated = Now
    ' Display object properties.
    MsgBox "Name: " & Name & vbCrLf & "Created: " _
    & Created, , "Thing Initialize"
End Sub

Private Sub Class_Terminate()
    ' Display object properties.
    MsgBox "Name: " & Name & vbCrLf & "Created: " _
    & Created, , "Thing Terminate"
End Sub
```

71

Usually, the Initialize event procedure contains any code that needs to be executed at the moment the object is created, such as providing the time stamp for the Created property. The Terminate event contains any code you need to execute in order to clean up after the object when it is being destroyed.

In this example, the two events are being used primarily to give you a visual indication that a Thing object is being created or destroyed.

## Using the Thing Object

Add this declaration to the Declarations section of the form module:

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

72

The variable mth will hold a reference to a Thing object, which will be created in the form's Load event. Put the following code in the Form_Load event procedure, and in the Click event procedures for the four buttons.

```
Private Sub Form_Load()
    Set mth = New Thing
    mth.Name = InputBox("Enter a name for the Thing")
End Sub

' Button "Show the Thing"
Private Sub Command1_Click()
    MsgBox "Name: " & mth.Name & vbCrLf _
    & "Created: " & mth.Created, , "Form Thing"
End Sub

' Button "Reverse the Thing's Name"
Private Sub Command2_Click()
    mth.ReverseName
    ' Click "Show the Thing"
    Command1.Value = True
End Sub

' Button "Create New Thing"
Private Sub Command3_Click()
    Set mth = New Thing
    mth.Name = InputBox( _
    "Enter a name for the new Thing")
End Sub

' Button "Temporary Thing".
Private Sub Command4_Click()
    Dim thTemp As New Thing
    thTemp.Name = InputBox( _
    "Enter a name for the Temporary Thing")
End Sub
```

73

## Running the Project

Press F5 to run the project. Looking at the code in the Form_Load event procedure, you can see that the New operator is used to create a Thing object. A reference to this Thing is assigned to the variable mth.

You will see the InputBox asking you for a name for the Thing. When you type a name and press ENTER, the return value is assigned to the Name property of the Thing object.

### Show the Form Thing

You can verify that the Name property has been assigned by pressing the first button, "Show the Thing," which displays a message box with all the properties of the Thing object.

### Reverse the Thing's Name

Press the second button, "Reverse the Thing's Name." This button calls the ReverseName method to turn the Thing object's name around, and then clicks the first button to display the updated property values.

### Create New Thing

Click the "Create New Thing" button to destroy the existing Thing object and create a new one. (Or, as it turns out, to create a new Thing and then destroy the old one.)

The New operator causes a new Thing to be created, so you'll see the MsgBox displayed by the new Thing's Initialize event. When you click OK, a reference to the new Thing is placed in the form-level variable `mth`.

This wipes out the reference to the old Thing. Because there are no more references to it, it's destroyed, and you'll see its Terminate event message box. When you click OK, the InputBox statement requests a name for the new Thing.

**Note** If you want to destroy the old Thing before creating the new one, you can add the line of code `Set mth = Nothing` at the beginning of the event procedure.

### Temporary Thing

The fourth button demonstrates another aspect of object lifetime. When you press it, you'll be prompted for a name for the temporary Thing.

But wait — there isn't a temporary Thing object yet. You haven't seen its Initialize message box. How can you assign it a name?

Because the variable `thTemp` was declared As New, a Thing object will be created the moment one of its properties or methods is invoked. This will happen when the return value of the InputBox is assigned to the Name property. Type a name and click OK on the InputBox.

You'll now see the Thing Initialize message box, which shows you that the Name property is still blank. When you click OK to dismiss the message box, the value from the InputBox statement is finally assigned to the Name property. That's a lot of activity for one line of code.

Of course, as soon as you've done that, the Click event procedure ends, and the variable `thTemp` goes out of scope. The object reference for the temporary Thing is released, so you'll see the Thing Terminate message box. Notice that it contains the name you supplied.

Each time you click this button, another temporary Thing will be created, named, and destroyed.

### Closing the Program

Close the program by clicking the form's close button. *Do not use the End button on the toolbar.* When the program closes, Form1 is destroyed. The variable `mth` goes out of scope, and Visual Basic cleans up the reference to the Thing. There are no

remaining references to the Thing, so it's destroyed, and its Terminate event message box is displayed.

Run the program again, and this time end it using the End button on the toolbar. Notice that the Terminate message box for the Thing object is *not* displayed.

It's important to remember that ending your program with the End button, or with an End statement in your code, halts the program *immediately,* without executing the Terminate events of any objects. It's always better to shut down your program by unloading all the forms.

You may find it useful to run the example by pressing F8 to step through the code one line at a time. This is a good way to understand the order of events for object creation and destruction.

**Important** In an actual application, the Initialize and Terminate events should not contain message boxes, or any other code that allows Windows messages to be processed. In general, it's better to use Debug.Print statements when debugging object lifetimes.

74

**For More Information** Forms and controls are a bit different from other objects, as discussed in "Life Cycle of Visual Basic Forms."

You can read more about what you can do with classes and class modules in "Adding Properties and Methods to a Class" and "Adding Events to a Class."

75

# Debugging Class Modules

Debugging class modules differs slightly from debugging ordinary programs. This is because an error in a property or method of a class module always acts like a handled error. (That is, there's always a procedure on the call stack that can handle the error — namely the procedure that called the class module's property or method.)

Visual Basic compensates for this difference by providing the error-trapping option Break in Class Module, in addition to the older options Break on Unhandled Errors and Break on All Errors.

**Note** You can set the Error Trapping option on the General tab of the Options dialog box, available from the Tools menu. The option you select lasts until you close the development environment. Each time you start Visual Basic, the Error Trapping option is reset to Break in Class Module.

76

For example, suppose the class module Class1 contains the following code:

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

77

Now suppose a procedure in another class module, form, or standard module calls the member Oops:

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

78

If the error trapping option is set to Break on Unhandled Errors, execution will not stop on the zero divide. Instead, the error will be raised in the calling procedure, Command1_Click. Execution will stop on the call to the Oops method.

You could use Break on All Errors to stop in the zero divide, but Break on All Errors is a very inconvenient option for most purposes. It stops on every error, even errors for which you've written error handling code.

Break in Class Module is a compromise setting:

- Execution will not stop on class module code for which you've written an error handler.
- Execution only stops on an error that's unhandled in the class module, and therefore would be returned to the caller of the method.
- When the Visual Basic development environment is started, it defaults to Break in Class Module.
- If there are no class modules involved, Break in Class Module is exactly the same as Break on Unhandled Errors.

12

**Tip** When you hit a break point using Break in Class Module or Break on All Errors, you can step or run past the error — into your error handling code or into the code that called procedure in which the error occurred — by pressing ALT+F8 or ALT+F5.

79

**For More Information** Debugging is discussed in detail in "Debugging Your Code and Handling Errors."

80

# Life Cycle of Visual Basic Forms

Because they're visible to the user, forms and controls have a different life cycle than other objects. For example, a form will not close just because you've released all your references to it. Visual Basic maintains a global collection of all forms in your project, and only removes a form from that collection when you unload the form.

In similar fashion, Visual Basic maintains a collection of controls on each form. You can load and unload controls from control arrays, but simply releasing all references to a control is not sufficient to destroy it.

**For More Information** The Forms and Controls collections are discussed in "Collections in Visual Basic" earlier in this chapter.

81

# States a Visual Basic Form Passes Through

A Visual Basic form normally passes through four states in its lifetime:

1. Created, but not loaded.
2. Loaded, but not shown.
3. Shown.
4. Memory and resources completely reclaimed.

13

There's a fifth state a form can get into under certain circumstances: Unloaded and unreferenced while a control is still referenced.

This topic describes these states, and the transitions between them.

## Created, But Not Loaded

The beginning of this state is marked by the Initialize event. Code you place in the Form_Initialize event procedure is therefore the first code that gets executed when a form is created.

In this state, the form exists as an object, but it has no window. None of its controls exist yet. *A form always passes through this state*, although its stay there may be brief.

For example, if you execute `Form1.Show`, the form will be created, and Form_Initialize will execute; as soon as Form_Initialize is complete, the form will be loaded, which is the next state.

The same thing happens if you specify a form as your Startup Object, on the General tab of the Project Properties dialog box (which is available from the Project menu). A form specified as the Startup Object is created as soon as the project starts, and is then immediately loaded and shown.

**Note** You can cause your form to load from within Form_Initialize, by calling its Show method or by invoking its built-in properties and methods, as described below.

82

### Remaining Created, But Not Loaded

By contrast, the following code creates an instance of Form1 without advancing the form to the loaded state:

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

83

Once Form_Initialize has ended, the only procedures you can execute without forcing the form to load are Sub, Function, and Property procedures you've added to the form's code window. For example, you might add the following method to Form1:

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

84

You could call this method using the variable frm (that is, frm.ANewMethod) without forcing the form on to the next state. In similar fashion, you could call ANewMethod in order to create the form:

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

85

Because frm is declared As New, the form is not created until the first time the variable is used in code — in this case, when ANewMethod is invoked. After the code above is executed, the form remains created, but not loaded.

**Note** Executing Form1.ANewMethod, without declaring a form variable, has the same effect as the example above. As explained in "Customizing Form Classes," Visual Basic creates a hidden global variable for each form class. This variable has the same name as the class; it's as though Visual Basic had declared Public Form1 As New Form1.

86

You can execute as many custom properties and methods as you like without forcing the form to load. However, the moment you access one of the form's built-in properties, or any control on the form, the form enters the next state.

**Note** You may find it helpful to think of a form as having two parts, a code part and a visual part. Before the form is loaded, only the code part is in memory. You can call as many procedures as you like in the code part without loading the visual part of the form.

87

### The Only State All Forms Pass Through

Created, But Not Loaded is the only state *all* forms pass through. If the variable frm in the examples above is set to Nothing, as shown here, the form will be destroyed before entering the next state:

```
Dim frm As New Form1
frm.ANewMethod
Set frm = Nothing       ' Form is destroyed.
```

88

A form used in this fashion is no better than a class module, so the vast majority of forms pass on to the next state.

## Loaded, But Not Shown

The event that marks the beginning of this state is the familiar Load event. Code you place in the Form_Load event procedure is executed as soon as the form enters the loaded state.

When the Form_Load event procedure begins, the controls on the form have all been created and loaded, and the form has a window — complete with window handle (hWnd) and device context (hDC) — although that window has not yet been shown.

*Any form that becomes visible must first be loaded*.

Many forms pass automatically from the Created, But Not Loaded state into the Loaded, but Not Shown state. A form will be loaded automatically if:

- The form has been specified as the Startup Object, on the General tab of the Project Properties dialog box.
- The Show method is the first property or method of the form to be invoked, as for example `Form1.Show`.
- The first property or method of the form to be invoked is one of the form's built-in members, as for example the Move method.

  3**Note** This case includes any controls on the form, because each control defines a property of the form; that is, in order to access the Caption property of Command1, you must go through the form's Command1 property: `Command1.Caption`.

14

- The Load statement is used to load the form, without first using New or As New to create the form, as described earlier.

15

### Forms That Are Never Shown

In the first two cases listed above, the form will continue directly on to the visible state, as soon as Form_Load completes. In the last two cases, the form will remain loaded, but not shown.

It has long been common coding practice in Visual Basic to load a form but never show it. This might be done for several reasons:

- To use the Timer control to generate timed events.
- To use controls for their functionality, rather than their user interface — for example, for serial communications or access to the file system.
- To execute DDE transactions.

16

**Note** With the Professional or Enterprise edition, you can create ActiveX components (formerly called OLE servers), which are often better at providing code-only functionality than controls are. See *Creating ActiveX Components* in the *Component Tools Guide*.

89

#### Always Coming Home

Forms return from the visible state to the loaded state whenever they're hidden. Returning to the loaded state does not re-execute the Load event, however. Form_Load is executed only once in a form's life.

### Shown

Once a form becomes visible, the user can interact with it. Thereafter, the form may be hidden and shown as many times as you like before finally being unloaded.

## Interlude: Preparing to Unload

A form may be either hidden or visible when it's unloaded. If not explicitly hidden, it remains visible until unloaded.

The last event the form gets before unloading is the Unload event. Before this event occurs, however, you get a very important event called QueryUnload. QueryUnload is your chance to stop the form from unloading. If there's data the user might like to save, this is the time to prompt the user to save or discard changes.

**Important** Setting the Cancel argument of the QueryUnload to True will stop the form from unloading, negating an Unload statement.

90

One of most powerful features of this event is that it tells you *how* the impending unload was caused: By the user clicking the Close button; by your program executing the Unload statement; by the application closing; or by Windows closing. Thus QueryUnload allows you to offer the user a chance to cancel closing the form, while still letting you close the form from code when you need to.

**Important** Under certain circumstances, a form will not receive a QueryUnload event: If you use the End statement to terminate your program, or if you click the End button (or select End from the Run menu) in the development environment.

91

## Returning to the Created, But Not Loaded State

When the form is unloaded, Visual Basic removes it from the Forms collection. Unless you've kept a variable around with a reference to the form in it, the form will be destroyed, and its memory and resources will be reclaimed by Visual Basic.

If you kept a reference to the form in a variable somewhere, such as the hidden global variable described in "Customizing Form Classes," then the form returns to the Created, But Not Loaded state. The form no longer has a window, and its controls no longer exist.

The object is still holding on to resources and memory. All of the data in the module-level variables in the form's code part are still there. (Static variables in event procedures, however, are gone.)

You can use that reference you've been keeping to call the methods and properties that you added to the form, but if you invoke the form's built-in members, or access its controls, the form will load again, and Form_Load will execute.

## Memory and Resources Completely Reclaimed

The only way to release all memory and resources is to unload the form and then set all references to Nothing. The reference most commonly overlooked when doing this is the hidden global variable mentioned earlier. If at any time you have referred to the form by its class name (as shown in the Properties Window by the Name property), you've used the hidden global variable. To free the form's memory, you must set this variable to Nothing. For example:

```
Set Form1 = Nothing
```

92

Your form will receive its Terminate event just before it is destroyed.

**Tip** Many professional programmers avoid the use of the hidden global variable, preferring to declare their own form variables (for example, Dim dlgAbout As New frmAboutBox) to manage form lifetime.

93

**Note** Executing the End statement unloads all forms and sets all object variables in your program to Nothing. However, this is a very abrupt way to terminate your program. None of your forms will get their QueryUnload, Unload, or Terminate events, and objects you've created will not get their Terminate events.

## Unloaded and Unreferenced, But a Control Is Still Referenced

To get into this odd state, you have to unload and free the form while keeping a reference to one of its controls. If this sounds like a silly thing to do, rest assured that it is.

```
Dim frm As New Form1
Dim obj As Object
frm.Show vbModal
' When the modal form is dismissed, save a
' reference to one of its controls.
Set obj = frm.Command1
Unload frm
Set frm = Nothing
```

94

The form has been unloaded, and all references to it released. However, you still have a reference to one of its controls, and this will keep the code part of the form from releasing the memory it's using. If you invoke any of the properties or methods of this control, the form will be reloaded:

```
obj.Caption = "Back to life"
```

95

The values in module-level variables will still be preserved, but the property values of all the controls will be set back to their defaults, as if the form were being loaded for the first time. Form_Load will execute.

**Note** In some previous versions of Visual Basic, the form did not completely re-initialize, and Form_Load did not execute again.

96

**Note** Not all forms behave as Visual Basic forms do. For example, the Microsoft Forms provided in Microsoft Office don't have Load and Unload events; when these forms receive their Initialize events, all their controls exist and are ready to use.

97

**For More Information** Forms are discussed in "Designing a Form" in "Forms, Controls, and Menus".

98

# Class Modules vs. Standard Modules

Classes differ from standard modules in the way their data is stored. There's never more than one copy of a standard module's data. This means that when one part of your program changes a public variable in a standard module, and another part of your program subsequently reads that variable, it will get the same value.

Class module data, on the other hand, exists separately for each instance of the class (that is, for each object created from the class).

By the same token, data in a standard module has program scope — that is, it exists for the life of your program — while class module data for each instance of a class exists only for the lifetime of the object; it's created when the object is created, and destroyed when the object is destroyed.

Finally, variables declared Public in a standard module are visible from anywhere in your project, whereas Public variables in a class module can only be accessed if you have an object variable containing a reference to a particular instance of a class.

All of the above are also true for public procedures in standard modules and class modules. This is illustrated by the following example. You can run this code by opening a new Standard Exe project and using the Project menu to add a module and a class module.

Place the following code in Class1:

```
' The following is a property of Class1 objects.
Public Comment As String

' The following is a method of Class1 objects.
Public Sub ShowComment()
    MsgBox Comment, , gstrVisibleEverywhere
End Sub
```

99

Place the following code in Module1:

```
' Code in the standard module is global.
Public gstrVisibleEverywhere As String

Public Sub CallableAnywhere(ByVal c1 As Class1)
    ' The following line changes a global variable
    ' (property) of an instance of Class1. Only the
    ' particular object passed to this procedure is
    ' affected.
    c1.Comment = "Touched by a global function."
End Sub
```

100

Put two command buttons on Form1, and add the following code to Form1:

```
Private mc1First As Class1
Private mc1Second As Class1

Private Sub Form_Load()
    ' Create two instances of Class1.
    Set mc1First = New Class1
    Set mc1Second = New Class1
    gstrVisibleEverywhere = "Global string data"
End Sub

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

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

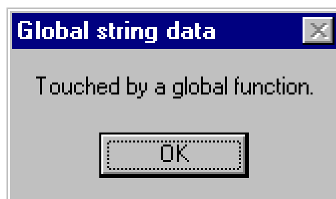
101

Press F5 to run the project. When Form1 is loaded, it creates two instances of Class1, each having its own data. Form1 also sets the value of the global variable `gstrVisibleEverywhere`.

Press Command1, which calls the global procedure and passes a reference to the first Class1 object. The global procedure sets the Comment property, and Command1 then calls the ShowComment method to display the object's data.

As Figure 9.6 shows, the resulting message box demonstrates that the global procedure CallableAnywhere set the Comment property of the object that was passed to it, and that the global string is visible from within Class1.

**Figure 9.6 Message box from the first Class1 object**



17

Press Command2, which simply calls the ShowComment method of the second instance of Class1.

As Figure 9.7 shows, both objects have access to the global string variable; but the Comment property of the second object is blank, because calling the global procedure CallableAnywhere only changed the Comment property for the first object.

**Figure 9.7 Message box from the second Class1 object**

18

**Important** Avoid making the code in your classes dependent on global data — that is, public variables in standard modules. Many instances of a class can exist simultaneously, and all of these objects share the global data in your program.

Using global variables in class module code also violates the object-oriented programming concept of encapsulation, because objects created from such a class do not contain all their data.

102

## Static Class Data

There may be occasions when you want a single data item to be shared among all objects created from a class module. This is sometimes referred to as *static class data*.

You cannot implement true static class data in a Visual Basic class module. However, you can simulate it by using Property procedures to set and return the value of a Public data member in a standard module, as in the following code fragment:

```
' Read-only property returning the application name.
Property Get CommonString() As String
    ' The variable gstrVisibleEverywhere is stored in a
    ' standard module, and declared Public.
    CommonString = gstrVisibleEverywhere
End Property
```

103

**Note** You cannot use the Static keyword for module-level variables in a class module. The Static keyword can only be used within procedures.

104

It's possible to simulate static class data that's not read-only by providing a corresponding Property Let procedure — or Property Set for a property that contains an object reference — to assign a new value to the standard module data member.

Using global variables in this fashion violates the concept of encapsulation, however, and is not recommended.

For example, the variable gstrVisibleEverywhere can be set from anywhere in your project, even from code that doesn't belong to the class that has the CommonString property. This can lead to subtle errors in your program.

105

**For More Information** Global data in ActiveX components requires different handling than in ordinary programs. If you have the Professional or Enterprise Edition of Visual Basic, see "Standard Modules vs. Class Modules" in "General Principles of Component Design."

106

# Adding Properties and Methods to a Class

The properties and methods of a class make up its default interface. The default interface is the most common way of manipulating an object.

In general, properties represent *data about an object*, while methods represent *actions an object can take*. To put it another way, properties provide the description of an object, while methods are its behavior.

**Note** Events aren't part of the default interface. Events are *outgoing* interfaces (that is, interfaces that reach out and touch other objects), while properties and methods belong to *incoming interfaces* (that is, interfaces whose members are invoked by other objects). The default interface of a Visual Basic object is an incoming interface.

107

**For More Information** Events are discussed in "Adding Events to a Class" later in this chapter.

108

## Adding Properties to a Class

The easiest way to define properties for a class is by adding public variables to the class module. For example, you could very easily create an Account class by declaring two public variables in a class module named Account:

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

109

This is pretty easy. It's just as easy to create private data for a class; simply declare a variable Private, and it will be accessible only from code within the class module:

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

110

## Data Hiding

The ability to protect part of an object's data, while exposing the rest as properties, is called *data hiding.* This is one aspect of the object-oriented principle of encapsulation, as explained in "Classes: Putting User-Defined Types and Procedures Together."

Data hiding means that you can make changes in the implementation of a class — for example, increasing the Account class's private variable `mintWithdrawalsMonthToDate` from an Integer to a Long — without affecting existing code that uses the Account object.

Data hiding also allows you to define properties that are read-only. For example, you could use a Property Get procedure to return the value of the private variable containing the number of withdrawals in a month, while only incrementing the variable from within the Account object's code. Which brings us to property procedures.

# Property Procedures

Data hiding wouldn't be much use if the only way you could create properties was by declaring public variables. How much good would it do you to give the Account class a Type property, if any code that had a reference to an Account object could blithely set the account type to any value at all?

Property procedures allow you to execute code when a property value is set or retrieved. For example, you might want to implement the Type property of the Account object as a pair of Property procedures:

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

' Private data storage for the Type property.
Private matType As AccountTypes

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

Public Property Let Type(ByVal NewType As AccountTypes)
    Select Case NewType
        Case atChecking, atSavings, atLineOfCredit
            ' No need to do anything if NewType is valid.
        Case Else
            Err.Raise Number:=vbObjectError + 32112, _
            Description:="Invalid account type"
    End Select
    If mbytType > NewType Then
        Err.Raise Number:=vbObjectError + 32113, _
```

```
        Description:="Cannot downgrade account type"
    Else
        mbytType = NewType
    End If
End Property
```

111

Now suppose you have a variable named acct that contains a reference to an Account object. When the code x = acct.Type is executed, the Property Get procedure is invoked to return the value stored in the class module's private data member mbytType.

When the code acct.Type = atChecking is executed, the Property Let is invoked. If the Account object is brand new, mbytType will be zero, and any valid account type can be assigned. If the current account type is atSavings, the account will be upgraded.

However, if the current account type is atLineOfCredit, the Property Let will raise an error, preventing the downgrade. Likewise, if the code acct.Type = 0 is executed, the Select statement in the Property Let will detect the invalid account type and raise an error.

In short, *property procedures allow an object to protect and validate its own data*.

**For More Information** Are public variables good for anything, then? "Property Procedures vs. Public Variables" outlines the appropriate uses of both. The capabilities of property procedures are explored further in "Putting Property Procedures to Work for You."

112

# Property Procedures vs. Public Variables

Property procedures are clearly such a powerful means for enabling encapsulation that you may be wondering if you should even bother with public variables. The answer, as always in programming, is "Of course — sometimes." Here are some ground rules:

Use property procedures when:

- The property is read-only, or cannot be changed once it has been set.
- The property has a well-defined set of values that need to be validated.
- Values outside a certain range — for example, negative numbers — are valid for the property's data type, but cause program errors if  the property is allowed to assume such values.
- Setting the property causes some perceptible change in the object's state, as for example a Visible property.
- Setting the property causes changes to other internal variables or to the values of other properties.

19

Use public variables for read-write properties where:

- The property is of a self-validating type. For example, an error or automatic data conversion will occur if a value other than True or False is assigned to a Boolean variable.
- Any value in the range supported by the data type is valid. This will be true of many properties of type Single or Double.
- The property is a String data type, and there's no constraint on the size or value of the string.

20

**Note** Don't implement a property as a public variable just to avoid the overhead of a function call. Behind the scenes, Visual Basic will implement the public variables in your class modules as pairs of property procedures anyway, because this is required by the type library.

113

**For More Information** The capabilities of property procedures are explored further in "Putting Property Procedures to Work for You."

114

# Putting Property Procedures to Work for You

Visual Basic provides three kinds of property procedures, as described in the following table.

| Procedure | Purpose |
| --- | --- |
| Property Get | Returns the value of a property. |
| Property Let | Sets the value of a property. |
| Property Set | Sets the value of an object property (that is, a property that contains a reference to an object). |

21

As you can see from the table, each of these property procedures has a particular role to play in defining a property. The typical property will be made up of a pair of property procedures: A Property Get to retrieve the property value, and a Property Let or Property Set to assign a new value.

These roles can overlap in some cases. The reason there are two kinds of property procedures for assigning a value is that Visual Basic has a special syntax for assigning object references to object variables:

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

115

The rule is simple: Visual Basic calls Property Set if the Set statement is used, and Property Let if it is not.

**Tip** To keep Property Let and Property Set straight, harken back to the Basics of yore, when instead of x = 4 you had to type Let x = 4 (syntax supported by Visual Basic to this very day). Visual Basic always calls the property procedure that corresponds to the type of

assignment — Property Let for Let x = 4, and Property Set for Set c1 = New Class1 (that is, object properties).

116

**For More Information** "Working with Objects" in "Programming Fundamentals" explains the use of the Set statement with object variables.

117

## Read-Write Properties

The following code fragment shows a typical read-write property:

```
' Private storage for property value.
Private mintNumberOfTeeth As Integer

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

Public Property Let NumberOfTeeth(ByVal NewValue _
    As Integer)
    ' (Code to validate property value omitted.)
    mintNumberOfTeeth = NewValue
End Property
```

118

The name of the private variable that stores the property value is made up of a scope prefix (m) that identifies it as a module-level variable; a type prefix (int); and a name (NumberOfTeeth). Using the same name as the property serves as a reminder that the variable and the property are related.

As you've no doubt noticed, here and in earlier examples, the names of the property procedures that make up a read-write property must be the same.

**Note** Property procedures are public by default, so if you omit the Public keyword, they will still be public. If for some reason you want a property to be private (that is, accessible only from within the object), you must declare it with the Private keyword. It's good practice to use the Public keyword, even though it isn't required, because it makes your intentions clear.

119

## Property Procedures at Work and Play

It's instructive to step through some property procedure code. Open a new Standard Exe project and add a class module, using the Project menu. Copy the code for the NumberOfTeeth property, shown above, into Class1.

Switch to Form1, and add the following code to the Load event:

```
Private Sub Form_Load()
    Dim c1 As Class1
    Set c1 = New Class1
    ' Assign a new property value.
    c1.NumberOfTeeth = 42
    ' Display the property value.
    MsgBox c1.NumberOfTeeth
```

```
End Sub
```

120

Press F8 to step through the code one line at a time. Notice that when the property value is assigned, you step into the Property Let, and when it's retrieved, you step into the Property Get. You may find it useful to duplicate this exercise with other combinations of property procedures.

## Arguments of Paired Property Procedures Must Match

The property procedure examples you've seen so far have been simple, as they will be for most properties. However, property procedures can have multiple arguments — and even optional arguments. Multiple arguments are useful for properties that act like arrays, as discussed below.

When you use multiple arguments, the arguments of a pair of property procedures must match. The following table demonstrates the requirements for arguments in property procedure declarations.

| Procedure | Declaration syntax |
| --- | --- |
| Property Get | Property Get *propertyname*(1,..., *n*) As *type* |
| Property Let | Property Let *propertyname*(1,..., *n*, *n*+1) |
| Property Set | Property Set *propertyname*(1,..., *n*, *n*+1) |

121

The first argument through the second-to-last argument (1,..., *n*) must share the same names and data types in all Property procedures with the same name. As with other procedure types, all of the required parameters in this list must precede the first optional parameter.

You've probably noticed that a Property Get procedure declaration takes one less argument than the related Property Let or Property Set. The data type of the Property Get procedure must be the same as the data type of the last argument (*n*+1) in the related Property Let or Property Set.

For example, consider this Property Let declaration, for a property that acts like a two-dimensional array:

```
Public Property Let Things(ByVal X As Integer, _
ByVal Y As Integer, ByVal Thing As Variant)
    ' (Code to assign array element omitted.)
End Property
```

122

The Property Get declaration must use arguments with the same name and data type as the arguments in the Property Let procedure:

```
Public Property Let Things(ByVal X As Integer, _
ByVal Y As Integer) As Variant
    ' (Code for retrieval from array omitted.)
End Property
```

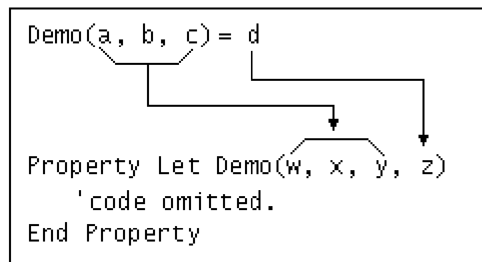
123

The data type of the final argument in a Property Set declaration must be either an object type or a Variant.

### Matching Up the Arguments

The reason for these argument matching rules is illustrated in Figure 9.8, which shows how Visual Basic matches up the parts of the assignment statement with the arguments of a Property Let.

**Figure 9.8   Calling a Property Let procedure**




22

The most common use for property procedures with multiple arguments is to create property arrays.

## Read-Only Properties

To create a read-only property, simply omit the Property Let or (for object properties) the Property Set.

## Object Properties

If you're creating a read-write object property, you use a Property Get and a Property Set, as here:

```
Private mwdgWidget As Widget

Public Property Get Widget() As Widget
    ' The Set statement must be used to return an
    ' object reference.
    Set Widget = mwdgWidget
End Property

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

124

## Variant Properties

Read-write properties of the Variant data type are the most complicated. They use all three property procedure types, as shown here:

```
Private mvntAnything As Variant

Public Property Get Anything() As Variant
```

```
	' The Set statement is used only when the Anything
	' property contains an object reference.
	If IsObject(mvntAnything) Then
		Set Anything = mvntAnything
	Else
		Anything = mvntAnything
	End If
End Property

Public Property Let Anything(ByVal NewValue As Variant)
	' (Validation code omitted.)
	mvntAnything = NewWidget
End Property

Public Property Set Anything(ByVal NewValue As Variant)
	' (Validation code omitted.)
	Set mvntAnything = NewWidget
End Property
```

125

The Property Set and Property Let are straightforward, as they're always called in the correct circumstances. However, the Property Get must handle both of the following cases:

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

126

In the first case, the Anything property contains a string, which is being assigned to a String variable. In the second, Anything contains an object reference, which is being assigned to an object variable.

The Property Get can be coded to handle these cases, by using the IsObject function to test the private Variant before returning the value.

Of course, if the first line of code is called when Anything contains an object reference, an error will occur, but that's not Property Get's problem — that's a problem with using Variant properties.

## Write-Once Properties

There are many possible combinations of property procedures. All of them are valid, but some are relatively uncommon, like write-only properties (only a Property Let, no Property Get). And some depend on factors other than the kinds of property procedures you combine.

For example, when you organize the objects in your program by creating an object model, as described in "Object Models" later in this chapter, you may want an object to be able to refer back to the object that contains it. You can do this by implementing a Parent property.

You need to set this Parent property when the object is created, but thereafter you may want to prevent it from being changed — accidentally or on purpose. The following example shows how the Account object might implement a Parent property that points to the Department object that contains the account.

```
' Private data storage for Parent property.
Private mdeptParent As Department

Property Get Parent() As Department
    ' Use the Set statement for object references.
    Set Parent = mdeptParent
End Property

' The property value can only be set once.
Public Property Set Parent(ByVal NewParent _
As Department)
    If deptParent Is Nothing Then
        ' Assign the initial value.
        Set mdeptParent = NewParent
    Else
        Err.Raise Number:=vbObjectError + 32144, _
        Description:="Parent property is read-only"
    End If
End Property
```

127

When you access the parent of an Account object, for example by coding `strX = acctNew.Parent.Name` to get the department name, the Property Get is invoked to return the reference to the parent object.

The Property Set in this example is coded so that the Parent property can be set only once. For example, when the Department object creates a new account, it might execute the code `Set acctNew.Parent = Me` to set the property. Thereafter the property is read-only.

**For More Information** Because forms in Visual Basic are classes, you can add custom properties to forms. See “Customizing Form Classes” earlier in this chapter.

128

# Adding Methods to a Class

The methods of a class are just the public Sub or Function procedures you’ve declared. Since Sub and Function procedures are public by default, you don’t even have to explicitly specify the Public keyword to create a method.

For example, to create a Withdrawal method for the Account class, you could add this Public Function procedure to the class module:

```
Public Function WithDrawal(ByVal Amount As Currency, _
ByVal TransactionCode As Byte) As Double
    ' (Code to perform the withdrawal and return the
    ' new balance, or to raise an Overdraft error.)
End Function
```

129

**Tip** Although you don’t have to type the Public keyword, doing so is good programming practice, because it makes your intent clear to people maintaining your code later.

130



Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: .
Revision #: 7 Page: 50 of 96 Printed: 09/16/96 04:48 PM

### Declaring Methods as Public Subs

Returning the new balance is optional, since you could easily call the Balance property of the Account object after calling the Withdrawal method. You could thus code Withdrawal as a Public Sub procedure.

**Tip** If you find yourself calling Balance almost every time you call Withdrawal, returning the new balance will be slightly more efficient. This is because, as noted in “Adding Properties to Class Modules,” any property access, even reading a public variable, means a function call — an explicit or implicit Property Get.

131

**For More Information** For more information on Sub and Function procedures, see “Introduction to Procedures” in “Programming Fundamentals.”

132

## Protecting Implementation Details

The public interface of a class is defined by the property and method declarations in the class module. As with data hiding, procedures you declare as Private are not part of the interface. This means that you can make changes to utility procedures that are used internally by a class module, without affecting code that uses the objects.

Even more important, you can also change the code inside the public Sub or Function procedure that implements a method, without affecting code that uses the method. As long as you don’t change the data types of the procedure’s arguments, or the type of data returned by a Function procedure, the interface is unchanged.

Hiding the details of an object’s implementation behind the interface is another facet of encapsulation. Encapsulation allows you to enhance the performance of methods, or completely change the way a method is implemented, without having to change code that uses the method.

**Note** The guidelines for naming interface elements — discussed in “Naming Properties, Methods, and Events” — apply not only to property and method names, but to the names of parameters in the Sub and Function procedures that define your methods. These parameter names are visible when you view the methods in the Object Browser, and can be used as named parameters (that is, *parametername:=value*) when the methods are invoked.

133

**For More Information** Named arguments are introduced in “Passing Arguments to Procedures” in “Programming Fundamentals.” Adding methods to form classes is a powerful programming technique, discussed in “Customizing Form Classes.” Sometimes it’s not clear whether a member should be a property or a method. “Is It a Property or a Method?” offers some guidelines.

134

# Is It a Property or a Method?

In general, a property is data about an object, while a method is an action the object can be asked to perform. Some things are obviously properties, like Color and Name, and some are obviously methods, like Move and Show.

As with any facet of human endeavor, however, there's a gray area in which an argument can be made either way.

For example, why is the Item method of the Visual Basic Collection class a method and not an indexed property? Aren't the items in the collection just data? The Item method of a hypothetical Widgets collection class could be implemented either way, as shown here:

```
' Private storage for the objects in the Widgets
' collection (same for both implementations).
Private mcol As New Collection

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

- or -

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

135

There's not a whole lot of difference between these two implementations. Both are read-only, so both depend on the Add method of the Widgets class to get Widget objects into the collection. Both delegate everything to a Collection object — even their errors are generated by the Collection!

**For More Information** Delegation is explained in "The Many (Inter)Faces of Code Reuse" and "Creating Your Own Collection Classes" later in this chapter.

136

You can get really nit-picky trying to decide whether a member is data about the object or object behavior. For example, you could argue that Item is a method because the collection is doing something for you — looking up the Widget you want. This kind of argument can usually be made with equal validity on either side, however.

You may find it more useful to turn the argument on its head, and ask yourself how you *want* to think of the member. If you want people to think of it as data about the object, make it a property. If you want them to think of it as something the object does, make it a method.

### The Syntax Argument

A strong reason for implementing a member using property procedures depends on the way you want to use the member in code. That is, will the user of a Widgets collection be allowed to code the following?

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

137

If so, implement the member as a read-write property, using Property Get and Property Set, because methods don't support this syntax.

**Note** In most collection implementations you encounter, this syntax is not allowed. Implementing a Property Set for a collection is not as easy as it looks.

138

### The Property Window Argument

You can also suppose for a moment that your object is like a control. Can you imagine the member showing up in the Property window, or on a property page? If that doesn't make sense, don't implement the member as a property.

### The Sensible Error Argument

If you forget that you made Item a read-only property and try to assign a value to it, you'll most likely find it easier to understand the error message Visual Basic raises for a Property Get — "Can't assign to read-only property" — than the error message it raises for a Function procedure — "Function call on left-hand side of assignment must return Variant or Object."

### The Argument of Last Resort

As a last resort, flip a coin. If none of the other arguments in this topic seem compelling, it probably doesn't make much difference.

**For More Information** Property procedures are introduced in "Adding Properties to Classes" earlier in this chapter. Methods are discussed in "Adding Methods to Classes."

139

# Making a Property or Method the Default

You can give objects created from your classes default properties, like the default properties of objects provided by Visual Basic. The best candidate for default member is the one you use most often.

### To set a property or method as the default

9 On the **Tools** menu, select **Procedure Attributes** to open the **Procedure Attributes** dialog box.

10 Click **Advanced** to expand the **Procedure Attributes** dialog box.

11 In the **Name** box, select the property or method that is currently the default for the class. If the class does not currently have a default member, skip to step 5.

4**Note** You can use the Object Browser to find out what the current default member of a class is. When you select the class in the Classes list, you can scroll through the members in the Members list; the default member will be marked with a small blue globe beside its icon.

23

12 In the **Procedure ID** box, select **None** to remove the default status of the property or method.

13 In the **Name** box, select the property or method you want to be the new default.

14 In the **Procedure ID** box, select **(Default)**, then click **OK**.

24

**Important** A class can have only one default member. If a property or method is already marked as the default, you must reset its procedure ID to None before making another property or method the default. No compile errors will occur if two members are marked as default, but there is no way to predict which one Visual Basic will pick as the default.

140

You can also open the Procedure Attributes dialog box from the Object Browser. This is convenient when you're changing the default member of a class, because it allows you to locate the existing default member quickly.

### To change a default property using the Object Browser

15 Press F2 to open the **Object Browser**.

16 In the **Classes** list, select the class whose default you want to change.

17 In the Members list, right-click the member with the small blue globe beside its icon to open the context menu. Click **Properties** to show the **Property Attributes** dialog box.

18 Click **Advanced** to expand the **Procedure Attributes** dialog box.

19 In the **Procedure ID** box, select **None** to remove the default status of the property or method, then click **OK**.

20 In the **Members** list, right-click the member you want to be the new default to open the context menu. Click **Properties** to show the **Property Attributes** dialog box.

21 Click **Advanced** to expand the **Procedure Attributes** dialog box.

22 In the **Procedure ID** box, select **(Default)**, then click **OK**.

25

**Note** You cannot use the Procedure Attributes dialog box to change the default member of a class provided by Visual Basic.

141

# Friend Properties and Methods

In addition to declaring properties and methods Public and Private, you can declare them Friend. Friend members look just like Public members to other objects in your project. That is, they appear to be part of a class's interface. They are not.

In the ActiveX components you can create with the Professional and Enterprise editions of Visual Basic, Friend members play an important role. Because they're not part of an object's interface, they can't be accessed by programs that use the

component's objects. They're visible to all the other objects within the component, however, so they allow safe internal communication within the component.

**Important** Because Friend members aren't part of an object's public interface, they can't be accessed late bound — that is, through variables declared As Object. To use Friend members, you must declare variables with early binding — that is, As *classname*.

142

Standard Exe projects can't be ActiveX components, because their class modules can't be Public, and thus can't be used by other applications. All communication between objects in a Standard Exe project is therefore private, and there's no need for Friend members.

However, Friend members have one particularly useful feature. Because they're not part of an ActiveX interface, they can be used to pass user-defined types between objects. For example, suppose you have the following user-defined type in a standard module:

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

143

You can define the following private variable and Friend members in Class1:

```
Private mDemo As udtDemo

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

' Note that udtDemo must be passed by reference.
Friend Property Let Demo(NewDemo As udtDemo)
    mDemo = NewDemo
End Sub

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 & vbCrLf & mDemo.strC
End Sub
```

144

**Note** When you pass user-defined types as Sub, Function, or property procedure arguments, you must pass them by reference. (ByRef is the default for procedure arguments.)

145



Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: .
Revision #: 7 Page: 55 of 96 Printed: 09/16/96 04:48 PM

You can then write the following code to use Class1:

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

146

The message box will display 42, 1138, and "Howdy."

**Note** Because Friend procedures are not part of a class's interface, they are not included when you use the Implements statement to implement multiple interfaces, as described in "Polymorphism."

147

**For More Information** The use of Friend members in components is discussed in "Private Communication Between Your Objects" in "General Principles of Component Design."

148

# Adding Events to a Class

Okay, let's say you've created a dinosaur simulation, complete with Stegosaur, Triceratops, and Tyrannosaur classes. As the final touch, you want the Tyrannosaur to roar, and when it does you want every other dinosaur in your simulation to sit up and take notice.

If the Tyrannosaur class had a Roar event, you could handle that event in all your other dinosaur classes. This topic discusses the declaration and handling of events in your class modules.

**Note** Kids, don't try this at home, at least not with more than a few dinosaurs. Connecting every dinosaur with every other dinosaur using events could make your dinosaurs so slow that mammal objects would take over the simulation.

149

Properties and methods are said to belong to *incoming interfaces*, because they're invoked from outside the object. By contrast, events are called *outgoing interfaces*, because they're initiated within the object, and handled elsewhere.

**For More Information** "Creating an ActiveX Control" discusses the use of events in designing your own software components. For a discussion of a better way to handle dinosaurs, see "Polymorphism" later in this chapter.

150

# Declaring and Raising Events

Assume for the moment that you have a Widget class. Your Widget class has a method that can take a long time to execute, and you'd like your application to be able to put up some kind of completion indicator.

Of course, you could make the Widget object show a percent-complete dialog box, but then you'd be stuck with that dialog box in every project in which you used the Widget class. A good principle of object design is to let the application that uses an object handle the user interface — unless the whole purpose of the object is to manage a form or dialog box.

The Widget's purpose is to perform other tasks, so it's reasonable to give it a PercentDone event, and to let the procedure that calls the Widget's methods handle that event. The PercentDone event can also provide a mechanism for canceling the task.

You can start building the code example for this topic by opening a Standard Exe project, and adding two buttons and a label to Form1. On the Project menu, select Add Class Module to add a class module to the project. Name the objects as shown in the following table.

| Object | Property | Setting |
|---|---|---|
| Class module | Name | Widget |
| First Button | Caption | Start Task |
| Second Button | Caption | Cancel |
| Label | Name<br>Caption | lblPercentDone<br>"0" |

151

## The Widget Class

You declare an event in the Declarations section of a class module, using the Event keyword. An event can have ByVal and ByRef arguments, as the Widget's PercentDone event demonstrates:

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

152

When the calling object receives a PercentDone event, the Percent argument contains the percentage of the task that's complete. The ByRef Cancel argument can be set to True to cancel the method that raised the event.

### Raising the PercentDone Event

The PercentDone event is raised by the LongTask method of the Widget class. The LongTask method takes two arguments: the length of time the method will pretend to be doing work, and the minimum time interval before LongTask pauses to raise the PercentDone event.

```
Public Sub LongTask(ByVal Duration As Single, _
```

```
ByVal MinimumInterval As Single)
    Dim sngThreshold As Single
    Dim sngStart As Single
    Dim blnCancel As Boolean

    ' The Timer function returns the fractional number
    ' of seconds since Midnight, as a Single.
    sngStart = Timer
    sngThreshold = MinimumInterval

    Do While Timer < (sngStart + Duration)
        ' In a real application, some unit of work would
        ' be done here each time through the loop.

        If Timer > (sngStart + sngThreshold) Then
            RaiseEvent PercentDone( _
            sngThreshold / Duration, blnCancel)
            ' Check to see if the operation was canceled.
            If blnCancel Then Exit Sub
            sngThreshold = sngThreshold + MinimumInterval
        End If
    Loop
End Sub
```

153

Every MinimumInterval seconds, the PercentDone event is raised. When the event returns, LongTask checks to see if the Cancel argument was set to True.

**Note** For simplicity, LongTask assumes you know in advance how long the task will take. This is almost never the case. Dividing tasks into chunks of even size can be difficult, and often what matters most to users is simply the amount of time that passes before they get an indication that something is happening.

154

**For More Information** Now that you've declared an event and raised it, how do you get another object to handle it? "Handling an Object's Events" continues the saga of the Widget object.

155

# Handling an Object's Events

An object that raises events is called an *event source.* To handle the events raised by an event source, you can declare a variable of the object's class using the WithEvents keyword.

This topic continues the Widget object example begun in "Declaring and Raising Events." To handle the PercentDone event of a Widget, place the following code in the Declarations section of Form1:

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

156

The WithEvents keyword specifies that the variable mWidget will be used to handle an object's events. You specify the kind of object by supplying the name of the class from which the object will be created.

The variable mWidget is declared in the Declarations section of Form1 because WithEvents variables must be module-level variables. This is true regardless of the type of module you place them in.

The variable mblnCancel will be used to cancel the LongTask method.

### Limitations on WithEvents Variables

You should be aware of the following limitations on the use of WithEvents variables:

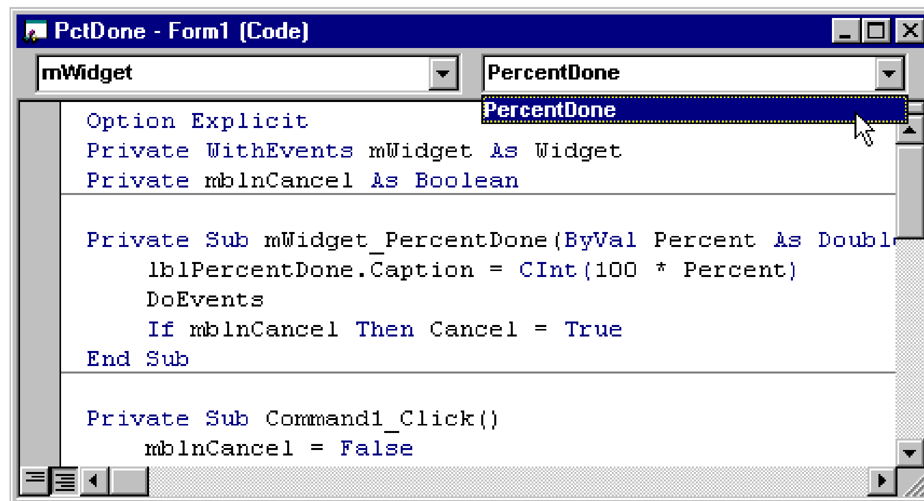
- A WithEvents variable cannot be a generic object variable. That is, you cannot declare it As Object — you must specify the class name when you declare the variable.
- You cannot declare a WithEvents variable As New. The event source object must be explicitly created and assigned to the WithEvents variable.
- You cannot declare WithEvents variables in a standard module. You can declare them only in class modules, form modules, and other modules that define classes.
- You cannot create arrays of WithEvents variables.

26

## Writing Code to Handle an Event

As soon as you declare a variable WithEvents, the variable name appears in the left-hand drop down of the module's code window. When you select mWidget, the Widget class's events will appear in the right-hand drop down, as shown in Figure 9.9.

**Figure 9.9   An event associated with a WithEvents variable**



27

Selecting an event will display the corresponding event procedure, with the prefix mWidget_. All the event procedures associated with a WithEvents variable will have the variable name as a prefix. Add the following code to the mWidget_PercentDone event procedure.

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

157

Whenever the PercentDone event is raised, the event procedure displays the percent complete in a Label control. The DoEvents statement allows the label to repaint, and also gives the user the opportunity to click the Cancel button. Add the following code for the Click event of the button whose caption is Cancel.

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

158

If the user clicks the Cancel button while LongTask is running, the Command2_Click event will be executed as soon as the DoEvents statement allows event processing to occur. The module-level variable mblnCancel is set to True, and the mWidget_PercentDone event then tests it and sets the ByRef Cancel argument to True.

## Connecting a WithEvents Variable to an Object

Form1 is all set up to handle a Widget object's events. All that remains is to find a Widget somewhere.

When you declare a variable WithEvents at design time, there is no object associated with it. A WithEvents variable is just like any other object variable. You have to create an object and assign a reference to the object to the WithEvents variable. Add the following code to the Form_Load event procedure to create the Widget.

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

159

When the code above is executed, Visual Basic creates a Widget and connects its events to the event procedures associated with mWidget. From that point on, whenever the Widget raises its PercentDone event, the mWidget_PercentDone event procedure will be executed.

To call the LongTask method, add the following code to the Click event of the button whose caption is Start Task.

```
' Start Task button.
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
```

160

Before the LongTask method is called, the label that displays the percent complete must be initialized, and the module-level Boolean flag for canceling the method must be set to False.

LongTask is called with a task duration of 14.4 seconds. The PercentDone event is to be raised once every two-thirds of a second. Each time the event is raised, the mWidget_PercentDone event procedure will be executed.

When LongTask is done, mblnCancel is tested to see if LongTask ended normally, or if it stopped because mblnCancel was set to True. The percent complete is updated only for the former case.

## Running the Program

Press F5 to put the project in Run mode. Click the Start Task button. Each time the PercentDone event is raised, the label is updated with the percentage of the task that's complete. Click the Cancel button to stop the task. Notice that the appearance of the Cancel button doesn't change immediately when you click it. The Click event can't happen until the DoEvents statement allows event processing.

You may find it instructive to run the program with F8, and step through the code a line at a time. You can clearly see how execution enters LongTask, and then re-enters Form1 briefly each time the PercentDone event is raised.

What would happen if, while execution was back in Form1's code, the LongTask method was called again? Confusion, chaos, and eventually (if it happened every time the event was raised) a stack overflow.

## Handling Events for a Different Widget

You can cause the variable mWidget to handle events for a different Widget object by assigning a reference to the new Widget to mWidget. In fact, you can make the code in Command1 do this every time you click the button, by adding two lines of code:

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

161

The code above creates a new Widget each time the button is pressed. As soon as the LongTask method completes, the reference to the Widget is released by setting mWidget to Nothing, and the Widget is destroyed.

A WithEvents variable can only contain one object reference at a time, so if you assign a different Widget object to mWidget, the previous Widget object's events

will no longer be handled. If mWidget is the only object variable containing a reference to the old Widget, the object will be destroyed.

**Note** You can declare as many WithEvents variables as you need, but arrays of WithEvents variables are not supported.

162

### Terminating Event Handling for a WithEvents Variable

As long as there is a Widget object assigned to the variable mWidget, the event procedures associated with mWidget will be called whenever the Widget raises an event. To terminate event handling, you can set mWidget to Nothing, as shown in the following code fragment.

```
' Terminate event handling for mWidget.
Set mWidget = Nothing
```

163

When a WithEvents variable is set to Nothing, Visual Basic disconnects the object's events from the event procedures associated with the variable.

**Important** A WithEvents variable contains an object reference, just like any other object variable. This object reference counts toward keeping the object alive. When you are setting all references to an object to Nothing in order to destroy it, don't forget the variables you declared WithEvents.

164

**For More Information** The event procedures associated with WithEvents variables look a lot like event procedures for controls on forms. "Comparing WithEvents to Control Events on Forms" discusses the similarities and differences.

165

# Comparing WithEvents to Control Events on Forms

You've probably noticed some similarities between the way you use WithEvents variables and the way you handle the events raised by controls on a form. In both cases, when you select the event in the right-hand drop down of a code window, you get an event procedure containing the correct arguments for the event.

In fact, the mechanism is exactly the same. A control is treated as a *property* of the form class, and the name of that property is the value you assigned to the control's Name property in the Properties window.

It's as if there's a Public module-level variable with the same name as the control, and all of the control's event procedure names begin with that variable name, just as they would with a WithEvents variable.

You can easily see this by declaring the variable mWidget Public instead of Private. The moment you do this, mWidget will show up in the Object Browser as a property of Form1, just like the controls on the form.

The difference between the two cases is that Visual Basic automatically creates instances of all the controls on a form when the form is created, whereas you have to create your own instances of classes whose events you want to handle, and assign references to those objects to WithEvents variables.

**For More Information** You can add your own events to forms, as discussed in "Adding an Event to a Form."

166

# Adding an Event to a Form

The following step by step procedure shows how you can create custom events for forms. To try this exercise, open a new Standard Exe project and do the following:

### To add an event to Form1

23 On the **Project** menu, select **Add Class Module** to add a class module to the project. Place the following code in the **Declarations** section of Class1:

```
1Public Property Get Form1() As Form1
2   Set Form1 = mForm1
3End Property
4
5Public Property Set Form1(ByVal NewForm1 As Form1)
6   Set mForm1 = NewForm1
7End Property
```

28

11If you're using Procedure View, the property procedures can't be viewed at the same time. Click the **Full Module View** button at the bottom left corner of the code window to switch to Full Module View. You can return to Procedure View by clicking the **Procedure View** button next to it. (Hover the mouse over the buttons to see which is which.)

24 Add the following code to the **Declarations** section of Form1:

```
8Event Gong
9Private mc1 As Class1
```

29

12Now that Class1 has been created, it's possible to create a variable of type Class1. This procedure switches between Form1 and Class1 several times, because a step in one module requires first adding code to the other.

25 Go back to Class1 and add the following code to the **Declarations** section.

```
10Private WithEvents mForm1 As Form1
```

30

13As discussed in "Adding Events to Classes," the WithEvents keyword means this instance of Form1 is associated *with events*. Note that this step wasn't possible until the Gong event had been created.

26 In the left-hand (**Object**) drop down on Class1's **Code** window, select **mForm1** to get the event procedure for the Gong event. Add the following code to the event procedure:

```
11Private Sub mForm1_Gong()
12  MsgBox "Gong!"
```

```
13End Sub
```

31

5**Note** Why can’t you see the other events of Form1 in the Procedure drop down? The events of the Visual Basic Form object are private. You can handle them in your form class, Form1, but they’re not visible outside. The events you declare in a form class are public, and can be handled by other objects.

32

27 Go back to Form1. In the **Object** drop down, select **Form**. In the right-hand (**Procedure**) drop down, select **Load**. Add the following code to the event procedure:

```
14Private Sub Form_Load()
15  Set mc1 = New Class1
16  Set mc1.Form1 = Me
17End Sub
```

33

14The first line creates a Class1 object, and the second assigns to its Form1 property (created in step 1) a reference to Form1 (that is, Me — when you’re in Form1’s **Code** window, Me refers to Form1; when you’re in Class1’s **Code** window, Me refers to Class1).

28 Put three text boxes on Form1. Use the **Object** and **Procedure** drop downs to select the **Change** event procedure for each control in turn, and place the same line of code in each:

```
18Private Sub Text1_Change()
19  RaiseEvent Gong
20End Sub
```

34

15Each time the contents of a text box change, the form’s Gong event will be raised.

29 Press F5 to run the project. Each time you type a character in one of the text boxes, the message box rings a bell. It's very annoying, but it shows how you can add an event to a form, and thus get notifications from several controls.

35

As shown in “Adding Events to Classes,” you can add arguments to events. For example, you might pass the name of the control — or better still, a reference to the control — to the receiver of the event.

**Note** Remember — as described in step 4 — the only form events you can handle in other modules are user-defined events.

167

# Summary of Declaring, Raising, and Handling Events

To add an event to a class and then use the event, you must:

- In the Declarations section of the class module that defines the class, use the Event statement to declare the event with whatever arguments you want it to have. Events are always Public.

- At appropriate places in the class module's code, use the RaiseEvent statement to raise the event, supplying the necessary arguments.
- In the Declarations section of the module that will handle the event, add a variable of the class type, using the WithEvents keyword. This must be a module-level variable.
- In the left-hand drop down of the code window, select the variable you declared WithEvents.
- In the right-hand drop down, select the event you wish to handle. (You can declare multiple events for a class.)
- Add code to the event procedure, using the supplied arguments.

36

**For More Information** Details and code examples are provided in "Adding Events to a Class."

168

# Naming Properties, Methods, and Events

The properties, methods, and events you add to a class module define the interface that will be used to manipulate objects created from the class. When naming these elements, and their arguments, you may find it helpful to follow a few simple rules.

- Use entire words whenever possible, as for example SpellCheck. Abbreviations can take many forms, and hence can be confusing. If whole words are too long, use complete first syllables.
- Use mixed case for your identifiers, capitalizing each word or syllable, as for example ShortcutMenus or AsyncReadComplete.
- Use the correct plural for collection class names, as for example Worksheets, Forms, or Widgets. If the collection holds objects with a name that ends in "s," append the word "Collection," as for example SeriesCollection.
- Use either verb/object or object/verb order consistently for your method names. That is, use InsertWidget, InsertSprocket, and so on, or always place the object first, as in WidgetInsert and SprocketInsert.

37

One of the chief benefits of programming with objects is code reuse. Following the rules above, which are part of the ActiveX guidelines for interfaces, makes it easier to remember the names and purposes of properties, methods, and events.

**For More Information** If you have the Professional or Enterprise Edition of Visual Basic, see the expanded list in "What's In a Name?" in "General Principles of Component Design."

169

# Polymorphism

*Polymorphism* means that many classes can provide the same property or method, and a caller doesn't have to know what class an object belongs to before calling the property or method.

For example, a Flea class and a Tyrannosaur class might each have a Bite method. Polymorphism means that you can invoke Bite without knowing whether an object is a Flea or a Tyrannosaur — although you'll certainly know afterward.

**For More Information** With the Professional and Enterprise editions of Visual Basic, Polymorphism becomes a powerful mechanism for evolving systems of software components. This is discussed in "General Principles of Component Design."

170

# How Visual Basic Provides Polymorphism

Most object-oriented programming systems provide polymorphism through *inheritance*. That is, the hypothetical Flea and Tyrannosaur classes might both inherit from an Animal class. Each class would override the Animal class's Bite method, in order to provide its own bite characteristics.

The polymorphism comes from the fact that you could call the Bite method of an object belonging to any class that derived from Animal, without knowing which class the object belonged to.

## Providing Polymorphism with Interfaces

Visual Basic doesn't use inheritance to provide polymorphism. Visual Basic provides polymorphism through multiple ActiveX *interfaces*. In the Component Object Model (COM) that forms the infrastructure of the ActiveX specification, multiple interfaces allow systems of software components to evolve without breaking existing code.

An *interface* is a set of related properties and methods. Much of the ActiveX specification is concerned with implementing standard interfaces to obtain system services or to provide functionality to other programs.

In Visual Basic, you would create an Animal interface and implement it in your Flea and Tyrannosaur classes. You could then invoke the Bite method of either kind of object, without knowing which kind it was.

## Polymorphism and Performance

Polymorphism is important for performance reasons. To see this, consider the following function:

```
Public Sub GetFood(ByVal Critter As Object, _
ByVal Food As Object)
    Dim dblDistance As Double
    ' Code to calculate distance to food (omitted).
```

```
    Critter.Move dblDistance  ' Late bound
    Critter.Bite Food         ' Late bound
End Sub
```

171

The Move and Bite methods are *late bound* to Critter. Late binding happens when Visual Basic can't determine at compile time what kind of object a variable will contain. In this example, the Critter argument is declared As Object, so at run time it could contain a reference to any kind of object — like a Car or a Rock.

Because it can't tell what the object will be, Visual Basic compiles some extra code to ask the object if it supports the method you've called. If the object supports the method, this extra code invokes it; if not, the extra code raises an error. Every method or property call incurs this additional overhead.

By contrast, interfaces allow *early binding*. When Visual Basic knows at compile time what interface is being called, it can check the type library to see if that interface supports the method. Visual Basic can then compile in a direct jump to the method, using a virtual function table (vtable). This is many times faster than late binding.

Now suppose the Move and Bite methods belong to an Animal interface, and that all animal classes implement this interface. The Critter argument can now be declared As Animal, and the Move and Bite methods will be early bound:

```
Public Sub GetFood(ByVal Critter As Animal, _
ByVal Food As Object)
    Dim dblDistance As Double
    ' Code to calculate distance to food (omitted).
    Critter.Move dblDistance  ' Early bound (vtable).
    Critter.Bite Food         ' Early bound (vtable).
End Sub
```

172

**For More Information** "Creating and Implementing an Interface" creates an Animal interface and implements it in Flea and Tyrannosaur classes.

173

# Creating and Implementing an Interface

As explained in "How Visual Basic Provides Polymorphism," an interface is a set of properties and methods. In the following code example, you'll create an Animal interface and implement it in two classes, Flea and Tyrannosaur.

You can create the Animal interface by adding a class module to your project, naming it Animal, and inserting the following code:

```
Public Sub Move(ByVal Distance As Double)

End Sub

Public Sub Bite(ByVal What As Object)

End Sub
```

174

Notice that there's no code in these methods. Animal is an *abstract class,* containing no implementation code. An abstract class isn't meant for creating objects — its purpose is to provide the template for an interface you add to other classes. (Although, as it turns out, sometimes it's useful to implement the interface of a class that isn't abstract; this is discussed later in this topic.)

Now you can add two more class modules, naming one of them Flea and the other Tyrannosaur. To implement the Animal interface in the Flea class, you use the Implements statement:

```
Option Explicit
Implements Animal
```

175

As soon as you've added this line of code, you can click the left-hand (Object) drop down in the code window. One of the entries will be Animal. When you select it, the right-hand (Procedure) drop down will show the methods of the Animal interface.

Select each method in turn, to create empty procedure templates for all the methods. The templates will have the correct arguments and data types, as defined in the Animal class. Each procedure name will have the prefix Animal_ to identify the interface.

**Important** An interface is like a contract. By implementing the interface, a class agrees to respond when any property or method of the interface is invoked. Therefore, you must implement *all* the properties and methods of an interface.

176

You can now add the following code to the Flea class:

```
Private Sub Animal_Move(ByVal Distance As Double)
    ' (Code to jump some number of inches omitted.)
    Debug.Print "Flea moved"
End Sub

Private Sub Animal_Bite(ByVal What As Object)
    ' (Code to suck blood omitted.)
    Debug.Print "Flea bit a " & TypeName(What)
End Sub
```

177

You may be wondering why the procedures are declared Private. If they were Public, the procedures Animal_Jump and Animal_Bite would be part of the Flea interface, and we'd be stuck in the same bind we were in originally, declaring the Critter argument As Object so it could contain either a Flea or a Tyrannosaur.

## Multiple Interfaces

The Flea class now has two interfaces: The Animal interface you've just implemented, which has two members, and the default Flea interface, which has no members. Later in this example you'll add a member to one of the default interfaces.

You can implement the Animal interface similarly for the Tyrannosaur class:

```
Option Explicit
```

```
Implements Animal

Private Sub Animal_Move(ByVal Distance As Double)
    ' (Code to pounce some number of yards omitted.)
    Debug.Print "Tyrannosaur moved"
End Sub

Private Sub Animal_Bite(ByVal What As Object)
    ' (Code to take a pound of flesh omitted.)
    Debug.Print "Tyrannosaur bit a " & TypeName(What)
End Sub
```

178

# Exercising the Tyrannosaur and the Flea

Add the following code to the Load event of Form1:

```
Private Sub Form_Load()
    Dim fl As Flea
    Dim ty As Tyrannosaur
    Dim anim As Animal

    Set fl = New Flea
    Set ty = New Tyrannosaur
    ' First give the Flea a shot.
    Set anim = fl
    Call anim.Bite(ty)  'Flea bites dinosaur.
    ' Now the Tyrannosaur gets a turn.
    Set anim = ty
    Call anim.Bite(fl)   'Dinosaur bites flea.
End Sub
```

179

Press F8 to step through the code. Notice the messages in the Immediate window. When the variable anim contains a reference to the Flea, the Flea's implementation of Bite is invoked, and likewise for the Tyrannosaur.

The variable anim can contain a reference to any object that implements the Animal interface. In fact, it can *only* contain references to such objects. If you attempt to assign a Form or PictureBox object to anim, an error will occur.

The Bite method is early bound when you call it through anim, because Visual Basic knows at compile time that whatever object is assigned to anim will have a Bite method.

## Passing Tyrannosaurs and Fleas to Procedures

Remember the GetFood procedure from "How Visual Basic Provides Polymorphism?" You can add the *second* version of the GetFood procedure — the one that illustrates polymorphism — to Form1, and replace the code in the Load event with the following:

```
Private Sub Form_Load()
    Dim fl As Flea
    Dim ty As Tyrannosaur

    Set fl = New Flea
```

```
    Set ty = New Tyrannosaur
    'Flea dines on dinosaur.
    Call GetFood(fl, ty)
    ' And vice versa.
    Call GetFood(ty, fl)
End Sub
```

180

Stepping through this code shows how an object reference that you pass to an argument of another interface type is converted into a reference to the second interface (in this case, Animal). What happens is that Visual Basic queries the object to find out whether it supports the second interface. If the object does, it returns a reference to the interface, and Visual Basic places that reference in the argument variable. If the object does not support the second interface, an error occurs.

## Implementing Methods That Return Values

Suppose the Move method returned a value. After all, you know how far you want an Animal to move, but an individual specimen might not be able to move that far. It might be old and decrepit, or there might be a wall in the way. The return value of the Move method could be used to tell you how far the Animal actually moved.

```
Public Function Move(ByVal Distance As Double) _
As Double

End Function
```

181

When you implement this method in the Tyrannosaur class, you assign the return value to the procedure name, just as you would for any other Function procedure:

```
Private Function Animal_Move(ByVal Distance _
As Double) As Double
    Dim dblDistanceMoved As Double
    ' Code to calculate how far to pounce (based on
    ' age, state of health, and obstacles) is omitted.
    ' This example assumes that the result has been
    ' placed in the variable dblDistanceMoved.
    Debug.Print "Tyrannosaur moved"; dblDistanceMoved
    Animal_Move = dblDistanceMoved
End Function
```

182

To assign the return value, use the full procedure name, including the interface prefix.

**For More Information** The interfaces you implement can have properties as well as methods. “Implementing Properties” discusses some differences in the way properties are implemented.

183

# Implementing Properties

This topic continues the code example begun in “Creating and Implementing an Interface,” adding properties to the Animal interface that was implemented in the

Flea and Tyrannosaur classes. You may find it helpful to read that topic before beginning this one.

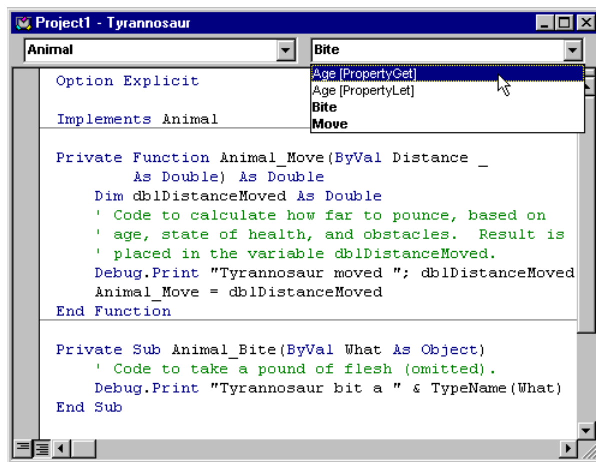
Suppose we give the Animal class an Age property, by adding a Public variable to the Declarations section:

```
Option Explicit
Public Age As Double
```

184

The Procedure drop downs in the code modules for the Tyrannosaur and Flea classes now contain property procedures for implementing the Age property, as shown in Figure 9.10.

**Figure 9.10   Implementing property procedures**




38

This illustrates a point made in “Adding Properties to a Class” earlier in this chapter. Using a public variable to implement a property is strictly a convenience for the programmer. Behind the scenes, Visual Basic implements the property as a pair of property procedures.

You must implement both procedures. The property procedures are easily implemented by storing the value in a private data member, as shown here:

```
Private mdblAge As Double

Private Property Get Animal_Age() As Double
    Animal_Age = mdblAge
End Property

Private Property Let Animal_Age(ByVal RhsVal As Double)
    mdblAge = RhsVal
End Property
```

185

The private data member is an implementation detail, so you have to add it yourself.



Filename: in.doc     Project: A Traditional User's Guide Template
Template:     Author: Ruth Silverio     Last Saved By: .
Revision #: 7     Page: 71 of 96     Printed: 09/16/96 04:48 PM

**Note** When Implements provides the template for a Property Set or Property Let, it has no way of determining the name of the last argument, so it substitutes the name RhsVal, as shown in the code example above.

186

There's no data validation on a property implemented as a public data member, but that doesn't mean you can't add validation code to the Property Let for Animal_Age. For example, you might want to restrict the values to ages appropriate for a Tyrannosaur or a Flea, respectively.

In fact, this shows the independence of interface and implementation. As long as the interface matches the description in the type library, the implementation can be anything.

Before you go on to the next step, remove the implementation of the read-write Age property from both class modules.

## Implementing a Read-Only Property

Of course, allowing the age of an animal to be set arbitrarily is bad object design. The object should know its own age, and provide it to the user as a read-only property. Remove the public variable Age from the Animal class, and add the template for a read-only age property, like this:

```
Public Property Get Age() As Double

End Property
```

187

Now the Procedure drop downs in the code windows for the Tyrannosaur and Flea classes contain only a single entry, Age [PropertyGet]. You might implement this for the Tyrannosaur as follows:

```
Private mdblBirth As Double

Private Property Get Animal_Age() As Double
    Animal_Age = Now - mdblBirth
End Property
```

188

The code above returns the age of the Tyrannosaur in days. You could set mdblBirth in the Initialize event of the Tyrannosaur class, as here:

```
Private Sub Class_Initialize()
    mdblBirth = Now
End Sub
```

189

And of course you could return the property value in more commonly used units, such as dog years.

**For More Information** We've been tossing interfaces and objects around like they were the same thing, seemingly putting references to objects into one object variable, and references to interfaces into another. "Time Out for a Brief Discussion of Objects and Interfaces" clears matters up.

190

# Time Out for a Brief Discussion of Objects and Interfaces

This topic completes the code example begun in “Creating and Implementing an Interface,” and continued in “Implementing Properties.” You may find it helpful to read those topics before beginning this one.

The Tyrannosaur and Flea code example seems to play fast and loose with interfaces and objects. References to objects are assigned to one object variable, and references to interfaces to another.

In fact, *all of the references are object references*. A reference to an interface is also a reference to the object that implements the interface. Furthermore, an object may have multiple interfaces, but it’s still the same object underneath.

In Visual Basic, each class has a default interface that has the same name as the class. Well, almost the same. By convention, an underscore is prefixed to the class name. The underscore indicates that this interface is hidden in the type library.

Thus the Tyrannosaur class has a default interface called _Tyrannosaur. Because Tyrannosaur also implements Animal, the class has a second interface named Animal.

However, underneath it all, the object is still a Tyrannosaur. Place a command button on Form1, and add the following code:

```
Private Sub Command1_Click()
    Dim ty As Tyrannosaur
    Dim anim As Animal

    Set ty = New Tyrannosaur
    Set anim = ty
    MsgBox TypeName(anim)
End Sub
```



You might expect the message box to display “Animal,” but in fact it displays "Tyrannosaur.”

## Querying for Interfaces

When you assign a Tyrannosaur object to variable of type Animal, Visual Basic asks the Tyrannosaur object if it supports the Animal interface. (The method used for this is called QueryInterface, or QI for short; you may sometimes hear QI used as a verb.) If the answer is no, an error occurs.

If the answer is yes, the object is assigned to the variable. Only the methods and properties of the Animal interface can be accessed through this variable.

## Generic Object Variables and Interfaces

What happens if you assign the object reference to a generic object variable, as in the following code?

```
Private Sub Command1_Click()
    Dim ty As Tyrannosaur
    Dim anim As Animal
    Dim obj As Object

    Set ty = New Tyrannosaur
    Set anim = ty
    Set obj = anim
    MsgBox TypeName(obj)
End Sub
```

192

The result is again Tyrannosaur. Now, what interface do you get when you call properties and methods through the variable `obj`? Add the following method to the Tyrannosaur class:

```
Public Sub Growl()
    Debug.Print "Rrrrrr"
End Sub
```

193

The Growl method belongs to the Tyrannosaur object's default interface. In the code for the command button's Click event, replace the MsgBox statement with the following two lines of code:

```
obj.Move 42
obj.Growl
```

194

When you run the project and click the button, execution stops on the Growl method, with the error "Object does not support this property or method." Clearly, the interface is still Animal.

This is something to bear in mind when using variables of type Object with objects that have multiple interfaces. The interface the variable will access is the *last interface assigned.* For example:

```
Private Sub Command1_Click()
    Dim ty As Tyrannosaur
    Dim anim As Animal
    Dim obj As Object

    Set ty = New Tyrannosaur
    Set anim = ty
    Set obj = anim
    obj.Move 42         ' Succeeds
    obj.Growl       ' Fails

    Set obj = ty
    obj.Move 42         ' Fails
    obj.Growl       ' Succeeds
End Sub
```

195

Fortunately, there's very little reason to use the slower, late-bound Object data type with objects that have multiple interfaces. One of the main reasons for using multiple interfaces is to gain the advantage of early binding through polymorphism.

### Other Sources of Interfaces

Visual Basic class modules are not your only source of interfaces to implement. You can implement any interface contained in a type library, as long as that interface supports Automation.

If you have the Professional or Enterprise Edition of Visual Basic, you can create your own type libraries of abstract classes. These type libraries can be used in many projects, as described in "General Principles of Component Design."

The Professional and Enterprise editions also include the MkTypLib (Make Type Library) utility in the Tools directory. If you've used this utility with Microsoft Visual C++, you may find it a more congenial way to create interfaces.

#### Using Interfaces in Your Project

To use an interface in your project, click References on the Project menu to open the References dialog box. If the type library is registered, it will appear in the list of references, and you can check it. If the type library is not in the list, you can use the Browse button to locate it.

Once you have a reference to a type library, you can use Implements to implement any Automation interfaces the type library contains.

**For More Information** You're not limited to implementing abstract interfaces. "The Many (Inter)Faces of Code Reuse" describes how you can implement an interface and selectively reuse the properties and methods of the class that provides the interface.

196

# The Many (Inter)Faces of Code Reuse

There are two main forms of code reuse — binary and source. Binary code reuse is accomplished by creating and using an object, while source code reuse is achieved by inheritance, which isn't supported by Visual Basic. (Source code reuse can also be achieved by copying and modifying the source code, but this technique is nothing new, and has many well-known problems.)

Visual Basic has been a pioneer of binary code reuse — controls being the classic example. You reuse the code in a control by placing an instance of the control on your form. This is known as a *containment* relationship or a *has-a* relationship; that is, the form *contains* or *has a* CommandButton.

**For More Information** Containment relationships are discussed in "Object Models" later in this chapter.

197

## Delegating to an Implemented Object

Implements provides a powerful new means of code reuse. You can implement an abstract class (as discussed in "Creating and Implementing an Interface"), or you can implement the interface of a fully functional class. You can create the *inner object*

(that is, the implemented object) in the Initialize event of the *outer object* (that is, the one that implements the inner object's interface).

As noted in "Creating and Implementing an Interface," an interface is like a contract — you must implement all the members of the inner object's interface in the outer object's class module. However, you can be very selective in the way you delegate to the properties and methods of the inner object. In one method you might delegate directly to the inner object, passing the arguments unchanged, while in another method you might execute some code of your own before calling the inner object — and in a third method you might execute only your own code, ignoring the inner object altogether!

For example, suppose you have a OneManBand class and a Cacophony class, both of which generate sounds. You'd like to add the functionality of the Cacophony class to the OneManBand class, and reuse some of the implementation of the Cacophony class's methods.

```
' OneManBand implements the Cacophony interface.
Implements Cacophony

' Object variable to keep the reference in.
Private mcac As Cacophony

Private Sub Class_Initialize()
    ' Create the object.
    Set mcac = New Cacophony
End Sub
```

198

You can now go to the Object drop down and select Cacophony, and then get procedure templates for the methods of the Cacophony interface. To implement these methods, you can delegate to the Cacophony object. For example, the Beep method might look like this:

```
Private Sub Cacophony_Beep(ByVal Frequency As Double, _
ByVal Duration As Double)
    ' Delegate to the inner Cacophony object.
    Call mcac.Beep(Frequency, Duration)
End Sub
```

199

The implementation above is very simple. The outer object (OneManBand) delegates directly to the inner (Cacophony), reusing the Cacophony object's Beep method without any changes. This is a good thing, but it's only the beginning.

The Implements statement is a very powerful tool for code reuse, because it gives you enormous flexibility. You might decide to alter the effects of the OneManBand class's Beep method, by inserting your own code before (or after) the call to the inner Cacophony object:

```
Private Sub Cacophony_Beep(ByVal Frequency As Double, _
ByVal Duration As Double)
    ' Bump everything up an octave.
    Frequency = Frequency * 2
```

```
    ' Based on another property of the OneManBand
    ' class, Staccato, cut the duration of each beep.
    If Staccato Then Duration = Duration * 7 / 8
    Call mcac.Beep(Frequency, Duration)
    ' You can even call other methods of OneManBand.
    If Staccato Then Pause(Duration * 1 / 8)
End Sub
```

200

For some of the methods, your implementation may delegate directly to the inner Cacophony object, while for others you may interpose your own code before and after delegating — or even omit delegation altogether, using entirely your own code to implement a method.

Because the OneManBand class implements the Cacophony interface, you can use it with any musical application that calls that interface. Your implementation details are hidden from the calling application, but the resulting sounds are all your own.

**Note** COM provides another mechanism for binary code reuse, called *aggregation*. In aggregation, an entire interface is reused, without any changes, and the implementation is provided by an instance of the class being aggregated. Visual Basic does not support this form of code reuse.

201

### Doesn't This Get Tedious?

Writing delegation code can indeed become tedious, especially if most of the outer object's properties and methods simply delegate directly to the corresponding properties and methods of the inner object.

If you have the Professional or Enterprise Edition of Visual Basic, you can use the Visual Basic Extensibility model to create your own delegation wizard to automate the task, similar to the Class Wizard that's included in the Professional and Enterprise editions.

**For More Information** The use of polymorphism and multiple interfaces in component software is discussed in "General Principles of Component Design."

# Programming with Your Own Objects

You can start using objects gradually, finding useful tasks for which combining code and data is an advantage. You can use the functionality of these objects by declaring object variables, assigning new objects to them, and calling the objects' properties and methods.

As you add more and more objects to your programs, you'll start to see relationships between them. You can begin making program design more dependent on objects and their relationships, and you can begin using more robust techniques — like creating custom collection classes — for expressing those relationships in code.

At some point, you'll suddenly see how linking objects together changes the very nature of your program, and you'll be ready to start designing object-based programs from the ground up.

The following topics provide an overview of these evolutionary changes in your coding style. Read them now, to give yourself a rough picture of where you're headed, and read them again when your ideas of object-based programming begin to gel.

**For More Information** ActiveX components open up yet another dimension of code reuse and object-based programming. If you have the Professional or Enterprise Edition of Visual Basic, you can begin to explore that dimension through "Creating an ActiveX Control."

202

# Object References and Reference Counting

The primary rule for object lifetime is very simple: An object is destroyed when the last reference to it is released. However, as with so much of life, simple doesn't always mean easy.

As you use more objects, and keep more variables containing references to those objects, you may go through periods when it seems impossible to get your objects to go away when you want them to.

At some point, it will occur to you that Visual Basic must be keeping track of object references — otherwise how could it know when the last reference to an object is released? You may start thinking that if only you could get access to Visual Basic's reference counts, debugging would be much easier.

Unfortunately, that's not true. To make using objects more efficient, the Component Object Model (COM) specifies a number of complex shortcuts to its reference counting rules. The net result is that you couldn't trust the value of the reference count even if you had access to it.

According to COM rules, the only information you can depend on is *whether or not the reference count is zero*. You know when the reference count reaches zero, because your object's Terminate event occurs. Beyond that, there's no reliable information to be gleaned from reference counts.

**Note** The fact that you don't have to remember the COM reference counting rules is no small thing. Managing reference counts yourself is a lot more difficult than keeping track of which object variables in your program contain references to objects.

203

**Tip** Declare your object variables as class types, instead of As Object. That way, if you have a Widget object that isn't terminating, the only variables you need to worry about are those declared As Widget.



Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: .
Revision #: 7 Page: 78 of 96 Printed: 09/16/96 04:48 PM

For collections of object references, don’t use the Visual Basic Collection object by itself. Object references in a Visual Basic Collection object are stored in Variants — which, like variables declared As Object, can hold references to objects of any class. Instead create collection classes of your own that accept objects of only one class, as described in “Creating Your Own Collection Classes.” That way, the only collections you need to search for your Widget object are those of type Widget.

Organize your object into a hierarchy, as described in “Object Models.” If all of your objects are connected, It’s easy to write a procedure that walks through the whole model and reports on all the existing objects.

Don't declare variables As New. They're like those birthday candles that reignite after you blow them out: If you use one after you've set it to Nothing, Visual Basic obligingly creates another object.

**For More Information** Circular references are the most difficult kind to shut down cleanly. See “Object Models.”

The ProgWOb.Vbp sample application demonstrates a number of techniques for keeping track of objects, ranging from the simple and mundane to the downright dangerous.

204

# Object Models

Once you’ve defined a class by creating a class module and giving it properties and methods, you can create any number of objects from that class. How do you keep track of the objects you create?

The simplest way to keep track of objects is to declare an object variable for each object you plan to create. Of course, this places a limit on the number of objects you can create.

You can keep multiple object references in an array or a collection, as discussed in “Creating Arrays of Objects” and “Creating Collections of Objects” earlier in this chapter.

In the beginning, you’ll probably locate object variables, arrays, and collections in forms or standard modules, as you do with ordinary variables. As you add more classes, though, you’ll probably discover that the objects you’re using have clear relationships to each other.

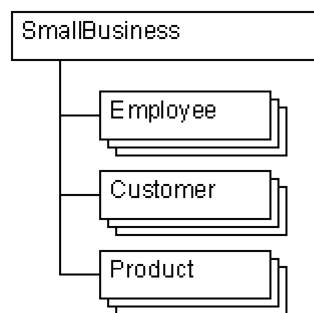
# Object Models Express Containment Relationships

*Object models* give structure to an object-based program. By defining the relationships between the objects you use in your program, an object model organizes your objects in a way that makes programming easier.

Typically, an object model expresses the fact that some objects are “bigger,” or more important than others — these objects can be thought of as containing other objects, or as being made up of other objects.

For example, you might create a SmallBusiness object as the core of your program. You might want the SmallBusiness object to have other types of objects associated with it, such as Employee objects and Customer objects. You would probably also want it to contain a Product object. An object model for this program is shown in Figure 9.11.

**Figure 9.11   An object model**




39

You can define four class modules, named SmallBusiness, Employee, Customer, and Product, and give them each appropriate properties and methods, but how do you make the connections between objects? You have two tools for this purpose: Object properties and the Collection object. The following code fragment shows one way to implement the hierarchy in Figure 9.11.

```
' Code for the Declarations section of the
' SmallBusiness class module.
Public Name As String
Public Product As New Product
Public Employees As New Collection
Public Customers As New Collection
```

205

The first time you refer to the Product property, the object will be created, because it was declared As New. For example, the following code might create and set the name and price of the SmallBusiness object’s Product object.

```
' Code for a standard module.
Public sbMain As New SmallBusiness
Sub Main
```

```
    sbMain.Name = "Velociraptor Enterprises, Inc."
    ' The first time the Product variable is used in
    ' code, the Product object is created.
    sbMain.Product.Name = "Inflatable Velociraptor"
    sbMain.Product.Price = 1.98
    .
    .   ' Code to initialize and show main form.
    .
End Sub
```

206

**Note** Implementing an object property with public variables is sloppy. You could inadvertently destroy the Product object by setting the property to Nothing somewhere in your code. It's better to create object properties as read-only properties, as shown in the following code fragment.

207

```
' Code for a more robust object property. Storage for
' the property is private, so it can't be set to
' Nothing from outside the object.
Private mProduct As New Product

Property Get Product() As Product
    ' The first time this property is called, mProduct
    ' contains Nothing, so Visual Basic will create a
    ' Product object.
    Set Product = mProduct
End If
```

208

## One-to-Many Object Relationships

Object properties work well when the relationship between objects is one-to-one. It frequently happens, however, that an object of one type contains a number of objects of another type. In the SmallBusiness object model, the Employees property is implemented as a Collection object, so that the SmallBusiness object can contain multiple Employee objects. The following code fragment shows how new Employee objects might be added to this collection.

```
Public Function NewEmployee(Name, Salary, HireDate, _
ID) As Employee
    Dim empNew As New Employee
    empNew.Name = Name       ' Implicit object creation.
    empNew.Salary = Salary
    empNew.HireDate = HireDate
    ' Add to the collection, using the ID as a key.
    sbMain.Employees.Add empNew, CStr(ID)
    ' Return a reference to the new Employee.
    Set NewEmployee = empNew
End Function
```

209

The NewEmployee function can be called as many times as necessary to create employees for the business represented by the SmallBusiness object. The existing employees can be listed at any time by iterating over the Employees collection.

**Note** Once again, this is not a very robust implementation. Better practice is to create your own collection classes, and expose them as read-only properties. This is discussed in "Creating Your Own Collection Classes."

210

**Tip** The Class Builder utility, included in the Professional and Enterprise editions of Visual Basic, can generate much of the code you need to implement an object model. Class Builder creates robust object properties and collection classes, and allows you to rearrange your model easily.

211

# Parent Properties

When you have a reference to an object, you can get to the objects it contains by using its object properties and collections. It's also very useful to be able to navigate up the hierarchy, to get to the object that contains the object you have a reference to.

Navigating upward is usually done with Parent properties. The Parent property returns a reference to the object's container.

You can find an example of a Parent property in "Adding Properties to Classes" earlier in this chapter.

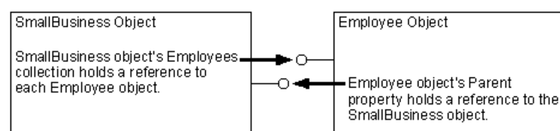
**Tip** When you assign a Parent property to an object in a collection, don't use a reference to the Collection object. The real parent of the object is the object that contains the collection. If the Parent property points to the collection, you'll have to use two levels of indirection to get to the real parent — that is, obj.Parent.Parent instead of obj.Parent.

212

## Parent Properties, Circular References, and Object Teardown

One of the biggest problems with Parent properties is that they create circular references. The "larger" object has a reference to the object it contains, and the contained object has a reference through its Parent property, creating a loop as shown in Figure 9.12.

**Figure 9.12 A case of circular references**




40

What's wrong with this picture? The way you get rid of objects when you're done with them is to release all references to them. Assuming the reference to the

SmallBusiness object is in a variable named sbMain, as earlier in this topic, you might write the following code:

```
Set sbMain = Nothing
```

213

Unfortunately, there's still a reference to the SmallBusiness object — in fact, there may be many references, because each Employee object's Parent property will hold a reference to the SmallBusiness object.

Since the SmallBusiness object's Employees collection holds a reference to each Employee object, none of the objects ever get destroyed.

### TearDown Methods

One solution is to give the SmallBusiness object a TearDown method. This could set all of the SmallBusiness object's object properties to Nothing, and also set all the Collection objects (Employees, Customers) to Nothing.

When a Collection object is destroyed, Visual Basic sets all the object references it was holding to Nothing. If there are no other references to the Employee and Customer objects that were contained in the Employees and Customers collections, they'll be destroyed.

Of course, if the Employee object is made up of finer objects, it will have the same circular reference problem its parent does. In that case, you'll have to give the Employee class a TearDown method. Instead of just setting the Employees Collection object to Nothing, the SmallBusiness object will first have to iterate through the collection, calling the TearDown method of each Employee object.

### It's Not Over Yet

Even then, not all the objects may be destroyed. If there are variables anywhere in your program that still contain references to the SmallBusiness object, or to any of the objects it contains, those objects won't be destroyed. Part of the cleanup for your program must be to ensure that all object variables everywhere are set to Nothing.

To test whether this is happening, you may want to add some debugging code to your objects. For example, you can add the following code to a standard module:

```
' Global debug collection
Public gcolDebug As New Collection

' Global function to give each object a unique ID.
Public Function DebugSerial() As Long
    Static lngSerial As Long
    lngSerial = lngSerial + 1
    DebugSerial = lngSerial
End Function
```

214

In each class module, you can put code similar to the following. Each class provides its own name where "Product" appears.

```
' Storage for the debug ID.
Private mlngDebugID As Long
```

```
Property Get DebugID() As Long
    DebugID = mlngDebugID
End Property

Private Sub Class_Initialize()
    mlngDebugID = DebugSerial
    ' Add a string entry to the global collection.
    gcolDebug.Add "Product Initialize; DebugID=" _
    & DebugID, CStr(DebugID)
End Sub

Private Sub Class_Terminate()
    ' Remove the string entry, so you know the object
    ' isn't around any more.
    gcolDebug.Remove CStr(DebugID)
End Sub
```

215

As each object is created, it places a string in the global collection; as it's destroyed it removes the string. At any time, you iterate over the global collection to see what objects haven't been destroyed.

**For More Information** This and other debugging techniques are illustrated in the ProgWOb.vbg sample application.

Object models assume new importance, and a different set of problems, when you use the Professional or Enterprise Edition of Visual Basic to create ActiveX components. See "General Principles of Component Design."

216

# Creating Your Own Collection Classes

There are three general approaches you can take to implementing object containment using collections. Consider the Employees collection of the SmallBusiness object discussed in "Object Models." To implement this collection you might:

- In the SmallBusiness class module, declare an `Employees` variable As Collection, and make it Public. This is the cheap solution.
- In the SmallBusiness class module, declare an `mcolEmployees` variable As Collection, and make it Private. Give the SmallBusiness object a set of methods for adding and deleting objects. This is the least object-oriented of the three designs.
- Implement your own collection class, by creating a collection class module named Employees, as described later in this chapter. Give the SmallBusiness object a read-only property of the Employees class.

41

The strategies are listed in order of increasing robustness. They could be characterized as the house of straw, house of sticks, and house of bricks approaches.

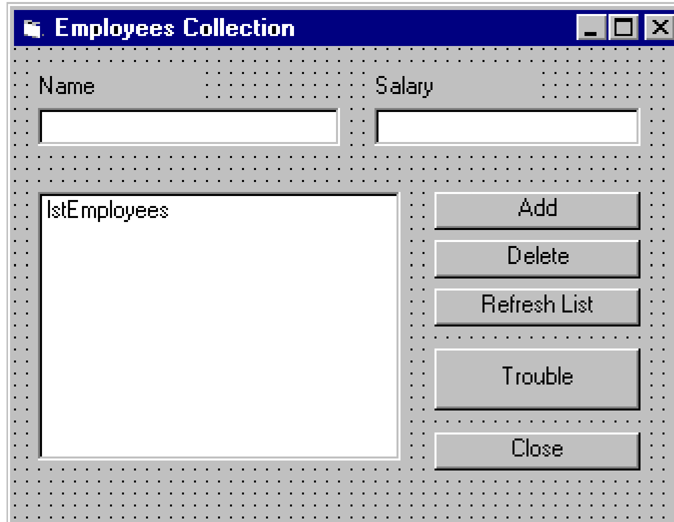


Filename: in.doc Project: A Traditional User's Guide Template
Template: Author: Ruth Silverio Last Saved By: .
Revision #: 7 Page: 84 of 96 Printed: 09/16/96 04:48 PM

# Public Collection Example: The House of Straw

To create the example, open a new project and insert two class modules. Draw five command buttons, a list box, two text boxes, and two labels on the form, as shown in Figure 9.13.

**Figure 9.13 Employees collection example**




42

The following table lists the property values you need to set for this example.

| Object | Property | Setting |
|---|---|---|
| Class module | Name | Employee |
| Class module | Name | SmallBusiness |
| Form | Caption | Employees Collection |
| First command button | Caption<br>Name | Add<br>cmdAddEmployee |
| Second command button | Caption<br>Name | Delete<br>cmdDeleteEmployee |
| Third command button | Caption<br>Name | Refresh List<br>cmdListEmployees |
| Fourth command button | Caption<br>Name | Trouble<br>cmdTrouble |
| Fifth command button | Caption<br>Name | Close<br>cmdClose |
| First label control | Caption | Name |
| Second label control | Caption | Salary |
| First text box | Name | txtName |

| Object | Property | Setting |
|---|---|---|
| | Text | (blank) |
| Second text box | Name<br>Text | txtSalary<br>(blank) |
| List Box | Name | lstEmployees |

217

In the Employee class module, add the following declarations and property procedures:

```
Option Explicit
' Properties of the Employee class.
Public Name As String
Public Salary As Long

' Private data for the write-once ID property.
Private mstrID As String

Property Get ID() As String
    ID = mstrID
End Property

' The first time the ID property is set, the static
' Boolean is also set.  Subsequent calls do nothing.
' (It would be better to raise an error, instead.)
Property Let ID(strNew As String)
    Static blnAlreadySet As Boolean
    If Not blnAlreadySet Then
        blnAlreadySet = True
        mstrID = strNew
    End If
End Property
```

218

The ID property is the key for retrieving or deleting an Employee object from the collection, so it must be set once and never changed. This is accomplished with a Static Boolean variable that is set to True the first time the property is set. The property can always be read, because there is a Property Get.

In the SmallBusiness class module, add the following declaration. The collection object will be created the first time the Employees variable is referred to in code.

```
Option Explicit
Public Employees As New Collection
```

219

## The Form Does All the Work

All of the remaining code goes into the form module. Add the following declaration in the Declarations section.

```
Option Explicit
Public sbMain As New SmallBusiness
```

220

The code in the cmdEmployeeAdd_Click event adds a member to the collection.

```
Private Sub cmdEmployeeAdd_Click()
```

```
    Dim empNew As New Employee
    Static intEmpNum As Integer
    ' Using With makes your code faster and more
    ' concise (.ID vs. empNew.ID).
    With empNew
        ' Generate a unique ID for the new employee.
        intEmpNum = intEmpNum + 1
        .ID = "E" & Format$(intEmpNum, "00000")
        .Name = txtName.Text
        .Salary = CDbl(txtSalary.Text)
        ' Add the Employee object reference to the
        ' collection, using the ID property as the key.
        sbMain.Employees.Add empNew, .ID
    End With
    txtName.Text = ""
    txtSalary.Text = ""
    ' Click the Refresh List button.
    cmdListEmployees.Value = True
End Sub
```

221

The code in the cmdListEmployees_Click event procedure uses a For Each ... Next statement to add all the employee information to the Listbox control.

```
Private Sub cmdListEmployees_Click()
    Dim emp As Employee
    lstEmployees.Clear
    For Each emp In sbMain.Employees
        lstEmployees.AddItem emp.ID & ", " & emp.Name _
        & ", " & emp.Salary
    Next
End Sub
```

222

The cmdEmployeeDelete_Click event uses the Collection object's Remove method to delete the collection member currently selected in the ListBox control.

```
Private Sub cmdEmployeeDelete_Click()
    ' Check to make sure there's an employee selected.
    If lstEmployees.ListIndex > -1 Then
        ' The first six characters are the ID.
        sbMain.Employees.Remove _
        Left(lstEmployees.Text, 6)
    End If
    ' Click the Refresh List button.
    cmdListEmployees.Value = True
End Sub
```

223

Add the following code to the Trouble button.

```
Private Sub cmdTrouble_Click()
    ' Say what!?
    sbMain.Employees.Add Me
End Sub
```

224

The cmdClose_Click event closes the application. When you close projects that use objects, do so by unloading all the forms, to ensure that any Terminate event

procedures in your class modules will get executed. By contrast, using the End statement stops a program abruptly, without executing Terminate events.

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

225

To add employees in the example, run the application, enter values in the two text boxes, and then choose the Add button. Add a few employees, and then experiment with the delete and list buttons.

### Robust as a Straw House

This simple implementation is not very robust. Because the Employees property is just a public Collection object, you could inadvertently access it from anywhere in your program. Furthermore, the Add method of the Collection object doesn't do any type checking. For example, the code in the Trouble button's Click event blithely inserts an object reference to the form into the collection of employees.

Click the Trouble button, and notice that no error occurs. Now click the Refresh List button. When the For Each ... Next loop encounters the unexpected object type, it causes error 13, Type mismatch.

This is an example of the kind of error you're exposed to when you build an object model with public Collection objects. Objects can be added from anywhere in your project, and there's no guarantee that they'll be properly initialized. If a programmer clones the code to add an employee, and the original code is later changed, the resulting errors can be very difficult to track down.

**For More Information** The example begun in this topic is continued in "Private Collection Example: The House of Sticks."

226

# Private Collection Example: The House of Sticks

This topic continues the code example begun in "Public Collection Example: The House of Straw." You may want to read that topic before beginning this one.

A somewhat more robust way to link Employee objects with the SmallBusiness object is to make the Collection object private. For this example, you'll reuse the form and most of the code from the "Public Collection" example.

The Employee class module is unchanged. The SmallBusiness class module, however, gets a complete facelift. Replace the declaration of the public Collection object with the following declaration, and add the Sub and Function procedures described in the following paragraphs.

```
Option Explicit
Private mcolEmployees As New Collection
```

227

As before, the code that adds an employee does most of the work. (You can take the block of code between the dotted lines out of the cmdEmployeeAdd_Click event procedure in the previous example.)

The important change is that the Add method of the Collection object can no longer be called from any module in your program, because colEmployees is Private. You can only add an Employee object using the EmployeeAdd method, which correctly initializes the new object:

```
' Method of the SmallBusiness class.
Public Function EmployeeAdd(ByVal Name As String, _
ByVal Salary As Double) As Employee
    ' - - - - - - - - - - - - - - - -
    Dim empNew As New Employee
    Static intEmpNum As Integer
    ' Using With makes your code faster and more
    ' concise (.ID vs. empNew.ID).
    With empNew
        ' Generate a unique ID for the new employee.
        intEmpNum = intEmpNum + 1
        .ID = "E" & Format$(intEmpNum, "00000")
        .Name = Name
        .Salary = Salary
        ' Add the Employee object reference to the
        ' collection, using the ID property as the key.
        ' - - - - - - - - - - - - - - - -
        mcolEmployees.Add empNew, .ID
    End With
    ' Return a reference to the new Employee.
    Set EmployeeAdd = empNew
End Function
```

228

The EmployeeAdd method returns a reference to the newly added Employee object. This is a good practice, because as soon as you create an object you will most likely want to do something with it.

The EmployeeCount, EmployeeDelete, and Employees methods *delegate* to the corresponding methods of the Collection object. Delegation means that the Collection object does all the work.

```
' Methods of the SmallBusiness class.
Public Function EmployeeCount() As Long
    EmployeeCount = mcolEmployees.Count
End Function

Public Sub EmployeeDelete(ByVal Index As Variant)
    mcolEmployees.Remove Index
End Sub

Public Function Employees(ByVal Index As Variant) _
As Employee
    Set Employees = mcolEmployees.Item(Index)
End Function
```

229

**Note** You can add extra functionality to these methods. For example, you can raise your own errors if an index is invalid.

230

The last method is Trouble. This method attempts to add an uninitialized Employee object to the collection. Any guesses what will happen?

```
' Method of the SmallBusiness class.
Public Sub Trouble()
    Dim x As New Employee
    mcolEmployees.Add x
End Sub
```

231

## Changes to the Form

You'll have to make a few changes to the form module. You can use the same module-level declarations used for the previous example, and the Click event for the Close button is the same, but the other event procedures have changed — the Add button code is much shorter, while the code for the Delete and List Employees buttons have changed in small but significant ways:

```
Private Sub cmdEmployeeAdd_Click()
    sbMain.EmployeeAdd txtName.Text, txtSalary.Text
    txtName.Text = ""
    txtSalary.Text = ""
    cmdListEmployees.Value = True
End Sub

Private Sub cmdEmployeeDelete_Click()
    ' Check to make sure there's an employee selected.
    If lstEmployees.ListIndex > -1 Then
        ' The first six characters are the ID.
        sbMain.EmployeeDelete 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.EmployeeCount
        With sbMain.Employees(lngCt)
            lstEmployees.AddItem .ID & ", " & .Name _
            & ", " & .Salary
        End With
    Next
End Sub
```

232

But what's all this extra code in cmdListEmployees_Click? Unfortunately, in pursuit of robustness you've given up the ability to use For Each ... Next to iterate through the items in the collection, because the Collection object is now declared Private. If you try to code the following, you'll just get an error:

```
' Won't work, because Employees isn't really a
' collection.
```

```
For Each emp In sbMain.Employees
```

233

Fortunately, the EmployeeCount method can be used to delimit the iteration range.

The Trouble button changes a little, too, but it's still, well, Trouble.

```
Private Sub cmdTrouble_Click()
    sbMain.Trouble
End Sub
```

234

Run the project and experiment with the Add, Delete, and Refresh List buttons. Everything works just like before.

When you click the Trouble button, once again no error is generated. However, if you now click the Refresh List button, you can see that the uninitialized Employee object has somehow been added to the collection.

How can this be? By making the Collection object private, you protect it from all the code in your program that's *outside* the SmallBusiness object, but not from the code *inside*. The SmallBusiness object may be large and complex, with a great deal of code in it. For example, it will very likely have methods like CustomerAdd, ProductAdd, and so on.

A coding error, or the creation of a duplicate of the EmployeeAdd method, can still result in erroneous data — even invalid objects — being inserted into the collection, because the private variable is visible throughout the class module.

**For More Information** This example is continued in "Creating Your Own Collection Class: The House of Bricks."

235

# Creating Your Own Collection Class: The House of Bricks

This topic continues the code example begun in "Public Collection Example: The House of Straw" and "Private Collection Example: The House of Sticks." You may want to read those topics before beginning this one.

The most robust way to implement a collection is by making it a class module. In contrast to the preceding examples, moving all the code for object creation into the collection class follows good object design principles.

This example uses the same form and the same Employee class module as the previous examples. Insert a new class module, and set its Name property to "Employees." Insert the following declarations and code into the new class module.

```
Option Explicit
Private mcolEmployees As New Collection
```

236

The Add, Count, and Delete methods of the Employees class are essentially the same as those of the old SmallBusiness class. You can simply remove them from the

SmallBusiness class module, paste them into the Employees class module, and change their names.

The names can change because it's no longer necessary to distinguish EmployeeAdd from, say, CustomerAdd. Each collection class you implement has its own Add method.

```
' Methods of the Employees collection class.
Public Function Add(ByVal Name As String, _
ByVal Salary As Double) As Employee
    Dim empNew As New Employee
    Static intEmpNum As Integer
    ' Using With makes your code faster and more
    ' concise (.ID vs. empNew.ID).
    With empNew
        ' Generate a unique ID for the new employee.
        intEmpNum = intEmpNum + 1
        .ID = "E" & Format$(intEmpNum, "00000")
        .Name = Name
        .Salary = Salary
        ' Add the Employee object reference to the
        ' collection, using the ID property as the key.
        mcolEmployees.Add empNew, .ID
    End With
    ' Return a reference to the new Employee.
    Set Add = empNew
End Function

Public Function Count() As Long
    Count = mcolEmployees.Count
End Function

Public Sub Delete(ByVal Index As Variant)
    mcolEmployees.Remove Index
End Sub
```

237

The Employees method of the SmallBusiness object becomes the Item method of the collection class. It still delegates to the Collection object, in order to retrieve members by index or by key.

```
' Method of the Employees collection class.
Public Function Item(ByVal Index As Variant) _
As Employee
    Set Item = colEmployees.Item(Index)
End Function
```

238

There's a nice touch you can add here. By making Item the default method of the Employees class, you gain the ability to code `Employees("E00001")`, just as you could with the Collection object.

## To make Item the default property

30 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box. In **Name** box, select the Item method.

31 Click **Advanced** to show the advanced features. In the **Procedure ID** box, select **(Default)** to make the Item method the default. Click **OK**.

43

**Note** A class can have only one default member (property or method).

239

## Enabling For Each … Next

Along with robustness, you get For Each … Next back. Once again you can delegate all the work to the Collection object, by adding the following method:

```
' NewEnum must return the IUnknown interface of a
' collection's enumerator.
Public Function NewEnum() As IUnknown
    Set NewEnum = mcolEmployees.[_NewEnum]
End Function
```

240

The important thing you're delegating to the Collection object is its *enumerator*. An enumerator is a small object that knows how to iterate through the items in a collection. You can't write an enumerator object with Visual Basic, but because the Employees class is based on a Collection object, you can return the Collection object's enumerator — which naturally enough knows how to enumerate the items the Collection object is holding.

The square brackets around the Collection object's _NewEnum method are necessary because of the leading underscore in the method name. This leading underscore is a convention indicating that the method is hidden in the type library. You can't name your method _NewEnum, but you can hide it in the type library and give it the procedure ID that For Each … Next requires.

### To hide the NewEnum method and give it the necessary procedure ID

32 On the **Tools** menu, click **Procedure Attributes** to open the **Procedure Attributes** dialog box. In **Name** box, select the NewEnum method.

33 Click **Advanced** to show the advanced features. Check **Hide this member** to make NewEnum hidden in the type library.

34 In the **Procedure ID** box, type **–4** (minus four) to give NewEnum the procedure ID required by For Each … Next. Click **OK**.

44

**Important** In order for your collection classes to work with For Each … Next, you must provide a hidden NewEnum method with the correct procedure ID.

241

## Not Much Left of the SmallBusiness Class

The SmallBusiness class will have considerably less code in it now. To replace the Collection object and all the methods you removed, there's a new declaration and a read-only property:

```
Option Explicit
Private mEmployees As New Employees

Public Property Get Employees() As Employees
    Set Employees = mEmployees
End If
```

242

This deserves a word of explanation. Suppose for a moment that you left out the Property Get, and simply declared `Public Employees As New Employees`.

Everything would work fine as long as nobody made any mistakes, but what if you accidentally coded `Set sbMain.Employees = Nothing`? That's right, the Employees collection would be destroyed. By making Employees a read-only property, you avert that possibility.

## Changes to the Form

The code for the form module is very similar to the preceding example. You can use the same module-level declarations, and the Click event for the Close button is the same.

The only change in most of the event procedures is replacing the old methods of the SmallBusiness class with the new methods of the Employees collection object:

```
Private Sub cmdEmployeeAdd_Click()
    sbMain.Employees.Add txtName.Text, txtSalary.Text
    txtName.Text = ""
    txtSalary.Text = ""
    cmdListEmployees.Value = True
End Sub

Private Sub cmdEmployeeDelete_Click()
    ' Check to make sure there's an employee selected.
    If lstEmployees.ListIndex > -1 Then
        ' The first six characters are the ID.
        sbMain.Employees.Delete _
        Left(lstEmployees.Text, 6)
    End If
    cmdListEmployees.Value = True
End Sub

Private Sub cmdListEmployees_Click()
    Dim emp As Employee
    lstEmployees.Clear
    For Each emp In sbMain.Employees
        lstEmployees.AddItem emp.ID & ", " & emp.Name _
        & ", " & emp.Salary
    Next
End Sub
```

243

Notice that you can use For Each … Next again to list the employees.

Run the project and verify that everything works. There's no code for the Trouble button this time, because encapsulation has banished trouble.

**For More Information** Read "The Visual Basic Collection Object" and "Collections in Visual Basic" for background on collections. The Class Builder utility included in the Professional and Enterprise editions will create collection classes for you.

The lessons of the House of Straw, House of Sticks, and House of Bricks examples are summed up in "The Benefits of Good Object-Oriented Design."

244

# The Benefits of Good Object-Oriented Design

This topic summarizes the results of the code example begun in "Public Collection Example: The House of Straw," and continued in "Private Collection Example: The House of Sticks" and "Creating Your Own Collection Class: The House of Bricks." You may want to read those topics before beginning this one.

Creating the Employees collection class results in a very clean, modular coding style. All the code for the collection is in the collection class (encapsulation), reducing the size of the SmallBusiness class module. If collections of Employee objects appear in more than one place in your object hierarchy, reusing the collection class requires no duplication of code.

## Enhancing Collection Classes

You can implement additional methods and properties for your collection classes. For example, you could implement Copy and Move methods, or a read-only Parent property that contains a reference to the SmallBusiness object.

You could also add an event. For example, every time the Add or Remove method changed the number of items in your collection, you could raise a CountChanged event.

## Robustness, Robustness, Robustness

You don't always have to implement collections in the most robust way possible. However, one of the benefits of programming with objects is code reuse; it's much easier to reuse objects than to copy source code, and it's much safer to use robust, encapsulated code.

A wise man once said, "If you want to write really robust code, you have to assume that really bad things will happen."

## Collection Classes and Component Software

If you're using the Professional or Enterprise Edition of Visual Basic, you can turn your project into an ActiveX component, so that other programmers in your organization can use the objects you've created.

## Steps to Implement a Collection Class

The following list summarizes the steps required to create a collection class.

5. Add a class module to your project, and give it a name — usually the plural of the name of the object the collection class will contain. (See "Naming Properties, Methods, and Events" earlier in this chapter.)
6. Add a private variable to contain a reference to the Collection object your properties and methods will delegate to.
7. In the Class_Initialize event procedure, create the Collection object. (If you want to defer creation of this object until it's needed, you can declare the private variable in step 2 As New Collection. This adds a small amount of overhead each time the Collection is accessed.)
8. Add a Count property and Add, Item, and Remove methods to your class module; in each case, delegate to the private Collection by calling its corresponding member.
9. When you implement the Add method, you can override the behavior of the Collection object's undiscriminating Add method by accepting only objects of one type. You can even make it impossible to add externally created objects to your collection, so that your Add method completely controls the creation *and initialization* of objects.
10. Use the Procedure Attributes dialog box to make the Item method the default for your collection class.
11. Add a NewEnum method, as shown below. Use the Procedure Attributes dialog box to mark it as hidden, and to give it a Procedure ID of –4 so that it will work with For Each … Next.

```
21Public Function NewEnum() As IUnknown
22  Set NewEnum = mcol.[_NewEnum]
23End Function
```

45

6**Note** The code above assumes that the private variable in step 2 is named mcol.

46

12. Add custom properties, methods, and events to the collection class.

47

**Note** The Class Builder utility, included in the Professional and Enterprise editions of Visual Basic, will create collection classes for you. You can customize the resulting source code.

245

**For More Information** You can read more about software components in *Creating ActiveX Components,* in the *Component Tools Guide*. If you have the Enterprise Edition of Visual Basic, you can read about using components in your business in *Building Client/Server Applications with Visual Basic.*

246