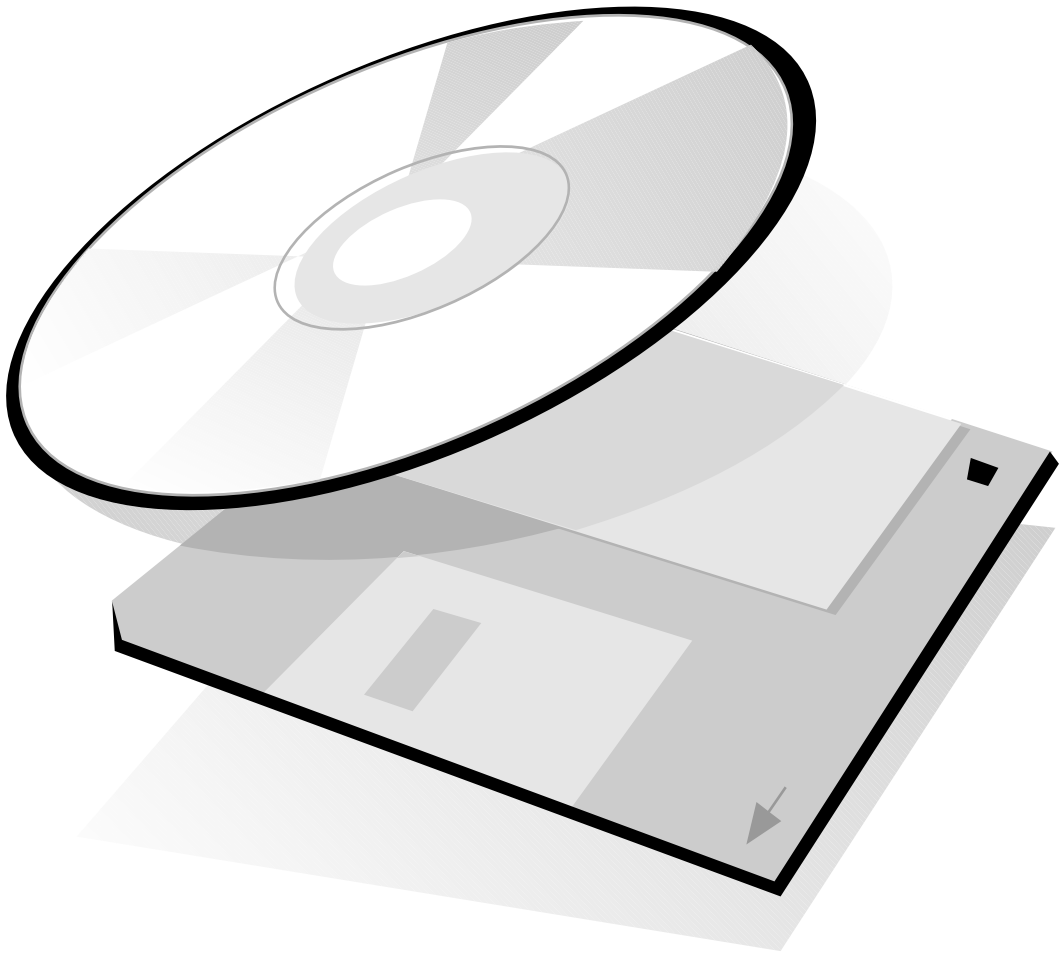
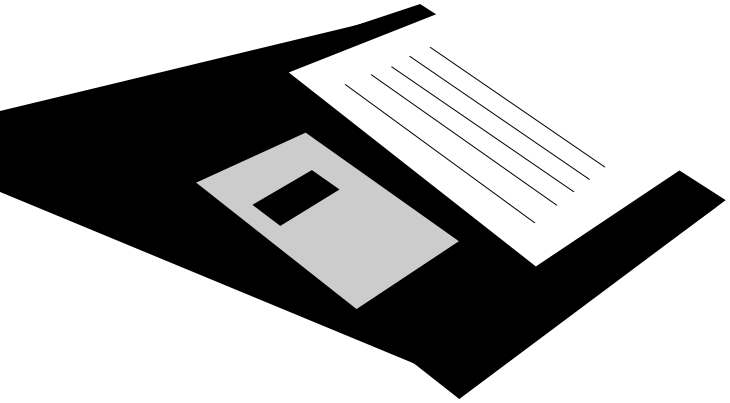


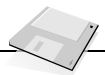
Chapter Eleven

Objects



At this point, you are probably pretty comfortable with the basic techniques for manipulating Visual Basic's built-in objects. To go further with Visual Basic, you will need to know more than just how to manipulate its objects by setting properties and applying methods. This chapter will show you new ways of manipulating Visual Basic's built-in objects (and get you started on creating your own). In particular, Visual Basic 4 is the first version of Visual Basic that gives you access to some of the power and so some of the advantages of *object-oriented programming*.

Object-oriented programming (OOP) seems to be the dominant programming paradigm these days, having replaced the *structured* programming techniques that were developed in the early 1970s. If you haven't worked with OOP before, you are probably wondering what all the hoopla is about. This chapter is designed to show you (well, at least to give you a glimpse). Since there's a fair amount of terminology needed to make sense of OOP, the chapter starts with some concepts and definitions. After this, you'll learn the basics of how Visual Basic implements the part of OOP that it supports.



note: *Visual Basic lets you add other objects (such as ones representing Word or Excel) to the toolbox. You can use those objects in ways similar to ordinary controls. See Chapter 18 for more on this.*

Getting Started with Object-Oriented Programming

Let's start with a question that, on the surface, seems to have nothing to do with programming: how did Gateway 2000 become a billion-dollar company faster than any other company in American history? Most people would probably say they made good computers and sold them at rock-bottom prices. But go further—how did they do *that*? Well, a big part of the answer is that they farmed out a lot of the work. They bought components from reputable vendors and

then assembled them. They didn't invest any money in designing and building power supplies, disk drives, motherboards, and so on. This made it possible for them to have a good product at a low price.

Ask yourself for a second how this could work. The obvious (and to a large extent, correct) answer is that what they were buying was "prepackaged functionality." For example, when they bought a power supply, they were buying something with certain properties (size, shape, and so on) and a certain functionality (smooth power output, amount of power available, and so on). Object-oriented programming springs from the same idea. Your program is made up of objects with certain properties and functions. You depend on the objects not to interact in undocumented ways with other objects or the code in your project. Whether you build the object or buy it might depend on the state of your wallet or how much time you have free. In either case, as long as the objects satisfy your specifications, you don't much care how the functionality is implemented. In OOP, the way people put it is that what you care about is what the objects *expose*.

So, just as Gateway doesn't care about the internals of a power supply as long as it does what they want, most programmers don't need to care how command buttons are implemented in Visual Basic as long as they do what *they* want. And, as you certainly know by now, on the whole, Visual Basic's objects do what you would expect them to!

The key to being most productive in object-oriented programming is to make your objects as complete as possible and, *as much as possible,* have the other objects and parts of your program tell the objects what to do. OOP jargon describes this by saying that what you do in object-oriented programming is *have clients send messages to objects*. By designing your objects to handle all appropriate messages and manipulate their data internally, you maximize reusability and minimize debugging time.

By now you have seen pretty clear evidence that Visual Basic's built-in objects fit this paradigm well. They are extremely rich in functionality. Of course, occasionally you may need to add your own objects to Visual Basic by either buying them from third-party vendors or writing them yourself. Generally speaking, it is rarely worth reinventing the wheel. If it takes you 40 hours to build a component in C++ and you can buy it for $200, one should really ask: is it worth it? Commercial vendors such as MicroHelp produce quality products that are worth the (usually) small cost. I have always used commercial products in my applications and have been generally happy with the results.

Unfortunately, Visual Basic misses implementing one of the key features of OOP here. Ideally, if you do have to write your own objects, another tenet of OOP would make this easier as well: objects can be built on other objects.

When you do this, the new object starts out by inheriting all the properties and functions of its parent—you can pick and choose whether you want to keep or modify any property or function of the parent. Visual Basic 4 simply doesn't allow this.

Vocabulary of OOP

Traditional structured programming consists of manipulating data. (This is one reason computer programming used to be called data processing.) You manipulate the data in specific ways that are theoretically sure to terminate. (These are usually called *algorithms*.) Now, for the next bit of terminology you have to know: computer scientists talk about *data structures* when they want to single out the arrangements used in your program for the data. All this explains in part why one of the most important computer scientists (he designed Pascal, for example), a Swiss professor named Niklaus Wirth, called his famous book on programming *Algorithms + Data Structures = Programs* (Prentice Hall, 1976). Notice that in Wirth's title the algorithms come first and the data structures come after. This mimics the way programmers worked at that time. First, you decided how to manipulate the data, then you decided what structure to impose on the data in order to make the manipulations easier. OOP puts both algorithms and data structures on the same level. With OOP you work with packages consisting of both data and the functions that manipulate them.

The rest of this section explains the basic terminology of OOP. There's a fair amount of it but it is worth learning for two reasons. The first is that you will need some of this terminology to understand the discussions in this chapter, the second is that knowing this terminology is useful if you move on to a full OOP language such as C++ or Delphi in order to extend Visual Basic.

CLASSES

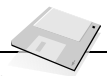
A *class* is usually described as the template or blueprint from which the object is actually made. The standard way of thinking about classes is to think of them as the cookie cutter and the actual object as the cookie. The "dough" in the form of memory will sometimes need to be allocated as well. Visual Basic is pretty good about hiding this "dough preparation" step from you. You almost never have to worry about creating memory for an object or releasing memory when a program ends under Visual Basic.

When you create an object from a class, you are said to have *created an instance* of the class. For example, all forms in Visual Basic are instances of the Form class, and an individual form in your application is actually a class you can use

to create new forms. (See the section on the New keyword a bit later in the chapter.) On the other hand, the controls on the toolbox represent individual classes, but an individual control on a form does not.

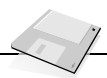
ENCAPSULATION

Encapsulation is the key concept in working with objects. Formally, encapsulation is nothing more than combining the data and behavior in one package.



note: *The data in an object is usually called its* instance variables *or* fields, *and the functions and procedures are its methods.*

A key rule in making encapsulation work is that programs should never (well, almost never) access the instance variables (fields) in an object directly. Programs should interact with this data only through the object's methods. (Properties in Visual Basic are designed to give you a way to interact with the instance variables without violating encapsulation.) Encapsulation is the way to give an object its "black box"-like behavior, which is the key to reuse and debugging efficiency.



note: *Visual Basic fully supports encapsulation.*

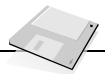
INHERITANCE

The ability to make classes that descend from other classes is called *inheritance*. The purpose of inheritance is to make building code for specialized tasks easier. The instance variables and methods of the descendent (sometimes called the *subclasses*) start out being the same.

note: *Visual Basic does not support inheritance for creating new subclasses.*

Subclasses will usually use (inherit) the same methods as the parent class. OOP (but not Visual Basic) allows you to define a new method in a subclass but give it the same name. This is called *overriding*. A true object-oriented language such as C++ (and not, alas, Visual Basic) allows you to go beyond simply overriding a method into what is usually called *polymorphism*. The idea

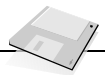
behind polymorphism is that while the message may be the same, the *object* determines how to respond. Polymorphism can apply to any method that is inherited from a base class.



note: *Visual Basic does not support polymorphism in any form for the objects you create.*

Polymorphism is helpful when creating new objects from old ones because it makes a programmer's job simpler. When you define a new object based on an existing object, you do not want to rebuild the parent's code in order to take into account that a new object exists. (Imagine some giant case statement in the parent class that just grows and grows!)

The key to making polymorphism work is called *late binding*. This means the compiler doesn't generate the code to call a method at compile time. Instead, every time you use a method with an object, the compiler generates code that lets it calculate which method to call using pointer information built into the object that called it. Methods that allow late binding are called *virtual methods* because they do not exist in the .EXE file but are only potentially there.



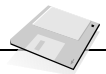
note: *We can only hope that the next version of Visual Basic will support class hierarchy and polymorphism. Microsoft has made no announcement on this, and your guess is as good as mine.*

Manipulating Objects Built into Visual Basic

The key to working with objects in Visual Basic is using variables of the special *object* type. For example, when you use the Me keyword to refer to the current object, you are using an object variable.

In general, you declare an object variable with the same Dim, Private, Public, Static, and so on keywords that you've already seen. Thus, you can have local, form-level, or public (global) object variables. Here are some examples:

```
Dim AForm As Form
Dim InfoBox As TextBox
Public AButton As CommandButton
Private MyBar As ScrollBar
```

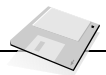


note: *Microsoft suggests using an "obj" prefix for object variables. For example:*

```
Dim objTheLabel As Label
```

In general, the name used for an object variable of a given control type is the name given in the help file for that control. You can define arrays of object variables in the same way you define an ordinary array:

```
Dim LotsOfTextBoxes(1 To 100) As TextBox
```



note: *You can use variant variables as the type instead of object variables, but this will slow your program down and make it harder to debug.*

When you want to make an object variable refer to a specific object of that type in your project, use the Set keyword. For example, if your project has a form named Form1 and a command button named Command1, your code would look like this:

```
Set AForm = Form1
Set AButton = Command1
```

tip: *The Set command can also be used to simplify lengthy control references.*

Here's an example:

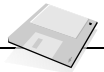
```
Dim Foo As TextBox
Set Foo = frmHelp.txtHelp
```

Now you can write

```
Foo.BackColor
```

instead of

```
frmhelp.txtHelp.BackColor
```



note: *It is important to note that the Set command does not make a copy of the object as a variable assignment would. Instead, the Set command points the object variable to the other object.*

In particular, you cannot use an assignment statement to make an object variable equal to, say, a text box. Trying to use code like this

```
Dim Foo As TextBox
Foo = Text1
```

will give you an error message.

That all the Set keyword does is point your variable to an object can occasionally lead to problems. For example, if you change a property of an object variable that is set to another object, the property of the original object changes as well (much like passing by reference in procedures does).

You can use the Is keyword to test whether two object variables refer to (have been set to) the same object. Suppose AControl and BControl are two control object variables. A line of code such as

```
If AControl Is BControl Then
```

lets you test whether they refer to the same object. (It is a wise precaution to find out whether changing the properties of one variable will also change the properties of the other!)

THE NEW KEYWORD

One case where you can create a new instance of a Visual Basic object at run time is when you use an existing form as the class. The syntax for this is a little different. Assume you have a form named Form1 in your project already. Then the statement

```
Dim AForm As New Form1
```

creates a new instance of Form1. This new instance has the same properties as the original Form1 at the time the code is executed. Use the New keyword only when you want Visual Basic to create a new instance of the original object. For example,

```
Dim AForm As New Form1
Dim BForm As New Form1
AForm.Show
BForm.Show
AForm.Move Left - 199, Top + 100
BForm.Move Left - 399, Top + 200
```

shows two copies of the original Form1. The locations are determined by the value of the Left and Top properties of the original Form1. (We needed to change them to prevent them from stacking one on another because instances inherit all the properties of their parent.)

It would be logical if you could also use the New keyword to create controls at run time; unfortunately, that isn't the way it works. Visual Basic 4 still uses the older (and somewhat clumsier) method of control arrays that you saw in the previous chapter. Only forms in Visual Basic 4 are classes (templates for new objects).

Thus a statement such as

```
Dim Foo As New Text1
```

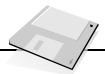
gives an error message when you try to compile it.

You usually use the New keyword with objects you create yourself. (See the sections "Collections" and "Creating a New Class Module" later in the chapter for more examples of using the New keyword.)

THE NOTHING KEYWORD

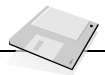
Once you use the Set keyword to assign an object to an object variable, you need to release the memory used for the object. You do this by setting the object variable to the keyword Nothing. For example:

```
Dim objFrm As New Form1
  ' code to manipulate the new instance of Form1 would go here
  '
  '
Set objForm = Nothing
```



note: *Since object variables merely point to the object, it is possible for several object variables to refer to the same object. When several object variables refer to the same object, you must set all of them to Nothing in order to release the memory and system resources associated with the object.*

(Memory may be released automatically: this happens, for example, after the last object variable referring to the object goes out of scope. However, relying on this is sloppy. For example, if you set a local object variable inside a procedure, set it to Nothing before the Sub is exited; don't rely on Visual Basic to clean up after you!)



note: *An uninitialized object variable can be thought of as having a current value of Nothing.*

GENERAL OBJECT VARIABLES

There are a few general types of object variables for use when you need to refer to objects of many different types. For example,

```
Dim objFoo As Control
```

gives you a way to refer to any control. Similarly,

```
Dim objGeneral As Object
```

lets you set the variable named objGeneral to *any* Visual Basic object.

tip: *Always use the most specific object variable you can find.*

For example, code with this statement

```
Dim txtFoo As TextBox
Set Foo = Text1
```

will always run faster than

```
Dim Foo As Control
Set Foo = Text1
```

which in turn will always run faster than

```
Dim Foo As Object
Set Foo = Text1
```

Manipulating Object Variables via Code

You have already seen how to manipulate individual objects by setting their properties or applying one of their methods to them. Suppose, however, you want to write a general procedure to manipulate properties of forms or controls or the forms and controls themselves—you simply don't yet have the techniques needed for this. This section explains them.

First off, *properties* of forms and controls can only be passed by value. For example, consider the following simple Sub procedure:

```
Sub ChangeText (ByVal X As String, Y As String)
  Y = X
End Sub
```

If you call it using the following code,

```
Call ChangeText(Form1.Caption, Y$)
```

then the current value of Y$ is the caption for Form1.

tip: *If you set the Tag property of the form or control to contain information otherwise not available at run time, you can write a general procedure using this technique to analyze the Control.Tag property in order to find information about the control that would otherwise not be available at run time.*

On the other hand, you will often want to affect the properties of a form or control by using a general procedure. For this, you have to pass the form or control as a parameter by reference. To do this, declare the argument to the procedure to be one of the object types. (You could use variants too, of course, but this should be avoided unless absolutely necessary because it is slower and also leads to code that is harder to debug.) For example, the following code makes a command button visible if it is invisible:

```
Sub MakeVisible (X As CommandButton)
  If X.Visible = False Then
    X.Visible  = True
  End If
End Sub
```

Notice that the parameter is declared to be an object variable of Command Button type and will be passed by reference (the default behavior for procedures). Otherwise, the code is pretty straightforward. Since X is being passed by reference, Visual Basic knows where in memory the object is located. Since it knows this location, it can change the properties of the object. You access properties of an object variable inside a procedure using the dot notation you have become familiar with. In this case, the code is straightforward: if the button isn't visible (so X.Visible = False), the procedure makes it visible.

tip: *This kind of procedure would usually be in a code module, since you will want to use it for many different buttons.*

As another example, if you often find yourself writing code to center a form on the screen, why not use the following general procedure:

```
Public Sub CenterForm(X As Form)
    X.Move (Screen.Width - X.Width)/2, _
(Screen.Height - X.Height)/2
End Sub
```

Then whenever you are in a procedure attached to a specific form, you can simply write

```
Center Me
```

to center the form on the screen.

Similarly, you can have a Sub or Function procedure that affects a property of a control. For example, a first approximation to a general procedure to change the caption on a control might look like this:

```
Sub ChangeCaption (X As Control, Y As String)
  X.Caption = Y
End Sub
```

Notice that this procedure uses the general Control type. However, suppose you tried to use this procedure in the form of

```
Call ChangeCaption(Text1, "New text")
```

where Text1 was the name of a text box. Then Visual Basic would give you a run-time error because text boxes do not have a Caption property.

The solution for this is to use a variant on the If-Then-Else loop in Visual Basic that allows you to determine what type of control is being manipulated. This takes the following form,

```
If TypeOf Control Is ControlType Then
 .
 .
 .
Else
 .
 .
 .
End If
```

where the *ControlType* parameter is the same as is used in declaring an object variable (Form, Label, TextBox, and so on).

For example, if all you wanted to do was work with both text boxes and all the other controls you wanted to change *did* have Caption properties, you could use

```
Sub ChangeCaptionOrText (X As Control, Y As String)
  If TypeOf X Is TextBox Then
    X.Text = Y
   Else
    X.Caption = Y
   End If
End Sub
```

You cannot use the keyword Not in this type of control structure, so you will often find yourself using an empty If clause. For example, if you wanted to play it safer:

```
Sub ChangeCaption (X As Control, Y As String)
  If TypeOf X Is TextBox Then
```

```
    '  Do Nothing
  Else
    X.Caption = Y
  End If
End Sub
```

Since there is also no version of the Select Case for controls, you may need the If-Then-ElseIf version of this control structure:

```
If TypeOf X Is...Then
 .
 .
ElseIf TypeOf X Is...Then
 .
 .
ElseIf TypeOf X Is...Then
 .
 .
Else
 . .
 .
End If
```

Collections

A Collection object is an object whose parts can be referred to individually as needed, *and* you still can refer to the object as a whole when needed. You have already seen the Printers collection in Chapter 6. Visual Basic also has built-in collections that give you information about all the forms in a project and all the controls on a specific form. They are called Forms and Controls. Just as with the Printers collection, the Count property of the Forms or Controls collection tells you how many forms you have loaded or how many controls are loaded on a specific form.

You can access individual forms or controls by writing, for example, Forms(0), Forms(1), and so on. Unfortunately, although the count starts at 0, Forms(0) is not necessarily the startup form. The order of the Forms( ) collection is unpredictable as is the order of the Controls collection. For example, the following code prints the captions of all the *loaded* forms in your project in the Debug window.

```
Dim I As Integer
For I = 0 To Forms.Count - 1
  Debug.Print Forms(I).Caption
Next I
```

(Since the Count property starts at 0, we go to one less than Forms.Count - 1.)

Although the preceding code works fine, most programmers would use the For-Each structure for iterating through a collection. They feel the For-Each structure makes the code a bit clearer when you need to iterate through all the elements in a collection. A framework for this structure takes the following form,

```
For Each Element In TheCollection

Next
```

as shown in the following rewritten version of the program to print the captions of all the loaded forms in a project:

```
Dim objForm As Form
For Each objForm In Forms
  Debug.Print objForm.Caption
Next
```

An Example of Using the Controls Collection with Set

The Set statement is also useful when working with collections of objects. For example, suppose you need to know a non-enabled control on your form. The following code finds one and assigns it to an object variable named NotEnabledControl:

```
Dim AControl As Control
Dim NotEnabledControl As Control
For Each AControl in Form1.Controls
  If Not(AControl.Enabled) Then
     Set NotEnabledControl = AControl
     Exit For
  End If
Next
```

This code moves through all the controls on a form until it finds one that is not enabled and then sets the AControl object variable to it.

Building Your Own Collections

Occasionally it is useful to build your own collections. The items in a collection (usually called its *members* or *elements)* can be of any type, and you can mix types in a collection if necessary.

Since a collection is an object, you must create it as an instance of a built-in class in Visual Basic. The class you need is called, naturally enough, the Collection class. For example

```
Dim X As New Collection
```

creates a new collection as an instance of the Collections class.

Just as with the Forms, Controls, or Printers collection, the Count property of each collection you create tells you the number of items in a collection. (Collections start out with no elements, so Count is 0.) Each element in a collection can be referred to by its index—just as you saw in the Forms and Controls collections. This means that the following gives you one way of dealing with all the elements in a collection:

```
For I = 1 To NameOfCollection.Count-1
  'work with NameOfCollection(I)
Next I
```

However, it is usually a bit clearer to use the For-Each structure.

Of course, you still don't know how to add or remove elements from a collection. But before moving on to the important Add and Remove methods that do this, you need to learn about one other method for working on a collection.

THE ITEM METHOD

The Item method is the default method for a collection; it is how you refer to (or return) a specific element of a collection. Its syntax is

CollectionObject.Item(index)

The *index* parameter specifies the position of a member of the collection. It is a long integer (you can have *lots* of elements in a collection) and goes to the number of items in the collection. For example

```
MyCollection.Item(1)
```

is the first item in the collection.

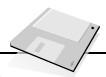
caution: *Collections you create start with an index of 1 and go up to the count of the collection. The built-in collections (Forms, Controls, and Printers) start at 0 and go up to the Count -1.*

In general, Visual Basic lets you use a key to access the elements in a collection. This key is set up at the time you add the element to the collection. Using a key rather than an index is often more effective: you can easily associate a useful mnemonic as the key. For example, this means a statement such as

```
Debug.Print Forms(1).Caption
```

is actually the same as:

```
Debug. Print Forms.Item(1).Caption
```



note: *The Item method is the default method for a collection.*

THE ADD METHOD

Once you create the collection by using the New keyword, you use the Add method to add items to it. The trick in using the Add method is to remember you have to set up a variant variable first to hold the information before you use the Add method. For example:

```
Dim Versions As New Collection
Dim Foo As Variant
Foo = "Visual Basic 3.0"
Versions.Add(Foo)
Foo = "Visual Basic 4.0"
Versions.Add(Foo)
```

In general, the Add method has the following syntax (it supports named arguments by the way):

CollectionObject.Add *item as Variant [, key as string][, before As Long] [, after As Long]*

Here are short descriptions of the parts of the Add method:

- *CollectionObject* is any object or object variable that refers to a collection.
- The *item* parameter is required. Since the information will be held in a variant variable, it can be an expression of any type. (As mentioned previously, you can mix types in a collection.)
- The *key* parameter is optional. It must be a string expression, and within the collection it must be unique or you'll get a run-time error. For example:

```
Dim Presidents As New Collection
Dim Foo As Variant
Foo = "George Washington"
Presidents.Add Foo, "Didn't lie"
Foo = "John Adams"
Presidents.Add item := Foo, key:= "Proper Bostonian"
```

  Now you can access George by:

```
Presidents.Item("Didn't lie")
```

  But remember that the match to the string in the key must be perfect. For example, this doesn't work:

```
Presidents.Item("didn't lie")
```

- The optional *before* and *after* parameters are usually numeric expressions that evaluate to a (long) integer. The new member is placed right before (right after) the member identified by the before (after) argument. If you use a string expression, it must correspond to one of the keys that was used to add elements to the collection. You can specify before or after positions but not both.

THE REMOVE METHOD

When you need to remove items form a collection, you use the Remove method. It too supports named arguments and its syntax is

CollectionObject.Remove *index*

Here, as you might expect, the *index* parameter is used to specify the element you want removed. If *index* is a numeric expression, it must be a number between 1 and the collection's Count property. If it's a string expression, it must exactly match a key to an element in the collection.

The Object Browser

You've seen how to use the Object Browser (shown in Figure 11-1) to look at the built-in constants in Visual Basic and to navigate among the procedures you have written. The Object Browser can do far more. In particular, it gives you complete access to the classes, objects, and their methods and properties that you can use in your Visual Basic projects.

More precisely, objects that are usable in Visual Basic are usually collected into *object libraries*. For example, there are Visual Basic's object library, the Visual Basic for Applications' object library, Excel's object library, and so on. An object library contains the information that Visual Basic needs to build instances of its objects as well as information on the methods and properties of the object in the library. One of the purposes of the Object Browser is to help you understand the object libraries in your project better.

To bring up the Object Browser shown in Figure 11-1:

- Choose View | Object Browser, press F2, or use the toolbar shortcut (the eighth tool).

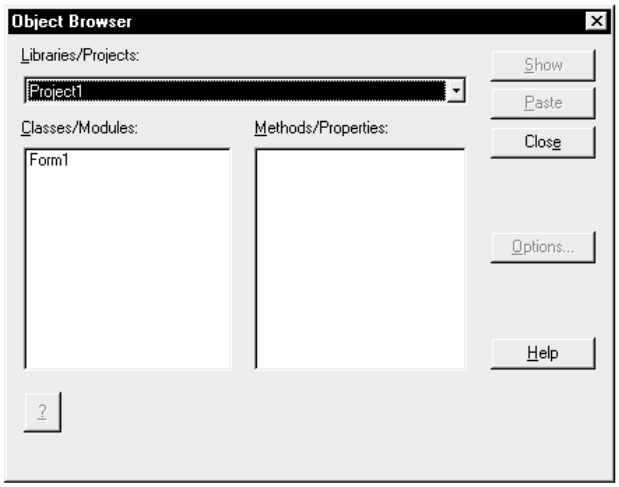
To close the Object Browser:

- Click the Close button.

(The Object Browser also closes whenever you choose the Paste button in order to paste the current selection into the active Code window.)

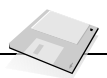
What follows are short descriptions of the parts of the Object Browser.

Figure 11-1
The Object Browser



LIBRARIES/PROJECTS DROP-DOWN LIST BOX

This list box displays the libraries available to your project and, as you have seen earlier, it also lists the modules in your project. In general, you can select from available object libraries, including the ones in the current Visual Basic project. Once you choose a library, other parts of the Object Browser let you look at the classes, modules, procedures, methods, and properties of that library.



note: *When you use Tools|References to add another object library to Visual Basic, you will add information about its object library to the Object Browser, and it is automatically listed in this drop-down list box.*

CLASSES/MODULES

As you have already seen in Chapter 9, if you select your own Visual Basic project in the Libraries/Projects box, this box displays modules from your project, including any classes you defined in the current project. You also have already used this box to get at the library of built-in constants in both Visual Basic and Visual Basic for Applications. When you select another object library in the Libraries/Projects list box, the Classes/Modules pane displays the classes available in that library.

METHODS/PROPERTIES

You already saw in Chapter 9 how to use Methods/Properties to display the procedures in the current project. Once you select an object library in the Libraries/Projects list box, the Object Browser gives you a list of the methods and properties for the class that you have selected in the Classes/Modules box.

THE "?" BUTTON

Clicking this button displays the online Help topic for the item selected in the Classes/Modules or Methods/Properties box. In particular, the Object Browser places the syntax for the item you have selected next to this button.

SHOW

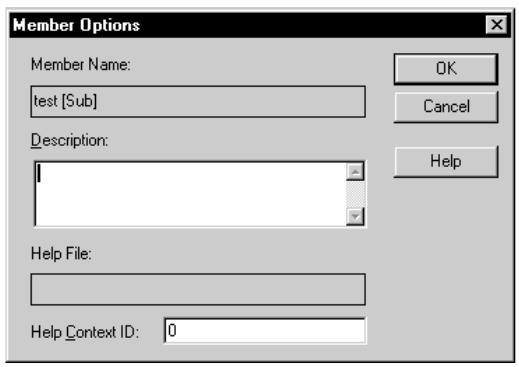
Clicking on Show takes you to the code for the selected procedure in the Code window.

PASTE

Clicking on Paste pastes a template of the Visual Basic code needed for the procedure, property, or method directly into your Code window at the current insertion point.

OPTIONS

Clicking on Options displays the Member Options dialog box, as shown here. You can use this box to add help information or comments about any modules, procedures, classes, properties, and methods that you define in your project.



Creating an Object in Visual Basic

You can build objects in two ways in Visual Basic. One is by adding custom properties to an existing form and then using that form as a class (template) for new instances of the form. Each new instance of the form will have the new properties. The second is by using a special type of module called a *class module*. Class modules have the advantage that they can be compiled separately (see Chapter 18) and used by other Windows applications. The disadvantage is that, at present, they can have no visual components associated with them. Classes created out of a form, on the other hand, are obviously visual, but they cannot be compiled separately for use in some other project. (They would have to be added at the design stage whenever they were used.)

Still, in both cases, you use the same ideas for adding properties to these classes (templates). Once you think about it the right way, the terminology used actually makes sense—but at first it can be confusing. First off, what are the most basic things you will want to be able to do with a new property?

- You want to get its current value.
- You want to assign a new value to it.

For the first situation, you use a special type of procedure called a *Property Get* procedure. For the second you use a *Property Let* procedure. (The way I remember the terminology is that a Let statement is the way you make assignments in BASIC.)

For example, suppose you want to add a custom property to a form that will tell you whether a form named frmNeedsToBeCentered is centered and also center it if you set the property to True. Here's what you need to do.

First, set up a Private variable in the declarations section of the form. (It's a Private variable to enforce encapsulation.)

```
Private IsCentered As Boolean
```

Then add the following procedures to the form:

```
Public Property Let Center(X As Boolean)
  IsCentered = X 'used for the current state of the property
  If X then
    Me.Move (Screen.Width - Me.Width)/2, _
Screen.Height - Me.Height)/2
```

```
  End If
End Sub
```

The first line of code uses the Private variable to store the current value of the property. Now you can use a line of code like

```
Me.Center = True
```

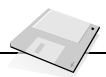
to center the form (or any instance of it). From another form or code module, you can use a line of code like:

```
frmNeedsToBeCentered.Center = True
```

Of course, it might also be useful to know whether a form is centered. For this you need to use a Property Get procedure that returns a Boolean:

```
Public Property Get Center() As Boolean
 Center = IsCentered
End Sub
```

This picks up the current value of the IsCentered Private variable that you are using to hold the information about the current value of the property. (Notice that Property Get procedures are a bit like Function procedures: you assign a value to them within the body of the procedure.)



note: *You may be wondering why all this bother? You can certainly use a Public function to determine whether a form is centered or not. The point is that the designers of Visual Basic 4 are trying to give objects as much "black box" behavior as they can. Using a Public function to determine whether a form is centered would partially defeat this. All this being said, in my opinion, given the current limited object orientation of Visual Basic, all Property procedures in a form are really doing is setting the stage for what will be in later versions of Visual Basic!*

In any case, you can use code like this:

```
If Not(frmNeedsToBeCentered.Center) Then MsgBox _
("Why did you move the form?")
```

General Property Procedures

Did you notice how the Property Let and Property Get procedures in the example worked in tandem? The value returned by the Property Get procedure is of the same type as the one used in the assignment for Property Let. In general, the number of arguments for a Property Get is also one less than that of the corresponding Property Let (the last argument being the one that will be changed). A Property Get procedure that you write without a corresponding Property Let procedure gives you a read-only property—since you have no way to change it.

The full syntax for a Property Let procedure template looks like this:

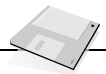
```
[Public | Private][Static] Property Let name [(arglist)]
    [statements]
    [name = expression]
    [Exit Property] ' if need be
End Property
```

- Use Public to make the Property Let procedure accessible to every procedure in every module.
- Use Private to make the Property Let procedure accessible only to other procedures in the module where it is declared.

The other keywords work as they would in any procedure. Use the Static keyword if you need the Property Let procedure's local variables preserved between uses. The Exit Property keywords give you a way to immediately exit from a Property Get procedure, and so on. The name of the Property Let procedure must follow standard variable naming conventions, except that the name can, and will most often, be the same as a Property Get or Property Set procedure in the same module.

The full syntax for a Property Get procedure template looks like this:

```
[Public | Private][Static] Property Get name [(arglist)][As type]
    [statements]
    [name = expression]
    [Exit Property] ' if need be
End Property
```



note: *The name and type of each argument in a Property Get procedure must be the same as the corresponding arguments in the Property Let procedure—if it exists. The type of the value returned by a Property Get procedure must be the same data type as the last argument in the corresponding Property Let procedure if it exists.*

The last type of Property procedure is the Property Set statement. Use this when you need to set a reference to an object instead of just setting a value (for example, when you want to set a printer different than the current one). Here is the syntax:

```
[Public | Private][Static] Property Set name [(arglist)]
    [statements]
    [Exit Property]
    [statements]
End Property
```

Building Your Own Classes

Although you can add custom properties to a form and then use them as templates for new objects, the most common way to build a new class (template) for new objects in Visual Basic is to use a class module. A class module object contains the code for the custom properties (via Property procedures) and methods that objects defined from it will have.

You can then create new instances of the class from any other module or form in your project. (You can even compile class modules for use by other applications as in-process OLE servers—see Chapter 18 for more on this important concept.) A class module cannot have a visible interface of its own. Each class module you create gives you, naturally enough, a single class (template) for building new instances of that class. However, you can have as many class modules in a project as you like (subject only to operating system constraints of course).

As you might expect, once you have a class module, you use the New keyword to create new instances of it. For example, if FirstClass is the name of a class module in your project:

```
Dim AnInstance As New FirstClass
```

you use Property procedures to define the properties of your class and use Public Sub and Public Function procedures for its methods.

Creating a New Class Module

You create a new class module at design time by choosing Insert | Class Module. Each class module can respond to only two events: Initialize and Terminate. They are triggered when someone creates an instance of your class or terminates it. (As you might expect, the Terminate event for a class module is triggered when the class created via the New keyword is set to Nothing. *It does not occur if the application stops because of the End statement*.)

There are three properties for a class module that you set at design time via the Properties window. As you might expect, the Name property determines the name of the class. The other two properties are described next.

PUBLIC

The Public property determines whether other applications can write code to invoke the properties and methods of your class. This property can only be set to True in the Professional and Enterprise editions of Visual Basic.

INSTANCING

If you set the Instancing property to True, then other applications outside your project can declare new instances of your class. (This is appropriate when using a class module in OLE—see Chapter 18.) You must have the Public property set to True to be able to set the Instancing property to True.

An Example: A Deck of Cards Class Module

Start by imagining you want to provide a toolkit for the designers of computer-generated card games. You obviously need an object that takes the place of a deck of cards. Since class modules in Visual Basic can't be visual, you only need to be concerned about the data and methods this deck of cards object must support.

Let's call this class module CardDecks. This object needs to expose individual cards and have methods for shuffling the deck and dealing the cards. (You will use the Initialize event to build up the deck of cards.)

The code for creating this nonvisual object might look like this. First off, start with the Private variables used for the data:

```
Private Deck(0 To 51) As Integer
Private TheCard As String
Private Position As Integer
```

The Deck array will be used to hold the integers that are the internal representation of the cards. The Private TheCard variable will be used for the Property procedures to encapsulate the card. (This will make it easy to change the names of the cards for a different country, for example.)

The Initialize procedure simply fills the array with 52 consecutive integers:

```
Private Sub Class_Initialize()
Dim I As Integer
For I = 0 To 51
  Deck(I) = I
Next I
End Sub
```

Now it's on to the methods. First off, there's got to be a Shuffle method for shuffling the deck. It might look like this:

```
Public Sub Shuffle()
 Dim X As Integer, I As Integer
 Dim Temp As Integer, Place As Integer
 Randomize
  For I = 0 To 5199 '10 times through the deck should be
enough
    Place = I Mod 52
    X = Int(52 * Rnd)
    Temp = Deck(Place)
    Deck(Place) = Deck(X)
    Deck(X) = Temp
  Next I
End Sub
```

Of course, you can easily add an argument to this procedure to control how many "shuffles" are made. If you do, then the method this procedure generates would have an argument:

```
Foo.Shuffle 10
```

Next, give the read-only property that tells you the current card. It simply looks up the current value of the Private ThisCard variable:

```
Public Property Get CurrentCard() As String
  TheCard = CalculateCard(Deck(Position))
  CurrentCard = TheCard
End Property
```

(It is read-only because there is no associated Property Let procedure to change the current card.)

Notice we call a private procedure that converts the integer in the Deck array to the card called TheCard. The DealCard method might look like this:

```
Public Function DealCard() As String
  If Position > 51 Then Err.Raise Number :=vbObjectError + 32144,_
Description := "Only 52 cards in deck!"
  TheCard = CalculateCard(Deck(Position))
  DealCard = TheCard
Position = Position + 1
End Function
```

Finally, here's the private procedure for converting a number in the card array to a string describing the card:

```
Private Function CalculateCard(X As Integer) As String
  Dim Suit As Integer, CardValue As Integer
  Suit = X \ 13
  Select Case Suit
   Case 0
     TheCard = "Clubs"
   Case 1
     TheCard = "Diamonds"
   Case 2
     TheCard = "Hearts"
   Case 3
     TheCard = "Spades"
   End Select

  CardValue = X Mod 13
  Select Case CardValue
   Case 0
    TheCard = "Ace of " + TheCard
   Case 1 To 9
    TheCard = Str$(CardValue + 1) + " of " + TheCard
   Case 10
     TheCard = "Jack of " + TheCard
```

```
    Case 11
      TheCard = "Queen of " + TheCard
    Case 12
      TheCard = "King of " + TheCard
    End Select
CalculateCard = TheCard
End Function
```

Now all you have to do to use this class module is have a line such as

```
Dim MyDeck As New CardDeck
```

before you start working with it. For example, you could test it with the following code:

```
Private Sub Form_Load()
Dim MyDeck As New CardDeck, I As Integer
  MyDeck.Shuffle
  For I = 1 To 20
    MyDeck.DealCard
    MsgBox MyDeck.CurrentCard
  Next I
End Sub
```