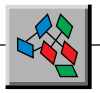

Take Advantage of Property Procedures

Don't be intimidated by Property procedures. Like most of VB4's new features, they've been added for good reasons.

by Chris Barlow

With the long-awaited release of Visual Basic 4.0 and its integrated VBA engine comes some difficult new concepts for Visual Basic 3.0 programmers to grasp as they move to this new form of VB. In the October issue of *VBPJ*, I gave an overview of the VBA engine's new features and explained how to use some of them. But there is a lot to learn and some of the features don't seem to make sense at first glance. Since that issue, many readers have asked questions revolving around the new feature called Property procedures.

If you've worked with Visual Basic or OLE Objects you understand the concept of *methods* and *properties*. Although there are some gray areas, generally properties hold a value such as the Text property of the text box or the Width property of the Screen object. Methods, on the other hand, perform some action or function such as the Move method of a form or the AddItem method of a list box.

You're also familiar with Sub and Function procedures—the Function procedures return a value and Sub procedures do not. In the past most programmers thought of methods as Sub procedures and properties as public variables.

It is interesting to examine some of these gray areas that separate properties and methods. For example, the Width method property of the Form object acts like a normal property when you use it on the right side of the equal sign to get the value of the form width:

```
SaveWidth = Me.Width
```

But when you use the Width property on the left side of the equal sign to set the value for the form width, it seems to act more like a method because it also performs the action of changing the dimensions of the form:

```
Me. Width = Save Width + 100
```

Chris Barlow is president and CEO of SunOpTech, a developer of manufacturing decision-support applications including the ObjectBank and the ObjectJob Systems, where Chris and Ken Henderson hold a software patent related to decentralized distributed asynchronous object-oriented systems. Chris holds degrees from Harvard Business School and Dartmouth College where he worked with Drs. Kemeny and Kurtz on the BASIC language. Reach Chris on the Internet at ChrisB@SunOpTech.com or through SunOpTech's World Wide Web server at www.SunOpTech.com.

On the other hand, the OpenRecordset method of the Database object acts like a method when it performs the action of opening the recordset. But it also acts like a property when it returns a value—the Recordset object:

```
MyRecordset = Mydb.OpenRecordset(MyTable)
```

These gray areas can lead to some confusion when trying to distinguish between properties and methods. Before the release of Visual Basic 4.0, however, you could not create custom properties and methods. So, the distinction was not too important! You just took the properties and methods as the developer defined them. Now that you can create custom properties and methods for your forms and classes, you need to be careful how you make the distinction so that you won't confuse the programmers who use your code or classes.

As if this is not complicated enough, Visual Basic 4.0 introduced a new type of procedure called Property procedures. I received this question more than once: "How can Properties be procedures?"

These procedures add powerful capabilities to the Visual Basic language that you'll need to understand if you want to write robust classes and forms. It is these new types of procedures that allow you to expose new properties of a form or class, yet retain control over how these properties are set and retrieved.

First, I'll examine Visual Basic 4.0's ability to add new properties and methods to a form. One of your goals should be to write code that is easy for another programmer to pick up and understand. The ability to create new properties and methods makes it much easier to meet this goal.

For example, when working with a form the program must know if the user has changed something on the form—this is commonly called making the form "dirty." So most programmers would like to be able to check a Dirty property of the form to know whether something has changed. Prior to Visual Basic 4.0, you would have had to create a variable, perhaps called gDirty, to contain a Boolean value. Then you could check this global variable from anywhere in the project. However, suppose this was defined as a global variable and this program was changed to an MDI application where you could have multiple instances of this form. Now you would need an array of gDirty() to track the state of each instance of the form. This would get very messy!

GETTING DOWN AND DIRTY

But Visual Basic 4.0 makes this kind of work easy. Simply define a public form-level variable, Dirty, and refer to it just as you would any other property of the form. This code checks the active MDI child's new Dirty property and calls a procedure called SaveForm if the form is dirty:

```
If frmMain.ActiveForm.Dirty then SaveForm
```

Similarly, you can define custom methods for a form by adding public Sub or Function procedures. A common one is a Center method that will center a form by calling Me.Center:

```
Public Sub Center()
Move (Screen.Width - Width) \ 2, (Screen.Height - Height) \ 2
End Sub

Private Sub Form_Load()
Me.Center
End Sub
```

The Center method defined in this Sub procedure looks and acts like any other method of the form. However, there is a real weakness when you create a new property with the simple technique of adding a public variable to a form. Because the variable is public, you can change its value from anywhere in the project and you can set the variable to any value without any kind of validation. In practice this is not a big problem with form properties because usually you are the person who develops the form and then writes the code to work with the form. You know what you want to do with the properties on the form. But this weakness quickly becomes a real problem when you look at the properties you create in a class—particularly if these properties will be the exposed properties in an OLE Server. In this case you may be the one developing the OLE Server, but you have no idea who will develop the program to use your new class. Do you really want that programmer to be able to change the value of any of the properties in your class at any time without any chance to validate these changes? I don't think so!

So how do you protect the properties in your form or class so that you can create truly robust classes? Use the new Property procedures. Visual Basic 4.0 implements three Property procedures that can be used to set and get property values:

- Property Get returns the value of a property.
- Property Set sets the value of a property to an object as in a Set statement.
- Property Let sets the value of a property to other than an object as in the Let statement that is implicit in the "=" operation.

The Property Get procedure is a block of code that returns the property value. The Property Let or Property Set is used to set the value. A complete property would have a Property Get procedure and either a Property Let or a Property Set procedure.

For example, as in the first example I presented, suppose you want to have a Dirty property that indicates whether the form has changed. You want that property to be accessible from anywhere in the project. However, unlike the first example, you don't want that property to be set by any routine outside of the form. You want to make the property appear to be "read-only." You would only implement a Property Get procedure to return the value from a private variable accessible only within the form. Meanwhile, the Property Let procedure would not change the value of this private variable:

```
Option Explicit
Private bDirty As Boolean

Public Property Get Dirty() As Boolean
Dirty = bDirty
End Property

Public Property Let Dirty(val As Boolean)
End Property
```

Notice how the Property Get procedure returns the value in the private variable bDirty, but the Property Let procedure is "empty." Because it does nothing, no one else can change thevalue of bDirty. The private variable bDirty holds the actual value, but the programmer using the Dirty property would not use the bDirty variable. Because this variable is private to the form it can only be set by procedures within the form. The programmer can use this code to get the value of the Dirty property:

```
If frmMain.Activeform.Dirty then SaveForm
```

KEEP IT IN RANGE

You can also use Property procedures to make sure a certain property is within an expected range. For example, suppose you want to set up a property called WorkWeek as the number of days a person works each week. Whenever this property is set, you want to make sure it is greater than or equal to zero and less than eight. You would start out by defining the private variable to hold the value for WorkWeek and write a standard Property Get procedure to return the value in this private variable:

```
Private iWorkWeek As Integer
Public Property Get WorkWeek() As Integer
WorkWeek = iWorkWeek
End Property
```

Then you would write the Property Let procedure to set the value of the private variable iWorkWeek to a valid number:

```
Public Property Let WorkWeek(val As Integer)
Select Case val
Case Is < 0: iWorkWeek = 0
Case Is > 7: iWorkWeek = 7
Case Else: iWorkWeek = val
End Select
End Property
```

This procedure will set iWorkWeek to zero if given a value of less than zero and seven if given a value greater than seven. Notice the use of the colon to combine two Visual Basic statements on the same line. I usually don't recommend that you use this feature because it can lead to confusing code. However, in short Case statements such as the ones I just presented, I believe it helps to make the code more readable.

Finally, you use Property Set procedures in place of Property Let procedures when a reference to an object needs to be returned. Suppose you want to contain an object such as a text box within a property. You would create a standard Property Get procedure with an object type to return a reference to your private object:

```
Private ctlTxt As Object
Public Property Get TxtBox() As Object
TxtBox = ctlTxt
End Property
```

Then you would create a matching Property Set procedure to set your private object to the passed reference:

```
Property Set TxtBox(T As Object)
   Set ctlTxt = T
End Property
```

It is always difficult to get used to new features in a language, and judging by my e-mail, these Property procedures have been one of the more intimidating areas. As you can see, like most of the new features in Visual Basic 4.0, Property procedures have been added for good reasons and it is worth the effort to learn how to use them effectively. ☒