Chapter 6
OOP Enhancements

Object orientation is one of Visual FoxPro's most powerful features. VFP 7 has a variety of improvements in this area, including new properties and methods, and even a pair of new events.

The massive changes in VFP 3 changed FoxPro from a procedural language to an object-oriented one. Each subsequent version has added new classes or enhanced existing classes. While VFP 7 doesn't include any new base classes, it does add a number of features to existing classes, as well as improving the way we work with them.

Creating classes

Creating classes in code (rather than with the Class Designer) is taking on increasing importance for several reasons.

The Session class (introduced in VFP 6 Service Pack 3), which is extremely useful for creating Automation servers, can't be subclassed in the Class Designer. In addition, some of the new features in VFP 7, such as strong typing, apply only to code classes (that is, non-visual classes). (See "Using the session class in COM servers" and "Storing VFP 7 COM classes" in Chapter 12, "Building World-Class COM Servers in VFP 7," to learn more about the importance of the Session class and strong typing.)

In VFP 6 and earlier versions, when you have a class definition in code, you have to be sure to SET PROCEDURE (or SET CLASSLIB) to the file containing the class library for the parent class before instantiating the class. VFP 7 makes it easier to write class definitions by adding an optional OF <classlib> clause to DEFINE CLASS. For example, to subclass the _form class in the FoxPro Foundation Classes, you can begin the class definition like this:

```
DEFINE CLASS frmMyForm AS _form OF HOME()+"FFC\_BASE.VCX"
```

DEFINE CLASS has another new keyword: IMPLEMENTS. See Chapter 13, "Implementing Interfaces," for an explanation of the meaning and use of this keyword.

There's another change that only affects coded class libraries. In earlier versions, when you instantiate an object from a PRG-based class, if the Name property isn't explicitly given a value in the properties section of the class definition, the Name of the object is set to the class name plus a number. Numbers are assigned sequentially, increasing each time you instantiate a new object of that class (that is, myObject1, myObject2, myObject3, and so forth). This approach to object naming is different from that used for VCX-based classes, and, more importantly, the need to determine the next available number can slow down code considerably when many objects of a single class are instantiated.

In VFP 7, this behavior is gone. All unnamed objects are assigned the name of the class they belong to rather than a unique name.

Exploring classes

The AMEMBERS() function lets you explore a class or an object. Depending on the value you pass for the third parameter, it fills an array with a list of the properties of the object, a list of all members of the object (properties, methods, and contained objects), or just a list of contained objects. In VFP 6 and earlier, AMEMBERS() works only on native VFP objects.

VFP 7 extends AMEMBERS() in several ways. First, the function now works on COM objects, as well as native objects. It also provides additional information about native objects. Passing 3 for the third parameter indicates that the function should return a four-column array—the contents of the columns are shown in **Table 1**. A third parameter of 3 is also the key to exploring COM objects.

***Table 1**. What's in an object? Calling AMEMBERS() with 3 as the third parameter returns a four-column array.*

Column	Contents
1	Name of the property, event, or method.
2	Type of item. For VFP objects, the possible values are "Property", "Event", or "Method". For COM objects, the possible values are "PropertyPut", "PropertyGet", "PropertyPutRef", and "Method".
3	Empty for properties of VFP objects. For methods of VFP objects, the parameter list. For properties and methods of COM object, the member's signature, consisting of the parameter list, plus the return value.
4	The help string for the item.

For example, to get a list of the members of the Windows Scripting Host's FileSystemObject, you can use this code:

```
oWSH = CreateObject("Scripting.FileSystemObject")
nMemberCount = AMEMBERS( aWSHMembers, oWSH, 3 )
```

Here's a partial listing of the array created. (The actual array has 27 rows.)

```
   (    1,    1)     C     "BuildPath"
   (    1,    2)     C     "Method"
   (    1,    3)     C     "(Path as String, Name as String) as String"
   (    1,    4)     C     "Generate a path from an existing path and a name"
   (    2,    1)     C     "CopyFile"
   (    2,    2)     C     "Method"
   (    2,    3)     C     "(Source as String, Destination as String,
[OverWriteFiles as Logical=.T.])"
   (    2,    4)     C     "Copy a file"
   (    3,    1)     C     "CopyFolder"
   (    3,    2)     C     "Method"
   (    3,    3)     C     "(Source as String, Destination as String,
[OverWriteFiles as Logical=.T.])"
   (    3,    4)     C     "Copy a folder"
   (    4,    1)     C     "CreateFolder"
   (    4,    2)     C     "Method"
   (    4,    3)     C     "(Path as String) as IFolder"
   (    4,    4)     C     "Create a folder"
   (    5,    1)     C     "CreateTextFile"
   (    5,    2)     C     "Method"
```

```
   (    5,    3)     C     "(FileName as String, [Overwrite as Logical=.T.],
[Unicode as Logical=.F.]) as ITextStream"
   (    5,    4)     C     "Create a file as a TextStream"
```

AMEMBERS() has also acquired a cFlags parameter (the fourth parameter) that lets you specify, for native objects, which members to return. **Table 2** lists the flag characters. The flags in each filter group are mutually exclusive. However, by default, when you concatenate multiple flag characters into the cFlags parameter, they're combined with OR, so a cFlags parameter of "HP" includes all hidden and protected properties, events, and methods (PEMs). Passing "GU" includes all members that are either public or user-defined in the result. (Be aware that the "C" flag doesn't catch changes to array properties.)

***Table 2**. Choosing members—AMEMBERS() new, fourth, parameter lets you filter the list of members returned.*

Flag character	Filter group	Meaning
P	Visibility	Protected
H	Visibility	Hidden
G	Visibility	Public
N	Origin	Native
U	Origin	User-defined
I	Inheritance	Inherited
B	Inheritance	Base
C	Changed	Changed
R	Read-only	Read-only

There are two special flags. Including the "+" anywhere in the cFlags parameter indicates that the filters should be combined with AND rather than OR. So, for example, passing "GU+" includes only members that are both public and user-defined.

The second special flag is "#", which adds a column to the resulting array. The new column shows the flags (the same values as in Table 2) that apply to each member.

This code creates an instance of the _MoverLists class from the FoxPro Foundation Classes, and then lists the Protected members of the class, including their flags:

```
oObject = NewObject( "_MoverLists",HOME()+"FFC\_CONTROLS" )
AMEMBERS( aMemberList, oObject, 3, "P#" )
LIST MEMORY LIKE aMemberList
AMEMBERLIST         Pub      A
   (    1,    1)     C     "ADDTOPROJECT"
   (    1,    2)     C     "Method"
   (    1,    3)     C     ""
   (    1,    4)     C     "Dummy code for adding files to project."
   (    1,    5)     C     "CPUI"
   (    2,    1)     C     "NINSTANCES_ACCESS"
   (    2,    2)     C     "Method"
   (    2,    3)     C     ""
   (    2,    4)     C     "Access method for nInstances property."
   (    2,    5)     C     "CPUI"
   (    3,    1)     C     "NINSTANCES_ASSIGN"
   (    3,    2)     C     "Method"
```

```
   (    3,    3)    C     "vNewVal"
   (    3,    4)    C     "Assign method for nInstances property."
   (    3,    5)    C     "CPUI"
   (    4,    1)    C     "NOBJECTREFCOUNT_ACCESS"
   (    4,    2)    C     "Method"
   (    4,    3)    C     ""
   (    4,    4)    C     "Access method for nObjectRefCount property."
   (    4,    5)    C     "CPUI"
   (    5,    1)    C     "NOBJECTREFCOUNT_ASSIGN"
   (    5,    2)    C     "Method"
   (    5,    3)    C     "m.vNewVal"
   (    5,    4)    C     "Assign method for nObjectRefCount property."
   (    5,    5)    C     "CPUI"
```

This example includes all members for the class (because all three visibilities are listed), but also includes the flags. (Only a portion of the output is shown.)

```
AMEMBERS(aMemberList, oObject, 3, "GPH#")
LIST MEMORY LIKE aMemberList

AMEMBERLIST         Pub      A
   (    1,    1)    C     "ACTIVECONTROL"
   (    1,    2)    C     "Property"
   (    1,    3)    C     ""
   (    1,    4)    C     "References the active control on an object."
   (    1,    5)    C     "GRNI"
   (    2,    1)    C     "ADDOBJECT"
   (    2,    2)    C     "Method"
   (    2,    3)    C     "cName, cClass"
   (    2,    4)    C     "Adds an object to a container object at run time."
   (    2,    5)    C     "GNI"
   (    3,    1)    C     "ADDPROPERTY"
   (    3,    2)    C     "Method"
   (    3,    3)    C     "cPropertyName,eNewValue"
   (    3,    4)    C     "Adds a new property to an object."
   (    3,    5)    C     "GNI"
   (    4,    1)    C     "ADDTOPROJECT"
   (    4,    2)    C     "Method"
   (    4,    3)    C     ""
   (    4,    4)    C     "Dummy code for adding files to project."
   (    4,    5)    C     "CPUI"
   (    5,    1)    C     "AOBJECTREFS"
   (    5,    2)    C     "Property"
   (    5,    3)    C     ""
   (    5,    4)    C     "Array of object references properties."
   (    5,    5)    C     "GUI"
   (    6,    1)    C     "BACKCOLOR"
   (    6,    2)    C     "Property"
   (    6,    3)    C     ""
   (    6,    4)    C     "Specifies the background color used to display text
and graphics in an object."
   (    6,    5)    C     "GNI"
```

The form ShowAMembers.SCX in the Developer Downloads available at www.hentzenwerke.com lets you experiment with AMEMBERS().

New and enhanced PEMs

VFP 7 includes a variety of changes to the properties, events, and methods (PEMs) of its classes. There are brand-new PEMs, PEMs added to additional objects, and new values for some properties. This section looks first at the changes that affect multiple classes, and then explores those affecting only a single class.

Multi-class changes

Many of the changes to PEMs apply to more than one class, some of them to quite a few classes.

MouseEnter and MouseLeave events

Exposing new events is unusual for Visual FoxPro, but in VFP 7, all visible controls have new MouseEnter and MouseLeave events that give you the opportunity to take action as the mouse passes through the control. These events take the same parameters as MouseMove: the mouse button or buttons that are currently pressed; which, if any, of the Ctrl, Shift, and Alt keys is pressed; and the current mouse position. You can use these methods to enable and disable certain options based on the mouse position. They're also convenient for manipulating the new VisualEffect property of buttons (described in "Changes to controls" later in this chapter).

Objects collection

Every container class in VFP has a way of accessing its members, generally through a collection named for the member type. For example, PageFrame has a Pages collection and Grid has a Columns collection. However, some of the containers also offer a more generic way to access their members—the Objects collection. Objects is a COM collection complete with Count and Item properties.

In VFP 7, all container classes have an Objects collection, including those that didn't in VFP 6 (CommandGroup, DataEnvironment, Grid, PageFrame, and OptionGroup).

SpecialEffect property

Many controls have a new "Hot Tracking" setting (2) for SpecialEffect that makes them flat except when the mouse passes over them. At that point, depending on the particular control, they become either raised or depressed. For check boxes and option buttons, the Hot Tracking setting works only when Style is set to Graphical. Like the changes to menus discussed in Chapter 3, "New and Better Tools," this setting makes it easier to write applications that follow the new flat-look interface style.

The form UIChanges.SCX in the Developer Downloads available at www.hentzenwerke.com demonstrates the MouseEnter and MouseLeave events, as well as the Hot Tracking setting for SpecialEffect and command buttons' new VisualEffect property (discussed in "Changes to controls" later in this chapter).

WriteMethod() method

The WriteMethod() method allows you to programmatically add code to a method. In VFP 7, it has a new (third) parameter that allows you to create methods on the fly. When the method specified by the first parameter does not exist and the lCreateMethod parameter is True, the method is created. This approach works only at design time, and the form or class must be saved after the method is added. These limitations are reasonable since WriteMethod() is intended for use in builders.

Form and toolbar changes

A couple of changes affect forms and/or toolbars. All of them help to build more standard Windows applications.

hWnd property

Every window in Windows has a "handle" that identifies it. In VFP 7, the window handle of forms and toolbars is finally directly accessible—through the new hWnd property. In older versions, you need to make a couple of function calls to get this information. **Figure 1** shows a form that passes its hWnd to an API function and changes itself to a circle. The code for the form (as exported by the Class Browser) is shown in **Listing 1**.

The form in Figure 1 is available as MkCircle.SCX in the Developer Downloads at www.hentzenwerke.com.

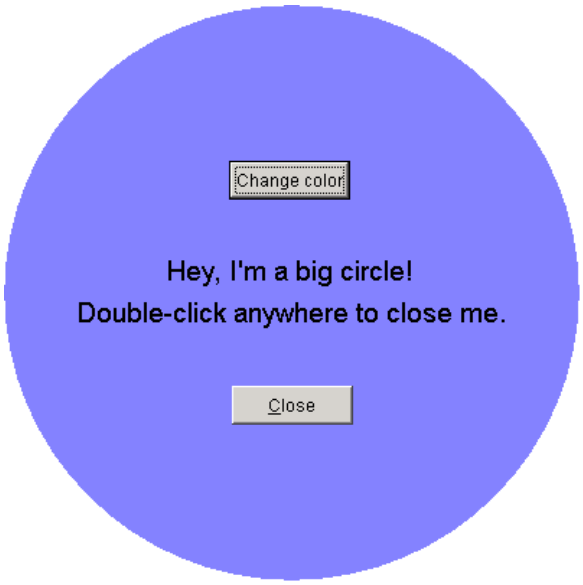

***Figure 1**. Getting a handle on a form. This form passes its window handle to the API function SetWindowRgn in order to change the form into a circle. Double-click anywhere on the form to close it.*

***Listing 1**. Using a form handle—this code creates the form shown in Figure 1.*

```
PUBLIC ofrmcircle

ofrmcircle=NEWOBJECT("frmcircle")
ofrmcircle.Show
RETURN

DEFINE CLASS frmcircle AS form


   Height = 400
   Width = 400
   DoCreate = .T.
   AutoCenter = .T.
   BorderStyle = 0
   Caption = "Form"
   Movable = .F.
   TitleBar = 0
   WindowType = 1
   BackColor = RGB(128,128,255)
   Name = "frmCircle"


   ADD OBJECT lblcircle AS label WITH ;
      AutoSize = .T., ;
      FontSize = 14, ;
      BackStyle = 0, ;
      Caption = "Hey, I'm a big circle!", ;
      Height = 25, ;
      Left = 113, ;
      Top = 173, ;
      Width = 173, ;
      Name = "lblCircle"


   ADD OBJECT lbldblclick AS label WITH ;
      AutoSize = .T., ;
      FontSize = 14, ;
      BackStyle = 0, ;
      Caption = "Double-click anywhere to close me.", ;
      Height = 25, ;
      Left = 50, ;
      Top = 202, ;
      Width = 300, ;
      Name = "lblDblClick"


   ADD OBJECT cmdcolor AS commandbutton WITH ;
      Top = 108, ;
      Left = 156, ;
      Height = 27, ;
      Width = 84, ;
      Caption = "Change color", ;
      Name = "cmdColor"
```

```
   ADD OBJECT cmdclose AS commandbutton WITH ;
      Top = 264, ;
      Left = 158, ;
      Height = 27, ;
      Width = 84, ;
      Caption = "\<Close", ;
      Name = "cmdClose"


   PROCEDURE DblClick
      ThisForm.Release()
   ENDPROC


   PROCEDURE Init
      LOCAL nhWnd, nWidth, nHeight, nRegion

      DECLARE INTEGER CreateEllipticRgn IN gdi32 ;
              INTEGER X1 , INTEGER Y1 , INTEGER X2 , INTEGER Y2
      DECLARE INTEGER SetWindowRgn IN user32 ;
              INTEGER HWND, INTEGER hRgn , INTEGER bRedraw

      nhWnd = This.HWnd
      nWidth = This.WIDTH / 1 && change ratio
      nHeight = This.HEIGHT / 1 && change ratio
      * Call API to convert an otherwise regular form into a circular one.
      nRegion = CreateEllipticRgn(0, 0, nWidth, nHeight)
      SetWindowRgn(nhWnd, nRegion, 1)
   ENDPROC


   PROCEDURE lblcircle.DblClick
      ThisForm.DblClick()
   ENDPROC


   PROCEDURE lbldblclick.DblClick
      ThisForm.DblClick()
   ENDPROC


   PROCEDURE cmdcolor.Click
      nColor = GETCOLOR(ThisForm.BackColor)
      IF nColor <> -1
         ThisForm.BackColor=nColor
      ENDIF
   ENDPROC


   PROCEDURE cmdclose.Click
      ThisForm.Release()
   ENDPROC


ENDDEFINE
```

The main VFP window, accessed through the system variable _VFP, and its client area, accessed using _Screen, also have the hWnd property. Each has a different value for it. See

Chapter 8, "Resource Management," for more about the differences between _VFP and _Screen in VFP 7.

ShowInTaskBar property

Forms have a new ShowInTaskBar property that determines whether top-level forms appear in the Windows taskbar. By default, the property is True and any top-level form has an independent presence in the taskbar. When ShowInTaskBar is set to False, there's no taskbar item for the form and it minimizes to the desktop, rather than to the taskbar.

Style property

The Style property has been added to Separators, the objects that let you space items in a toolbar. By default, Style is set to 0-Normal. But you can set Style to 1-Vertical Rule. This new setting allows you to create toolbars that look like those in commercial applications—with a sunken vertical bar between groups of buttons. Note that the vertical line shows up only at run time, not at design time.

Figure 2 shows a "standard" toolbar where the Separators have their Style set to 0. **Figure 3** shows the same toolbar with the Separators' Style property set to 1.

***Figure 2**. Invisible separators—the Style property for the separators in this toolbar is set to 0-Normal.*

***Figure 3**. Visible separators—in this version of the toolbar, the Style property for the separators is set to 1-Vertical Rule, and thus they have a visual presence at run time.*

The toolbar class shown in Figure 2 and Figure 3 is in the Chapter6.VCX class library in the Developer Downloads available at www.hentzenwerke.com.

To test the example toolbar class, instantiate it to a variable and set Visible to True. If you pass False or no parameters to the object, you get the toolbar in Figure 2 (invisible separators):

```
oToolbar = NewObject("tbrStandard", "chapter6.vcx")
oToolbar.Visible = .T.
```

To see the version with the vertical rules, pass True to the Init() method:

```
oToolbar = NewObject("tbrStandard", "chapter6.vcx", "", .T.)
oToolbar.Visible = .T.
```

Changes to controls

Finally, there are a few items that affect individual controls. As with the others, many of these respond to long-time developer requests.

Buttons

Command buttons have a new property, VisualEffect, that lets you raise or depress the button at run time. VisualEffect is read-only at design time. VisualEffect has three available settings: 0-None, 1-Raised, and 2-Depressed.

Setting VisualEffect to 1-Raised in MouseEnter and restoring it to the default setting of 0-None in MouseLeave gives the same results as setting SpecialEffect to the new 2-Hot Tracking setting.

The Close *button on the form UIChanges.SCX in the Developer Downloads available at www.hentzenwerke.com demonstrates the VisualEffect property.*

Grids

Since Visual FoxPro 3.0 was first released, developers have wondered why the BeforeRowColChange and AfterRowColChange events weren't divided into separate BeforeRowChange, BeforeColChange, AfterRowChange, and AfterColChange events. While VFP 7 doesn't go quite that far, the new RowColChange property does make it easy to know why the two events fired. It contains a value that indicates what changed: the row (1), the column (2), both (3), or neither (0). (You get the "neither" value when the grid is first displayed and when it's refreshed.) Your code can check that value and act accordingly.

The new HighlightRowLineWidth property indicates how many pixels should be used to create a highlight around the current row. This setting is only used when HighlightRow is set to its default value of True.

Headers

A WordWrap property has been added to the Header object in grids, so that headers can occupy more than one line. This may be the single most requested grid-related feature.

ProjectHook changes

When project hooks were added in VFP 6, developers immediately found uses for them. But a few features were missing. VFP 7 plugs those holes with three new events: QueryNewFile, Activate, and Deactivate.

QueryNewFile fires when you begin the process of adding a file to a project. That happens when you click the New button in the Project Manager. Previously, no event fired when a new file was added.

The Activate and Deactivate events for the ProjectHook object are like those of other classes—they fire when the object becomes active and when it loses focus, respectively. In the case of project hooks, however, that's when the project associated with the project hook is activated or deactivated. This means you now have the ability to modify the VFP environment

as you switch between projects, offering the chance to change things like the VFP PATH, field mappings, and other project-specific settings.

Summary

VFP 7's changes to the OOP part of the language make it easier to create classes and work with them, as well as giving forms and controls more of the behaviors developers and users want. VFP 7 also provides the tools needed to create interfaces in the Windows 2000 style.