New Language Features in Visual FoxPro 6.0

By Tamar E. Granor, Ph.D.

Each new version of FoxPro has introduced some new commands and functions (and since VFP 3, new properties, events and methods). VFP 6.0 is no exception. As in previous versions, many of the language enhancements are focused in a few well-defined areas, but there are changes all over the language.

VFP 6 provides a project object that exposes many of its properties and methods in code. While the project object is an automation object, not a native VFP object, the ProjectHook object that can be associated with each project is pure VFP. These two objects can be used to automate various project-related processes, as well as to provide tracking and other management services to projects.

OLE drag-and-drop is new in VFP 6. Beginning in this version, it's possible to drag data between ActiveX controls and native controls, and between VFP apps and other applications. To support this technology requires a number of properties, events and methods.

Another large group of new language elements is a group of file-handling functions promoted from the FoxTools library to full-fledged members of VFP.

Beside these three substantial groups, there are literally dozens of other new or enhanced commands, functions, properties, events, methods and even a few new system variables in VFP 6. This session takes a look at many of the changes, digging deep into a few of them.

Project Management Tools

FoxPro has had good project management tools since the introduction of the Project Manager in FoxPro 2. The integration of source control in VFP 5 made these tools even more useful. However, until now, manipulating a project programmatically meant either opening the .PJX as a table or using a lot of KEYBOARD commands. No more.

VFP 6 exposes open projects as objects of type Project. (Project is an Automation object, so it can be accessed by other applications through the VFP Automation server.) VFP itself hosts a collection (called Projects) of open projects, as well as a pointer to the ActiveProject. Each project is, in turn, composed of a collection of Files. Almost every operation you'd want to perform on a project, from adding an object to it to building it to cleaning it up has a corresponding method. In addition, almost every piece of information stored in a project is available via a property of either the project itself or one of its files.

Copyright 1998, Tamar E. Granor, Ph.D.

Here's a simple example. To go through a whole project and display the name and description of each file in it, you can use code like this:

```
PROCEDURE ShowProjFiles
LPARAMETER oProject

LOCAL oFile

FOR EACH oFile IN oProject.Files
   ? oFile.Name,oFile.Description
ENDFOR

RETURN
```

You can get your hands on the active project at any time by using _VFP.ActiveProject, and look at all open projects by working with _VFP.Projects. A new NOSHOW clause of MODIFY PROJECT means that you can open a project for manipulation without making it visible.

In addition to the ability to directly manipulate projects and their contents so that we can now write code for all the tasks normally performed interactively using the Project Manager, there's another level of control. Every project object can have an associated ProjectHook object. ProjectHook is a new VFP base class with events that fire before and after significant project actions.

For example, the ProjectHook's BeforeBuild method fires when a project's Build method has been called, but before the build actually occurs, giving us the opportunity to take some action, whether it's logging the build, checking for certain conditions, displaying a dialog, or something else. A number of projecthook events' names begin with "Query," like "QueryAddFile". These events fire just before the indicated project action occurs and let us decide whether that action should proceed. For example, issuing NODEFAULT in QueryAddFile keeps the specified file from being added to the project.

Here's the code for a simple ProjectHook class (found in NewLang.VCX in the conference materials) that simply tells you what's going on in the project.

```
DEFINE CLASS prjshowme AS projecthook

PROCEDURE Destroy
WAIT WINDOW "Closing project" NOWAIT
ENDPROC

PROCEDURE Init
WAIT WINDOW "Opening project" NOWAIT
ENDPROC

PROCEDURE QueryAddFile
LPARAMETERS cFileName
WAIT WINDOW "Adding file "+cFileName NOWAIT
ENDPROC

PROCEDURE QueryModifyFile
LPARAMETERS oFile, cClassName
IF EMPTY(cClassName)
   WAIT WINDOW "Modifying file " + oFile.Name NOWAIT
ELSE
   WAIT WINDOW "Modifying class " + cClassName + " of " + oFile.Name NOWAIT
ENDIF
```

Copyright 1998, Tamar E. Granor, Ph.D.

```
ENDPROC

PROCEDURE QueryRemoveFile
LPARAMETERS oFile, cClassName, lDeleteFile

LOCAL cDelete
IF lDeleteFile
   cDelete = "Deleting"
ELSE
   cDelete = "Removing"
ENDIF

IF EMPTY(cClassName)
   WAIT WINDOW cDelete + " file " + oFile.Name NOWAIT
ELSE
   WAIT WINDOW "Removing class " + cClassName + " from " + oFile.Name NOWAIT
ENDIF
ENDPROC

PROCEDURE QueryRunFile
LPARAMETERS oFile
WAIT WINDOW "Running file "+oFile.Name NOWAIT
ENDPROC

PROCEDURE BeforeBuild
LPARAMETERS cOutputName, nBuildAction, lRebuildAll, lShowErrors, lBuildNewGuids

LOCAL cBuildType
DO CASE
CASE nBuildAction = 1
   cBuildType = "rebuild project"
CASE nBuildAction = 2
   cBuildType = "build APP"
CASE nBuildAction = 3
   cBuildType = "build EXE"
CASE nBuildAction = 4
   cBuildType = "build DLL"
ENDCASE

WAIT WINDOW "About to " + cBuildType + " " + cOutputName NOWAIT
ENDPROC

PROCEDURE AfterBuild
LPARAMETERS nError
WAIT WINDOW "Finished building project"
ENDPROC

ENDDEFINE
```

Other things you might do with project hooks include creating a sort of "poor man's Source Safe" by logging every project action, enforcing project security by only allowing certain developers to modify some pieces, or making global changes to a project to reflect some change in circumstance.

To have a project hook created with each project, you use the Projects page of the Tools | Options dialog. Check the Project Class checkbox and specify the project hook class. To make the setting persistent beyond the current session, choose Set as Default. Once you've done this, each time you open a project, a projecthook of the specified class is created and linked to the project through the ProjectHook property.

Copyright 1998, Tamar E. Granor, Ph.D.

You can also link a project hook to an individual project using the Project Info dialog, but it doesn't get instantiated and attached until you close and reopen the project. In addition, you can manually link a project hook to a project by creating the project hook, then setting the project's ProjectHook property. In fact, it's possible for several projects to share the same project hook object simultaneously.

OLE Drag-and-Drop

In many situations, drag-and-drop makes interfaces easier-to-use (at least for users comfortable with a mouse). Drag-and-drop was first offered to FoxPro developers in VFP 3. However, the drag and drop properties and methods available in VFP 3 and VFP 5 apply only to native objects. If you have ActiveX controls in your forms, you can't drag between them and native controls. VFP's native drag-and-drop also doesn't let you drag things between VFP apps and other applications, something that's allowed by more and more applications. (Ever dragged a group of files from Explorer into, say, WinZip? Very cool!)

VFP 6 addresses this whole area with a new group of properties, events and methods that deal with OLE drag-and-drop. The PEMs for OLE drag-and-drop are quite similar to those used for conventional drag-and-drop. However, OLE drag-and-drop offers more control and seems smarter. As with native drag-and-drop, there are two objects involved, the *drag source* and the *drop target*. Each has a number of PEMs affecting OLE drag-and-drop.

On the drag source side, OLEDragMode indicates whether OLE drag-and-drop is automatic or manual. If manual dragging is chosen, the OLEDrag method can be called to start a drag operation. Whichever way you start dragging, OLEStartDrag fires, so you can make any changes you need immediately. Once you start dragging, the drag source's OLEGiveFeedback event fires regularly to tell it what's going on and to allow it to change the cursor shown. OLEDragPicture controls the picture to use while you drag.

From the drop target side, OLEDragOver fires when an object is dragged over the object, while OLEDragDrop fires when an object is dropped. OLEDropMode indicates whether or not an object is a drop target. By default, OLE objects do accept OLE drops, while native VFP objects do not. But changing the status of an object is as easy as modifying OLEDropMode.

Because the data you can drag with OLE drag-and-drop can be quite complex, there are additional properties and even an object containing the data that let you decide how a particular object should respond to a drop.

The list of events that fire may seem somewhat ambiguous. The drag source and drop target have events that fire based on the same action. For example, when you're dragging, OLEDragOver fires, followed by OLEGiveFeedback. This is intentional because it's possible for either the drag source or the drop target to be something other than a VFP object. For example, you can drag from a VFP form into, say, Word, or vice versa. The duplication of events means that the VFP object gets a chance to respond, no matter where the drag comes from or where the data is being dragged or dropped.

Copyright 1998, Tamar E. Granor, Ph.D.

The VFP interface also includes OLE drag-and-drop functionality. So you can do things like drag files from Explorer into VFP. For example, dragging a .DBC into the Command Window opens the database and issues MODIFY DATABASE.

FoxTools functions

Since FoxPro 2.x, each version of FoxPro has included the FoxTools library. This library includes an assortment of functions that call on the Windows API for various tasks.

The set that lets you parse and manipulate filenames and paths has been added to the language in VFP 6 and no longer requires FoxTools. This means you don't have to distribute FoxTools with your applications, nor SET LIBRARY in your code to be able to manipulate file names with ease.

The first subset of these functions pulls filenames and paths apart.

JustDrive(cFileOrPath) returns the drive portion (letter plus colon) of a filename or path.

JustPath(cFile) returns the path portion of a filename.

JustFname(cFile) returns the filename without the path, but including the extension.

JustStem(cFile) goes one step further. It returns only the "stem" portion of the filename. That is, it returns the filename without the path and without the extension.

JustExt(cFile) returns only the extension, very handy if you're trying to figure out what kind of file you have.

Here are examples of each of these functions.

Example	Result
`JustDrive("g:\vfp6\gallery")`	`"g:"`
`JustPath("g:\vfp6\browser.app")`	`"g:\vfp6"`
`JustFname("g:\vfp6\browser.app")`	`"browser.app"`
`JustStem("g:\vfp6\browser.app")`	`"browser"`
`JustExt("g:\vfp6\browser.app")`	`"app"`

Several of the functions let you make changes to a filename or path.

AddBS(cPath) adds a final backslash to a path, if it doesn't already have one.

ForcePath(cFileName, cPath) returns the specified filename with its path changed to the specified path.

ForceExt(cFileName, cExtension) and DefaultExt(cFileName, cExtension) let you manipulate file extensions. ForceExt() changes the file's extension to the one specified, while DefaultExt adds the extension only if the file doesn't already have an extension.

Here are examples of this group of functions.

Copyright 1998, Tamar E. Granor, Ph.D.

Example	Result
`AddBS("g:\vfp6")`	`"g:\vfp6\"`
`AddBS("g:\vfp6\")`	`"g:\vfp6\"`
`ForcePath("g:\vfp6\browser.app", ; "c:\program files")`	`"c:\program files\browser.app"`
`ForceExt("g:\vfp6\config.fpw", "old")`	`"g:\vfp6\config.old"`
`DefaultExt("g:\vfp6\config.fpw","old")`	`"g:\vfp6\config.fpw"`

DefaultExt() is most useful when you want to make sure you have an extension to work with, such as when the user has entered a filename. For instance, say cOutFile contains the name of an output file specified by the user. To make sure it has an extension, with .TXT as the default, you can use:

```
DefaultExt(cOutFile, "TXT")
```

Finally, there are a couple of functions that tell you about the environment or files. DriveType(cPath) returns an integer indicating the kind of drive. For example:

```
DriveType("c:")
```

returns 3, indicating a hard disk.

The conference materials include a function ADrives that fills an array parameter with the list of drives on the machine and their types. It returns the number of drives found.

The last function in this group doesn't exactly come from FoxTools. But it is a replacement for a FoxTools function. The new function is AGetFileVersion(aOutput, cFileName) and it returns version information about the file, superseding FoxTools' GetFileVersion() function (added in VFP 5). It's intended for files like .EXEs and .DLLs that have version information stored in them. The array that's created contains 15 elements, including such information as the File Description and File Version, Product Name and Product Version, and much more. To use it, you pass an array and the file name, like this:

```
AGetFileVersion(aVersionInfo, "e:\msoffice\office\outlook.exe")
```

The function returns the number of elements in the array (15 in VFP 6). Note that it works on all kinds of .EXEs and .DLLS, not just .EXEs created by VFP.

OOP Enhancements

VFP 6 has a number of enhancements to the object-oriented part of the FoxPro language. The most significant change in this area is the addition of access and assign methods. However, there are also lots more changes to the OOP sub-language, including new base classes, new properties, new values for old properties and new methods. Several new functions make working with the OOP parts of the language easier as well.

Access and Assign Methods

Any property, whether built-in or user-defined, can have both an access and an assign method. The assign method for a property fires when a new value is assigned to the

Copyright 1998, Tamar E. Granor, Ph.D.

property. In fact, the new value can be the same value. What's trapped is the assignment of any value to the specified property, whether through code or an action.

The access method for a property fires whenever the value of that property is used. Any reference to the property, other than an assignment, fires the access method.

The addition of these methods has a number of consequences. First, since these methods fire based on things happening, it provides us with many more programmable events. Until now, we've been restricted to responding to the events that the designers of VFP have given us. For the first time, we can add our own events.

Access and assign methods also let us answer one of the "pure OOP" criticisms that's been leveled at VFP over the years. In the pure OOP languages, properties can't be accessed directly – everything goes through methods. Access and assign methods let us simulate that kind of environment. Since the access method associated with a property fires whenever the property is used, we can take action just as if a method had to be called to return the property value. Similarly, the assign method lets us control changes to the value of the property.

Finally and most importantly, access and assign methods encapsulate individual properties. Using these methods, a property can control its own destiny. The property can react to changes and uses of itself. You don't have to rely on the control that displays a property or remember to make a method call when a property changes. As a current TV ad says, "This changes everything."

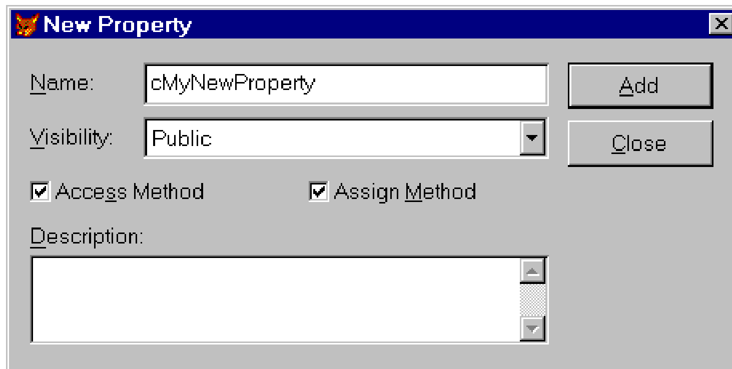

Figure 1. Adding access and assign methods to new properties is as easy as checking a couple of checkboxes.

When you add a new property, you can give it an access and/or assign method by checking the appropriate checkboxes in the New Property dialog (figure 1). For existing properties (either inherited properties or those previously added), the Edit Property/Method dialog (figure 2) lets you indicate whether a property should have access and assign methods. (In forms, the Edit Property/Method dialog doesn't show the properties inherited from the Form class. However, you can still add access and assign methods by using the New Method dialog. Just give the method the appropriate name: property_access or property_assign, where you substitute the name of the relevant

Copyright 1998, Tamar E. Granor, Ph.D.

property.) There is one limitation, here: read-only properties can have access methods, but not assign methods.

Once you add an access or assign method for a property, it's available in all the usual ways. The method names are the property name followed by an underscore, then either "access" or "assign". So, for example, the access method for the Caption property is caption_access. Access and assign methods are treated like custom methods and show at the end of the Methods page in the property sheet.

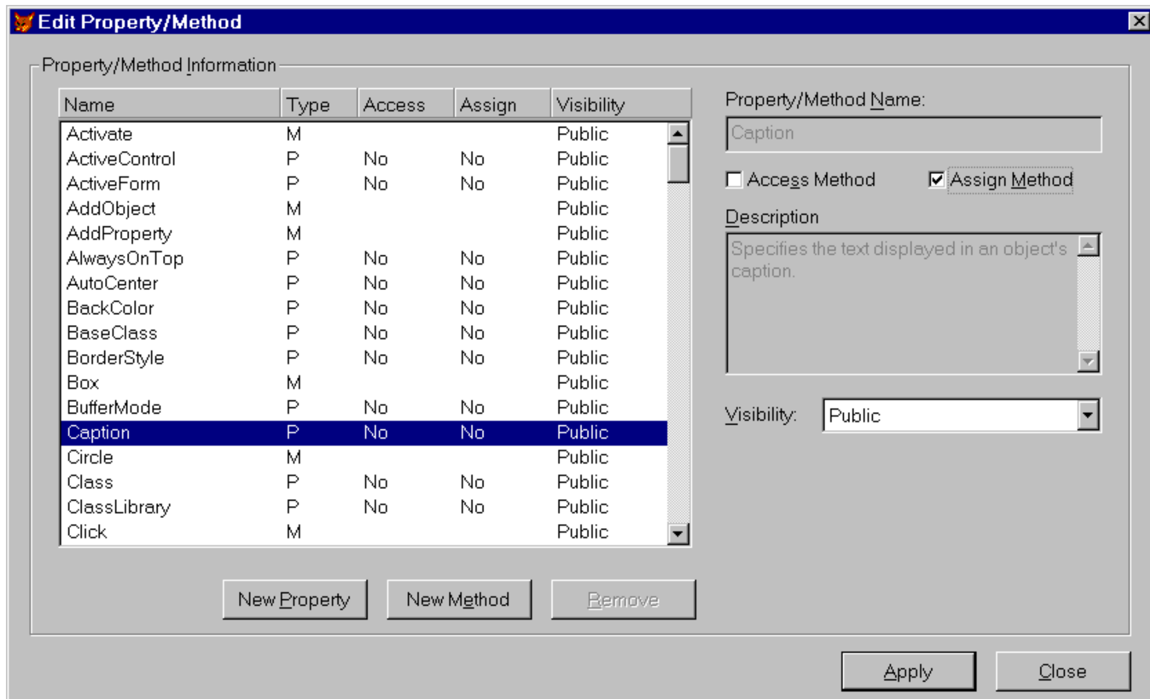

Figure 2. For existing properties, use the Edit Property/Method dialog to control access and assign methods.

In the Code window, however, access and assign methods are always at the top of the list because they have code added on creation. Here's the initial code for an access method for Caption:

```
*To do: Modify this routine for the Access method
RETURN THIS.Caption
```

For an assign method for Top, the initial code looks like:

```
LPARAMETERS vNewVal
*To do: Modify this routine for the Assign method
THIS.Top = m.vNewVal
```

These examples show a key point about the methods. Once you add an access method to a property, when the property is used, the actual value used is whatever the access method returns. It's up to you whether that's the value stored in the property or something else. Similarly, the assign method gives you control over what is really stored to the property. It can be the value received as a parameter, a different value or nothing at all.

Copyright 1998, Tamar E. Granor, Ph.D.

Why would you use access and assign methods? How about logging changes to a property? Every time the property changes, you could store a record in a table. Another possibility is controlling access to a particular property more finely than just making it protected or hidden. You can check who's asking for the value and only provide it to those who have a right to it. Another use for the assign method is to limit the values a property can take. Perhaps a certain property should only be able to have a value between 1 and 10 – use the assign method to control it.

Here's the code for a phone number class (found in NewLang.VCX) that stores the phone type internally as a number, but displays it to the world as a string.

```
DEFINE CLASS phonenumber AS custom

   *-- What type of phone is this: voice, fax, modem?
   phonetype = 1
   *-- The actual phone number
   phonenumber = ""
   Name = "phonenumber"

   PROCEDURE phonetype_access
      * Convert the internally stored number to a string.
      LOCAL cPhoneType
      DO CASE
      CASE THIS.PhoneType = 1
         cPhoneType = "Voice"
      CASE THIS.PhoneType = 2
         cPhoneType = "Fax"
      CASE THIS.PhoneType = 3
         cPhoneType = "Modem"
      ENDCASE

      RETURN m.cPhoneType
   ENDPROC

   PROCEDURE phonetype_assign
      * While phonetype displays as a string, internally it's stored
      * as a number, allowing more flexibility. This also allows us
      * to normalize the case used and not worry about upper, lower, etc.
      LPARAMETERS vNewVal
      DO CASE
      CASE UPPER(m.vNewVal) = "VOICE"
         THIS.PhoneType = 1
      CASE UPPER(m.vNewVal) = "FAX"
         THIS.PhoneType = 2
      CASE UPPER(m.vNewVal) = "MODEM"
         THIS.PhoneType = 3
      OTHERWISE
         * Reject the value by not assigning it
      ENDCASE

      RETURN
   ENDPROC

ENDDEFINE
```

Several of the demo forms for this session use Access methods to update things on the form as soon as a change occurs. For example, look at the filename_assign method in FileVer.SCX.

Copyright 1998, Tamar E. Granor, Ph.D.

Expect to see a lot of articles about using access and assign methods. The interest in these methods among the expert/writer community is huge.

Other OOP Changes

The VFP developers have added all kinds of little things (and a couple of bigger ones) that make using VFP's OOP portions easier and more productive.

Creating Objects

One of the little annoyances of writing VFP code has been having to SET CLASSLIB or SET PROCEDURE before issuing CreateObject(). Forgetting that line is easy. VFP 6 makes it unnecessary with the NewObject() function. This function lets you pass both the class name and the class library to create a new object. It finds the class in the specified class library (which can be either a .VCX or a .PRG) and instantiates it. The class library is not added to the search list.

The function expects two parameters, the class name and the library name. If the library is a .VCX, you can omit the extension. For a .PRG, it's required.

For example, suppose you have a class called frmDialog in a class library called forms.vcx. You can create a new instance of frmDialog like this:

```
oDialog = NewObject("frmDialog","forms")
```

If the class library has already been set, you can omit it from the NewObject() call, so you can use NewObject() for all your instantiation, if you choose.

In addition, the various container classes now have a NewObject method that lets you add objects to them without issuing SET CLASSLIB or SET PROCEDURE. In many ways, this may be more important than the function, since setting libraries inside a class can have far-reaching consequences.

Finding Out About Classes

Two new functions make it easier to build development tools involving class libraries. A third function has been enhanced to provide more information.

The first new function, AVCXClasses() fills an array with information about the classes in a class library. You pass an array and the name of the class library, and the array is filled with information on each class in the library. The information returned includes the name, base class, parent class, parent class' library, icons, description and more.

For example, if you have a library called MyClasses.VCX, you can fill an array with its classes by calling:

```
nClassCount = AVCXClasses(aMyClasses,"MyClasses")
```

The function returns the number of classes in the library. The form VCXClas.SCX in the conference materials demonstrates AVCXClasses.

Copyright 1998, Tamar E. Granor, Ph.D.

The new function AGetClass() displays the Open Class dialog and returns the class library and name of the chosen class in an array. As with GetFile(), you can customize the appearance of the dialog, specifying the class to highlight initially, the dialog's caption, the caption next to the text box, and the OK button's caption. Figure 3 shows the dialog produced by this call:

```
AGetClass(aclass, "_base", "_form", "Choose a Class", "Class Lib", "Choose")
```

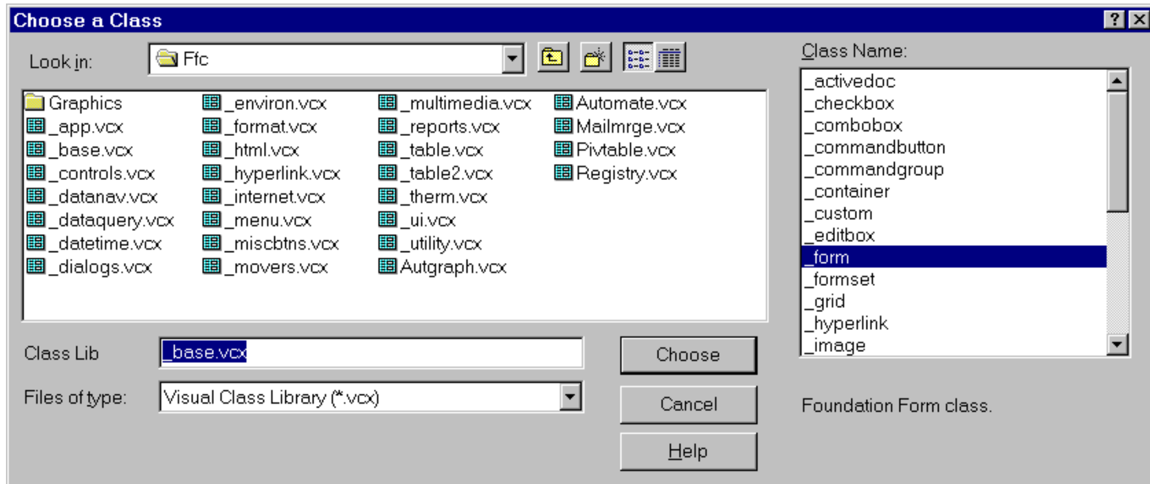

Figure 3 - AGetClass() lets you choose a class from a class library. Various captions can be customized, as can the initial highlight.

The program EditCls.PRG in the conference materials shows how you can use AGetClass() to choose a class to edit, then open it for editing.

In addition to these new functions, PEMStatus() has a new option. Passing 6 for the last parameter lets you determine whether a property, event, or method originates in this class or was inherited from a superclass.

New Base Classes

Three new base classes have been added. One is ProjectHooks, discussed above. The others are for Internet applications. ActiveDoc is for building active documents, while Hyperlink is for putting links into applications.

The ActiveDoc class lets you build VFP applications that can be run inside Active Document containers (of which Internet Explorer is the best known, of course). The ActiveDoc object is the main "program" of such an application and handles all interaction with the container. ActiveDocs have no interface of their own - they call on other VFP objects to present a user interface and perform the actual application actions.

Running an ActiveDoc application requires the VFP runtime libraries on the local machine, so this is not a way to deploy programs on the Internet. It has some uses for intranets, though, especially as an easy way to use an existing application.

Copyright 1998, Tamar E. Granor, Ph.D.

The Hyperlink object has several methods that let you jump from one document or URL to another. You can use hyperlinks both in applications running in a container like IE and to launch a browser from a VFP application.

New Properties and Methods

Several new properties and methods have been added to forms to make them run better inside a browser. A lot of them deal with scrolling. First, ScrollBars now applies to forms as well as to grids. As with grids, you can have horizontal or vertical scroll bars or both. In order to have scrollbars available at run-time, you must set ScrollBars to a non-zero value in the Property Sheet. (Zero indicates no scrollbars.) This is because providing scrollbars to a form requires a great deal of work from the VFP engine – it only does that work if the form indicates that it's going to use scrollbars.

Many of the other new properties support scrolling and give you information about the current status of your form. VScrollSmallChange and HScrollSmallChange let you decide how much the form scrolls when the user clicks on an arrow. ContinuousScroll determines whether the form scrolls while the "thumb" is still moving or waits until the mouse is released to scroll. A set of properties (ViewportLeft, ViewportTop, ViewportHeight, ViewportWidth) let you know what part of the form is visible. Finally, the Scrolled event fires when scroll bars are used. These new properties do not apply to other controls that support scrollbars, however. The conference materials contain a formset (ScrollBr.SCX) that demonstrates these properties.

Forms have a new property, TitleBar, that lets you determine whether the form has a title bar or not. In VFP 3 and 5, you have to set a whole bunch of properties to get a form without a title bar. Now it's as simple as This.TitleBar = .F.

Like Scrollbars, TitleBar was added for use in ActiveDocs, but has a place in conventional applications as well. One obvious use is for splash screens (although there's a problem with form classes with no titlebar and ShowWindow set to 2 – As Top-Level Window). Figure 4 shows a sample splash screen – you'll find it in the conference materials as Splash.SCX.

Copyright 1998, Tamar E. Granor, Ph.D.

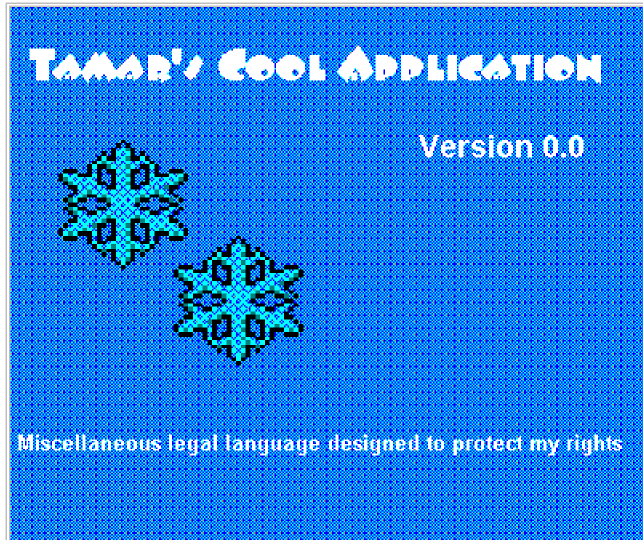

Figure 4 – The new TitleBar property makes it much easier to create splash screens, as well as being useful for ActiveDocs.

By default, when a combo box drops open, it shows seven items. Another new property, DisplayCount, lets you determine how many items show. While seven is a reasonable default (based on research into what people can remember and so forth), in some situations, it's not a good choice. There are two cases where changing DisplayCount makes sense. First, when the whole list contains more than seven items, but not many more. It's silly for users to have to scroll when the whole list has eight or nine items. The other case is for very large lists, where seven items simply isn't enough. (Of course, usually, there are better controls to use in those situations.) DispCnt.SCX in the conference materials demonstrates DisplayCount.

Grids have a new method that makes it much easier to drag-and-drop into grid cells. The method, called GridHitTest, lets you pass a point and find out whether the point is in the grid and, if so, where it is in the grid. Using the information returned, it takes only a few lines of code in the DragDrop or OLEDragDrop method to put the dropped information into the right cell. In DragDrop, you can use code like this:

```
LPARAMETERS oSource, nXCoord, nYCoord
cData = oSource.Value

LOCAL nComp, nRow, nCol
IF THIS.GridHitTest(nXCoord,nYCoord,@nComp,@nRow,@nCol)
   IF nComp = 3
      THIS.SetFocus()
      THIS.ActivateCell(nRow,nCol)
      THIS.Columns[THIS.ActiveColumn].Controls[2].Value = cData
   ENDIF
ENDIF
```

This code assumes that the current control in the column is Controls[2]. To avoid that assumption requires some more code to check CurrentControl and so forth. The form TestGrHt.SCX in the conference materials demonstrates the use of GridHitTest.

Copyright 1998, Tamar E. Granor, Ph.D.

One of the buzzwords of object orientation is polymorphism, meaning that different objects may have PEMs with the same name, but slightly different effects. That way, you can learn, for example, that whenever an error occurs, the Error method of the object is called. However, until now, some of the VFP base classes didn't include some of the basic properties and events.

That problem is fixed in VFP 6. A number of properties, events and methods have been added to objects that didn't have them before to make things more uniform. For example, the Tag property has been added to those classes that didn't already have it. Similarly, all the base classes now have Init, Destroy and Error events.

Finally, the new AddProperty method lets you add properties to objects at run-time. For example:

```
IF NOT PEMStatus(THIS,"MyNewProperty",5)
   THIS.AddProperty("MyNewProperty")
ENDIF
```

AddProperty seems particularly useful for development tools like Builders and Wizards.

Developer Enhancements

Several of the changes to the language make it easier for developers to get things done.

Since VFP was introduced, creating a new form based on the class of your choice has been tricky. We've always been able to specify a single template form class (through Tools-Options), but often we need more than that. In VFP 6, the CREATE FORM command has an optional AS clause that specifies the class on which the new form is based. So you can issue commands like this:

```
CREATE FORM Main AS frmBaseForm FROM BaseLib
```

Include files (also known as header files) are another area that's been too hard to get right. While you could specify an Include file for a form or class and have it apply to all methods, you had to remember to do it each time. Even doing it at the class level wasn't sufficient; the header file isn't inherited by subclasses or instances.

In VFP 6, there's a new system variable, _INCLUDE, that lets you specify a default header file. The specified file is automatically filled in as the include file for all new classes and forms. You can, of course, override this specification for an individual form or class.

But Wait, There's More!

There are a number of other small changes in VFP's OOP language. Some classes have new PEMs. Some existing properties have new potential values. Be sure to spend some time checking what appears in the property sheet for each object, and the choices on the various dropdowns.

Copyright 1998, Tamar E. Granor, Ph.D.

New Commands and Functions

It's not only the OOP sub-language that's been enhanced in VFP 6. This version also includes a number of new commands and functions and new options in existing language elements.

Date Handling

With the year 2000 on the horizon, the VFP developers have made it harder to get ambiguous code into your apps. A new command, SET STRICTDATE, lets you decide how date constants are handled. When strict dates are in use, any date constants that are ambiguous as to the year trigger an error. Strict date checking means you can't get away with code like:

```
IF DATE()<{01/01/99}
   * do something
ENDIF
```

Instead, you'll have to be explicit about the year, which will save you from code that doesn't work after the year 2000 (at which point that date would be interpreted as January 1, 2099). Be forewarned: by default, strict date checking is on, though not in its strictest form.

SET STRICTDATE applies independently at compile-time and runtime. The value at compile-time determines which errors are detected. Appropriate error messages are built into the compiled code. Then, at runtime, the STRICTDATE value is checked and only the errors that apply with that value are shown. (Of course, only errors up to the compile-time setting of STRICTDATE can show up at runtime.)

This approach lets you work on fixing your code while your users can still run the application. SET STRICTDATE TO 2 at compile-time to find all the ambiguities, but set it to 0 at runtime, so it still runs as expected.

In addition, it's now easier to specify non-ambiguous date constants. VFP 5 added one strict date notation, like this:

```
{^ 1998-09-28 09:02:37 AM}
```

Even though some of the parts can be omitted, writing dates that way can get a bit tedious. In addition, it doesn’t handle cases where you want to take a group of variables holding the parts of a date and turn them into a date variable. Say you have character variables holding the year, month and day you're interested in. In older versions, you create a date like this:

```
dDate = CTOD(cMonth + "/" + cDay + "/" + cYear)
```

But code like that depends on the current SET DATE settings and may evaluate differently at different times. To enable us to create dates and datetimes without ambiguity, the DATE() and DATETIME() functions have been enhanced. They now accept an appropriate number of numeric parameters and convert them into a date or datetime value. So you can now do something like:

```
dDate = DATE( VAL(cYear), VAL(cMonth), VAL(cDay) )
```

Copyright 1998, Tamar E. Granor, Ph.D.

The form StrictDt.SCX in the conference materials demonstrates both SET STRICTDATE and the new functionality of DATE().

Type Checking

Checking the type of variables can be faster in VFP 6 than in older versions. The new VARTYPE() function takes an value and returns a single letter indicating its data type. It's similar to TYPE(), but lacks the additional level of indirection of that function. That is, you traditionally check data type like this:

```
cType = TYPE("SomeVar")
```

but, with VARTYPE(), you omit the quotes:

```
cType = VARTYPE(SomeVar)
```

My tests (with a beta version of VFP 6) show VARTYPE() as three to five times faster than TYPE().

If the variable you pass VARTYPE() is null, it returns "X" rather than the variable's type. This makes it a good choice for checking whether an object variable currently points to an object or not. You can replace code like this:

```
IF TYPE("oObject")="O" AND NOT ISNULL(oObject)
```

with

```
IF VARTYPE(oObject)<>"X"
```

Be aware that you can't replace all uses of TYPE() with VARTYPE(). If you pass VARTYPE() a character string containing an expression, it returns "C", not the type to which the expression evaluates. For example:

```
?TYPE("x=1")
```

returns "L", while:

```
?VARTYPE("x=1")
```

returns "C". So, use VARTYPE() for checking the existence of variables or ensuring that objects are non-null, but not for finding the result type of expressions.

Searching for a Key

FoxPro has never had a way to check whether a particular index value exists without moving the record pointer. You either had to move the pointer and then put it back or open the table in another work area and move the pointer there. Both of those approaches are annoying when all you want to do is find out if a particular value is already in the index.

The new INDEXSEEK() function checks for an index value without moving the record pointer – it returns .T. or .F. It's great for checking whether a user entry already exists without committing the record the user is working on. For example, using the TasTrade Customer table, we can check whether there's already a customer with the id contained in cCustId like this:

```
lHasIt = INDEXSEEK(cCustId, .F., "Customer", "Customer_I")
```

Copyright 1998, Tamar E. Granor, Ph.D.

Only the first parameter, the value to search for, is required. The last two parameters indicate the alias and tag to use for the search.

The second parameter is interesting - it indicates whether or not to move the record pointer. Since the point of this function is to search without moving the pointer, it defaults to .F. However, its existence means that you can, in fact, use INDEXSEEK() all the time, and stop using SEEK or SEEK().

With so many choices for checking an index value, a little speed testing is in order. My tests (using a beta version of VFP 6) did batches of 10,000 searches in a table of more than 17,000 records, with a hit rate of less than 50% - that is, more than half my searches resulted in no record found. The time difference between the alternatives was small enough to be significant only inside tight loops. With all those disclaimers, it appears that SEEK with FOUND() is fastest, SEEK() is a little slower, INDEXSEEK() without moving the record pointer is next and INDEXSEEK() moving the record pointer is last.

The form SeekMatc.SCX in the conference materials lets you experiment with the different search commands. Run MakeSeek.PRG first to create the sample data set.

Text Handling

Several new functions deal with text, making it easier to work with both text files and blocks of text.

ALines() takes a character string (whether stored in a character or memo field or a variable) and breaks it up into lines. The results are put into an array, with one line per element. ALines() does not interact with SET MEMOWIDTH – lines are determined only by CHR(13) and CHR(10). This makes the function useful for breaking up data in paragraphs. Each paragraph goes into a single array element.

FileToStr() and StrToFile() move information between files and string variables. These functions offer a one-line way to do what used to require loops and low-level file functions. For example, to read a text file called readme.txt into a variable cReadMe, use:

```
cReadMe = FileToStr("readme.txt")
```

The inverse StrToFile() function lets you decide whether to overwrite an existing file or add to it. To add the contents of cMoreInfo to the readme.txt file, use:

```
StrToFile( cMoreInfo, "readme.txt", .T. )
```

It's likely that all three of these functions were added to support some of the tools written in VFP (like the Class Browser and Coverage Profiler) that come with the product and do significant text handling. The form TextHand.SCX in the conference materials demonstrates some of these functions.

Reporting

Several changes affect report previews, making them easier to manage. The best of the changes doesn't require you to do anything in your code. When a report preview is run

Copyright 1998, Tamar E. Granor, Ph.D.

from a top-level form, it appears in the top-level form. In VFP 5, the report preview window was always in the main VFP window.

In addition, the PREVIEW clause now supports not only the WINDOW clause to define a template for the preview window, but the IN WINDOW clause that lets you confine the preview window to another window. For example:

```
DEFINE WINDOW ContainerWindow FROM 5,5 TO 50,100
DEFINE WINDOW DefinitionWindow FROM 0,0 TO 40,80 TITLE "My Test Report"
ACTIVATE WINDOW ContainerWindow
REPORT FORM TestRep PREVIEW WINDOW DefinitionWindow IN WINDOW ContainerWindow
RELEASE WINDOW DefinitionWindow
RELEASE WINDOW ContainerWindow
```

This code creates a larger container window to hold the preview window, then defines a window from which the preview picks up some (but not all) characteristics.

Committing Local Views

In VFP 3 and VFP 5, if a local view contains more than 23 fields and specifies that conflicts are to be checked using the key plus updatable fields (and, in some cases, key plus modified fields), an error occurs when TableUpdate() is issued. The problem is that the SQL UPDATE command created is long and has many separate items that need to be evaluated. VFP only allocates a certain amount of stack space to hold those items and the most it can handle in the case is the number generated by 23 fields.

This has been a major problem for developers since it means that either they must use several views when one should do (and handle the resulting complexity) or they must choose less rigorous conflict checking. Neither is a good choice.

VFP 6 solves the problem. The new function SYS(3055) lets you determine the size of the stack used. The default value is also the minimum value and accommodates 40 fields. However, you can specify a value that allows 255 fields, the maximum permitted in a view. To do so, you simply call the function, like this:

SYS(3055, 8*255) && you could just use 2040, but this is clearer

Increasing the stack space doesn't appear to have any serious performance or memory consequences.

Odds and Ends

The new AMouseObj() function gives you information about the mouse position. It creates a four row array containing object references to both the object over which the mouse is positioned and that object's container, plus the x and y coordinates of the mouse position, relative to that container. AMouseObj() works at design-time as well as at run-time, making it useful for developer tools like builders.

ALTER TABLE now includes extensions that let you create filtered indexes for primary and candidate keys. This is especially welcome for those who recycle deleted records in their applications. To create a filtered primary key, use a command like:

Copyright 1998, Tamar E. Granor, Ph.D.

```
ALTER TABLE MyTable ADD PRIMARY KEY Pkey FOR NOT DELETED()
```

If you use filtered keys, remember that they are not used for Rushmore optimization, so you need an additional regular key for each expression that may be used in queries or other optimizable commands.

Several functions that provide information about the VFP or Windows environment have been enhanced to offer additional information. HOME() now accepts an optional parameter, allowing it to return not just the directory from which VFP was started (HOME() or HOME(0)), but the VFP root directory (HOME(1)), and the locations of various files or directory trees like the VFP samples. For example, to open the TasTrade database, you can use:

```
OPEN DATABASE HOME(2)+"\TasTrade\Data\TasTrade"
```

The new system variable _SAMPLES contains the same value as HOME(2), so you can also write that line as:

```
OPEN DATABASE _SAMPLES+"\TasTrade\Data\TasTrade"
```

You can find out whether you're on a machine that supports double-byte character sets with a new parameter to OS():

```
lHasDoubleByte = OS(2)
```

OS(1) is the same as OS() and tells you which version of Windows you're running.

ANetResources() fills an array with information about either network shares or printers on the network.

As with the OOP language, these items aren't all there is. Some other commands and functions have been enhanced, as well. Be sure to read the Help file, especially the section on New and Enhanced Language Elements.

Summary

As in every version to date, VFP 6 introduces a variety of enhancements in the programming language. The changes in this version range from simple to complex and offer a wide range of new capabilities.

Copyright 1998, Tamar E. Granor, Ph.D.