Base Classes Revisited

Doug Hennig

Most VFP developers know you should never use the VFP base classes, but instead create your own set of "base" classes. It's time to blow the dust off the set of base classes Doug has discussed for the past six years and see how they can be revamped in VFP 8.

Like many VFP developers, I created my own set of subclasses of the VFP base classes years ago (I first discussed these classes in the February 1997 issue of FoxTalk). Although my base classes, each called SF*baseclassname* and defined in SFCtrls.VCX, have served me well for a long time, I recently decided it was time to revisit them. They were all created in the VFP 3 days, and VFP has changed a lot since then. As I went through the SFCtrls classes, I found that a lot of code and custom properties were no longer required to create certain behavior because that behavior was now built into VFP. This article discusses the changes I made to the classes in SFCtrls.VCX, in addition to support for resizing that I discussed in the June 2003 issue ("Ahoy! Anchoring Made Easy").

Page and PageFrame

In earlier versions of VFP, increasing the PageCount property of a PageFrame added Page objects to the PageFrame. As base class objects, you had to manually code any behavior for these pages. One common behavior to add to pages is refreshing on activation. For performance reasons, when you call the Refresh method of a form, VFP only refreshes controls on the active page of a page frame. As a result, when the user activates another page, the controls on that page show their former values rather than current ones. To fix this, you could add This.Refresh() to the Activate method of each page, but that's tedious and doesn't work in cases where pages are added dynamically at runtime.

To prevent that, SFPageFrame had the following code in Init. This code added an SFPageActivate object to every page.

```
local llAdd, ;
  loPage
llAdd = '\SFCTRLS.VCX' $ upper(set('CLASSLIB'))
for each loPage in This.Pages
  if llAdd
    loPage.AddObject('oActivate', 'SFPageActivate')
  else
    loPage.NewObject('oActivate', 'SFPageActivate', ;
      'SFCtrls.vcx')
  endif llAdd
next loPage
```

SFPageActivate is a simple class with the following code in UIEnable:

```
lparameters tlEnable
with Thisform
  if tlEnable
    .LockScreen = .T.
    This.Parent.Refresh()
    if pemstatus(Thisform, 'SetFocusToFirstObject', 5)
      .SetFocusToFirstObject(This.Parent)
    endif pemstatus(Thisform, 'SetFocusToFirstObject', 5)
    .LockScreen = .F.
  endif tlEnable
endwith
```

This code does two things when a page is activated: it refreshes the controls on the page and sets focus to the first one. (We won't look at the SetFocusToFirstObject method in SFForm; feel free to examine that method yourself.)

VFP 8 makes this much simpler because we can now subclass Page visually and tell PageFrame to use our subclass by setting its MemberClass and MemberClassLibrary properties to the appropriate class and

library. My VFP 8 version of SFPageFrame no longer adds SFPageActivate objects to pages in Init and has MemberClass and MemberClassLibrary set to SFPage, a new class based on Page in SFCtrls.VCX. I also removed the SFPageActivate class from SFCtrls.VCX since it isn't needed anymore. SFPage has following code in Activate:

```
local llLockScreen
with This

* Lock the screen for snappier refreshes.

  llLockScreen = Thisform.LockScreen
  if not llLockScreen
    Thisform.LockScreen = .T.
  endif not llLockScreen

* If we're supposed to instantiate a member object and
* haven't yet done so, do that now.

  if not empty(.cMemberClass) and ;
    type('.oMember.Name') <> 'C'
    if '\' + upper(.cMemberLibrary) $ set('CLASSLIB')
      .AddObject('oMember', .cMemberClass)
    else
      .NewObject('oMember', .cMemberClass, ;
        .cMemberLibrary)
    endif '\' ...
    with .oMember
      .Top     = 10
      .Left    = 10
      .Visible = .T.
      .ZOrder(1)
    endwith
  endif not empty(.cMemberClass) ...

* Set focus to the first object.

  if pemstatus(Thisform, ;
    .SetFocusToFirstObject', 5)
    Thisform.SetFocusToFirstObject(This)
  endif pemstatus(Thisform ...

* Refresh all controls and restore the LockScreen
* setting.

  .Refresh()
  if not llLockScreen
    Thisform.LockScreen = .F.
  endif not llLockScreen
endwith
```

In addition to the behavior of the former SFPageActivate, this code supports one other feature: the ability to delay instantiation of the controls on the page until the page is activated. Why would you want to do that? One reason is performance. If you have a form containing a page frame with a lot of pages and a lot of controls on each page, it'll take a long time to instantiate. However, if only the controls on the first page are created when the form is instantiated, the form loads very quickly. When the user selects another page for the first time, the controls on that page will be instantiated. The user will take a slight performance hit when this happens, but it's minimal and doesn't occur the second and subsequent times. Another reason is flexibility. I've created wizard-based forms where which controls appear on page 2 depend on choices the user makes in page 1. Delaying the instantiation of the page 2 controls until page 2 is activated allows me to decide what controls to instantiate dynamically.

To support this feature, SFPage has two new properties: cMemberClass and cMemberLibrary, which contain the class and library for a container of controls. To use delayed instantiation, create a subclass of Container, add the controls that should appear in the page to the container, and set the cMemberClass and cMemberLibrary properties of a page in a page frame to the container's class and library names.

SFPage has other behavior as well, including support for shortcut menus and automatically disabling all the controls on the page when the page itself is disabled. The reason for the latter feature is that normally when you disable a page, the controls on the page don't appear to be disabled, even though they can't receive focus, so it's confusing to the user.

To see an example of these classes, run DemoPageClass.SCX. This form has two page frames: one PageFrame with regular Page objects and the other SFPageFrame with SFPage objects. Select page 2 of both page frames, note the customer information shown, then select page 1 of both and click on the Next button a few times. Select page 2 of both again and notice that the PageFrame doesn't show the correct information (that is, until you click in a textbox) but SFPageFrame does. Also, check and uncheck the Enabled check box and notice that while the controls in the PageFrame don't appear disabled (although you can't set focus to them), those in SFPageFrame do.

OptionGroup

Similar to PageFrame, increasing the ButtonCount property of an OptionGroup adds more base class OptionButton objects to the control. Because the default property values for OptionButton aren't correct, in my opinion, you always have to manually set the AutoSize property to .T. and the BackStyle property to 0-Transparent for every button.

Because VFP 8 visually supports OptionButton subclasses and OptionGroup has MemberClass and MemberLibrary properties, I created an SFOptionButton class and set MemberClass and MemberClassLibrary in SFOptionGroup to point to the new class. To see these buttons in effect, run DemoOptionButton.SCX. With both option groups, I simply changed the caption of the two buttons. Notice that the top one, which uses OptionGroup, isn't sized properly and the buttons don't have the proper background color. The bottom one, which is an instance of SFOptionGroup, looks correct.

Grid

Until VFP 8, one feature that distinguished VFP grids from grid controls in other languages or ActiveX grids was that there wasn't an easy way to highlight an entire row in VFP. This is a useful feature for a couple of reasons: it makes it easier to tell which row you're on when a grid doesn't have focus, and it makes a grid act like a list box but with much better performance when there are a lot of rows.

To allow highlighting of an entire row, I added a bunch of properties and methods to SFGrid: lAutoSetup to determine if SetupColumns should be called from Init, lHighlightRow to determine if we want that behavior, cSelectedBackColor and cSelectedForeColor to determine the colors for the selected row, nBackColor to save the former background color for a row so it can be restored when another row is selected, nRecno to keep track of the current row being highlighted, lGridHasFocus and lJustGotFocus so we can track when the grid is selected, code in Init to call SetupColumns if lAutoSetup is .T., SetupColumns to set DynamicBackColor and DynamicForeColor to expressions that highlight the current row, and code in When, Valid, BeforeRowColChange, and AfterRowColChange to support it. Whew!

In the VFP 8 version of SFGrid, I removed most of these custom properties and methods because VFP 8 has a much simpler way to do this: set the new HighlightStyle property to 2-Current row highlighting enable with visual persistence, and HighlightBackColor and HighlightForeColor to the desired colors.

I also added an lAutoFit property. If this property is .T., the SetupColumns method calls the AutoFit method so all the columns are sized appropriately for their data. While the user can do this manually by double-clicking in the grid, I prefer to do it for the user automatically.

DemoGrid.SCX shows some of the features of SFGrid, including automatic auto-fitting and automatic header captions, and resizing when the form resizes, which we discussed in the June 2003 issue.

Error Handling

One of the most important enhancements in VFP 8 is the addition of structured error handling. The TRY structure is part of a three-level error handling mechanism in VFP: true local error handling (using TRY), object error handling (using the Error method), and global error handling (via ON ERROR). I discussed structured error handling in the January 2003 issue of FoxTalk ("CATCH Me if You Can").

However, one complication is that the new THROW command, which raises an error in the next highest error handler, doesn't play well with either Error methods or ON ERROR routines. That's because THROW passes an Exception object (one of the new base classes in VFP 8) to the error handler, but only the CATCH clause of a TRY structure is prepared to receive such an object. As a result, information about

the error, such as the error number, line number, and so forth, pertains to the THROW statement, not to any original error that occurred.

One way to handle this is to store the exception object, which contains information about the original error, to a persistent location (such as a global variable or property) before using THROW. That way, if the next highest error handler is an Error method or ON ERROR routine, that error handler can get the correct information about the original error by examining the stored exception object.

To support this, I added an oException property to all base classes. Any code in an object that uses a THROW should first store an exception object in oException. The Error method checks to see if oException contains an object, and if so, gets information about the error from it. (Not all of the code in Error is shown here, only the code applicable to oException.)

```
lparameters tnError, ;
  tcMethod, ;
  tnLine
* LOCAL statement omitted

* Use AERROR() to get information about the error. If we
* have an Exception object in oException, get information
* about the error from it.

lnError  = tnError
lcMethod = tcMethod
lnLine   = tnLine
lcSource = message(1)
aerror(laError)
with This
  if vartype(.oException) = 'O'
    lnError  = .oException.ErrorNo
    lcMethod = .oException.Procedure
    lnLine   = .oException.LineNo
    lcSource = .oException.LineContents
    laError[cnAERR_NUMBER]  = .oException.ErrorNo
    laError[cnAERR_MESSAGE] = .oException.Message
    laError[cnAERR_OBJECT]  = .oException.Details
    .oException = .NULL.
  endif vartype(.oException) = 'O'
endwith
* Remainder of code omitted
```

To see how this works, run DemoErrorHandling.SCX. The two buttons have identical code which causes an error and throws an Exception object, but the second one sets oException to the Exception object before using THROW. Notice the difference in the error message displayed when you click on each button; the first one indicates that the error is an unhandled structured exception, which is caused by the THROW command, while the second displays the correct message for the error that occurred (the table the code is trying to open doesn't exist.)

Collection

Collections are a common way to store multiple instances of things. For example, a TreeView control has a Nodes collection and Microsoft Word has a Documents collection. Until recently, Visual FoxPro developers wanting to use collections often created their own classes that were nothing more than fancy wrappers for arrays. However, in addition to being a lot of code to write, home-built collections don't support the FOR EACH syntax, which is especially awkward when they're exposed in COM servers. (I wrote about such a class in the July 1998 issue of FoxTalk, "Collecting Objects".) VFP 8 solves this problem by providing a true Collection base class.

Because we should never use VFP base classes, I created a subclass of Collection called SFCollection. This subclass has the same About (used for documentation), Error (implementing the same error handling mechanism all SF classes use), and Release (so all classes can be released by calling this method) methods my other base classes have. It also has a GetArray method that provides a means of filling an array with information about items in the collection; this is handy when you want to base a combo box or list box on a collection of items, since these controls don't support collections but they do support arrays. GetArray calls the abstract FillArrayRow method, which must be coded in a subclass to put the desired elements of a

collection item into the current row of the array. SFCollection also has FillCollection and SaveCollection methods that allow you to fill a collection from and save a collection to persistent storage. FillCollection is called from Init if the custom lFillOnInit property is .T.; it's abstract in SFCollection and must be coded in a subclass. SaveCollection spins through the collection and calls the abstract SaveItem method, which a subclass will implement to save the current item.

MetaData.PRG shows an interesting use of collections: so meta data about tables, fields, and indexes appear as collections of collections. First, a generic MetaDataCollection class is defined as follows:

```
define class MetaDataCollection as SFCollection of SFCtrls.vcx
  protected function FillArrayRow(taArray, tnItem, toItem)
    dimension taArray[tnItem, 2]
    taArray[tnItem, 1] = toItem.Caption
    taArray[tnItem, 2] = This.GetKey(tnItem)
  endfunc
enddefine
```

The FillArrayRow method is overridden here so when the GetArray method is called, the specified array is filled with the caption and key name of each item in the collection.

Classes are defined to simply hold properties about fields and indexes:

```
define class Field as Custom
  DataType   = ''
  Length     = 0
  Decimals   = 0
  Binary     = .F.
  AllowNulls = .F.
  Caption    = ''
enddefine

define class Index as Custom
  Expression = ''
  Filter     = ''
  Type       = ''
  Ascending  = .F.
  Collate    = ''
  Caption    = ''
enddefine
```

The table class is slightly more complicated: in addition to properties about a table, it also adds fields and indexes collections:

```
define class Table as Custom
  CodePage  = 0
  BlockSize = 0
  Caption   = ''

  add object Fields  as MetaDataCollection
  add object Indexes as MetaDataCollection
enddefine
```

Finally, a class to manage a collection of tables is defined. Since lFillOnInit is set to .T., the FillCollection method is called when the class instantiates. The code in that method fills the collections of tables, fields, and indexes from the CoreMeta table of a DBCX-based set of meta data. (If you're not familiar with DBCX, it's a public domain data dictionary available for download from the Technical Papers page of http://www.stonefield.com.)

```
define class Tables as MetaDataCollection
  lFillOnInit = .T.

  procedure FillCollection
    local lcTable, ;
      loTable, ;
      lcField, ;
```

```
      loField, ;
      lcIndex, ;
      loIndex
    use CoreMeta
    scan
      do case

* If this is a table or view, add it to the collection.

        case cRecType $ 'TV'
          lcTable = trim(cObjectNam)
          loTable = createobject('Table')
          with loTable
            .CodePage  = nCodePage
            .BlockSize = nBlockSize
            .Caption   = trim(cCaption)
          endwith
          This.Add(loTable, lcTable)

* If this is a field, add it to the appropriate table.

        case cRecType = 'F'
          lcTable = juststem(cObjectNam)
          lcField = trim(justext(cObjectNam))
          loField = createobject('Field')
          with loField
            .DataType   = cType
            .Length     = nSize
            .Decimals   = nDecimals
            .Binary     = lBinary
            .AllowNulls = lNull
            .Caption    = trim(cCaption)
          endwith
          This.Item(lcTable).Fields.Add(loField, lcField)

* If this is an index, add it to the appropriate table.

        case cRecType = 'I'
          lcTable = juststem(cObjectNam)
          lcIndex = trim(justext(cObjectNam))
          loIndex = createobject('Index')
          with loIndex
            .Expression = mTagExpr
            .Filter     = mTagFilter
            .Type       = cTagType
            .Ascending  = lAscending
            .Collate    = trim(cCollate)
            .Caption    = trim(cCaption)
          endwith
          This.Item(lcTable).Indexes.Add(loIndex, ;
            lcIndex)
      endcase
    endscan
    use
  endproc
enddefine
```

DemoMetaData.SCX shows how these classes can be used. The table combo box gets its rows from an array of tables in the collection, retrieved by calling the GetArray method of the tables collection. The field combo box is based on an array of fields filled by calling the GetArray method of the fields collection of the table object for the table selected by the table combo box.

Summary

Since your subclasses of the VFP base classes form the foundation of all other classes and forms, it's a good idea to revisit them from time to time and freshen them up to match new capabilities in VFP. I hope you find the ideas in this article useful and that it motivates you to examine your own base classes.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com