Practical Uses for New Features in Visual FoxPro 8.0

Doug Hennig
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Voice: 306-586-3341
Fax: 306-586-5080
Email: dhennig@stonefield.com
Web: http://www.stonefield.com
Web: http://www.stonefieldquery.com

Overview

Going over a list of the new features in VFP 8 doesn't really tell you how and where to use them. This document takes a practical look at new features to create things like picklist grids, page frames with delayed instantiation of contents, labeled option boxes, objects that know how to resize themselves in a form, dynamic data objects, and many more.

Introduction

VFP 8 is probably the most significant upgrade to FoxPro since VFP 3 was released. Nearly every facet of the product has been improved in some way. That's the good news. The bad news is that it can take quite a while to figure out what's new in VFP 8 and how it applies to you and your development challenges.

The purpose of this document is to examine some practical uses for some of the new features in VFP 8. We'll start by looking at changes I've made to my base classes to accommodate new and improved features, such as page frames with my own Page classes and controls that know how to resize themselves. Then we'll look at some specialized classes to do things like provide picklist grids, labeled option boxes, and dynamic data objects.

Base Classes

Like many VFP developers, I created my own set of subclasses of the VFP base classes years ago. 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.

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; 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 has no code 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.
```

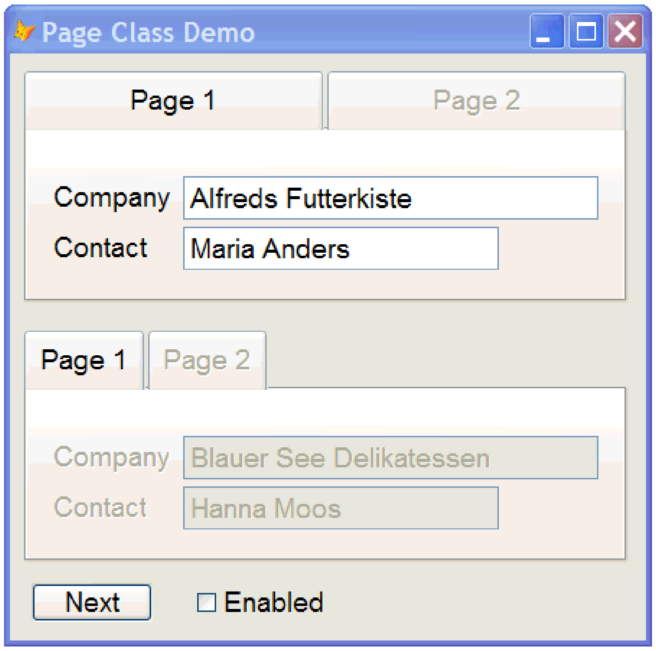
```
   if pemstatus(Thisform, ;
      'SetFocusToFirstObject', 5)
      Thisform.SetFocusToFirstObject(This)
   endif pemstatus(Thisform ...

* 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, automatic header captions, and resizing when the form resizes, which we'll discuss next.

Handling Container Resizing

Until VFP 8, there were two ways to deal with what happens when a container, such as a form, is resized. One was to put code into the Resize method of the container that resized all the controls. That usually resulted in a ton of manually written code setting the Top, Left, Height, and Width properties of each control by control name, always a dangerous proposition (what if you rename a control?). The other way was to use a resizer object responsible for iterating through the controls in the form, resizing each appropriately. The FoxPro Foundation Classes (FFC) that come with VFP provided a class, _Resizable, to handle this for you. I created a subclass of _Resizable called SFResizable that had better behavior, including allowing you to define how each control is resized. However, it again meant specifying controls by name and having to remember to update the SFResizable object when new controls were added to the form.

Although earlier versions of VFP had the ability to bind to events in COM objects, VFP 8 provides event binding to native controls. Event binding means that an object can be informed when some event is triggered. In the case of resizing, that means each control is informed when its container is resized and is itself responsible for what should happen.

To implement this, I added an lHandleContainerResize property, .F. by default, to most of the visual classes in SFCtrls.VCX: SFCheckBox, SFComboBox, SFCommandButton, SFContainer, SFEditBox, SFGrid, SFImage, SFLabel, SFLine, SFListBox, SFOptionGroup, SFPageFrame, SFShape, SFSpinner, and SFTextBox. They also have a HandleContainerResize method and cAnchor, nContainerHeight, nContainerWidth, nHeight, nLeft, nTop, and nWidth properties. The Init method binds the Resize event of the object's container (the page frame in the case of the parent being a page) to the HandleContainerResize method and saves the current size and position of the container and the object if lHandleContainerResize is .T.:

```
with This
```

```
* If we're supposed to bind to the Resize event
* of our container, and there is
* one, do so. If we're in a page of a pageframe,
* bind to the pageframe. Save the current size and position of the
* container and us.

   if .lHandleContainerResize and type('.Parent.Name') = 'C'
      if upper(.Parent.BaseClass) = 'PAGE'
         bindevent(.Parent.Parent, 'Resize', This, ;
            'HandleContainerResize')
      else
         bindevent(.Parent, 'Resize', This, 'HandleContainerResize')
      endif upper(.Parent.BaseClass) = 'PAGE'
      .nContainerHeight = .Parent.Height
      .nContainerWidth  = .Parent.Width
      .nWidth           = .Width
      .nHeight          = .Height
      .nLeft            = .Left
      .nTop             = .Top
   endif .lHandleContainerResize ...
endwith
```

The HandleContainerResize method, called when the container is resized, adjusts the Height, Left, Top, and Width properties according to the setting of the cAnchor property. cAnchor should contain some combination of the following values:

Value	Anchor control to
L	Left edge of container
R	Right edge of container
V	Vertical center of container
T	Top edge of container
B	Bottom edge of container
H	Horizontal center of container

“Anchoring” means that the various edges of the control are virtually (not actually) bound to part of the container. For example, if a control is anchored to the right edge of the form (cAnchor contains “R”), as the form is widened, the control will move to the right. If cAnchor is set to “LR”, the control won’t move since its left edge is bound to the left edge of the container but will instead become wider since its right edge is bound to the right edge of the container. You can use various combinations of anchor settings to achieve the desired result. For example, “LRTB” means the object will grow horizontally and vertically as the container is resized. Use “V” and “H” when you want an object to be centered (for example, “BV” will keep OK and Cancel buttons centered at the bottom of a form) or for objects that need to be resized that sit beside other objects that are resized.

Here’s the code in HandleContainerResize that does all the heavy lifting (AEVENTS() gives us a reference to the object whose event we’re bound to):

```
local laEvents[1], ;
   lnHeight, ;
   lnWidth, ;
   lnHeightAdjust, ;
   lnWidthAdjust, ;
   lcAnchor
```

```
with This

* Get the new height and width of the container
* and how much it changed by.

   aevents(laEvents, 0)
   lnHeight       = laEvents[1].Height
   lnWidth        = laEvents[1].Width
   lnHeightAdjust = lnHeight - .nContainerHeight
   lnWidthAdjust  = lnWidth  - .nContainerWidth

* Adjust the width and left as appropriate.

   lcAnchor = upper(.cAnchor)
   do case
      case 'L' $ lcAnchor and 'R' $ lcAnchor
         .Width = max(.nWidth + lnWidthAdjust, 0)
      case 'L' $ lcAnchor and 'V' $ lcAnchor
         .Width = max(.nWidth + lnWidthAdjust/2, 0)
      case 'R' $ lcAnchor and 'V' $ lcAnchor
         .Width = max(.nWidth + lnWidthAdjust/2, 0)
         .Left  = .nLeft + lnWidthAdjust/2
      case 'V' $ lcAnchor
         .Left  = .nLeft + lnWidthAdjust/2
      case 'R' $ lcAnchor
         .Left  = .nLeft + lnWidthAdjust
   endcase

* Adjust the height and top as appropriate.

   do case
      case 'T' $ lcAnchor and 'B' $ lcAnchor
         .Height = max(.nHeight + lnHeightAdjust, 0)
      case 'T' $ lcAnchor and 'H' $ lcAnchor
         .Height = max(.nHeight + lnHeightAdjust/2, 0)
      case 'B' $ lcAnchor and 'H' $ lcAnchor
         .Height = max(.nHeight + lnHeightAdjust/2, 0)
         .Top    = .nTop + lnHeightAdjust/2
      case 'H' $ lcAnchor
         .Top    = .nTop + lnHeightAdjust/2
      case 'B' $ lcAnchor
         .Top    = .nTop + lnHeightAdjust
   endcase
endwith
```

SFForm has This.Resize() in Init so all controls resize properly at startup.

To see how this feature works, run DemoResizeBinding.SCX and resize the form. Notice that the page frame, the grid in page 1, and the edit boxes in page 2 automatically resize themselves, and the OK and Cancel buttons stay centered. No code was required to do this, just setting lHandleContainerResize to .T. and cAnchor as appropriate.

There are a couple of issues with this mechanism I haven't worked out as of this writing. One is that when you programmatically change the Height and Width of a container, the HandleContainerResize of all member controls is called for each change, so this method gets called twice. It’d be nice to defer firing HandleContainerResize until both Height and Width are changed. The other issue is what happens when you click on the Minimize and Maximize buttons of the form. It appears that the Height of the page frame doesn’t change until after the HandleContainerResize method of the grid and edit box are fired, so while the width of these controls is adjusted properly, the height is not (although sometimes in my testing, it was Width that didn’t change).

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).

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.

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. VFP 8 solves this problem by providing a true Collection base class.

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
```

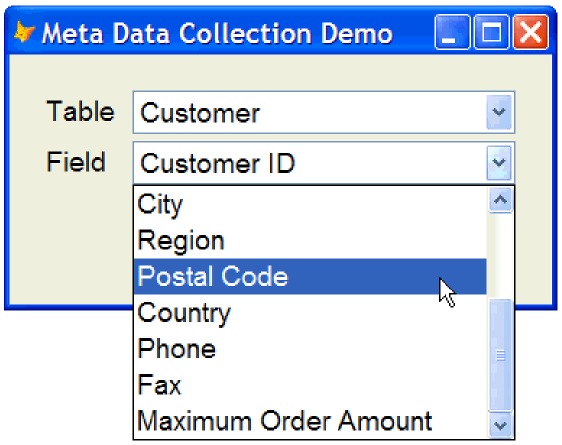
The table class is slightly more complicated: in addition to properties about a table, it also adds fields and indexes collections:

```
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 Paper 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.



Specialized Classes

Now that we've looked at new base classes and changes to existing ones, let's take a look at some specialized classes that provide specific functionality, such as picklist grids and labeled option boxes, using new features in VFP 8.

Picklist Grids

As I mentioned earlier when I discussed grids, creating a grid that behaves like a list box was a lot of work in earlier versions of VFP. In VFP 8, it's ridiculously easy because of some new grid properties: set HighlightStyle to 2-Current row highlighting enable with visual persistence, HighlightBackColor and HighlightForeColor to the desired colors, and AllowCellSelection to .F. The latter property is the key to a picklist-style grid: it prevents the user from setting focus to any cell, thus making each row in the grid behave like a single, multi-columned object.

Because picklists are often used in applications, I created a subclass of SFGrid called SFPickListGrid (in SFCCtrls.VCX). Its properties are set as follows:

Property	**Value**
AllowCellSelection	.F.
DeleteMark	.F.
HighlightStyle	2-Current row highlighting enable with visual persistence
RecordMark	.F.
ScrollBars	2-Vertical

To provide incremental searching and allow the user to quickly move to the first or last rows with the Home and End keys, the KeyPress method has the following code:

```
lparameters tnKeyCode, ;
   tnShiftAltCtrl
local lcAlias, ;
```

```
   lcOrder, ;
   lnRecno, ;
   lcCurrNear, ;
   lcCurrExact, ;
   lcKey, ;
lcType
with This

* Get the cursor alias and current order.

   lcAlias = .RecordSource
   lcOrder = order(lcAlias)
   do case

* If we have a character keystroke and an order for the table, do an
* incremental search. First, save some other things and set them the
* way we want.

      case between(tnKeyCode, 32, 127) and not empty(lcOrder)
         lnRecno     = iif(eof(lcAlias) or bof(lcAlias), 1, ;
            recno(lcAlias))
         lcCurrNear  = set('NEAR')
         lcCurrExact = set('EXACT')
         set near on
         set exact off

* If the user pressed BackSpace, delete the last character from the
* search string (if there is one).

         do case
            case tnKeyCode = 127 and len(.cSearchString) > 0
               .cSearchString = left(.cSearchString, ;
                  len(.cSearchString) - 1)

* If the user is entering characters within the "incremental search"
* time period, add them to the search string.

            case seconds() - .nKeyTime <= _incseek
               .cSearchString = .cSearchString + chr(tnKeyCode)

* The search string is just the character entered.

            otherwise
               .cSearchString = chr(tnKeyCode)
         endcase
         .nKeyTime = seconds()
         lcKey     = key(tagno(lcOrder, '', lcAlias), lcAlias)
         lcType    = type(lcKey)
         do case

* Look for the search string: if we can't find it, try seeking on the
* UPPER() function.

            case lcType = 'C'
               if not seek(.cSearchString, lcAlias, lcOrder) and ;
                  not seek(upper(.cSearchString), lcAlias, lcOrder)
                  go lnRecno in (lcAlias)
               endif not seek(.cSearchString, ...

* Handle other data types for the SEEK.

            case lcType $ 'NFIYB'
               = seek(val(.cSearchString), lcAlias, lcOrder)
            case lcType = 'D'
               = seek(ctod(.cSearchString), lcAlias, lcOrder)
            case lcType = 'T'
               = seek(ctot(.cSearchString), lcAlias, lcOrder)
         endcase

* Set things back the way they were.

         if lcCurrExact = 'ON'
            set exact on
         endif lcCurrExact = 'ON'
         if lcCurrNear = 'OFF'
            set near off
         endif lcCurrNear = 'OFF'

* If the user pressed Home, move to the first record.

      case tnKeyCode = 1
         go top in (lcAlias)

* If the user pressed End, move to the last record.

      case tnKeyCode = 6
         go bottom in (lcAlias)
   endcase
endwith
```

This code uses two custom properties of SFPickListGrid: cSearchString, which contains the string of characters to perform an incremental search on, and nKeyTime, which contains the last time a key was pressed, used to clear cSearchString after a pause.

Allowing the user to change how the list is sorted is a useful feature, so the SetupColumns method, defined in SFGrid but overridden here, has code that goes through each column and substitutes the header of any column displaying a field that has a tag on it with an SFHeaderSortable object, which allows the user to click on the header to sort on that field. (We won’t look at SFHeaderSortable, which is defined in SFClasses.PRG; feel free to examine it yourself.)

```
local lcOrder, ;
   laTags[1], ;
   lnTags, ;
   llUseAdd, ;
   loColumn, ;
   lcField, ;
   lnI, ;
   lcTag, ;
   lcCaption

* Do the usual behavior first so all the columns are set up properly.
```

```
dodefault()

* Get the current order for the table and a list of its tags, then go
* through each column, checking whether the field for the column is
* involved in a tag.

lcOrder  = order(This.RecordSource)
lnTags   = ataginfo(laTags, '', This.RecordSource)
llUseAdd = '\SFCLASSES.PRG' $ upper(set('PROCEDURE'))
for each loColumn in This.Columns
   lcField = loColumn.ControlSource
   for lnI = 1 to lnTags

* If this field is indexed, switch the header to an SFHeaderSortable
* object

      if atc(justext(lcField), laTags[lnI, 3]) > 0 or ;
         atc(lcField, laTags[lnI, 3]) > 0
         lcTag     = laTags[lnI, 1]
         lcCaption = loColumn.Header1.Caption
         if llUseAdd
            loColumn.AddObject('ColHeader', 'SFHeaderSortable')
         else
            loColumn.NewObject('ColHeader', 'SFHeaderSortable', ;
               'SFClasses.prg')
         endif llUseAdd

* Set the properties of the new header. If the tag for this field is
* the main one for the table, set the picture of the header. Set the
* caption and order that this column uses.

         with loColumn.ColHeader
            .Picture  = iif(lcOrder == lcTag, .cAscImage, '')
            .Caption  = lcCaption
            .FontName = This.FontName
            .FontSize = This.FontSize
            .cOrder   = laTags[lnI, 1]
         endwith

* If we're supposed to auto-fit, do so for the column since the picture
* could affect the width.

         if This.lAutoFit
            loColumn.AutoFit()
         endif This.lAutoFit
         exit
      endif atc(justext(lcField), laTags[lnI, 3]) > 0 ...
   next lnI
next loColumn
```

Rather than subclassing SFPickListGrid for each type of picklist, I decided to create a generic form called SFPickListForm, also defined in SFCCtrls.VCX, which accepts some parameters about what the picklist should contain. The Init method expects to be passed the name of the table or view (including path or database name if necessary, since the form has a private data session), the initial tag to use, a comma-delimited list of field names to display in the grid, the field or expression to return if the user chooses OK, and the caption for the form.

```
lparameters tcTable, ;
   tcOrder, ;
   tcColumns, ;
   tcReturn, ;
   tcCaption
local lcOrder, ;
   lcAlias, ;
   lnColumns, ;
   lnI
with This

* Open the specified table using the specified order. If we can't, exit
* now.

   lcOrder = iif(empty(tcOrder), '', 'order ' + tcOrder)
   try
      use (tcTable) &lcOrder again shared noupdate
   catch
   endtry
   lcAlias = alias()
   if empty(lcAlias)
      return .F.
   endif empty(lcAlias)

* Set the RecordSource for the grid and ControlSource for each column
* to the specified alias and fields.

   .grdPickList.RecordSource = lcAlias
   lnColumns = alines(laColumns, tcColumns, .T., ',')
   .grdPickList.ColumnCount = lnColumns
   for lnI = 1 to lnColumns
      .grdPickList.Columns[lnI].ControlSource = lcAlias + '.' + ;
         laColumns[lnI]
   next loColumn

* Store the name of the return column and set the form caption.

   .cReturn = tcReturn
   if not empty(tcCaption)
      .Caption = tcCaption
   endif not empty(tcCaption)

* Finish off with the default behavior.

   dodefault()
endwith
```

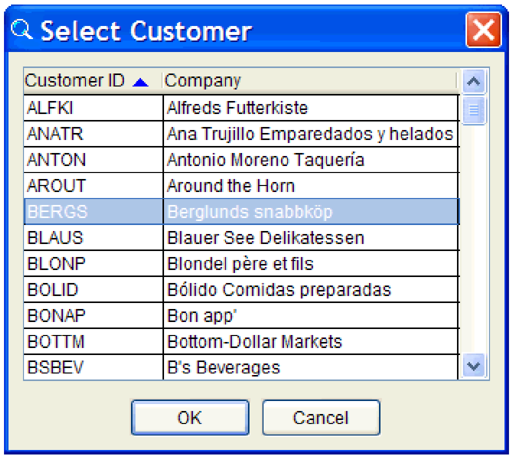
The Show method sets up the columns and adjusts the width of the grid and form so everything fits properly.

```
lparameters tnStyle
local lnWidth
with This
```

```
* Set up the columns.

   .grdPickList.SetupColumns()

* Determine the width of the grid and the form based on the column
* widths.

   lnWidth = 0
   for each loColumn in .grdPickList.Columns
      lnWidth = lnWidth + loColumn.Width
   next lnWidth
   .grdPickList.Width = lnWidth + sysmetric(5) + 6
   store .grdPickList.Width + .grdPickList.Left * 2 to .Width, ;
      .MinWidth, .MaxWidth
   .Resize()
endwith
```

The SelectRecord method, called if the user clicks on the OK button or presses Enter in the grid, sets the cSelected property to the value of the return field for the current record and hides the form so execution returns to the caller.

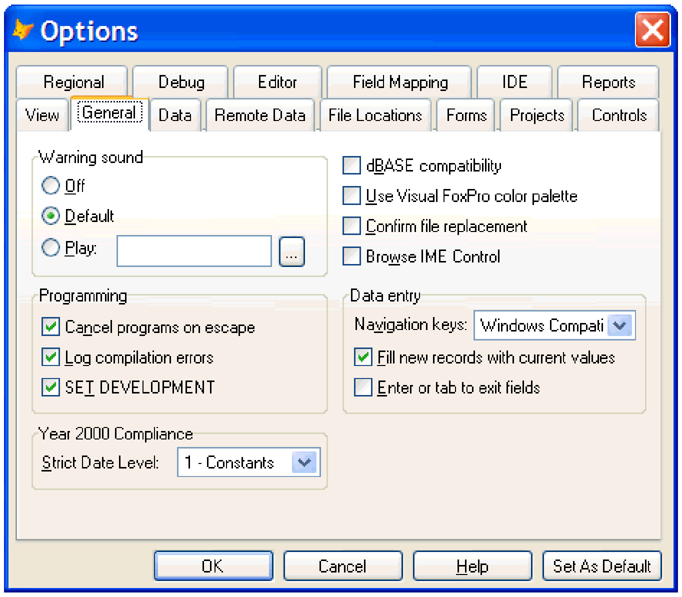
DemoPickList.PRG shows how this works. Try clicking in the header of either column, typing a few letters to jump to the first customer with those letters, and choosing OK or Cancel. Note that incremental searching is based on the index expression of a tag, so if UPPER() isn't used (as is the case for the Company field in this example), the search will be case-sensitive.



Customer ID ▲	Company
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel père et fils
BOLID	Bólido Comidas preparadas
BONAP	Bon app'
BOTTM	Bottom-Dollar Markets
BSBEV	B's Beverages

Labeled Option Boxes

Some forms group related controls visually using a labeled option box. For example, the Tools, Options dialog uses this type of box in several places. The image below shows three of them: the Warning sound, Programming, and Data entry areas of the General page.


Options

Regional | Debug | Editor | Field Mapping | IDE | Reports
View | General | Data | Remote Data | File Locations | Forms | Projects | Controls

Warning sound
- ○ Off
- ● Default
- ○ Play:

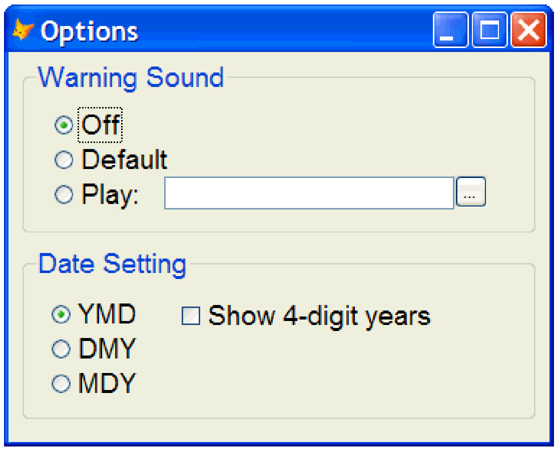
- ☐ dBASE compatibility
- ☐ Use Visual FoxPro color palette
- ☐ Confirm file replacement
- ☐ Browse IME Control

Programming
- ✓ Cancel programs on escape
- ✓ Log compilation errors
- ✓ SET DEVELOPMENT

Data entry
- Navigation keys: Windows Compati
- ✓ Fill new records with current values
- ☐ Enter or tab to exit fields

Year 2000 Compliance
- Strict Date Level: 1 - Constants

Creating a labeled option box is easy: simply add a label and a shape to a container and position and size them correctly. However, to make it look good in Windows XP requires a new feature in VFP 8: setting the Style property of each of the components to 3-Themed. Even though it doesn't seem like you need to, you must set the container's Style to 3-Themed as well. Otherwise, the labeled option box won't appear correctly when it's on a page of a page frame. SFLabelledBox, in SFCCtrls.VCX, is an example of such an object. To see an example of its use, run DemoLabelledBox.SCX.


Options

Warning Sound
- ● Off
- ○ Default
- ○ Play:

Date Setting
- ● YMD
- ○ DMY
- ○ MDY
- ☐ Show 4-digit years

One complication with using this control: when you size an instance of it at design time, the shape inside it isn't automatically sized. This means you have to drill into the container and resize the shape as well. Also, you have to drill into the container to change the caption of the label. This sounds like the perfect use for a builder, so set the Builder property of SFLabelledBox, which defines the builder to use for this class, to SFLabelledBoxBuilder.PRG. That PRG contains the following code, which simply adjusts the size of the shape to match the container and prompts for the caption of the label:

```
local laObjects[1], ;
   loObject, ;
   lcCaption
aselobj(laObjects)
loObject = laObjects[1]
with loObject
   .shpBox.Width  = .Width
   .shpBox.Height = .Height - .shpBox.Top
   lcCaption = inputbox('Caption:', 'Labeled Box Builder', ;
      .lblBox.Caption)
   if not empty(lcCaption)
      .lblBox.Caption = lcCaption
   endif not empty(lcCaption)
endwith
```

Dynamic Data Objects

Some people like to create "data" objects by specifying the NAME clause in the SCATTER command. This creates an object of class Empty, which has no properties, events, or methods (PEMs) at all, creates a property for each field in the current table, and fills those properties with the current record (unless you specify the BLANK clause). One reason to use such a data object is that it's a very lightweight object you can easily pass to anything, including non-VFP COM objects such as Microsoft Word and Excel.

However, one drawback of these data objects is that they are so lightweight, they contain nothing but the properties matching the fields in the table. Sometimes, it'd be useful to have an object that has behavior as well as data. For example, maybe you'd like to perform some rudimentary validation for the properties. Unfortunately, since SCATTER NAME always created a new object of class Empty, this wasn't possible in earlier versions of VFP.

Fortunately, VFP 8 has a great new clause in the SCATTER command: ADDITIVE. If you specify this clause and an object with the name specified in the NAME clause already exists, it'll be used rather than creating a new Empty object. The object doesn't have to have property names that match the fields in the table; VFP will automatically create new properties as necessary.

Here's a simple example, taken from DemoDataObject.PRG, which shows a data object that splits a single contact name into its component first, middle, and last names. The access methods for the FirstName, MiddleName, and LastName properties retrieve the appropriate part of the Contact property (which isn't defined in this class but will automatically be added by a SCATTER command), and the assign methods update the Contact property appropriately.

```
use _samples + 'data\customer'
browse
loCustomer = createobject('CustomerObject')
scatter name loCustomer additive
messagebox('Contact: ' + loCustomer.Contact + chr(13) + ;
   'First name: ' + loCustomer.FirstName + chr(13) + ;
   'Middle name: ' + loCustomer.MiddleName + chr(13) + ;
   'Last name: ' + loCustomer.LastName)
loCustomer.FirstName  = 'Sue'
loCustomer.MiddleName = 'Ellen'
messagebox('Contact: ' + loCustomer.Contact + chr(13) + ;
   'First name: ' + loCustomer.FirstName + ;
      chr(13) + ;
   'Middle name: ' + loCustomer.MiddleName + ;
      chr(13) + ;
   'Last name: ' + loCustomer.LastName)

define class CustomerObject as Custom
   FirstName  = ''
   MiddleName = ''
   LastName   = ''

   function FirstName_Access
      This.FirstName = getwordnum(This.Contact, 1)
      return This.FirstName
   endfunc

   function FirstName_Assign(tcValue)
      This.FirstName = tcValue
      This.BuildName()
   endfunc

   function MiddleName_Access
      local lcContact, ;
         lnWords, ;
         lnPos
      lcContact = This.Contact
      lnWords   = getwordcount(lcContact)
      do case
         case lnWords > 3
            lnPos = at(' ', lcContact) + 1
            This.MiddleName = substr(lcContact, lnPos, ;
               rat(' ', lcContact) - lnPos)
         case lnWords = 3
            This.MiddleName = getwordnum(lcContact, 2)
         otherwise
            This.MiddleName = ''
      endcase
      return This.MiddleName
   endfunc

   function MiddleName_Assign(tcValue)
      This.MiddleName = tcValue
      This.BuildName()
   endfunc

   function LastName_Access
      This.LastName = getwordnum(This.Contact, ;
         getwordcount(This.Contact))
      return This.LastName
   endfunc

   function LastName_Assign(tcValue)
      This.LastName = tcValue
      This.BuildName()
   endfunc

   function BuildName
```

```
      This.Contact = This.FirstName + ' ' + ;
         iif(empty(This.MiddleName), '', ;
         This.MiddleName + ' ') + ;
         This.LastName
   endfunc
enddefine
```

Parameters Objects

If you have a function that requires a large number of parameters, or needs to return more than a single value, a "parameters" object can be useful. A parameters object is simply one that contains the values of input and output parameters as properties. You create a parameters object, set the properties for input parameters appropriately, pass it to a function or method, and upon return, read the values of the output properties.

Until VFP 8, you had to do one of two things to use a parameter object: predefine a class with the appropriate properties, or instantiate a base class object that supports the AddProperty method and add the properties dynamically at runtime. VFP 8 provides a more lightweight object that's perfect for this: the Empty base class. As we saw in the previous section, Empty has actually existed for a long time in VFP, but until VFP 8, we couldn't create an Empty object directly.

Since Empty has no PEMs, it doesn't support the AddProperty method, so how do you add the properties you want? Using the new ADDPROPERTY() function.

DemoParametersObject.PRG shows an example of a parameters object. This PRG calls a form (Reindex.SCX) in which the user can select which tables should be reindexed, and whether they should also be packed and sorted. So, the form really has four return values: whether the user chose OK or Cancel, which tables they chose, and whether the Pack and Sort options were checked. Since you can only have one return value, a parameters object provides a way to return all four of these.

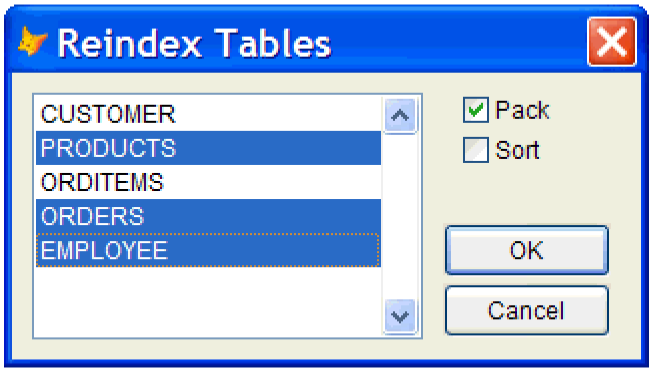
The code first creates an Empty object and adds the desired properties to it. Note that oTables is actually a collection. In previous versions of VFP, this would have been an array, but collections are often more convenient to work with than arrays. The collection is filled with Empty objects, each containing the name of a table in the TESTDATA database and a flag indicating whether the table should be reindexed. After running the Reindex form, the various properties of the parameters object are examined to see what choices the user made.

```
* Create a parameters object and add the parameters properties to it.

loParameters = createobject('Empty')
addproperty(loParameters, 'oTables',  createobject('Collection'))
addproperty(loParameters, 'lPack',     .T.)
addproperty(loParameters, 'lSort',     .F.)
addproperty(loParameters, 'lReindex', .F.)

* Get a list of the tables in the TESTDATA database and add "table"
* objects to the collection in the parameters object.

open database _samples + 'data\testdata'
lnTables = adbobjects(laTables, 'Table')
```

```
for each lcTable in laTables
   loTable = createobject('Empty')
   addproperty(loTable, 'cName',    lcTable)
   addproperty(loTable, 'lReindex', .F.)
   loParameters.oTables.Add(loTable, lcTable)
next lcTable

* Call the reindex form, passing it the parameters object.

do form Reindex with loParameters

* If the user chose OK, go through each table and see what we're
* supposed to do with it.

if loParameters.lReindex
   lcMessage = 'The following tables will be reindexed, ' + ;
      iif(loParameters.lPack, '', 'not ') + 'packed, and ' + ;
      iif(loParameters.lSort, '', 'not ') + 'sorted:' + chr(13)
   for each loTable in loParameters.oTables
      if loTable.lReindex
         lcMessage = lcMessage + chr(13) + loTable.cName
      endif loTable.lReindex
   next loTable
   messagebox(lcMessage)
else
   messagebox('You clicked on Cancel')
endif loParameters.lReindex
```

The Init method of the Reindex form saves the passed parameters object to a property and fills the list box with the names of the tables:

```
lparameters toParameters
with This
   .oParameters = toParameters

* Fill the list of tables from the collection.

   for each loTable in toParameters.oTables
      .lstTables.AddItem(loTable.cName)
   next loTable
   .lstTables.Requery()
endwith
```



The pack and sort check boxes are bound to the lPack and lSort properties of Thisform.oParameters. The Click method of the OK button sets the lReindex properties of the parameters object and the table objects in the tables collection:

```
local lnI, ;
   lcTable
with Thisform
   .oParameters.lReindex = .T.
   for lnI = 1 to .lstTables.ListCount
      lcTable = .lstTables.List[lnI]
      .oParameters.oTables(lcTable).lReindex = .lstTables.Selected[lnI]
   next lnI
endwith
Thisform.Release()
```

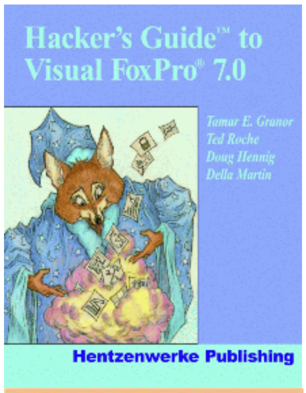
The only drawback of Empty is that it can't be subclassed, so you can't predefine a parameters object class. So, you have to use a series of ADDPROPERTY() statements each time you create a parameters object. However, other than that minor limitation, Empty is a great class to use.

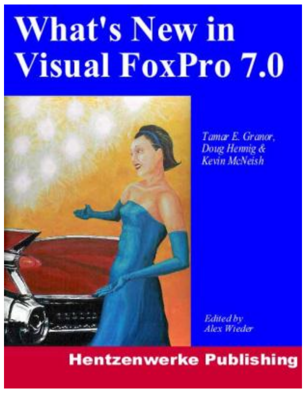
Summary

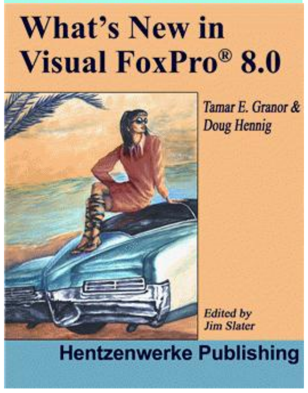
VFP 8 has so many new features that simply going over a list of them may be quite overwhelming. This document took a practical look at some of the new features to either do things we couldn't do before or we could but using a lot of ugly, hard-to-maintain code. I hope you find the classes included with this document useful!

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and the CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro 8. Doug is co-author of "What's New in Visual FoxPro 8.0", "The Hacker's Guide to Visual FoxPro 7.0", and "What's New in Visual FoxPro 7.0". He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals". All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). Doug writes the monthly "Reusable Tools" column in FoxTalk. He 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).




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Edited by Alex Wieder
Hentzenwerke Publishing


What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341 Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com
Web: www.stonefieldquery.com