Access and Assign Methods in VFP

By Doug Hennig

Introduction

In my opinion, one of the most useful and powerful additions in Visual FoxPro 6 is access and assign methods. It may seem strange that something as simple as this can be considered powerful, but in this document we'll see some very compelling reasons to use these methods, including the ability to strongly type properties, automatically validate the values of properties, create read-only properties, and support collection classes.

Access and Assign Methods

What are access and assign methods? If you're familiar with Visual Basic, access and assign methods are like Get and Let methods. The access method of a property is automatically fired whenever anything tries to access the value of the property in any way, such as displaying the value or using it in a calculation. The caller sees the return value of the method as the property's value, so typically the access method will return the value of the property. However, you can probably imagine cases where you do something first, like performing a calculation, before returning the value. Similarly, the assign method is fired whenever anything tries to change the value of the property. The assign method is passed the value the caller wants to assign to the property, so it will typically store that value to the property. However, you could do lots of things with the specified value first, such as ensuring it's the proper data type.

Access and assign methods are named the same as the property with "_access" or "_assign" as a suffix; for example, for the Item property, these methods are called Item_Access and Item_Assign. Access and assign methods are independent; you can have one without the other. There are two ways you can create access and assign methods: automatically and manually. To create them automatically, check the "Access method" or "Assign method" (or both) checkboxes in the New Property dialog when you create a new property or in the Edit Property/Method dialog for an existing property. To create them manually, create methods with the appropriate name. However you create them, VFP automatically puts code into these methods to accept the necessary parameters and get or set the value of the property (unless, of course, you create the method in a non-visual DEFINE CLASS definition). Here's a typical access method:

```
return This.Property
```

Here's a typical assign method:

```
lparameters vNewVal
This.Property = vNewVal
```

You can override the behavior of these methods to do whatever you want. As a silly example, instantiate the Fun class in TEST.VCX and display the value of the Count property several times (SILLY.PRG does this for you); look at the code in the Count_Access method to see where the values come from.

This_Access

In addition to property access and assign methods, VFP 6 supports a new global class method called This_Access. This method, which you have to create manually with the New Method dialog, is executed whenever you access a member (property, method, or contained object) of the object. For example, changing the Height of a form causes the form's This_Access method to fire (if there is one) before the Height is changed.

This_Access is passed the name of the member and must return an object reference, normally This (you could return a reference to a different object, but you'd only do that under very specific circumstances, since you're essentially redirecting access to a different object). Note that unlike specific property access and assign methods, This_Access has no control over the value of the member being accessed, since the method isn't passed the new value to assign and doesn't return the value for the member.

The sample form THISACCESS.SCX provides a simple example that merely demonstrates This_Access being fired by displaying the name of the member being accessed, using the following code:

```
lparameters cMember
? cMember
return This
```

The Init method of this form has the following code:

```
This.Caption = 'This_Access Test'
&& displays "caption"
This.SFTextBox1.Value = 'Test'
&& displays "sftextbox1"
```

When you run this form, you won't just see "caption" and "sftextbox1" displayed; you will also see the names of certain form methods and properties as they're accessed in the class this form is based on (SFForm in SFCTRLS.VCX).

Because This_Access gets fired a lot, you'll definitely want to minimize its use and the amount of work it has to do.

Notes Regarding Access and Assign Methods

While access and assign methods can be very powerful, there are some issues you need to be aware of when you consider using them.

- Access and assign methods can be created for native VFP properties. For example, say you want to save and restore the position of a form. You could create assign methods for the Top and Left properties that set a custom logical property called lMoved to .T. In the Destroy method of the form, you could check the lMoved property and if it's .T., save the Top and Left properties somewhere (such as the Windows Registry) and restore them in the form's Init method. However, currently the assign methods for certain properties don't fire when the value is changed in certain ways. For example, you may be aware that refreshing a pageframe only refreshes the active page; you have to manually refresh other pages when they're selected, usually by putting code in the Activate method of each page. A great use of an assign method would be to refresh the page being selected in an ActivePage_Assign method of the pageframe; this would eliminate the need to manually insert the same code into every page.

```
lparameters tnNewPage
This.ActivePage = tnNewPage
This.Pages[tnNewPage].Refresh()
```

Unfortunately, while this code fires if you change the value of ActivePage programmatically or using the keyboard (tabbing to a new page), it doesn't fire when the value is changed by clicking on a page with the mouse. Similarly, the Partition_Assign method of a grid doesn't fire when you drag the splitter bar. Hopefully, Microsoft will fix these shortcomings.

- The access and assign methods of an array property are passed the subscript(s) for the array element being accessed. In the case of assign, the subscripts follow the parameter for the new value.

- Access and assign methods aren't executed at design time; for example, changing the value of a property in the Property Sheet or in a builder doesn't fire the code in the assign method of the property. This makes sense if you think about it – the Init and other methods of the object don't fire either because the object isn't really executing. These methods also don't fire when properties are displayed or changed in the Watch or Locals windows.

- On my test system, accessing a property without an access method took 0.000118 seconds and 0.000429 with one. Assigning a property value without an assign method took 0.000127 seconds and 0.000540 with one. In both cases, this is about 4 times slower on a relative comparison but still very fast absolutely. So, while you'll want to use access and assign methods when they're useful, you'll want to minimize accesses or assigns in a loop. Instead, store the property value (which fires the access method) to a variable and use that variable for repetitive accesses when you know the value won't change. The same holds true with assign: use a variable when, for example, you're accumulating values in a loop, and then store the variable's value to the property (which fires the assign method) when the loop is done.

- One way to improve the performance of an access or assign method is to check the current value of the property and avoid executing any code if nothing needs to be changed. For example, in an assign method, you might compare the new value to the current one (after using VARTYPE() to ensure the new value is the correct data type, of course) and not do anything else if they're the same.

- Under some conditions, you might want to set a property's value to itself in the Init method of the class to ensure the assign method will fire and perform any required action.

- You can't create access and assign methods for the native properties of an ActiveX control (well, you can create them, but they're never fired). You can, however, create and use access and assign methods for the VFP OLEContainer the ActiveX control is hosted in.

- There's currently a bug with This_Access: it isn't called if a member is accessed in a WITH block. Here's an example; you'll get an error on the assignment to Property3 and Property4.

```
oObject = createobject('Test')
oObject.Property1 = 'test'
oObject.Property2 = 'test'
with oObject
    .Property3 = 'test'
    .Property4 = 'test'
endwith

define class Test as Custom
    function This_Access
        lparameters tcMember
        if not pemstatus(This, tcMember, 5)
            This.AddProperty(tcMember)
        endif not pemstatus(This, tcMember, 5)
        return This
    endfunc
enddefine
```

Uses for Access and Assign Methods

The uses I've outlined below are by no means exhaustive. This is a catalog of uses I've come up with or seen others use. In other words, it's a set of design patterns for access and assign methods.

Creating "Action" Properties

Some native VFP properties fire an action when their values are changed. For example, changing the Height or Width properties of a form causes the form to be resized. An assign method allows you to do the same thing for a custom property or native property that doesn't have this behavior. This is a very powerful use of assign methods: the gyrations you used to have to go through to ensure something happens when conditions change is eliminated because the proper code is executed automatically. In other words, you still have to write the code to perform the action, but can eliminate the code that manages when the action occurs, which can be complex, kludgy, and almost always more error prone than the actual action code.

For example, in one class I created, I wanted to know if the object was resized or moved, either visually or programmatically. Before VFP 6, I'd have to store the original values into custom properties (likely in the Init of the class) and then when I need to know if the object was resized or moved, compare the current values against the saved ones. Now, I simply created assign methods for the Height, Width, Top, and Left properties, and had these methods set custom lSized and lMoved properties to .T. There's less code and fewer custom properties required.

Here's another example: suppose you want all objects on a form to perform some action, such as resizing themselves when the form is resized. In order to pass a message to each object, telling it to perform the action, you have to spin through the Controls collection of the form. However, that's not enough to get to all controls; you also have drill down through containers (and containers they contain) to reach each object. Instead, if each object has a property (such as lResize) that has an assign method to perform the desired action, you'd simply have to use code like the following to trigger this action for all objects:

```
This.SetAll('lResize', .T.)
```

Here are some examples from the FoxPro Foundation Classes (FFC):

- _Output (in _REPORTS.VCX): the assign methods of several properties (such as cAlias and cDestination) call the SetDestinations() method to ensure that changes to these properties are applied to dependent properties.

- _HyperlinkCommandButton (_HYPERLINK.VCX): lVisited_Assign sets the color of the button to the "visited" color, while nVisitedForeColor_Assign sets the color of the button to the new color if lVisited is .T. cTarget_Assign sets the ToolTipText for the button to the specified URL.

- _FindDialog (_TABLE.VCX): lAdvanced_Assign makes those controls used in "advanced" mode visible or not.

Property Validation

VFP has built-in validity checking for native object properties, both for data type (for example, you can't set Height to a string) and range (a form's BorderStyle can be 0 to 3). While you can test custom properties for validity in methods that use them (often done with ASSERT statements, since invalid property values are usually a programmer error), wouldn't it be nice to enforce this immediately when an invalid value is stored like VFP does? With assign methods, you can. Here's an example:

```
lparameters tuNewValue
assert vartype(tuNewValue) = 'N' ;
    message 'Invalid data type'
assert between(tuNewValue, 1, 5) ;
    message 'Invalid value'
This.Property = tuNewValue
```

This code ensures that the property is set to a numeric value between 1 and 5. In this case, ASSERT was used for testing, but you could also use IF or CASE statements and trigger an error using the ERROR command if the test fails.

Here are some examples from the FFC:

- _FileVersion (_UTILITY.VCX): cFileName has an access method (property validation is usually, but not always, done in the assign method) that uses GETFILE() to obtain a file name if the property is empty, non-character, or the file doesn't exist.

- _Output (_REPORT.VCX): cReport_Assign ensures the specified file exists, and cDisplayFontName_Assign ensures that the specified font exists in AFONT().

- _FilterExpr (_TABLE.VCX): cFilter's assign method strips carriage return, line feed, and tab characters out of the specified value (these may have gotten into the string when the user typed into an edit box, for example).

Creating Read-Only Properties

Many VFP objects have read-only properties. For example, the ControlCount property of container classes (such as forms) contains the number of controls in the container. You can't change the value of ControlCount directly; attempting to do so causes a "Property is read-only" error. Sometimes, it'd be nice to make custom properties read-only. For example, as we'll see later, a collection class needs a Count property, and you wouldn't want anything outside the class to be able to change its value directly.

In VFP 5, there was no way to do this directly with custom properties; you had to make the property protected and create a public method to return its value. In VFP 6, this is easily done: simply create an assign method for the property that fails to store the value to the property (and possibly also triggers error number 1743 using the ERROR command). The ROTest1 class in TEST.VCX shows this; to see it in action, do the following or run ROTEST1.PRG:

```
x = newobject('rotest1', 'test')
x.Count = 10
```

However, there's a complication: this property is now completely read-only; not even methods of the class can change the property's value. To see this, try to call the Test() method of the ROTest1 class, which simply tries to set the Count value to 5. So, the problem is: how to make it read-only to everything outside the class but read-write inside the class?

There are (at least) a couple of ways to handle this. One is to use a protected logical property that, when .T., indicates an internal method is trying to change the value, so the change is allowed. Since the property (called, for example, lInternal) is protected, it can only be set to .T. by internal methods. The Count_Assign method of the ROTest2 class (in TEST.VCX) permits the value of the property to be changed when lInternal is .T.:

```
lparameters tuNewValue
if This.lInternal
    This.Count = tuNewValue
else
    error 1743
endif This.lInternal
```

Here's the Test method of this class:

```
This.lInternal = .T.
This.Count = 5
This.lInternal = .F.
```

To test this, instantiate ROTest2, try to set Count to 5, then call the Test method (ROTEST2.PRG does this). Unlike the ROTest1 class, the Test method works, showing that the property is read-write to internal methods.

A drawback to this approach is having to set lInternal to .T. and back to .F. in each method needing to change the value of a read-only property. Another scheme, developed by Toni Feltman of F1 Technologies, doesn't require anything special in any method changing the value; instead, the assign method calls a custom CalledByThisClass method to determine if the code that tried to change the value came from a method of the class. Here's the code for Count_Assign in the ROTest3 class in TEST.VCX:

```
lparameters tuNewValue
if This.CalledFromThisClass()
    This.Count = tuNewValue
else
    error 1743
endif This.CalledFromThisClass()
```

Here’s the code for CalledFromThisClass:

```
local lnLevel, ;
    lcProgram, ;
    lcObject, ;
    lcThisName, ;
    loParent, ;
    llReturn
lnLevel    = program(-1)
lcProgram  = iif(lnLevel > 2, ;
    upper(program(lnLevel - 2)), '')
lcObject   = left(lcProgram, rat('.', lcProgram) - 1)
lcThisName = This.Name
loParent   = iif(type('This.Parent') = 'O', ;
    This.Parent, .NULL.)
do while vartype(loParent) = 'O'
    lcThisName = loParent.Name + '.' + lcThisName
    loParent   = iif(type('loParent.Parent') = 'O', ;
    loParent.Parent, .NULL.)
enddo while vartype(loParent) = 'O'
llReturn = upper(lcObject) == upper(lcThisName)
return llReturn
```

ROTest3 works just like ROTest2, but its Test method is just like that in ROTest1 (that is, it just assigns a value to Count), which failed.

Calculating Property Values When Required

Some properties contain calculated values. A simple example is an Area property, which is the product of Height and Width properties. Deciding when to calculate the value can be complicated, especially if the calculation is time-consuming. You could change the calculated value whenever any of the things the calculation depends on changes, but this may not be efficient if the dependent values change frequently and the calculated value isn’t needed right away. An example is a data entry form in which the user is entering several values that are involved in the calculation; you may not want to slow down the data entry by recalculating the value after every entry. However, not forcing a recalculation on each dependent value change means specifically calling a method to perform the calculation once it’s needed.

Fortunately, there’s a simple solution to this in VFP 6: have the calculation performed in the access method of the property. That way, you don’t have the overhead of the calculation until you actually need it, but the mechanism to calculate it is transparent (that is, you don’t have to call a method to force the calculation before getting the property value). An example of this is the Count property of the collection class we’ll see later. This property really just contains the number of non-empty rows in an array property, but rather than having to update it every time a row is added to or deleted from the array, we just calculate it when required using the following code for its access method:

```
local lnCount
with This
    lnCount = alen(.Item)
    lnCount = iif(lnCount = 1 and ;
        isnull(.Item[1]), 0, lnCount)
endwith
return lnCount
```

Notice something interesting here: the value of Count is never in fact changed; the access method just returns the proper value. There isn’t really a need to have a value in Count, since every time something wants its value, we calculate and return the value. As a result, this property also has an assign method that makes the property read-only by simply triggering error 1743 and not worrying about who tried to change the value.

Delayed Instantiation

Related to the previous use is delaying the instantiation of a member object until it’s actually needed. For example, a Word merge class may have an oWord property that references the Word application. Since instantiating Word can take some time and memory, why do it until absolutely necessary? If the user clicks on the Cancel button in a dialog using this class, Word won’t actually be used, so there’s no point in instantiating it in the Init of the class. Instead, the access method of oWord can test if the property contains .NULL., and if so, instantiate Word the first time it’s needed.

Another example of this pattern is the FFC _HyperlinkCommandButton (_HYPERLINK.VCX) class. Its oHyperlink property contains a reference to a Hyperlink object which does the actual work of hyperlinking the button to something, but if the user never clicks on the button, this object won’t be required. So, this object isn’t instantiated until the oHyperlink property is accessed for the first time.

Data Conversion

Some properties contain values that could be expressed differently on a different scale. For example, the BackColor property of an object expects a numeric value (although the value displayed in the Property Sheet is usually the parameters that would be passed to the RGB() function, which returns a numeric result). Using an assign method, you could allow it to accept string values representing the desired color. Here’s an example:

```
lparameters tuColor
local lnColor, ;
    lcColor
do case
    case vartype(tuColor) $ 'NFIBY' and tuColor >= 0
        This.BackColor = int(tuColor)
    case vartype(tuColor) = 'C'
        lcColor = upper(tuColor)
        do case
            case lcColor == 'RED'
                lnColor = rgb(255, 0, 0)
            case lcColor == 'GREEN'
                lnColor = rgb(0, 255, 0)
            case lcColor == 'BLUE'
                lnColor = rgb(0, 0, 255)
            otherwise
                lnColor = -1
                error 1560
        endcase
        if lnColor >= 0
            This.BackColor = lnColor
        endif lnColor >= 0
    otherwise
        error 1732
endcase
```

This code will accept a numeric value, “red”, “blue”, and “green” as valid values. COLORS.SCX provides an example of this.

Another example is conversion between units of measurement, such as between the metric and “medieval” systems. The assign method of a temperature value, for example, could check the setting of a units property (either “C” for Celsius or “F” for Fahrenheit) and automatically calculate the temperature for the other scale. TEMPERATURE.SCX provides a simple example of this. Other examples are conversions between lengths (such as meters and feet/yards), areas (acres and hectares), volumes (gallons and liters), and currencies (American and Canadian dollars, which will require a variable conversion factor).

Maintaining a Facade

Well-designed classes separate visual interface from implementation; this allows you to vary either without affecting the other, and to provide the implementation features in non-visual settings. An example is a “find” object that locates records based on the specified search criteria. This class might be used in lots of places besides a form, so putting the code into the Click method of a command button isn’t the best approach. Instead, create a non-visual find component with properties for the search criteria, and use this component whenever searching is needed. You might then build this object into another reusable component.

However, a complication arises: how do you address the criteria properties and search method of the find object? You wouldn’t want to use a reference like Container.oFind.cSearchString and Container.oFind.Search() because that means you have to know that oFind is the name of the find object within the container, and hard-coding names is a bad thing. Instead, use a Facade design pattern: create criteria properties and a search method of the container and use them instead; for example, Container.cSearchString and Container.Search(). This is a much better approach because it hides the find object within the container from the outside world (nothing that uses the container should know how the container implements the searching). However, this requires that before Container.Search() calls oFind.Search(), each of the criteria properties is written to the appropriate oFind property, and when the search is done, the properties of oFind (such as a success flag) are written to the appropriate container properties.

We can simplify this a lot using access and assign methods. Each container criteria property has an assign method that writes the specified value to the appropriate oFind property, and an access method that returns the value of the oFind property. Now, Container.Search() has nothing more than a call to oFind.Search().

The FFC Find Object (_TableFind in _TABLE.VCX) is an example of this pattern. The _FindButton class (also in _TABLE.VCX) is a visual component that uses _TableFind to do its work, and its various properties (such as cAlias, cFindString, and lFindAgain) have access and assign methods as I’ve described above.

Another example of supporting a facade is the FFC _GraphByRecord class (_UTILITY.VCX). This class is a wrapper for a graph object, and it has several properties that affect the way the graph appears, such as lAddLegend, lSeriesByRow, and nChartType. Each of these has an assign method that sets the appropriate graph property and refreshes the graph. This way, you can easily have a visual control such as a checkbox that has the appropriate property of the _GraphByRecord object as its ControlSource and no coding will be required to make it work.

Supporting a Decorator

The Decorator design pattern provides a way to augment the behavior of an object without subclassing the object. Decorators are useful when, for example, you don’t have the source code for the object you want to augment so you can’t subclass it. The way a decorator works is that it sits between the object being decorated and anything needing the services of that object, and presents itself as if it were the object. Any message passed to the decorator is either passed straight through to the object, wrapped in other behavior (pre- and/or post-behavior) and passed to the object, or completely handled by the decorator.

Here’s an example where a decorator might be used. Your client has a data handling class with a Query method that creates a cursor from some data. They ask you to make the data available to VB and Excel. You realize that this can easily be done by subclassing the class, making it an OLEPublic class (so it’s a COM object that can be called from VB and Excel), and overriding the Query method to convert the resulting cursor into an ADO recordset that you then return. However, the client informs you that they don’t have the source code for the class (perhaps it’s part of a library they purchased). So, instead of a subclass, you create a decorator class.

One of the complexities of creating a decorator is exposing the same interface as the decorated object. After all, if the decorator presents itself as if it were the decorated object, it needs to have the same public properties and methods as the decorated object. This can be a time-consuming chore, and furthermore, it can be difficult to expose properties of the decorated object if those properties can be changed by the object itself.

This_Access to the rescue! This method, which is called when any member of the object is accessed, can return an object reference to the decorated object rather than This to redirect access to the decorated object. Certain properties and methods can be processed by the decorator, however, by returning This instead. Here’s what this code might look like:

```
lparameters tcMember
if tcMember = 'query' or ;
    vartype(This.oDecorated) <> 'O' or ;
    not pemstatus(This.oDecorated, tcMember, 5)
    return This
else
    return This.oDecorated
endif tcMember = 'query'…
```

This code will run the Query method of the decorator but redirect every other message to the decorated object.

DECORATOR.PRG, kindly provided by Steven Black, demos how a decorator object can expose the properties and methods of a decorated object automatically.

Collections

A collection is a group of related things. In VFP, collections are everywhere: _SCREEN has a Forms collection, a form has a Controls collection, a pageframe has a Pages collection, and so on. Although they’re implemented differently in different places, collections generally have some things in common:

- They have a property that returns the number of items in the collection. In the case of the Forms collection, this property is called FormCount; for Controls, it's called ControlCount; for Pages, it's called PageCount. For ActiveX collections, it's frequently called Count, making it easier to deal with in generic code. Sometimes this property is read-write (such as PageCount), but usually it's read-only.

- They have a way to add and remove items from the collection. With native VFP collections, the way varies; instantiating a new form automatically adds it to the Forms collection, while new pages are created by incrementing the PageCount property. With ActiveX collections, there are usually Add or AddItem, Remove or RemoveItem, and Clear methods.

- They have a way to access items in the collection, usually by an index number. For the Forms collection, it's Forms[<index>]. For Pages, it's Pages[<index>]. ActiveX collections frequently have an Item method that can accept either an index number or the name of the desired item. Referencing an item in a collection by name is easier (and often requires less code) than by index number. Some VFP collections (like the Forms collection of _SCREEN) don't allow this, but some (like the Forms and Projects collections of _VFP) do.

I've been finding myself creating and using collections a lot lately. Often, these collections just consist of a class that contains an array of object references. Sometimes, the objects in a collection are very simple: they don't have any methods, but are just holders of properties. Other times, they're more complex objects, such as forms. Here are a few of the reasons I've come up with for using collections:

- Frequently, collections are replacements for arrays; I originally created my collection class to replace an array property with many columns. I found I was always forgetting which column certain information had to go into. It's a lot easier to understand and maintain code that simply sets or gets properties of objects in a collection than working with rows and columns.

- Since a collection is an object, it's easier to pass to a method than an array. For example, with an array, you have to use @ to ensure it's passed by reference, but you can't do that with a array property, so you end up copying the array property to a local array, passing the local array to the method, then (if the method changed the array) copying the array back to the array property. Oh, but don't forget to redimension the array property first or you might get an error. With a collection, you can omit several lines of ugly code. Passing arrays is even more complex when the object you're passing it to is a non-VFP COM object, which may not deal with arrays the same way VFP does.

- It can be easier to search for something in a collection than in an array. ASCAN searches all columns, not just the one you're interested in, so it may find false matches. Also, if the array contains references to objects, you won't be able to use ASCAN at all. With a collection, you can code for the exact search behavior you want in your collection class, then simply call the appropriate method.

- You can more easily protect the contents of a collection than you can an exposed array property.

Here are some of the places where collections might be used:

- A menu wrapper class: a menu is a collection of pads, each of which is a collection of bars (actually, pads reference popups, but the wrapper class could hide this complexity). An object oriented menu would be much easier to work with than VFP's current menu system, which has changed little since FoxPro 2.0.

- A custom DataEnvironment class: VFP's DataEnvironment consists of two collections: cursors and relations. A substitute for the native DataEnvironment could provide more functionality, such as methods for saving and reverting tables rather than putting this code into a form.

- An application object's forms collection: unlike the Forms collection of _SCREEN, which just has an object reference to each open form, the forms collection of an application object could contain more useful information, such as which toolbars they reference, whether they support multiple instances or not, whether they add their captions to the Window menu, etc.

- VFP Automation server classes: one of the things that makes working with Automation servers like Word and Excel easy is their rich object models, which usually consist of many collections of similar objects. Many VFP developers are developing similar object models for their tools or applications, allowing them to be used in many more ways than just a VFP user interface.

OK, so what do collections have to do with access and assign methods? As we'll see in a few moments, collections can be implemented much better in VFP 6 because of access and assign methods.

To make it easier to work with collections of objects, I created a couple of abstract classes intended to be subclassed into specific collection classes: SFCollection and SFCollectionOneClass. SFCollection (found in SFCOLLECTION.VCX) is a subclass of SFCustom, my Custom base class found in SFCTRLS.VCX. The VFP 5 version of SFCollection has a protected aItems array that contains references to the items in the collection, and an Item method that returns an object reference to the specified item. Why use a method to access items in the collection rather than just addressing the aItems array directly? Because I wanted to be able to reference items in the collection the same way ActiveX controls and newer VFP collections allow, either by index number or name. Here's an example:

```
oCollection.Item(5)
oCollection.Item('Customer')
```

VFP 5 doesn't allow an element in an array to be accessed by anything other than a numeric subscript, so we can't access aItems like this.

The VFP 6 version of SFCollection is also based on SFCustom, but it does things a little differently. Because we now have access and assign methods, we can use an array called Item that contains the items in the collection and access this array directly.

The access method of Item accepts either a numeric or character parameter. If the parameter is numeric, we assume it's the element number in the array. If it's character, we look for an item with that name.

The advantage of using this approach is that in the VFP 6 version, Item is a property not a method. This means you can use it to talk directly to an item in the collection. For example, you can use:

```
oCollection.Item[<index>].Property
```

In the VFP 5 version, you must use:

```
loObject = oCollection.Item(<index>)
loObject.Property
```

because Item is a method, not a property, in this version. The VFP 6 version acts more like native collections in VFP and ActiveX controls. Another advantage of access and assign methods: Count is a read-only property in the VFP 6 version thanks to an assign method. In the VFP 5 version, improper values can be stored to it.

Here's the code for the AddItem method:

```
lparameters tcClass, ;
    tcLibrary, ;
    tcName
local lnCount, ;
    loItem
with This
    .lInternal = .T.
    lnCount = .Count + 1
    dimension .Item[lnCount], .aNames[lnCount]
    loItem = newobject(tcClass, tcLibrary)
    .Item[lnCount] = loItem
    if vartype(tcName) = 'C' and not empty(tcName)
        .aNames[lnCount] = tcName
        if not ' ' $ tcName and isalpha(tcName)
            loItem.Name = tcName
        endif not ' ' $ tcName ...
    endif vartype(tcName) = 'C' ...
    .lInternal = .F.
endwith
return loItem
```

This method accepts three parameters: the name of the class to create the new item from, the library that class is defined in, and the name to assign to the item. AddItem creates an object of the specified class, stores the object in a new row in the Item array, and stores the specified name so it can be used later to find the item (in the protected aNames array). If the name is a valid VFP name, the name is also stored in the Name property of the new object. An object reference to the new item is then returned so the code calling this method can set properties or call methods of the new item. Notice that the Count property isn't incremented; it's calculated when required in its access method.

The RemoveItem method accepts the index or name of the item to remove as a parameter. If the specified item can be found (using the GetIndex method to find the element number of the item), the item is removed from the items and names arrays. As with AddItem, the value of the Count property isn't changed. Here's the code for this method:

```
lparameters tuIndex
local llReturn, ;
    lnIndex, ;
    lnCount
with This
    .lInternal = .T.
    lnIndex = .GetIndex(tuIndex)
    if lnIndex > 0
        adel(.Item, lnIndex)
        adel(.aNames, lnIndex)
        lnCount = .Count
        if lnCount = 0
            .InitArray()
        else
            dimension .Item[lnCount], .aNames[lnCount]
        endif lnCount = 0
        llReturn = .T.
    endif lnIndex > 0
    .lInternal = .F.
endwith
return llReturn
```

Here’s Item_Access, the access method for the Item array property:

```
lparameters tuIndex
local lnIndex, ;
    loReturn
with This
    lnIndex  = iif(.lInternal, tuIndex, ;
         .GetIndex(tuIndex))
    loReturn = iif(lnIndex = 0, .NULL., .Item[lnIndex])
endwith
return loReturn
```

This method accepts the index or name of the desired item as a parameter, and if that item exists, returns an object reference to it.

The GetIndex method is called from several other methods to convert a specified item index or name to the appropriate element number in the items array (or 0 if the item wasn’t found).

```
lparameters tuIndex
local lnCount, ;
    lnIndex, ;
    lcName, ;
    lnI
with This
    lnCount = .Count
    lnIndex = 0
    do case

* If the index was specified as a string, try to find
* a member object with that name.

        case vartype(tuIndex) = 'C'
            lcName = upper(alltrim(tuIndex))
            for lnI = 1 to lnCount
                if upper(.aNames[lnI]) == lcName
                    lnIndex = lnI
                    exit
                endif upper(.aNames[lnI]) == lcName
            next lnI

* If the index wasn't numeric, give an error.

        case not vartype(tuIndex) $ 'NIFBY'
            error cnERR_DATA_TYPE_MISMATCH

* If the index was numeric, give an error if it's
* out of range.

        otherwise
            lnIndex = int(tuIndex)
            if not between(lnIndex, 1, lnCount)
                error cnERR_INVALID_SUBSCRIPT
                lnIndex = 0
            endif not between(lnIndex, 1, lnCount)
    endcase
endwith
return lnIndex
```

Count_Access, the access method for the Count property, automatically calculates the value whenever Count is accessed (in fact, you'll see from this code that the value of Count is never changed; the access method just returns the proper value). Here's the code:

```
local lnCount
with This
    lnCount = alen(.Item)
    lnCount = iif(lnCount = 1 and isnull(.Item[1]), ;
        0, lnCount)
endwith
return lnCount
```

There are a few other methods in this class: Count_Assign ensures this property is read-only, Init calls the protected InitArray method to intialize the items array, and Destroy nukes all object references.

I created a simple sample to show how collections work. FIELDS.PRG opens the VFP TESTDATA database and creates a collection of fields in the CUSTOMER table. It omits the primary key field (CUST_ID) since we likely wouldn't want the user to see it. It also uses a nice caption for each field rather than the field name. These captions are displayed in a listbox using FIELDS.SCX. Selecting a caption in the listbox causes the appropriate field name to appear in the textbox on the form.

Using This_Access to Create Properties Dynamically

In addition to access and assign methods, VFP 6 added an AddProperty method to classes. AddProperty adds a property to an object at runtime rather than design time. Why would you want to do this? You need this capability when you cannot predict the interface of an object at design time. A parameter object is an example.

A parameter object is simply an object passed to a method as if it were a package of parameters. This mechanism allows you to change a long list of parameters ("was the caption the eighth parameter or the ninth?") into a single one: an object with the "parameters" stored in properties of the object. It also allows multiple return values from a method call, since the method can easily change the values of many properties of the parameter object and the caller can check the values of those properties.

However, there's one complication with using parameter objects: you either need a different class for the parameter object for each interface (one that has the properties both the calling and called methods expect) or you must use an array property of the parameter object (which simply transfers the complexity of a long list of parameters to multiple columns in an array). However, now that we have AddProperty, we can easily create an object of any class (even a VFP base class), add the properties we need to it, assign values to those properties, and it's ready for use.

This_Access can make this mechanism even better. Why bother manually creating properties; why not let the class create them the first time they're needed? I created a class called SFParameter (in SFCTRLS.VCX) that has the following code in its This_Access method:

```
lparameters tcMember
if not pemstatus(This, tcMember, 5)
    This.AddProperty(tcMember)
endif not pemstatus(This, tcMember, 5)
return This
```

This code automatically adds a property to the object if it doesn't exist the first time it's accessed. For example, although this class doesn't have a cCaption property, the following code works just fine:

```
oParameter = newobject('SFParameter', 'SFCtrls.vcx')
oParameter.cCaption = 'My Caption'
? oParameter.cCaption    && displays "My Caption"
```

That's because the statement assigning a value to the cCaption property first fires This_Access, which discovers that the property doesn't exist and creates it, after which the assignment takes place. All this in seemingly (from the point of view of the user of the class) one line of code.

There's just one complication: if the property to be created must be an array, This_Access can't create it because it doesn't receive any information that the property is an array (the only parameter is the name of the property). So, SFParameter also has a CreateArray method to specifically create an array property. Here's the code:

```
lparameters tcArray, ;
    tnRows, ;
    tnColumns
local lnRows
lnRows = iif(vartype(tnRows) = 'N', tnRows, 1)
if vartype(tnColumns) = 'N'
    This.AddProperty(tcArray + '[lnRows, tnColumns]')
else
    This.AddProperty(tcArray + '[lnRows]')
endif vartype(tnColumns) = 'N'
```

Summary

On the surface, the addition of access and assign methods to VFP 6 may not seem all that important. However, in this document I showed you how these methods can provide solutions to many different kinds of problems. Like anything else, they need to be used properly, but can help your development efforts in many ways.

Acknowledgements

I would like to acknowledge the following people who directly or indirectly helped with the information in this document: Steven Black, Toni Feltman, Dan Freeman, Christof Lange, Ken Levy, Lisa Slater Nicholls, and Jim Slater.

Copyright © 2000 Doug Hennig. All Rights Reserved

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
World Wide Web: www.stonefield.com

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He writes the monthly "Reusable Tools" column for FoxTalk, and is the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP).