Persistence without Perspiration

Doug Hennig

This month's article presents a set of classes that persist values. A common application of this technique is restoring a form's last size and position when it's opened.

Have you ever had a client ask you why the forms in your application always come up in the center of the screen (or worse, in the upper left corner) rather than where they were when the form was closed? Want to do something about it? This month, we'll look at classes that allow you to persist (a $5 word meaning save and restore) values, such as form size and position.

In this article, I use the term "item" (for want of a better generic term) to mean the thing we're persisting values for. An example of an item is the Top property of a form. The persistent class we'll look at in a moment allows you to specify which items the persistent object is supposed to manage (meaning that we'll save and restore the values for it, either automatically when the object is created and destroyed or manually by calling methods).

Before we get into the class, let's talk about a storage mechanism for persisted values. There are lots of places we can store values: in a table, in a text file (such as an INI file), in the Windows Registry, etc. However, you can probably see that while the actual code for reading and writing values varies with the storage mechanism, the overall logic of persisting values (looking up all the values of the items we're managing somewhere and restoring them, and doing the opposite for saving) is the same. So, we'll first look at a class that manages the abstract behavior of persistence.

SFPersistent

SFPersistent (located in SFPERSIST.VCX) is the base class of persistent objects. It's an abstract class and can't be used directly, because while it defines the interface for persistent objects, it doesn't do any persisting itself. SFPersistent is based on SFCustom, our Custom base class in SFCTRLS.VCX.

The Init method of SFPersistent calls the DefineItems method to specify which items we're supposed to manage (we'll talk about DefineItems in a moment). Then, if Init is passed .F. (or nothing) and the lRestoreOnInit property is .T. (the default), it calls the Restore method to restore the persisted values of the items we're managing. Here's the code for Init:

```
lparameters tlNoRestore
with This

* Call the method where the items to be managed is
* filled in.

    llReturn = .DefineItems()

* If we're supposed to, restore the items managed by
* the object.

    if llReturn and not tlNoRestore and .lRestoreOnInit
        .Restore()
    endif llReturn ...
endwith
return llReturn
```

The advantage of calling Restore from Init is that you can simply instantiate an SFPersistent subclass (one that specifies how to persist values) or drop it on a form and it'll restore any saved values for the items it manages automatically. However, there are times when you may not want this behavior, such as if the items it manages can't be restored automatically (perhaps they're properties of an object that doesn't exist yet). So, you can either pass .T. to the Init method or set the lRestoreOnInit property to .F. The reason for having two ways to do this is that you could drop an SFPersistent subclass on a form, in which case nothing is passed to Init, or you could instantiate it using CREATEOBJECT(), in which case you don't have a chance to set lRestoreOnInit until after the Init method fires.

Destroy does just one thing: if the lSaveOnDestroy property is .T. (which it is by default), it calls Save to store the values of the items we're managing.

DefineItems is an abstract method in SFPersistent; it's the place where you can specify what items a subclass or instance of SFPersistent will manage. This method should specify these items by calling the AddItem method, passing it the name that the item's value will be stored under in whatever storage mechanism we're using, the name of the item (often a property of a form or some other object), and optionally the data type of the item. If the data type isn't passed, AddItem will use TYPE() to figure it out. However, if TYPE() would fail (for example, you're managing properties of an object that hasn't been created yet), be sure to pass the data type as the third parameter. Here's an example of how AddItems is called; this code, taken from the DefineItems method of the SFPersistentForm class (which we'll discuss later), specifies that we'll manage the Left, Top, Height, and Width properties of the form the object is on.

```
with This
    .AddItem('Left',   'Thisform.Left')
    .AddItem('Top',    'Thisform.Top')
    .AddItem('Height', 'Thisform.Height')
    .AddItem('Width',  'Thisform.Width')
endwith
```

The Restore method restores the persisted values for one or all items; pass the name of an item to restore that item's value or nothing to restore values for all managed items. This method calls the protected RestoreOne method to restore the value of a single item. RestoreOne is an abstract method because the exact means of restoring an item's value will depend on the storage mechanism.

```
lparameters tcItem
local llReturn, ;
    lnI
with This

* Ensure that if the property was specified, it's a
* valid name, and that we have some properties defined.

    if pcount() = 1 and (vartype(tcItem) <> 'C' or ;
        empty(tcItem))
        error cnERR_ARGUMENT_INVALID
        return .F.
    endif pcount() = 1 ...
    if alen(.aItems, 2) = 0 or empty(.aItems[1, 1]) or ;
        empty(.aItems[1, 2])
        error 'Items were not defined'
        return .F.
    endif alen(.aItems, 2) = 0 ...

* If no item was specified, restore them all.

    llReturn = .T.
    if pcount() = 0
        for lnI = 1 to alen(.aItems, 1)
            llReturn = llReturn and .RestoreOne(lnI)
        next lnI

* If an item was specified, try to find it. If we can,
* restore it. If not, give an error.

    else
        lnI = ascan(.aItems, tcItem)
        if lnI > 0
            lnI = asubscript(.aItems, lnI, 1)
            .llReturn = RestoreOne(lnI)
        else
            error cnERR_PROPERTY_NOT_FOUND, tcItem
            llReturn = .F.
        endif lnI > 0
    endif empty(tcItem)
endwith
```

```
return llReturn
```

The Save method, which saves the value of one or all items, is very similar to Restore, except it calls the protected SaveOne method. Like RestoreOne, SaveOne is an abstract method.

SFPersistentRegistry

SFPersistentRegistry (also located in SFPERSIST.VCX) is a subclass of SFPersistent that persists item values in the Windows Registry. Rather than reading from and writing to the Registry directly, it uses the services of a class that knows how to do that: SFRegistry, a subclass of the FoxPro Foundation Classes (FFC) Registry class. I presented the SFRegistry class in the December 1998 issue of FoxTalk.

SFPersistentRegistry has an oRegistry property into which the Init method instantiates an SFRegistry object. You can pass the Init method the Registry key that'll be used for items the object manages; you can also set the cKey property to the desired key (as with turning off automatic restoration, there are two ways you can set the key since there are two ways you can instantiate an object). Use something like "Software\Your Company Name\Application Name\Object Name" for the key; by default, SFRegistry uses HKEY_CURRENT_USER as the top Registry node (which I've sometimes heard referred to as "hive"). The ReleaseMembers method of the class (called when the object is destroyed) sets oRegistry to .NULL. so object references are cleaned up.

The RestoreOne and SaveOne methods of SFPersistentRegistry are where the mechanism for persisting is defined. RestoreOne ensures that cKey has been filled in, then uses SFRegistry's GetRegKey method to read the value for the specified item from the Registry. If GetRegKey returns .NULL. (meaning an error occurred or, more likely, the value didn't exist in the Registry because it hadn't been saved yet), the item's value isn't changed. ConvertCharToType is used because SFRegistry can only store character values in the Registry, so we need to convert the character string read from the Registry back to the correct data type for the item. Also, the value isn't written to the item unless it's different than the item's current value; this prevents an automatic method (such as Resize when Top is changed) or an Assign method of the item from firing needlessly. Here's the code for RestoreOne:

```
lparameters tnItem
local lcItem, ;
    luDefault, ;
    luValue
with This

* Ensure we have a valid key.

    if empty(.cKey)
        error 'You must specify the cKey property.'
        return .F.
    endif empty(.cKey)

* Use the oRegistry object to get the value, convert it
* to the proper data type, and store it in the bound
* item.

    lcItem    = .aItems[tnItem, 2]
    luDefault = evaluate(lcItem)
    luValue   = nvl(.oRegistry.GetRegKey( ;
        .aItems[tnItem, 1], .cKey), luDefault)
    luValue   = .ConvertCharToType(luValue, ;
        .aItems[tnItem, 3])
    if not luDefault == luValue
        store luValue to (lcItem)
    endif not luDefault == luValue
endwith
```

SaveOne is almost identical to RestoreOne, except it uses SetRegKey to write the item's value to the Registry after using ConvertTypeToChar to convert it to a string.

To use SFPersistentRegistry, drop it on a form, set the cKey value to the Registry key where values should be stored (for example, "Software\Stonefield Systems Group Inc.\Test"), and put some code in the DefineItems methods to specify what items are to be persisted. You can also instantiate a subclass of

SFPersistentRegistry which has DefineItems filled in; in the case, it's likely you'll want to vary the key used from instance to instance, so pass the key as the first parameter.

SFPersistentForm

SFPersistentForm is a subclass of SFPersistentRegistry that specifically handles form size and position. It simply has the code in DefineItems I showed earlier to manage the Top, Left, Height, and Width properties of the form it's on.

SFPersistentForm is really easy to use: just drop it on a form and set cKey to the desired key. If you want to do this generically (that is, dropping it on a form class), you'll likely want to vary the key with the name of the application and form, so in that case, set the lRestoreOnInit property to .F. and in the Init of the form, set the cKey property and call the Restore method. Here's an example:

```
This.oPersist.cKey = 'Software\My Company\' + ;
    oApp.cAppName + '\' + This.Name
This.oPersist.Restore()
```

This code assumes an application object (oApp) has a cAppName property that contains the application name, and that the name of the form is part of the key.

TEST1.SCX is a very simple example of using SFPersistentForm. It has a few controls and an SFPersistentForm object with cKey set to "Software\Stonefield Systems Group Inc.\Test". Run this form, move and resize it, then close it. Run it again and you'll see that it opens with the same size and position it had when you closed it.

You might notice that as you resize the form, the controls on the form don't resize. In the same December 1998 column where I discussed SFRegistry, I presented a subclass of the FFC _Resizable class called SFResizable. This class resizes controls when a form is resized; you simply specify which objects should be resized and how by filling in properties of the object (see the article for details) and call the AdjustControls method in the form's Resize method. TEST2.SCX is a copy of TEST1, but has an SFResizable object added to it and a call to This.oResizer.AdjustControls in Resize. When you run this form, you'll see that it opens at the same size and position as TEST1 (since they use the same Registry key) but when you resize it, the controls adjust as necessary. However, when you close and rerun the form, it opens at the last saved size and position, but the controls don't. Why is that?

It turns out to be related to the order of instantiation. Since the SFPersistentForm object was added to the form first, its Init fires first and resizes the form to its former size. Then, the SFResizable Init fires, and it stores the form's Height and Width to "initial" properties. The problem, of course, is that the form isn't its original size anymore. So, the controls don't get adjusted correctly.

One solution to this problem would be to ensure that the SFResizable object is added to the form first, but that means the behavior of the form depends on the order of instantiation of objects, which should set off alarm bells for you. Instead, since SFResizable and SFPersistentForm need to work together at form instantiation, I created a subclass of SFResizable called SFPersistResizer (in SFCCTRLS.VCX) that adds an SFPersistentForm object to itself so the two can work together in the order they need to. I added lRestoreOnInit, lSaveOnDestroy, and cKey properties, and a Restore method that simply calls This.oPersist.Restore to the SFPersistResizer class; we don't have multiple inheritance in VFP, so I did this so the form can just talk to these properties and method of the SFPersistResizer object rather than having to address them as Thisform.oPersistResizer.oPersist.<property or method>. Thus, in addition to providing its own behavior, SFPersistResizer is a wrapper for SFPersistentForm.

The Init method of SFPersistResizer does its base behavior first (saving the form Height and Width), then adds an SFPersistentForm object and passes it the values of the cKey and lRestoreOnInit properties. Finally, if lRestoreOnInit is .T., it calls the Resize method of the form because Resize didn't fire automatically when the Init of the SFPersistentForm object adjusted the form size; Resize doesn't fire unless the form has finished instantiating. Resize will call the SFPersistResizer's AdjustControls method, which will adjust the controls. Thus, once the Init method of SFPersistResizer is done, the form and its controls have been properly adjusted.

The Destroy method of SFPersistResizer sets the SFPersistentForm object's lSaveOnDestroy property to its own property, so when that object gets destroyed, it'll save the form size and position if desired.

To see SFPersistResizer in action, run TEST3.SCX. This form is a copy of TEST2.SCX, but I removed the SFPersistentForm and SFResizable objects and replaced them with SFPersistResizer. When you run this form, you'll see that it exhibits the desired behavior: not only is the form size and position restored, so are those of the controls.

Conclusion

Although I didn't create one, you could easily subclass SFPersistent to use tables or INI files as the storage mechanism. Regardless of what mechanism you use, these classes make persisting values very easy to do.

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 is also 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). He can be reached at dhennig@stonefield.com.