June, 2004

Advisor Answers

Exploring an object's PEMs

VFP 8/7/6

Q: I have a complex form with a lot of custom properties. I'd like to give my users a way to start over if they need to. To do that, I have to be able to restore the values of those custom properties to what they were when the form opened. How can I do that?

–Sue Cunningham (via CompuServe.COM)

A: To do what you want, you need some method for saving the values when the forms open so you can restore them later on. There are a variety of places you can store the saved value. One possibility that works in all versions of VFP is to add an array property to the form that holds the names and values of your properties. Another possibility, new in VFP 8, is to create an object based on the Empty class and then add a property to it for each property value you want to save. No doubt you can think of several other ways to store the property values.

The big question then is how to find out what properties are there, so you can save them. While you could hard-code the list of values to save, this has the major disadvantage that you have to remember to update the code every time you change the form.

A better approach is to ask the form what properties it has and then retrieve their values. The key to doing this is the AMEMBERS() function. It lets you dig into an object (or a class) and get a list of its members, that is, its properties, events, methods and contained objects. As the "A" at the beginning of AMEMBERS() suggests, the list of members is stored in an array. In its simplest form, you pass the array and the object and a one-dimensional array is filled with a list of the object's properties. In this case, the function returns the number of properties.

For example:

```
oForm = CREATEOBJECT("Form")
nPropCount = AMEMBERS(aFormProps, oForm)
```

Because each new version of VFP tends to introduce some new properties, the array is different in each version. In VFP 8, the resulting array has 107 items, beginning with "ACTIVECONTROL" and ending with "ZOOMBOX". In VFP 7, the array has 101 items.

Prior to VFP 7, AMEMBERS() could work only with native VFP objects. In VFP 7 and later, it can also tell you about COM objects (like Automation servers and ActiveX controls). However, I'm going to address AMEMBERS() only as it applies to native VFP objects here.

AMEMBERS() can do a lot more than just list properties. It has an optional third parameter that determines what's included in the array. Table 1 shows the values you can pass for the third parameter.

Table 1. Changing AMEMBERS() output—The optional third parameter of AMEMBERS() determines the structure of the resulting array.

Third parameter	**Meaning**
0 or omitted	Created a one-dimensional array containing the object's properties.
1	Create a two-column array listing all of the object's properties, events, methods, and member objects. The first column contains the name of the member. The second column contains the type of member ("Property", "Event", "Method" or "Object").
2	Create a one-dimensional array listing the object's member objects only.
3	Creates a four-column array listing all of the object's properties, events, methods and member objects. Table 2 shows the meaning of each column.

The most informative output comes from passing 3 for the third parameter. In this case, the resulting array has four columns, as shown in Table 2.

Table 2. What's in a member?—When you pass 3 for the third parameter of AMEMBERS(), you get a four-column array.

Column	Meaning
1	The member's name
2	The member's type ("Property", "Event", "Method" or "Object")
3	For methods and events, the list of parameters. For properties and contained objects, the empty string.
4	The member's help string (as displayed in the property sheet).

At this point, you may be wondering how AMEMBERS() is going to help in your situation where you want a list of just your custom properties. The secret is an optional fourth parameter that lets you limit AMEMBERS() results to just those members that meet particular requirements.

The fourth parameter, added in VFP 7, is called cFlags. It's a character string that specifies which members you want to include in the output, based on various characteristics. Table 3 shows the values you can include in cFlags, and breaks them into categories.

Table 3. AMEMBERS() flag values—Pass one or more of these flags as the fourth parameter to AMEMBERS() to restrict output to members with specified characteristics.

Character	Category	Meaning
"G"	Visibility	Include public members
"H"	Visibility	Include hidden members
"P"	Visibility	Include protected members
"N"	Origin	Include native members (those that are part of the base class)

Character	Category	Meaning
"U"	Origin	Include user-defined members, those defined anywhere in the inheritance hierarchy
"B"	Inheritance	Include members defined at this level
"I"	Inheritance	Include inherited members
"C"	Change	Include members whose value has been changed at some level in the inheritance hierarchy
"R"	Change	Include members that are read-only
"+"	Management	Combine the other flags with "and" rather than "or"
"#"	Management	Add a column to the output for flags

By default, flags are combined with "or." That means that passing "HP" for the fourth parameter includes all members that are either hidden or protected, but omits public members. Passing "UC" includes members that are either user-defined or changed. If you want to combine flags with "and" instead, include the "+" flag in cFlags. So to get a list of members that are both changed and user-defined, pass "UC+".

The "#" is also special. When you include it in the cFlags parameter, the resulting array has an additional column, which contains the flags for each member (using the same characters as in Table 3). Note that you can't pass "#" by itself. You have to include at least one other flag—fortunately, the "+" flag is sufficient, so if you want to list all members with their flags, call the function like this:

```
nMembCount = AMEMBERS(aMembList, oForm, 3, "#+")
```

So how does AMEMBERS() solve your problem? You can use it to get a list of all the custom properties of your form, and store their values. Put code like this in the form's Init method to create a property to hold the saved values and then fill it with the properties and their values. This version uses the technique of creating an Empty object and adding properties to it:

```
LOCAL oSaveValues, nPropCount, aProps[1], nProp

oSaveValues = CREATEOBJECT("Empty")

nPropCount = AMEMBERS(aProps, This, 0, "IU+")

FOR nProp = 1 TO nPropCount
   * Exclude any property with an access or assign method
   IF NOT (PEMSTATUS(This, aProps[nProp] + "_access",5) ;
      OR PEMSTATUS(This, aProps[nProp] + "_assign", 5))
       uValue = GETPEM(This, aProps[nProp])
       * Exclude contained and referenced objects
       IF TYPE("uValue") <> "O"
         ADDPROPERTY(oSaveValues, aProps[nProp], ;
                     GETPEM(This, aProps[nProp]))
      ENDIF
   ENDIF
ENDFOR

This.AddProperty("oSaveValues", m.oSaveValues)
```

Note that the code does not save values for two types of properties. The first is those that have Access and/or Assign methods. The problem there is that you don't know what else might happen when you store or restore such a property.

The second group omitted are properties storing references to other objects. There are a couple of issues here. The first is that the referenced object might no longer exist when you try to restore it. The second is that saving the object reference may not be good enough—you may actually want to restore the values of the contained object's properties, as well. If that's the case, you need to write more comprehensive code that stores all the properties of the contained object and at restore time, can create the appropriate object and set its properties.

Once you have this code, you can add code to your custom StartOver method to restore these values to the form properties:

```
LOCAL nPropCount, aProps[1], nProp

nPropCount = AMEMBERS(aProps, This.oSaveValues)

FOR nProp = 1 TO nPropCount
   cProp = "This." + aProps[nProp]

   STORE GETPEM(This.oSaveValues, aProps[nProp]) ;
      TO (cProp)
ENDFOR
```

This month's Professional Resource CD contains a class library (AMEMDEMO.VCX) with a two-level form class hierarchy. The top-level class, frmBase, has a single custom property, while the concrete subclass, frmConcrete, has two custom properties. The code to save and restore the properties is in the frmConcrete class. There's also a form (Restorable.SCX) based on the frmConcrete class. It has textboxes bound to the three custom properties. You can change the values in the textboxes, then click the Restore button to restore their original values.

I should also point out that the problem is a little more complicated if you want to restore native properties. In that case, you have to deal with the fact that some of them are read-only, so you can't restore them. The way to handle this is to add the "#" flag to your AMEMBERS() call. Then, check the flags in your processing loop and save values only for those properties that are not read-only.

–Tamar