September, 2005

Advisor Answers

Getting a list of objects in memory

VFP 9/8

Q: I want to fill an array with a list of all the non-visual objects created using CreateObject() during runtime. Then, I can scan through the list to get all the objects' properties and their values using AMEMBERS(). The only command I can find is List Objects but it cannot output to an array.

A: I was surprised when I started to explore your question to find that there's no function to do what you want. VFP does include AINSTANCE(), but it returns only objects created from a particular class; it also returns the names of the objects, not object references.

As you note, LIST OBJECTS gives you a list of those objects that are stored in variables, but not in the form you want. The best solution appears to be redirecting the results of LIST OBJECTS to a file and then parsing the file. Fortunately, the file is pretty regular and VFP includes a great collection of parsing tools.

A few warnings before we look at the necessary code. First, only objects stored in variables are included in the listing; objects stored in properties of other objects aren't included. Second, the output of LIST OBJECTS is localized; while it's in English in the VFP IDE, at runtime, it appears using the language of whichever VFP resource file you specify. If you distribute your application using a language other than English, you'll need to adapt the code accordingly. Finally, LIST OBJECTS fires the access method for every property, as well as firing the This_Access method for every PEM of every object.

Let's start by taking a look at the object generated by LIST OBJECTS. I used the following code to put a few objects in memory for testing:

```
o=CREATEOBJECT("custom")
o1 = CREATEOBJECT("session")
o2 = NEWOBJECT("_aboutbox",HOME()+"ffc\_dialogs")
o2.addproperty("aOneD[3]")
o2.aOneD[1] = "Something"
o2.aOneD[2] = 37
o2.aOneD[3] = .t.
```

Figure 1 shows a portion of the listing created for these objects.

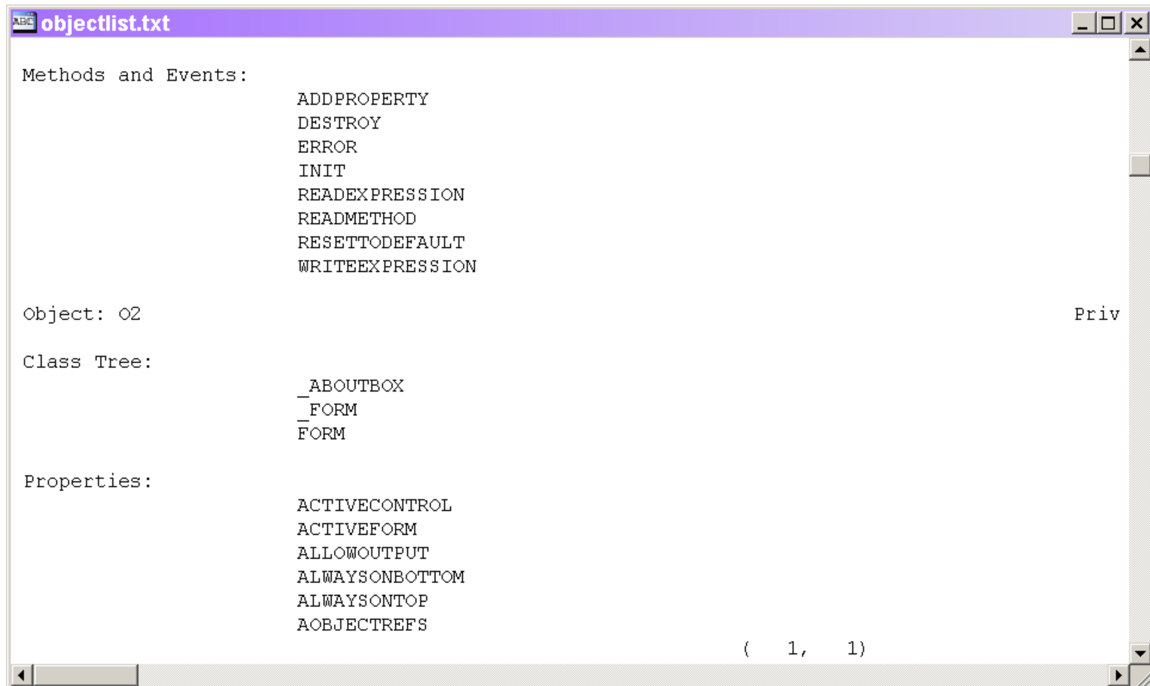

Figure 1. LIST OBJECTS includes the name, class hierarchy, properties and values, and methods and events for each object in memory.

Not only does LIST OBJECTS provide the names and classes of the objects in memory, it also includes the names and values of each object's properties, and the names of the methods and events for each object. This means that you don't need to follow this call with a call to AMEMBERS() for each object; you can just parse all of the information directly from the listing.

I wrote a function called AObjects.PRG (included on this month's Professional Resource CD) modeled after the various "A" functions included in VFP. It accepts an array as a parameter and fills the array with information about the objects in memory. The function is shown in Listing 1.

Listing 1. This function accepts an array and fills it with information about objects in memory. The array has four columns containing the name, the class, the properties and the methods of each object.

```
LPARAMETERS aObjectArray[1]
LOCAL cFileName, cContent, aObj[1], nObj, nObjectCount
LOCAL cProps, nPropCount, nProp, aProps[1], aPropInfo[1]
LOCAL cOldFont, nOldFontSize

SET LIBRARY TO HOME()+"FoxTools"

cFileName = FORCEPATH("ObjectList.TXT",SYS(2023))
```

```
cOldFont = _Screen.FontName
nOldFontSize = _Screen.FontSize
_Screen.FontName = "Arial"
_Screen.FontSize = 14
LIST OBJECTS TO FILE (cFileName) NOCONSOLE
_Screen.FontName = cOldFont
_Screen.FontSize = nOldFontSize
cContent = FILETOSTR(cFileName)

* Break up into one item per object
nObjCount = ALINES(aObj, cContent, 0, "Object:")
DIMENSION aObjectArray[nObjCount - 1, 4]

* Skip first item--contains leading blanks
FOR nObj = 2 TO nObjCount

   aObjectArray[ nObj-1, 1] = ;
      ALLTRIM(GETWORDNUM(aObj[ nObj ], 1))
   aObjectArray[ nObj-1, 2] = ;
      ALLTRIM(GETWORDNUM(aObj[ nObj ], 4))
   aObjectArray[ nObj-1, 3] = CREATEOBJECT("Collection")
   aObjectArray[ nObj-1, 4] = CREATEOBJECT("Collection")

   * Parse out property info
   cProps = STREXTRACT( aObj[ nObj], ;
                        "Properties:", ;
                        "Methods and Events:")
   IF "Member Objects:"$cProps
      cProps = LEFT(cProps, ;
                    AT("Member Objects:", cProps) -1)
   ENDIF

   * Break into lines
   nPropCount = ALINES(aProps, cProps)
   * Break each line into components
   FOR nProp = 2 TO nPropCount-1
      nCols = ALINES(aPropInfo, ;
                     REDUCE(aProps[ nProp ]), 0, " ")
      IF "("$aPropInfo[1]
         * Array element

         * Reparse the data
         cDimensions = STREXTRACT(aProps[nProp], ;
                                  "(", ")")
         nCommaPos = AT(",", cDimensions)
         IF nCommaPos = 0
            nRow = VAL(cDimensions)
            nCol = 0
            cType = aPropInfo[3]
            uValue = aPropInfo[4]
         ELSE
            nRow = VAL(LEFT(cDimensions, nCommaPos))
            nCol = VAL(SUBSTR(cDimensions, nCommaPos-1))
            cType = aPropInfo[4]
            uValue = aPropInfo[5]
```

```
      ENDIF

      oElement = CREATEOBJECT("empty")
      ADDPROPERTY(oElement, "Row", nRow)
      ADDPROPERTY(oElement, "Col", nCol)
      ADDPROPERTY(oElement, "Type", cType)
      ADDPROPERTY(oElement, "Value", uValue)

      oProp.oData.Add(oElement)

   ELSE
      oProp=CREATEOBJECT("Empty")
      ADDPROPERTY(oProp, "Name", aPropInfo[1])
      DO CASE
      CASE (UPPER(aPropInfo[2]) == "(NONE)")
         ADDPROPERTY(oProp, "Type", .null.)
         ADDPROPERTY(oProp, "Value", .null.)
      CASE UPPER(aPropInfo[2]) == "A"
         ADDPROPERTY(oProp, "Type", "A")
         ADDPROPERTY(oProp, "Value", .null.)
         ADDPROPERTY(oProp, "oData", ;
                     CREATEOBJECT("Collection"))
      OTHERWISE
         ADDPROPERTY(oProp, "Type", aPropInfo[2])
         ADDPROPERTY(oProp, "Value", aPropInfo[3])
      ENDCASE
      aObjectArray[nObj-1,3].Add(oProp)
   ENDIF
ENDFOR

* Now parse out methods
cMethods = STREXTRACT(aObj[nObj], ;
                      "Methods and Events:")
nMethodCount = ALINES(aMethods, cMethods)
FOR nMethod = 2 TO nMethodCount
   aObjectArray[nObj-1, 4].Add(aMethods[nMethod])
ENDFOR

ENDFOR

RETURN nObjCount-1
```

In testing AObjects() in an EXE, I ran into errors, which led me to discover that the output of LIST OBJECTS is sensitive to the current font settings for the main VFP window. So the function saves the current settings, then specifies a font and size that result in the desired format, then resets the font.

AObjects() uses a variety of string processing functions, most of which I've previously discussed in this column and all of which can be found in the VFP help file.

AObjects() uses one string function that's not built into VFP directly. Reduce() comes from the FoxTools library. It lets you compress a series of spaces into a single space. (Actually, it can do much more. See the August, 2004 ADVISOR Answers column.)

In addition to lots of string processing, the array filled by AObjects() uses a capability added in VFP 8, collections. Since the number of properties and methods may be different for each object, the third and fourth columns of the array contain collections. Column 3 has a collection of properties while column 4 has a collection of methods.

The approaches taken for the two columns vary. For methods, the method names are added directly to the collection as scalar values.

For properties, we want to grab the name, the type and the value. This is complicated by the fact that a property can be an array, in which case it has multiple values. (While a property can be a collection, LIST OBJECTS doesn't show the members of the collection.) To hold the property information, AObjects() creates a new object for each property. The object is based on the Empty base class; Name, Type, and Value properties are added. For arrays, a fourth property is added, oData, which is a collection that holds information about each element of the array.

In a function of less than 100 lines, AObjects() demonstrates the power and agility of VFP for processing data in whatever form it comes.

–Tamar