Collections: Managing Information the Object-Oriented Way

Tamar E. Granor
Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

When the collection base class was added to VFP 8, it gave us the opportunity to make our applications look more like those in other object-oriented languages. Collections let us manage groups of related objects through a straightforward interface and avoid the peculiarities of working with arrays in VFP.

In this session, we'll cover the basics of using collections, the reasons why they're better than arrays, and show how they improve object models. We'll also consider the weaknesses of collections in VFP and talk about workarounds.

Every procedural programming language I used before encountering FoxBase+ offered arrays as a way to hold an ordered collection of information. So when I started learning FoxBase+, its arrays made sense to me. In fact, the very first article I ever published in a Fox journal was about arrays. (It was in the June, 1990 issue of *FoxTalk*, and titled "Sorting and Searching in Arrays.")

Fox's arrays were different than those I'd used before in a couple of ways. First, most of the languages I'd worked with offered arrays of unlimited dimensions. FoxBase+ handled only one-dimensional and two-dimensional arrays; this is still true in VFP. Although this initially felt like a significant limit, I learned to work around it.

Second, the languages I'd used earlier were strongly typed, so every element of an array had to contain data of the same type. Fox is weakly typed and doesn't have this requirement. This made arrays very handy for storing copies of records, as well as other record-type information (for example, the file information returned by ADIR()).

Over the years, the functionality for working with arrays improved a lot. A group of functions was added long ago to make it easier to manipulate arrays; they include ASCAN() and ASORT() to search and sort (making my original *FoxTalk* article obsolete), ACOPY() to copy all or part of an array, and AINS() and ADEL() to add and remove data in the middle of an array.

In addition, VFP has acquired many functions that retrieve some information and store it in an array. For example, AFIELDS() puts the list of fields for a table into an array, APRINTERS() fills an array with the list of available printers, and AMEMBERS() retrieves the list of properties, events and methods for an object and stores that in an array. There are also commands to move table data directly into and out of arrays. (For detailed information about working with arrays, and the array functions, see http://tomorrowssolutionsllc.com/ConferenceSessions/You%20Need%20Arrays.pdf.)

With all these capabilities, arrays have been a valuable member of the VFP arsenal. However, the addition of a collection base class in VFP 8 gave us an alternative way to handle some groups of data. In particular, collections are a much more natural way than arrays to manage groups of related objects.

What is a collection?

A collection is a container for zero or more items. While the items can be scalar pieces of data (such as a string or a number), more often collections are used to contain a group of objects.

In VFP (and most other languages), a collection has a Count property that tells you how many items are in the collection and an Item method that provides access to the individual

members. In general, collection members can be accessed by either their position or by their key, a unique identifier assigned to each member.

Collections are naturally unordered, though VFP provides an ordered way to access their members. However, as members of the collection are added and removed, the position of an item can change.

Unlike VFP's arrays, collections handle the possibility of zero items with no problems. In that case, the collection's Count is 0.

COM Collections in VFP

Although the Collection base class was added in VFP 8, VFP has had tools for working with collections for much longer. Most Automation servers have lots of collections to represent the objects they deal with. For example, Microsoft Word has a Documents collection; its Document object has a Paragraphs collection. Microsoft Excel offers a Workbooks collection; each Workbook has a Worksheets collection, containing the individual sheets in the workbook. It's rare to encounter an Automation server that doesn't include at least a few collections in its object model.

VFP has been able to access and work with collections from Automation servers since VFP 3. VFP 5 added the FOR EACH loop construct (see "Looping through collections" later in this document) to make traversing a collection easier.

In addition to the collections from other Automation servers, VFP has several collections that belong to its own automation server. For example, the Projects collection was added in VFP 5 to provide access to all open projects. Project objects contain a Files collection with one member for each file in the project. The little program in **Listing 1** traverses the Files collection of a specified project; it let me quickly estimate the effort involved in making certain kinds of changes across a project I inherited.

Listing 1. The Projects and Files collections in VFP let us operate easily on projects. I used this code to quickly categorize the forms in a project I inherited, so I could estimate the effort in making certain changes.

```
LPARAMETERS cProject
MODIFY PROJECT (cProject) NOWAIT
oProject = _VFP.ActiveProject

DIMENSION aFormTypes[4]
aFormTypes=0

FOR EACH oFile IN oProject.Files
   IF oFile.Type = "K"
      MODIFY FORM (oFile.Name)
      cType = INPUTBOX( ;
         "1 for data entry, 2 for reporting, 3 for message, 4 for other")
      nType = val(cType)
      aFormTypes[nType] = aFormTypes[nType] + 1
   ENDIF
ENDFOR
```

```
?"Data entry forms = ", aFormTypes[1]
?"Reporting forms = ", aFormTypes[2]
?"Message forms = ", aFormTypes[3]
?"Other forms = ", aFormTypes[4]
```

The container classes in VFP (Form, Page, Grid, etc.) have an Objects property that points to a collection containing all the contained objects. Like Projects and Files, Objects is a COM collection, not an object native to VFP.

Collections of our own

It wasn't until VFP 8 that we got the ability to create our own collections. As noted above, they have a Count property and an Item method. They also include Add and Remove methods that let you add items to the collection and remove them from the collection. Not surprisingly, when you call Add to add an item, Count goes up. When you call Remove to remove an item, Count goes down.

VFP's collections also allow you to specify a *key* for each item. That is, you can associate a unique identifier with each member of a collection. Once you do so, you can access that item using its key.

To create a collection, you use CreateObject() as you would for any other class. Then, to add members, you call the collection's Add method, passing the item to add and, optionally, the key for that item. **Listing 2** shows code (MakeStatesCollection.PRG in the downloads for this session) to create a collection of strings representing the names of the first five US states alphabetically. The state's postal abbreviation is used as the key for the item. Each item can be accessed either by its position (index*)* in the list or by its key, as the last two lines show.

Listing 2. This code fragment creates a collection and adds the names of the first five US states. The postal abbreviation for each state is used as its key.

```
LOCAL oStates

oStates = CREATEOBJECT("Collection")
oStates.Add("Alabama", "AL")
oStates.Add("Alaska", "AK")
oStates.Add("Arizona", "AZ")
oStates.Add("Arkansas", "AR")
oStates.Add("California", "CA")

? oStates.Item[3]
? oStates.Item["AZ"]
```

In fact, in most cases, you can omit the Item keyword, as well, and just specify the index or key right after the collection name, as in **Listing 3**.

Listing 3. The Item keyword is usually optional.

```
? oStates[4]
```

```
? oStates["AR"]
```

Although collections can be used for scalar data as above, where they really shine is in working with objects. A collection can hold a group of related objects and provide easy access to them. We see this with the built-in collections like the Objects collection of container classes, and it's just as useful for your own objects.

For example, rather than just adding the state names to a collection, we might add state objects that contain the name, the abbreviation, the order in which the state joined the union, and the population at the 2020 census. We can still set the abbreviation as the key. (In an application, I'd probably create a State class, and offer a mechanism for creating and populating the objects. For this example, I'll do it by brute force, building the state objects on the fly.) **Listing 4** (MakeStatesCollectionObjects.PRG in the materials for this session) shows the code to create and populate two state objects and add them to a collection. As with scalars, you can access a member of a collection either by its index or by its key, and the Item keyword is optional. When a member of a collection is an object, you can then access its individual properties (and methods).

Listing 4. Collections are particularly useful for holding groups of objects.

```
LOCAL oStates, oState

oStates = CREATEOBJECT("Collection")

* Create and add a state
oState = CREATEOBJECT("Empty")
ADDPROPERTY(oState, "cName", "Alabama")
ADDPROPERTY(oState, "cAbbrev", "AL")
ADDPROPERTY(oState, "nOrder", 22)
ADDPROPERTY(oState, "nPopulation", 5024279)

oStates.Add(oState, oState.cAbbrev)

oState = CREATEOBJECT("Empty")
ADDPROPERTY(oState, "cName", "Alaska")
ADDPROPERTY(oState, "cAbbrev", "AK")
ADDPROPERTY(oState, "nOrder", 49)
ADDPROPERTY(oState, "nPopulation", 733391)

oStates.Add(oState, oState.cAbbrev)

* Now use it
? oStates[2].cName
? oStates["AK"].nOrder
```

Why collections?

When looking at the example in **Listing 4**, you may wonder why you should use a collection rather than an array. There are several reasons.

Access items by key

We've already seen the first reason, the ability to access members of a collection by their key instead of by their position. This can make code both more concise (since you don't have to go looking for the right item) and clearer (since the key in the code tells you exactly which item you're talking to).

Deal with no items

I also mentioned a second reason earlier. Collections handle the state of being empty very naturally. VFP's arrays can't be empty; they always have at least one element. So testing whether you have data requires either maintaining a separate counter variable or having a way to test whether the first element of the array is valid. **Listing 5** shows the test for an empty collection, and the more complex test for an empty array. In this example, the array test is that there's only one element and that element is empty. In other situations, you might need to test for a particular value or a particular data type.

Listing 5. Handling an empty collection is much easier than handling an empty array.

```
* To check whether a collection is empty, just look at its Count.
IF oCollection.Count = 0
   * Collection is empty. Act accordingly.
ENDIF

* To check whether an array is empty, you have to know what constitutes an empty
* element. The exact test varies with the way the array is being used. In this case,
* we consider the array empty if there's only one element and it's empty. Sometimes,
* you need to test for a specific value or type.
IF ALEN(aArray) = 1 AND ;
   EMPTY(aArray[1])
   * The array is empty. Act accordingly.
ENDIF
```

Add and remove items

Adding and removing items from a collection is easier than with arrays as well. To add an item to an array, you have to resize the array with DIMENSION, create a space where you want it with AINS(), and insert the data. With a collection, you just use the Add method. **Listing 6** shows the code tp insert a new item in the middle of a list using a collection and then using an array.

Listing 6. Adding a row to an array is a three-step process.

```
* Adding an item to a collection is easy. Here, we add five items.
* Keys are assigned to provide an ordering, so the last element added
* goes into the third position in key order.
oCollection.Add("A", "1")
oCollection.Add("B", "2")
oCollection.Add("C", "4")
oCollection.Add("D", "5")
oCollection.Add("Inserted", "3")
```

```
* Adding an item to an array takes more effort, especially if you
* want it to be other than the last item. Here, we create an
* array with four elements, then insert a new element into the
* third position.
LOCAL aArray[4]

* Populate it
aArray[1] = "A"
aArray[2] = "B"
aArray[3] = "C"
aArray[4] = "D"

* Add a row in the third position
LOCAL nRows
nRows = ALEN(aArray, 1)
DIMENSION aArray[m.nRows + 1]
AINS(aArray, 3)

aArray[3] = "Inserted"
```

Removing is similar. With an array, you have to use ADEL() to move data around, and then DIMENSION to resize the array. For a collection, you just call Remove.

While all those steps aren't too onerous for a one-dimensional array, with a two-dimensional array, it's more complex (particularly, if you want to add or remove a column). With a collection, whether it contains scalar items (as above) or objects, the mechanism is the same.

Pass as parameters

Passing arrays as parameters is a pain in VFP. You have to pass them by reference, which for function and method calls means prefixing the array name with "@", as in **Listing 7**. Because of this, array properties (that is, arrays that are properties of objects) can't be passed directly to other routines; you have to copy them to local arrays first, as in **Listing 8**. Then, if the other routine may have changed the array contents and you want to keep those changes, you have to copy back to the object property.

Listing 7. Arrays must be passed by reference.

```
LOCAL aMyArray[3], lResult
lResult = MyFunction(@aMyArray)
```

Listing 8. To pass an array property, you have to copy it to a local variable first.

```
* Assume oObject has an array property called aMyArray.
LOCAL aLocalArray[1], lResult

* Copy array property to local variable before passing it
ACOPY(oObject.aMyArray, aLocalArray)

lResult = MyFunction(@aLocalArray)
```

```
* Copy local array back to array property to preserve changes
ACOPY(aLocalArray, oObject.aMyArray)
```

Collections are easy to pass as parameters. You simply pass them like any other variable, as in **Listing 9**. Because the collection points to its actual contents, changes to the collection members in the routine are visible in the calling routine. You only need to pass the collection by reference when the collection itself may change in the routine, that is, when the routine might recreate the collection or destroy it.

Listing 9. Collections can be passed like any other variable.

```
* Assume oCollection is a collection
LOCAL lResult

lResult = MyFunction(oCollection)
```

Build hierarchies

If you need to deal with hierarchical objects, collections offer a clear benefit. You can walk down a hierarchy without having to use intermediate variables. **Listing 10** (included as MakeCountryCollectionNoClasses.PRG in the downloads for this session) shows the construction of a collection of countries. Each country contains two collections, a scalar list of languages, and a collection of state objects. (Once again, this is not the way I would build these objects in an application. See "Subclass the Collection class" later in this paper for an example showing roughly the same hierarchy built with classes.)

Listing 10. When you're working with hierarchical objects, collections make drilling down much easier than with arrays.

```
LOCAL oCountries, oCountry
LOCAL oStateProv, oStatesProvs

oCountries = CREATEOBJECT("Collection")

* Add the USA
oCountry = CREATEOBJECT("Empty")
ADDPROPERTY(oCountry, "cName", "United States of America")
ADDPROPERTY(oCountry, "cContinent", "North America")
ADDPROPERTY(oCountry, "oLanguages", CREATEOBJECT("Collection"))
oCountry.oLanguages.Add("English")

* Create a collection of its states/provinces
oStatesProvs = CREATEOBJECT("Collection")

oStateProv = CREATEOBJECT("Empty")
ADDPROPERTY(oStateProv, "cName", "Alabama")
ADDPROPERTY(oStateProv, "cAbbrev", "AL")

oStatesProvs.Add(m.oStateProv, m.oStateProv.cAbbrev)

oStateProv = CREATEOBJECT("Empty")
ADDPROPERTY(oStateProv, "cName", "Alaska")
ADDPROPERTY(oStateProv, "cAbbrev", "AK")
```

```
oStatesProvs.Add(m.oStateProv, m.oStateProv.cAbbrev)

* Etc. for the others

* Add the collection of states to the
* country object
ADDPROPERTY(oCountry, "oStatesProvs", m.oStatesProvs)

* Now add this country to the collection
oCountries.Add(m.oCountry, oCountry.cName)

* Add Canada
oCountry = CREATEOBJECT("Empty")
ADDPROPERTY(oCountry, "cName", "Canada")
ADDPROPERTY(oCountry, "cContinent", "North America")
ADDPROPERTY(oCountry, "oLanguages", CREATEOBJECT("Collection"))
oCountry.oLanguages.Add("English")
oCountry.oLanguages.Add("French")

* Create a collection of its states/provinces
oStatesProvs = CREATEOBJECT("Collection")

oStateProv = CREATEOBJECT("Empty")
ADDPROPERTY(oStateProv, "cName", "Ontario")
ADDPROPERTY(oStateProv, "cAbbrev", "ON")

oStatesProvs.Add(m.oStateProv, m.oStateProv.cAbbrev)

oStateProv = CREATEOBJECT("Empty")
ADDPROPERTY(oStateProv, "cName", "Quebec")
ADDPROPERTY(oStateProv, "cAbbrev", "PQ")

oStatesProvs.Add(m.oStateProv, m.oStateProv.cAbbrev)

* Etc. for the others

* Add the collection of provinces to the
* country object
ADDPROPERTY(oCountry, "oStatesProvs", m.oStatesProvs)

* Now add this country to the collection
oCountries.Add(m.oCountry, oCountry.cName)

* Now dig in and get some information
? oCountries["Canada"].oLanguages[1]
? oCountries["Canada"].oStatesProvs[2].cName
? oCountries["Canada"].oLanguages.Count
```

I use collections extensively in creating business objects for my applications. For that purpose, the ease of constructing and addressing hierarchies is critical. See "Collections for business objects" later in this paper for more on the subject.

Provide COM access

Finally, collections make sense if there's any chance that the code you're writing might need to be used from outside the application. That is, if you might end up creating a COM server, using collections will make it easier for the developers calling on your code, since it will behave like other COM server applications.

Working with collections

As the examples above show, working with collections is simple. However, in most cases, defining classes for both the collection and the items within is a better choice than building them on the fly as in the earlier examples.

Subclass the Collection class

As with other VFP classes, your first step in using collections should be to create your own "base" collection class, that is, a first-level subclass of Collection. I've typically added only two things to that subclass: a way to clean up object references as the collection is destroyed, and a method to make it easy to find an item based on its key. My “base” collection class is in CollectionBase.PRG in the downloads for this session.

The first issue is that when one object points to another and the second object points back to the first, neither object can be destroyed. Instead, they sit in memory, inaccessible, until you issue CLEAR ALL.

To prevent such *dangling references*, you can make sure to null all backward pointers before attempting to destroy the objects. I add a method called CleanUpReferences to my base collection class to help with that process. It contains the code shown in **Listing 11**, which visits every member of the collection and tells it to clean up its own backward references. The Destroy method of the class calls This.CleanUpReferences() to kick off the process.

Listing 11. Add this method to your "base" collection class as CleanUpReferences to help avoid dangling references.

```
LOCAL oItem
FOR EACH oItem IN This FOXOBJECT
   IF PEMSTATUS(oItem, "CleanUpReferences", 5)
      oItem.CleanUpReferences()
   ENDIF
ENDFOR
```

The second item, a method to look up a collection member by its key, is discussed in “Find collection items,” a little later in this document.

I typically subclass this "base" collection class to create collection classes for specific purposes. Most often, the items contained in the collection are also drawn from a specific class, generally a subclass of Custom. My "base" subclass of Custom, cusBase, also includes a CleanUpReferences method; it's abstract there. In subclasses, I add the necessary code to clean up specific references in that class.

Returning to the examples shown earlier in this paper, if I needed a collection of countries and states, I'd use several subclasses of collection: one for countries, one for states/provinces, and one for counties. I'd also create three subclasses of Custom: one to represent a country, one to represent a state or province, and one to represent a county. (In an application, I'd probably also change oLanguages from a collection of scalar items to a collection of objects, but I haven't done so for this example.) Using classes allows us to hide some of the details.

Listing 12 shows a subclass of my cusBase class used to represent an individual state or province. The Init method optionally populates the scalar properties; it also creates the empty collection of counties. The AddCounty method wraps up the steps to add a county and returns an object reference to the new county object, in case you want to do any more work with it. (The cusCountry class that represents an individual country has an analogous method for adding a state or province.) This class has code in CleanUpReferences to make sure that the two object references are cleared before the class is destroyed. (The version of this class in the downloads for this session contains some other methods, which are discussed later in this paper.)

Listing 12. Rather than creating objects and collections on the fly, in an application, it's best to define classes and build collections from them. Here's a custom StateProv class to use rather than the code in **Listing 10**.

```
DEFINE CLASS cusStateProv AS cusBase

cName = ""
cAbbrev = ""
oCountry = .null.
oCounties = .null.

PROCEDURE Init(cName, cAbbrev, oCountry)

IF PCOUNT() > 0 AND VARTYPE(m.cName) = "C"
   This.cName = m.cName
ENDIF

IF PCOUNT() > 1 AND VARTYPE(m.cAbbrev) = "C"
   This.cAbbrev = m.cAbbrev
ENDIF

IF PCOUNT() > 2 AND VARTYPE(m.oCountry) = "O"
   This.oCountry = m.oCountry
ENDIF

* Prepare oCounties to hold a collection
This.oCounties = CREATEOBJECT("colCounties")

RETURN

PROCEDURE CleanUpReferences

This.oCountry = .null.
This.oCounties.CleanUpReferences()
This.oCounties = .null.
```

```
RETURN

PROCEDURE AddCounty(cName)
* Add a county to this state or province

LOCAL oCounty

oCounty = CREATEOBJECT("cusCounty", m.cName)

IF VARTYPE(m.oCounty) = "O"
   This.oCounties.Add(m.oCounty)
ENDIF

RETURN m.oCounty

ENDDEFINE
```

Jumping farther up in the hierarchy, **Listing 13** shows the custom colCountries class, a subclass of collection that holds the whole group of countries. The custom AddCountry method is a wrapper for the native Add method. It accepts either a country object (that is, an object of class cusCountry) or the name and continent of a country, and adds the appropriate object to the collection, using its name as the key.

Listing 13. This class is used to hold the collection of countries. The custom AddCountry method wraps the details of creating a new country object and adding it to the collection.

```
DEFINE CLASS colCountries AS colBase

PROCEDURE AddCountry(uCountryOrName, cContinent)
* Add a country to the collection, using its name as the key.
* First parameter can be either country object or name of country

LOCAL oCountry, cName, cKey

DO CASE
CASE PCOUNT() < 1 OR NOT INLIST(VARTYPE(m.uCountryOrName), "O", "C")
   ERROR 11
   RETURN .F.

CASE VARTYPE(m.uCountryOrName) = "O"
   oCountry = m.uCountryOrName

CASE VARTYPE(m.uCountryOrName) = "C"
   cName= m.uCountryOrName
   oCountry = CREATEOBJECT("cusCountry", m.cName, m.cContinent)
ENDCASE


cKey = oCountry.cName
This.Add(m.oCountry, m.cKey)

RETURN m.oCountry
```

```
ENDDEFINE
```

One of the benefits of using classes with methods to handle much of the construction is that the code to build the collection becomes much simpler. **Listing 14** shows the main program for this version of the example.

Listing 14. When the classes you're using include the necessary properties and have methods to handle construction, the code to populate the collection is much simpler.

```
LOCAL oCountries

LOCAL oCountry, oStateProv

oCountries = CREATEOBJECT("colCountries")

* Add the US
oCountry = oCountries.AddCountry("United States of America", "North America")
oCountry.oLanguages.Add("English")

oStateProv = oCountry.AddStateProv("Alabama", "AL")
oStateProv.AddCounty("Lowndes")
oStateProv.AddCounty("Duval")
* Etc. for remaining counties

oStateProv = oCountry.AddStateProv("Alaska", "AK")
oStateProv.AddCounty("Bethel")
* Etc. for remaining counties

* Etc. for remaining states

* Add Canada
oCountry = oCountries.AddCountry("Canada", "North America")
oCountry.oLanguages.Add("English")
oCountry.oLanguages.Add("French")

oStateProv = oCountry.AddStateProv("Ontario", "ON")
oStateProv = oCountry.AddStateProv("Quebec", "PQ")
* Etc. for remaining provinces

RETURN
```

The complete code to recreate the earlier examples with classes is included with the downloads for this session as MakeCountryCollectionClasses.PRG. (In fact, this program creates a more extensive hierarchy than the earlier examples.)

Looping through collections

Sometimes, you need to traverse an entire collection. You have two options, a regular FOR loop and a FOR EACH loop. With the regular FOR loop, you can use the collection's Count property to set the limit. FOR EACH lets you go through a collection without counting. You're guaranteed to touch each item along the way. The order of traversal is based on the collection's KeySort property. With the default setting of 0, you go in ascending index order.

The other settings let you go in descending index order, and in ascending or descending key order.

The biggest advantage of FOR EACH is that each time through the loop, you're handed a reference to a collection member. When the items in a collection are objects, that's an object reference. With a counted FOR loop, you have to retrieve the item explicitly.

Listing 15 shows a simple FOR EACH loop that displays the name of each country in the collection created by **Listing 14**. **Listing 16** shows the equivalent counted FOR loop.

Listing 15. FOR EACH lets you walk through a collection, handing you a reference to each item.

```
LOCAL oCountry
FOR EACH oCountry IN oCountries
   ? oCountry.Name
ENDFOR
```

Listing 16. You can do the same thing with a counted FOR loop, but it takes a little more code.

```
LOCAL nCountry
FOR nCountry = 1 TO oCountries.Count
  ? oCountries[m.nCountry].Name
ENDFOR
```

When FOR EACH was added in VFP 5, there were no native collections. So it was designed to address COM objects. As a result, each object it creates is a COM object. After native collections were added in VFP 8, it became apparent that turning native objects into COM objects this way caused a number of problems. For example, when you use AMEMBERS() on such an object, you get different results than for the corresponding native object.

VFP 9 added the FOXOBJECT keyword, which tells FOR EACH to create native objects rather than COM objects. **Listing 17** (FoxobjectAndCom.PRG in the downloads for this session) demonstrates the problem with AMEMBERS(). FOR EACH without FOXOBJECT can also block access to some properties and methods.

Listing 17. You get different results in a FOR EACH loop with and without FOXOBJECT.

```
* Demonstrate the difference in FOR EACH with and without FOXOBJECT

* Build a collection
LOCAL oColl, oItem, nMembers, aMemberList[1]

oColl = CREATEOBJECT("Collection")

oColl.Add(CREATEOBJECT("cusItem", "First", 1))
oColl.Add(CREATEOBJECT("cusItem", "Second", 2))
oColl.Add(CREATEOBJECT("cusItem", "Third", 3))

* Loop without FOXOBJECT
FOR EACH oItem IN m.oColl
   nMembers = AMEMBERS(aMemberList, m.oItem, 3)
```

```
   ? "Member count for item " + oItem.cName + " = " + TRANSFORM(m.nMembers)
ENDFOR

* Loop with FOXOBJECT
FOR EACH oItem IN m.oColl FOXOBJECT
   nMembers = AMEMBERS(aMemberList, m.oItem, 3)
   ? "Member count for item " + oItem.cName + " = " + TRANSFORM(m.nMembers)
ENDFOR

RETURN

DEFINE CLASS cusItem AS Custom

cName = ''
nOrder = 0

PROCEDURE Init(cName, nOrder)

This.cName = m.cName
This.nOrder = m.nOrder

RETURN


ENDDEFINE
```

FOR EACH with FOXOBJECT is also much faster than without. In my tests, including FOXOBJECT makes the loop about an order of magnitude faster.

Unfortunately, FOR EACH with FOXOBJECT has one weakness. It ignores the KeySort value of the collection; the loop always proceeds in ascending index order. ForEachAndKeySort.PRG in the session materials demonstrates the problem.

There is one case where you must use a regular FOR loop rather than FOR EACH. That's when you're removing items from the collection as you go. In that case, you need to use a FOR loop that's counting backwards from the end of the collection. Otherwise, you won't visit every member of the collection. **Listing 18** (DeletingInALoop.PRG in the session materials) shows the problem and the solution.

Listing 18. To delete items from a collection based on their contents, use a regular FOR loop going backward.

```
* When deleting items, you need to be careful.

LOCAL oColl, oItem

PopulateCollection(@oColl)

* Using FOR EACH while deleting skips items
* Delete items whose names begin with "F"
FOR EACH oItem IN m.oColl FOXOBJECT
   IF UPPER(LEFT(oItem.cName, 1)) = "F"
      oColl.Remove(TRANSFORM(oItem.nOrder))
   ENDIF
```

```
ENDFOR

* What's left?
? " After deleting in a FOR EACH loop, the collection contains"
ShowItems(m.oColl)

* Now clear the collection and start over
PopulateCollection(@oColl)

* Using a regular FOR loop, going backward
LOCAL nItem

* Delete items whose names begin with "F"
FOR nItem = oColl.Count TO 1 STEP -1
   oItem = oColl[m.nItem]
   IF UPPER(LEFT(oItem.cName, 1)) = "F"
      oColl.Remove(TRANSFORM(oItem.nOrder))
   ENDIF
ENDFOR

? " After deleting in a backward FOR loop, the collection contains"
ShowItems(m.oColl)


RETURN

PROCEDURE PopulateCollection(oColl)

oColl = CREATEOBJECT("Collection")

oColl.Add(CREATEOBJECT("cusItem", "First", 1), "1")
oColl.Add(CREATEOBJECT("cusItem", "Second", 2), "2")
oColl.Add(CREATEOBJECT("cusItem", "Third", 3), "3")
oColl.Add(CREATEOBJECT("cusItem", "Fourth", 4), "4")
oColl.Add(CREATEOBJECT("cusItem", "Fifth", 5), "5")
oColl.Add(CREATEOBJECT("cusItem", "Sixth", 6), "6")

RETURN

PROCEDURE ShowItems(oColl)

FOR EACH oitem IN m.oColl FOXOBJECT
   ?oItem.cName
ENDFOR

RETURN


DEFINE CLASS cusItem AS Custom

cName = ''
nOrder = 0

PROCEDURE Init(cName, nOrder)
```

```
This.cName = m.cName
This.nOrder = m.nOrder

RETURN

ENDDEFINE
```

Find collection items

While looping through all items in a collection is a common operation, you also need ways to get to particular items. As I said earlier, giving an item a key offers direct access to that item. However, if you attempt to access an item that doesn't exist, you get an error message. (It's error 2061, "Index or expression does not match an existing member of the collection.") There are a couple of ways to avoid that problem.

The GetKey method gives you an easy way to switch between the index and the key of any element. If you pass a numeric value, it returns the key for the item with that index; if you pass a string, it returns the index for the item with that key. **Listing 19** demonstrates, using the collection of countries constructed in **Listing 14**.

Listing 19. The collection class's GetKey method switches between indexes and keys.

```
?oCountries.GetKey("Canada") && returns 2
?oCountries.GetKey(1)
  && returns "United States of America"
```

If you have the key for an item and you want to make sure the item exists, you can wrap the access in TRY-CATCH, as in **Listing 20**.

Listing 20. TRY-CATCH offers another way to avoid errors when addressing collection members by their keys.

```
TRY
   oFrance = oCountries.Item["France"]
CATCH
   oFrance = .null.
ENDTRY
```

Since I find that I need to look items up by their key in pretty much every collection I create, I've added a method called GetByKey to my collection base class. The code is shown in **Listing 21**.

Listing 21. This method, called GetByKey, is in my collection base class to provide a safe, easy way to look items up by their keys.

```
PROCEDURE GetByKey(cKey)
* Find a member of this collection using its key

LOCAL oReturn

TRY
   oReturn = This.Item[m.cKey]
```

```
CATCH
   oReturn = .null.
ENDTRY

RETURN m.oReturn
```

You can retrieve a particular country from the collection created earlier with code like that in **Listing 22**. In this case, oCanada would point to the cusCountry object for Canada, while oFrance would be null.

Listing 22. You can use the GetCountry method to retrieve the object for a particular country.

```
oCanada = oCountries.GetByKey("Canada")
oFrance = oCountries.GetByKey("France")
```

To find a particular item in a collection based on something other than the key, you generally have to use brute force. That usually involves looping through the collection until you find the item of interest.

For example, in the oStatesProvs collection for a country in the earlier example, states and provinces use their abbreviation as the key. If you need to look up a state or province based on its full name, you need a method (in colStatesProvs) like the one in **Listing 23**.

Listing 23. To search for an item in a collection based on something other than its key, you need brute force.

```
PROCEDURE GetStateProvByName(cName)

LOCAL oStateProv, lFound

FOR EACH oStateProv IN This FOXOBJECT
   IF UPPER(oStateProv.cName) == UPPER(m.cName)
      lFound = .T.
      EXIT
   ENDIF
ENDFOR

IF NOT m.lFound
   oStateProv = .null.
ENDIF

RETURN m.oStateProv
```

Using collections

With some idea of how to work with collections, let's look now at where collections are useful. The place I use collections most is in building business objects, but I've found lots of other uses for them.

Collections for business objects

A business object is an object that represents some real-world entity in an application. Business objects have properties that represent the characteristics of the entity and

methods to manage the entity and operate on it. Just as databases often represent hierarchical structures, it's not uncommon for business objects to form hierarchies, and collections can play a role in modeling those hierarchies.

The most typical use for collections in a business object hierarchy is to handle cases where an object can have zero or more of some other object. Add a property to the "parent" object to hold a collection of "child" objects.

In an application that needs to manage information about countries, their states and provinces, the counties within those states or provinces, and so forth, the classes developed earlier in this paper (**Listing 12**, **Listing 13** and **Listing 14**) would form a good foundation for the necessary business objects.

Some years ago, to demonstrate the use of business objects in VFP, I developed a Sudoku puzzle application. That application uses four business objects. One class (bizCell) represents a single cell in a Sudoku puzzle; another class (bizGame) represents the game as a whole. These two classes are connected using a pair of collection classes. First, a class I called cbzGroup represents one row, one column or one block in the puzzle. It's a collection of bizCell objects; in the standard Sudoku game, cbzGroup has nine members. Another class, cbzSetOfGroups, represents all the rows or all the columns or all the blocks in a puzzle. It's a collection of cbzGroup objects; in the standard game, it also has nine members. My paper on business objects, using this example, is available at http://tomorrowssolutionsllc.com/ConferenceSessions/Getting%20Your%20Head%20Around%20Business%20Objects.pdf.

Collections as managers

Most applications need *manager* objects that handle various kinds of things and the interactions among them. For example, most VFP frameworks have a Form Manager, which keeps track of which forms are open, and handles tasks like closing all forms on the way out of the application.

Collections are a great choice for manager objects. They let you hold the items that need to be managed and add methods to perform the various management tasks.

In one client application, requests are submitted to another application and processed when the response is received. The application needs to keep track of the requests that have been submitted for which it's awaiting responses. Responses include status information; the application needs to store that information temporarily in order to report success or failure to the user.

To handle these needs, I subclassed Collection, and added methods to add a request, set the status of a request, clear the list of requests, and answer various questions about the list (how many requests are we waiting for? which requests are we waiting for? was a particular request successful? and so forth). **Listing 24** shows a simplified version of this class; it's included in the session materials as WaitingRequests.COM. In my client's application, the application object has a property that points to an instance of this class.

Listing 24. This collection subclass manages a list of requests in an application.

```
DEFINE CLASS colWaitingRequests AS colBase

PROCEDURE AddToWaitingRequests(iRequestID)
* Add a request to the list of requests waiting
* to be processed.

LOCAL oRequest, nIndex

oRequest = CREATEOBJECT("Empty")
ADDPROPERTY(oRequest, "iRequestID", m.iRequestID)
ADDPROPERTY(oRequest, "nResultType", 0)

TRY
   THIS.ADD(m.oRequest, TRANSFORM(m.iRequestID))
CATCH
   oRequest = .null.
ENDTRY

RETURN m.oRequest
ENDPROC

PROCEDURE SetWaitingRequestResult(iRequestID, nResultType)
* Set the result type for a waiting request

LOCAL lSuccess, oRequest

oRequest = This.GetByKey(TRANSFORM(m.iRequestID))

IF ISNULL(m.oRequest)
   lSuccess = .F.
ELSE
   oRequest.nResultType = m.nResultType
   lSuccess = .T.
ENDIF
RETURN m.lSuccess
ENDPROC

PROCEDURE RemoveFromWaitingRequests(iRequestID)
* Remove a request from the list of requests waiting
* to be processed.

LOCAL nIndex

nIndex = This.GetByKey(TRANSFORM(m.iRequestID))
IF m.nIndex > 0
   THIS.REMOVE(m.nIndex)
ENDIF

RETURN
ENDPROC

PROCEDURE CountWaitingRequests
* Return the number of requests waiting for response.
```

```
LOCAL nCount

nCount = 0
FOR EACH oRequest IN This FOXOBJECT
   IF oRequest.nResultType = 0
      nCount = m.nCount + 1
   ENDIF
ENDFOR

RETURN m.nCount
ENDPROC

PROCEDURE AnyWaitingRequestsFailed
* Return a flag to indicate whether any of the waiting
* requests failed.

LOCAL lReturn

lReturn = .F.
FOR EACH oRequest IN This FOXOBJECT
   IF oRequest.nResultType < 0
      lReturn = .T.
      EXIT
   ENDIF
ENDFOR

RETURN m.lReturn
ENDPROC

PROCEDURE AllRequestsProcessed
* Return a flag to indicate whether there are any
* requests waiting to be processed.

LOCAL lReturn

lReturn = .T.

FOR EACH oRequest IN This FOXOBJECT
   IF oRequest.nResultType = 0
      lReturn = .F.
      EXIT
   ENDIF
ENDFOR

RETURN m.lReturn
ENDPROC

PROCEDURE ClearWaitingRequests
* Empty the list of nodes discovered

THIS.REMOVE(-1)

RETURN
ENDPROC
```

```
PROCEDURE IsSuccessful(iRequestID)
* Return a logical indicator for the specified request.

LOCAL lSuccess, oRequest

oRequest = This.GetByKey(TRANSFORM(m.iRequestID))

IF ISNULL(m.oRequest)
   lSuccess = .F.
ELSE
   lSuccess = (oRequest.nResultType = 1)
ENDIF

RETURN m.lSuccess
ENDPROC

ENDDEFINE
```

Collections for passing parameters and returning data

In VFP (as in most programming languages), the list of parameters for a procedure, function or method is fixed. While you can omit some of the parameters, you can't add them when you need to pass more than one instance of something. Return values are controlled even more strictly. A VFP function or method can return a single value.

Collections give you the opportunity to pass multiple items into a routine or return multiple items from one. That is, you can pass a collection as a parameter, and you can return a collection.

For example, in the countries/states example, we might want to get our hands on all countries where a particular language is spoken. The method in **Listing 25** (which is in colCountries) returns a collection containing the countries whose oLanguages collection includes the specified language. You can call this method as in **Listing 26**.

Listing 25. The method, added to colCountries, lets you retrieve a collection of all countries where a particular language is spoken.

```
PROCEDURE GetCountriesThatSpeak(cLanguage)

LOCAL oReturn, oCountry, cCountryLanguage

oReturn = CREATEOBJECT("colCountries")

FOR EACH oCountry IN This FOXOBJECT
   FOR EACH cCountryLanguage IN oCountry.oLanguages
      IF UPPER(m.cLanguage) == UPPER(m.cCountryLanguage)
         * Add this country to the collection to return
         oReturn.AddCountry(m.oCountry)
         EXIT
      ENDIF
   ENDFOR
ENDFOR
```

```
RETURN m.oReturn
```

Listing 26. When a method returns a collection, make sure to store the result in a variable.

```
oFrenchCountries = oCountries.GetCountriesThatSpeak("French")
```

Note that the GetCountriesThatSpeak method actually creates another instance of colCountries to hold the result. This means that a given cusCountry object can belong to more than one collection at the same time.

Just as this method can return a collection, you can pass a collection into another routine for processing. For example, once we have a collection of countries where French is spoken, we might want to process that list in some way; we can pass oFrenchCountries into the processing code. We might want to build a list of other languages spoken in such countries. **Listing 27** (FindOtherLanguages.PRG in the downloads for this session) shows a function that accepts a collection of countries and the name of a language and returns a collection of other languages spoken in any of those countries.

Listing 27. You can pass a collection as a parameter to make the whole thing available.

```
* Languages other than a specified language spoken in
* a group of countries.

LPARAMETERS oCountries, cCommonLanguage

LOCAL oCountry, oLanguages, cLanguage

oLanguages = CREATEOBJECT("Collection")

FOR EACH oCountry IN m.oCountries FOXOBJECT
   FOR EACH cLanguage IN oCountry.oLanguages
      IF UPPER(m.cLanguage) <> UPPER(m.cCommonLanguage)
         TRY
            * Add the language, using the name as a key
            oLanguages.Add(m.cLanguage, UPPER(m.cLanguage))
         CATCH
            * We'll land here if the language has already been added
            * since that would cause a duplicate key
         ENDTRY
      ENDIF
   ENDFOR
ENDFOR

RETURN m.oLanguages
```

Collections as temporary storage

Another way I often use collections is to hold a list of items that need some kind of processing. This is a variation of using collections as managers. While a manager object generally exists throughout an application, these collections tend to come and go. However, they perform many of the same kind of processes as manager collections.

For example, in the same client application that tracks waiting requests, there's one area where we need to perform a series of processes against a selected group of items. The items are objects. So I create a collection and add the items to be processed to the collection. Then, the processing code loops through the collection and handles each item. Since the collection is a local variable, when the processing code is done, the collection disappears, but the original objects still exist as they did beforehand.

A case from the Sudoku game I built demonstrates this, as well as providing another example of returning a collection from a method. In the game, when the user types a value into a cell, the game board is checked for any conflicts. (A conflict in Sudoku means that the same value appears twice in a row, column or block.) Any cells with conflicts appear in red. **Figure 1** shows a game in progress. The purplish numbers are those provided at the start of the game; black numbers are user entries that are (for the moment) acceptable; red shows conflicts.

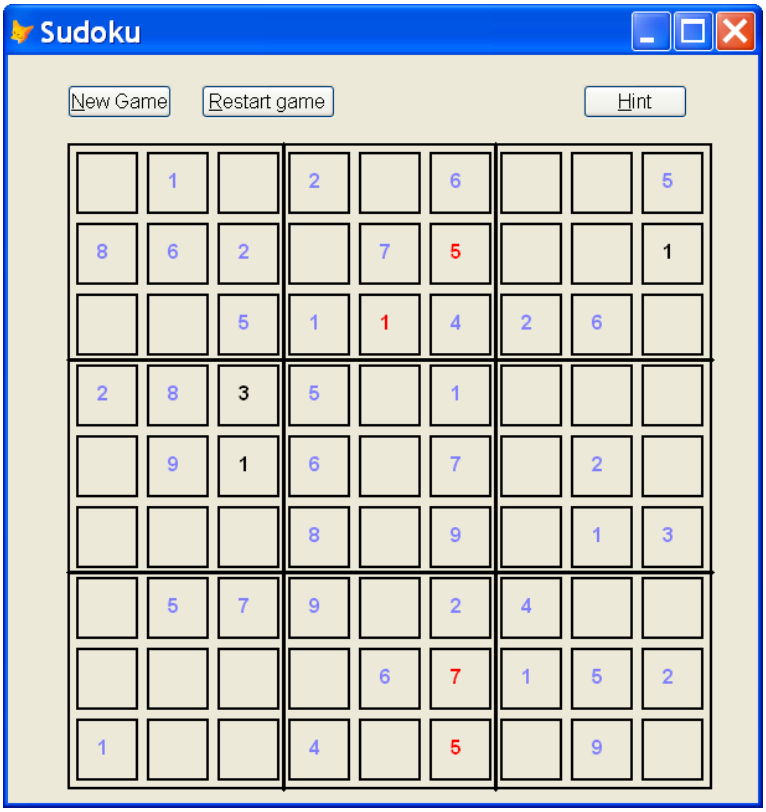

Figure 1. In the Sudoku game, values in red indicate conflicts, either with fixed values (those provided at the start of the game) or with other user entries.

To show the conflicts, a method of the board called CheckForConflicts (shown in **Listing 28**) is called. It calls on a method of the bizGame object, also called CheckForConflicts; that method returns a collection of bizCell objects that are in conflict. The board-level CheckForConflicts method then traverses that collection, finds the corresponding visible cell, and tells it to show a conflict.

Listing 28. The CheckForConflicts method of the board object (a subclass of Container) gets a collection of bizCell objects that are in conflict, finds the corresponding visual cell object for each and tells those objects to display the conflict.

```
LOCAL oConflicts, oBizCell, oCell

This.ClearConflicts()

oConflicts = This.oBizGame.CheckForConflicts()

IF NOT ISNULL(m.oConflicts) AND oConflicts.Count > 0
   FOR EACH oBizCell IN m.oConflicts FOXOBJECT
      oCell = This.GetCellByBizCell(m.oBizCell)
      IF NOT ISNULL(m.oCell)
         oCell.ShowConflict(.T.)
      ENDIF
   ENDFOR
ENDIF

RETURN oConflicts.Count > 0
```

Debugging with collections

There is one big negative to working with collections. VFP's debugging tools do not handle collections well. You can't drill down into a collection in either the Locals or the Watch window of the Debugger, though you can see the Count. In order to look at a particular item, you either have to enter a path to it into the Watch window, or save it to a variable and then examine that in the Locals or Watch window. In addition, IntelliSense for collections is weak.

To address some of the issues around debugging collections, I created a tool, the Object and Collection Inspector, which provides drill-down capability for objects and collections. The tool is discussed later in this section. More recently, Jim Nelson and Matt Slay created the Object Explorer that provides similar capabilities to my tool, as well as some additional abilities. It's discussed after the Object and Collection Inspector.

Collections in the Debugger

The VFP Debugger was introduced in VFP 5. Although that version included support for working with COM objects, including collections, the Debugger's support for those items was limited. Only some members of COM objects show in the Debugger; often, a property isn't visible in the Debugger until you've referenced it elsewhere. For example, if you open a project, grab a reference to it (oProj = _VFP.ActiveProject), expanding the object reference (**Figure 2**) omits the Files collection. However, if you then refer to the collection (oFile = oProj.Files[1]), the collection appears in the list. However, when you expand it (**Figure 3**), only the Count shows.

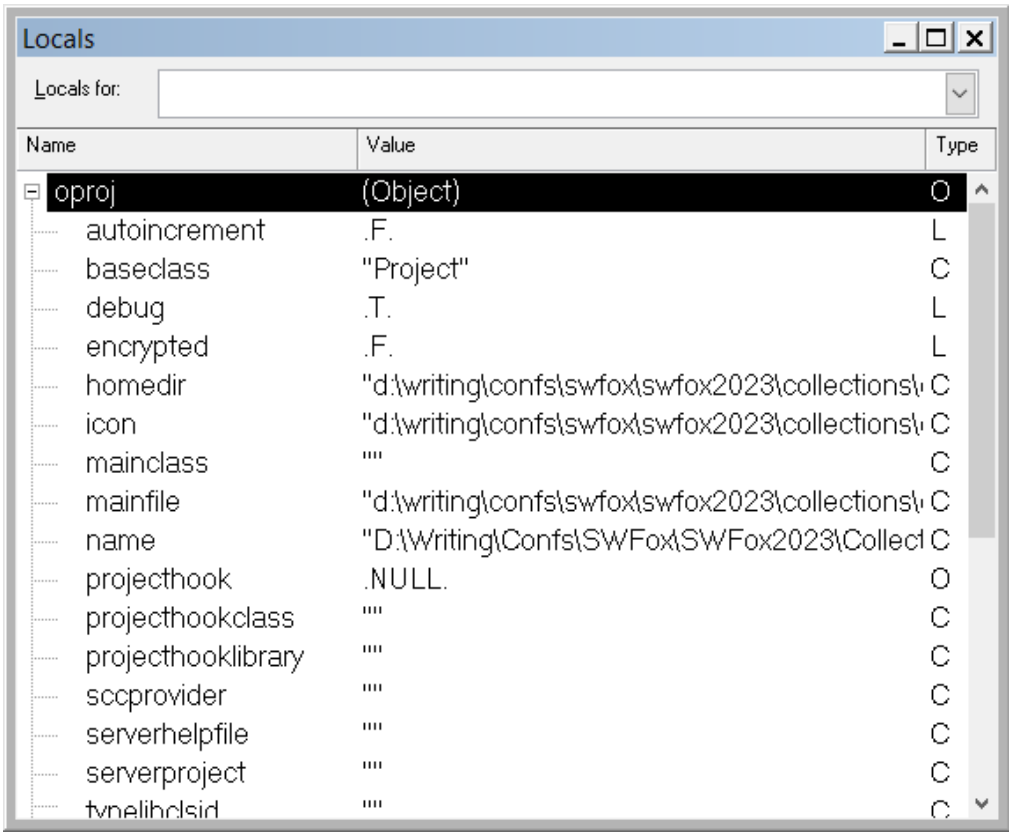

Figure 2. The Debugger doesn't handle COM objects or collections well. Here, the Files collection of the project isn't included in the list.

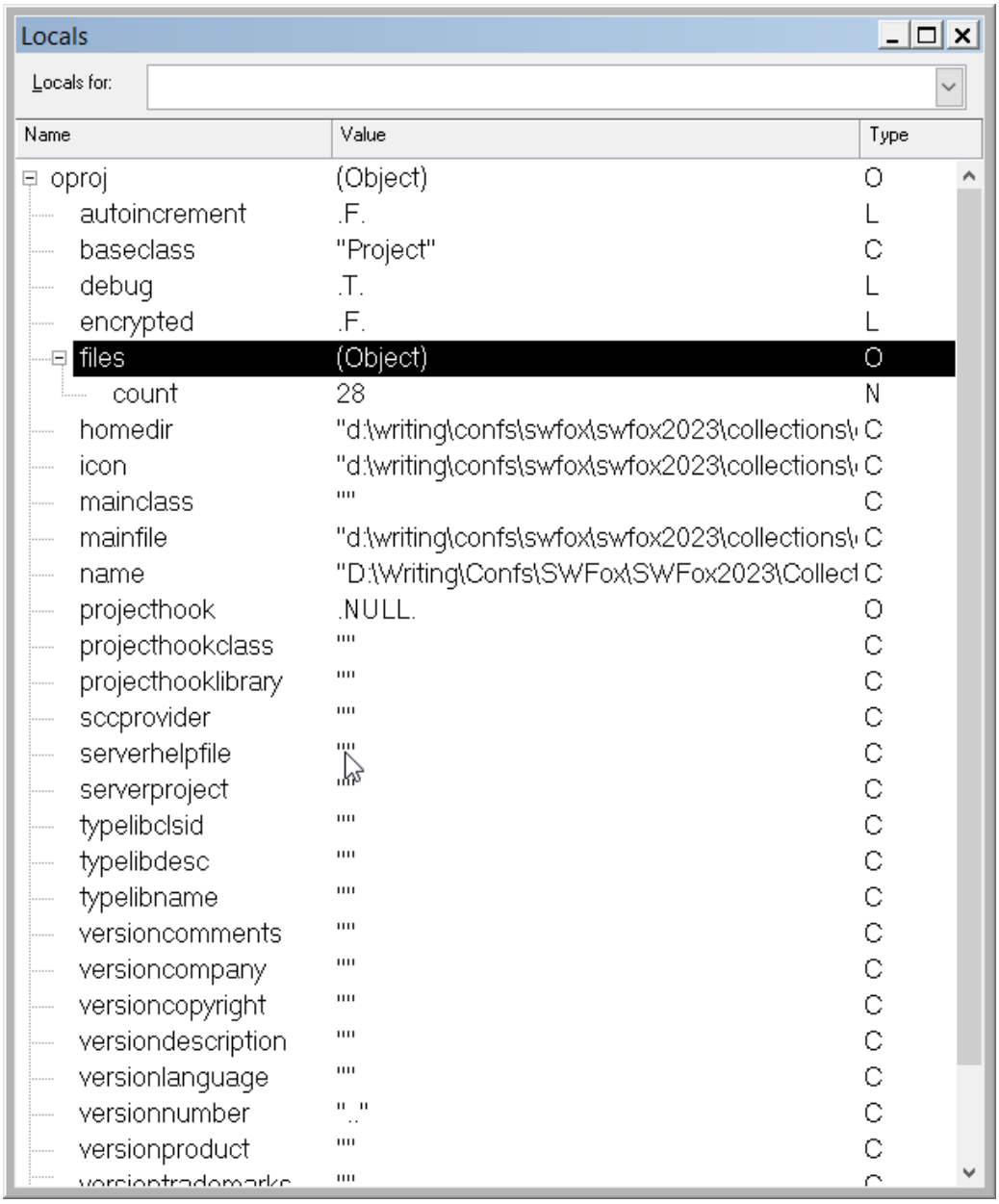

Figure 3. Once you refer to a property of a COM object, it may appear in the Debugger.

The Debugger handles some built-in collections (like Controls and Objects) properly. For those, drilling down allows you to access each member. **Figure 4** shows part of the Locals window, representing a form with two controls. Each of those objects can be expanded to see its properties and their values. However, other built-in collections like the List and ListItem properties of comboboxes and listboxes do not allow drill-down.

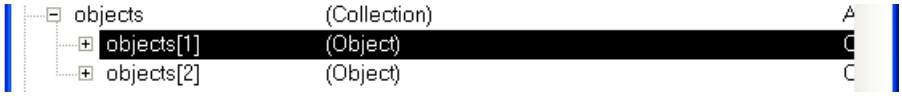

Figure 4. The built-in Controls and Objects collections can be expanded in the Debugger.

When native collections were added in VFP 8, unfortunately, the VFP team didn't update the Debugger to handle them properly. Opening a VFP collection in the Debugger shows its properties, including Count, but offers no way to drill down. **Figure 5** shows the oCountries collection (created in **Listing 14**) in the Locals window.

Figure 5. Expanding a collection in the Debugger shows all its properties, but offers no way to access its members.

The work-around is to drill down manually. In **Figure 6**, I've added oCountries[1] to the Watch window, allowing me to see the properties of the first member of the collection.

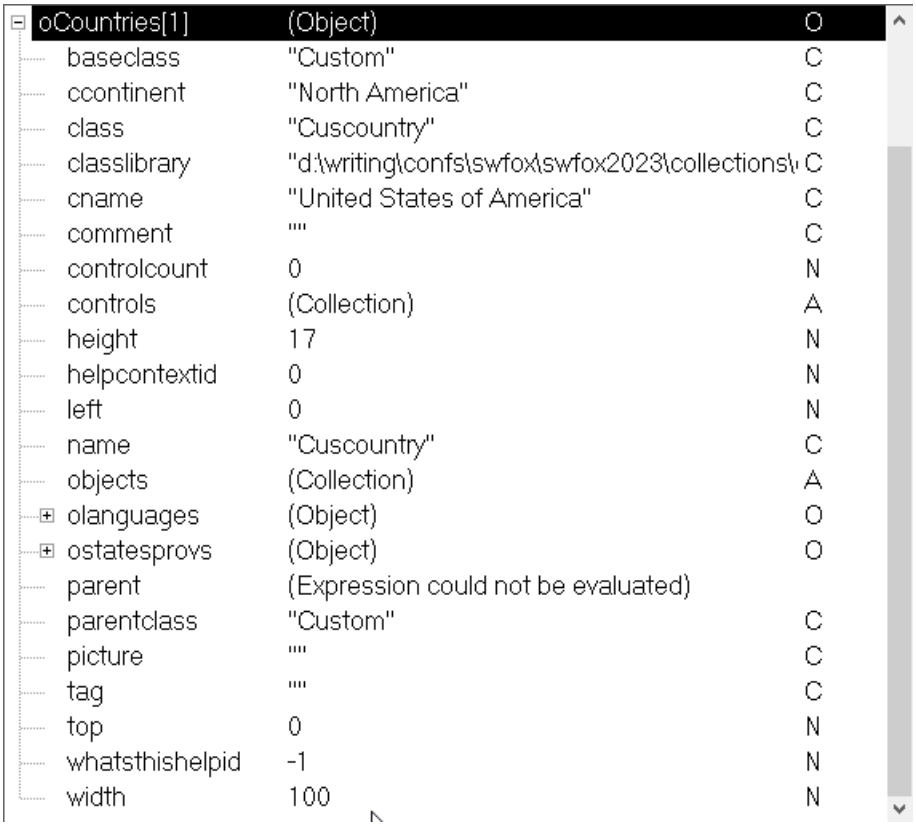

Figure 6. One work-around for the Debugger's inability to drill into collections is to add individual members to the Watch window.

Alternatively, you can assign a member to a variable and then look at that variable in the Locals window. Of course, to do that, you have to be at a breakpoint in your code that provides you with access to the Command Window.

Later in this document, I'll look at a couple of tools that provide better solutions.

Collections and IntelliSense

VFP's IntelliSense makes writing code and debugging it easier by providing cues as you type code. When debugging, you can type an object reference in the Command Window (or the Watch window) and see what properties and methods that object has. **Figure 7** shows an example, using the _VFP system variable.

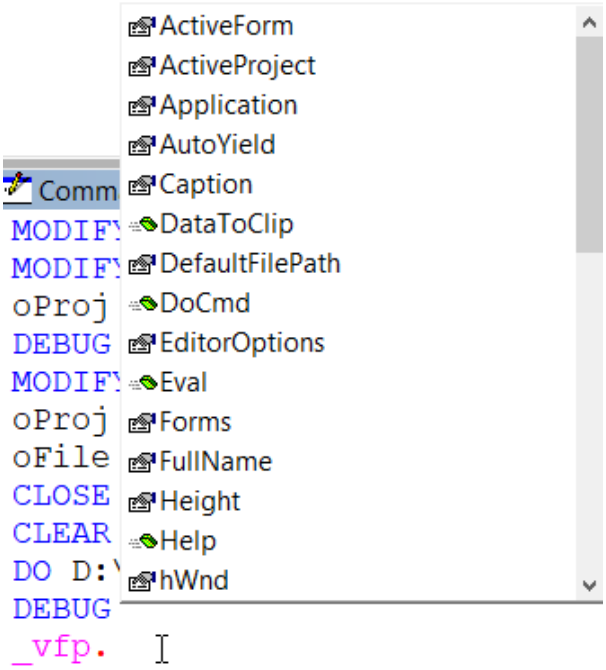

Figure 7. VFP's IntelliSense lets you see what properties and methods an object has, making debugging from the Command Window easier.

For collections, however, IntelliSense isn't as friendly as it could be. When you type the name of a variable representing a collection followed by a period, you see the collection's properties as you'd expect (**Figure 8**). However, once you drill into the collection, IntelliSense is much less help. As **Figure 9** demonstrates, when you drill down to a particular object, IntelliSense doesn't follow you. To get IntelliSense for a collection member, you have to save a reference to a variable and use the variable, as in **Figure 10**. (The line above the cursor, partially hidden by the IntelliSense pop-up, reads: oCanada=oCountries["Canada"].)

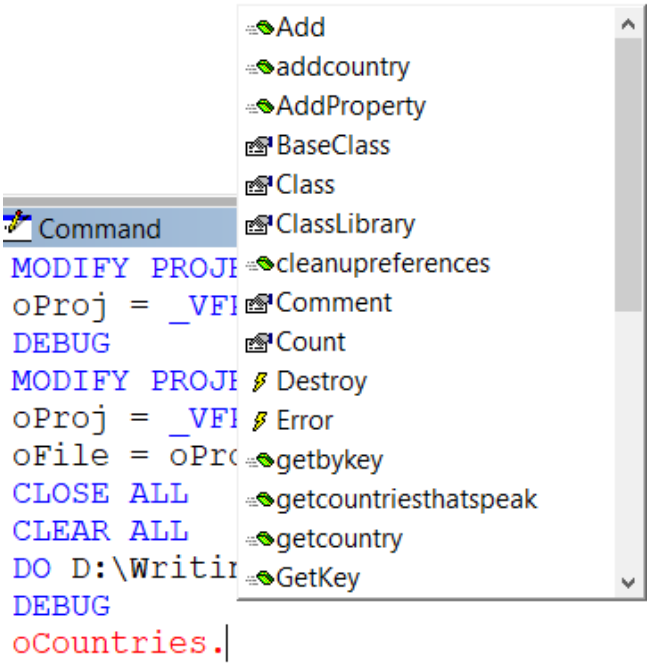

Figure 8. Typing a reference to a collection gives you the collection properties.

```
oCountries.Item(1).
```

Figure 9. When you drill into a collection, IntelliSense gives up.

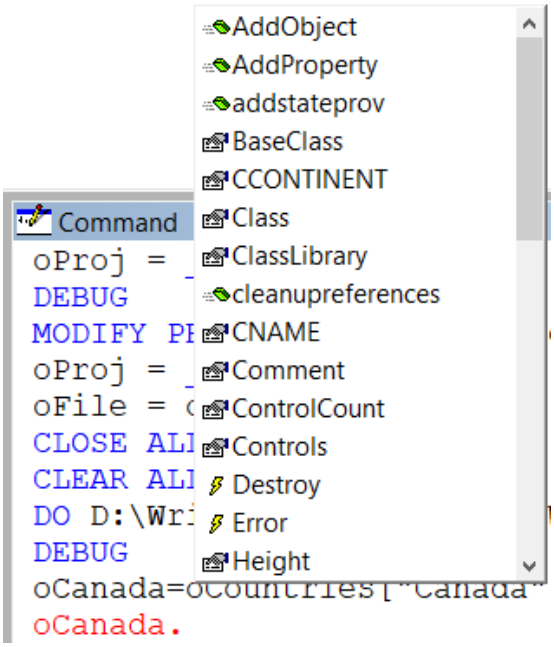

Figure 10. To use IntelliSense with a collection member, save an object reference to a variable and manipulate the variable.

Though IntelliSense can't drill into VFP's native collections, it does work for COM collections with one caveat. You have to use parentheses rather than square brackets to

specify a member. In **Figure 11**, oDoc is a reference to a Word document; specifying oDoc.Paragraphs(1). pops up the list of PEMs for a paragraph.

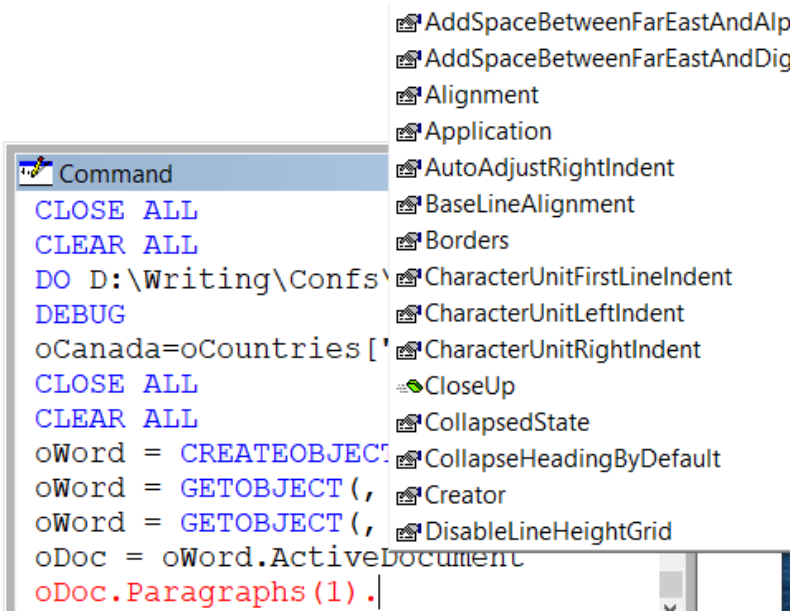

Figure 11. When working with COM collections, IntelliSense kicks in as long as you use parentheses to choose a member of the collection.

The Object and Collection Inspector

The most common reason I hear why people don't use collections is the difficulty in debugging them. So, when I was working on a project that used collections extensively, I decided to build a tool that would make it easier. The Object and Collection Inspector is written in VFP; you pass it a reference to an object and it displays a form showing the object hierarchy starting with that object, and allows you to drill down into any collections. **Figure 12** shows the Object and Collection Inspector for the collection of countries used as an example throughout this paper. I've drilled into that collection, and then drilled into its second member (Canada) and expanded its oStatesProvs collection.

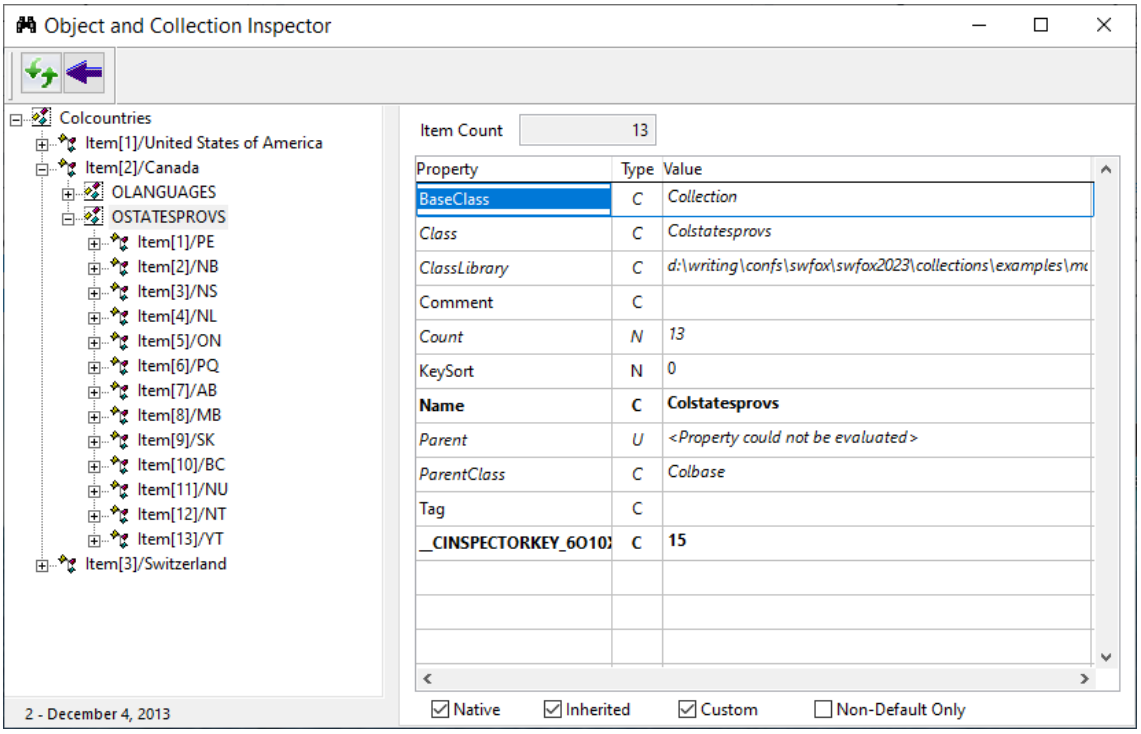

Figure 12. The Object and Collection Inspector lets you drill down into objects and collections.

The left-hand pane uses a treeview to show the hierarchy. For a collection, each item is shown with its index, and if it has one, its key. The display in the right pane varies depending what kind of item is chosen in the left-pane. For a collection as in **Figure 12**, it shows the count and then all the properties of the collection. For an object other than a collection, as in **Figure 13**, all properties are shown with their values. For objects and collections, you can narrow down which properties display in the right pane, using the checkboxes at the bottom of that pane.

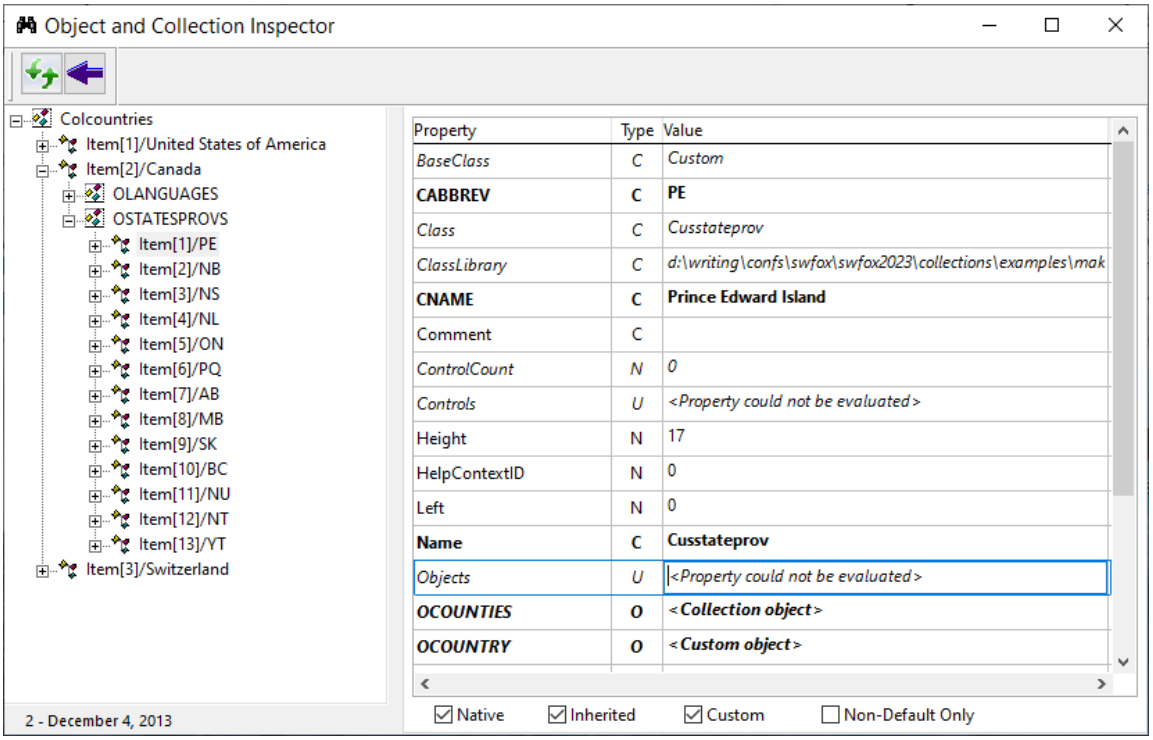

Figure 13. For objects other than collections, the Object and Collection Inspector just shows the list of properties in the right pane.

For scalar items (non-objects), the right pane shows the type and value of the item, as in **Figure 14**.

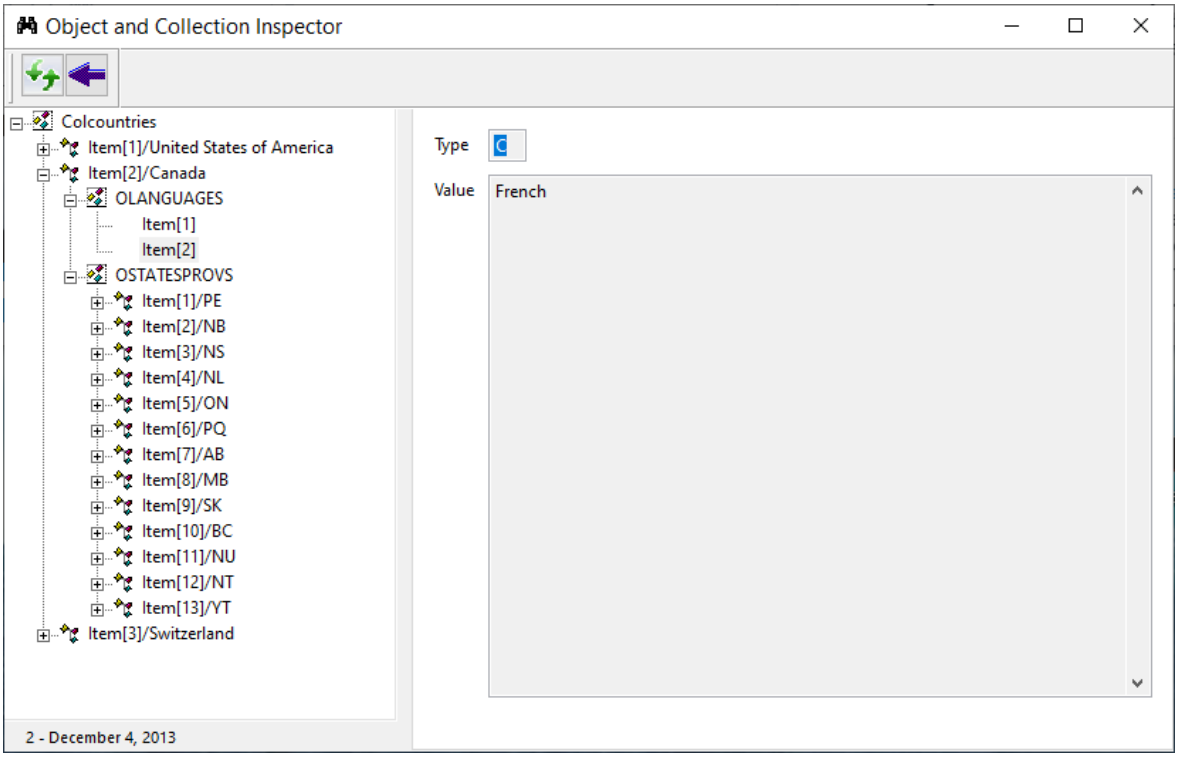

Figure 14. For scalar items, things that aren't objects, the Object and Collection Inspector shows the type and value.

It's not unusual for object hierarchies to include loops. The simplest case is where object A has a reference to object B, and object B has a reference to object A. But, in fact, much more complicated chains are not unusual. The Object and Collection Inspector can handle such situations. Rather than allowing you to drill down in perpetuity, the tool shows information for the first instance it finds of any object; for subsequent instances of that object, the right pane indicates the situation and provides a way to get to the object, as in **Figure 15**.

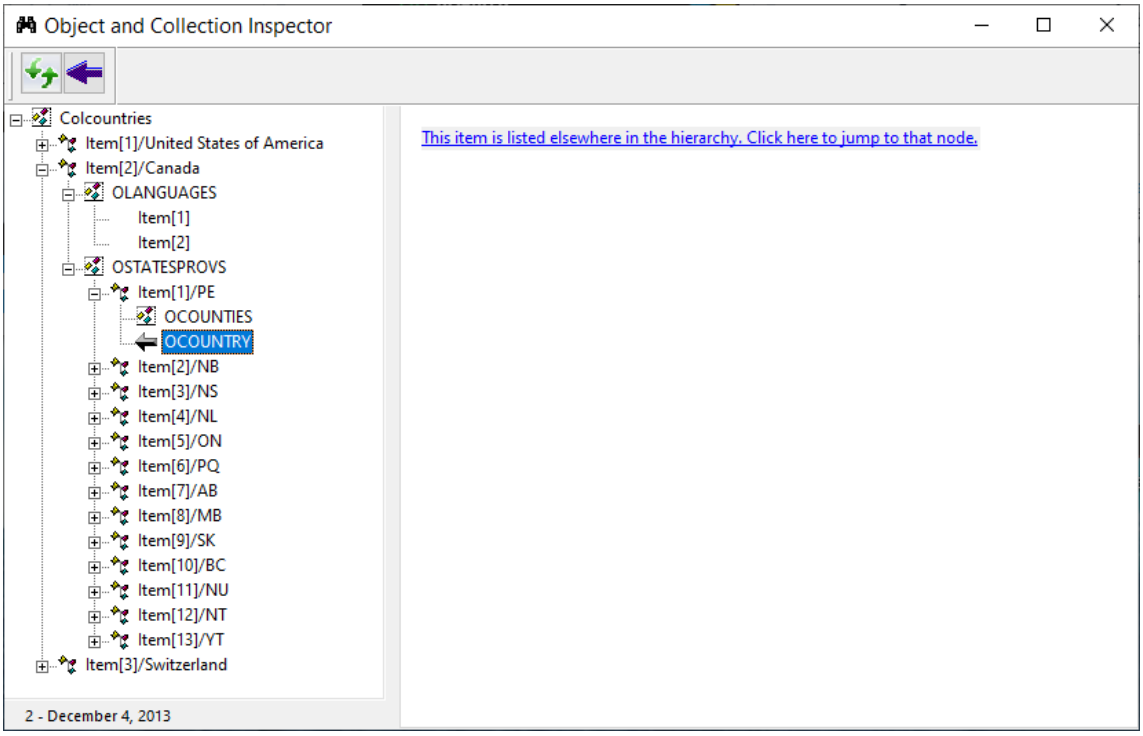

Figure 15. The Object and Collection Inspector shows the details of each object only once. When the same object is found again in the hierarchy, the reference uses a back-arrow icon and the right pane gives you a way to jump to the details of the object.

To avoid infinite recursion while constructing the treeview, the tool adds a property to every object it finds. For a given call to the Inspector, the property's name is generated, including a random value. The property value for a given object is the order in which it's seen in constructing the treeview data. **Figure 12** shows the property and its value for the item selected there; it's the last property.

The left button ("refresh") above the treeview lets you refresh the Inspector with the latest data from the object or collection you passed in. Since the Inspector runs in a top-level window, that means you can leave it open as you work and then come back to it and bring it up to date.

The right button ("back") lets you go back through the list of nodes you've looked at.

The Object and Collection Inspector is available through Thor. Once you've installed it, it appears in the Objects and PEMs submenu of the Thor Tools menu. What happens when you choose it in the Thor menu (or use an assigned hot key) depends on the position of the editing cursor. If the cursor is over the name of an object or collection when you choose it, the Object and Collection Inspector opens inspecting that object or collection. Otherwise, a call to the tool is pasted into the Command Window (as in **Listing 29**) with the cursor positioned for you to type a reference to the object you want to inspect. (The path will vary depending how you have Thor installed.)

Listing 29. Using the Object and Collection Inspector from Thor pastes a command like this into the Command Window. You can add a reference to relevant object and hit Enter to run the tool.

```
DO ("..\..\..\..\..\..\FOX\VFPX\THOR\THOR-MASTER\THOR\THOR\TOOLS\APPS\OBJECT
INSPECTOR\Inspector.App") WITH
```

The Object Explorer

Inspired by the Object and Collection Inspector and by a tool created by Mike Feltman called Collection Explorer, Thor giants Jim Nelson and Matt Slay[1] built another tool, the Object Explorer, that makes it easier to get a handle on the object of interest and lets you not just view, but also modify the properties of objects and collections.

The Object Explorer is also available through Thor. Once it's installed (via Thor's Check For Updates mechanism), the easiest way to put it to work is to define a hotkey for it. To start the tool, put focus on the object you're interested in (whether that's a form or a control) and hit your hotkey. The tool opens showing the entire form, with the object that had focus highlighted. For example, **Figure 16** shows the Object Explorer as it opens for the form Many.SCX from the VFP Solutions Samples; as **Figure 17** shows, the cursor is in the Company field in the form, so it's highlighted when the tool opens. The title bar contains the path to the form and its caption.

[1] Sadly, Matt passed away in 2021.

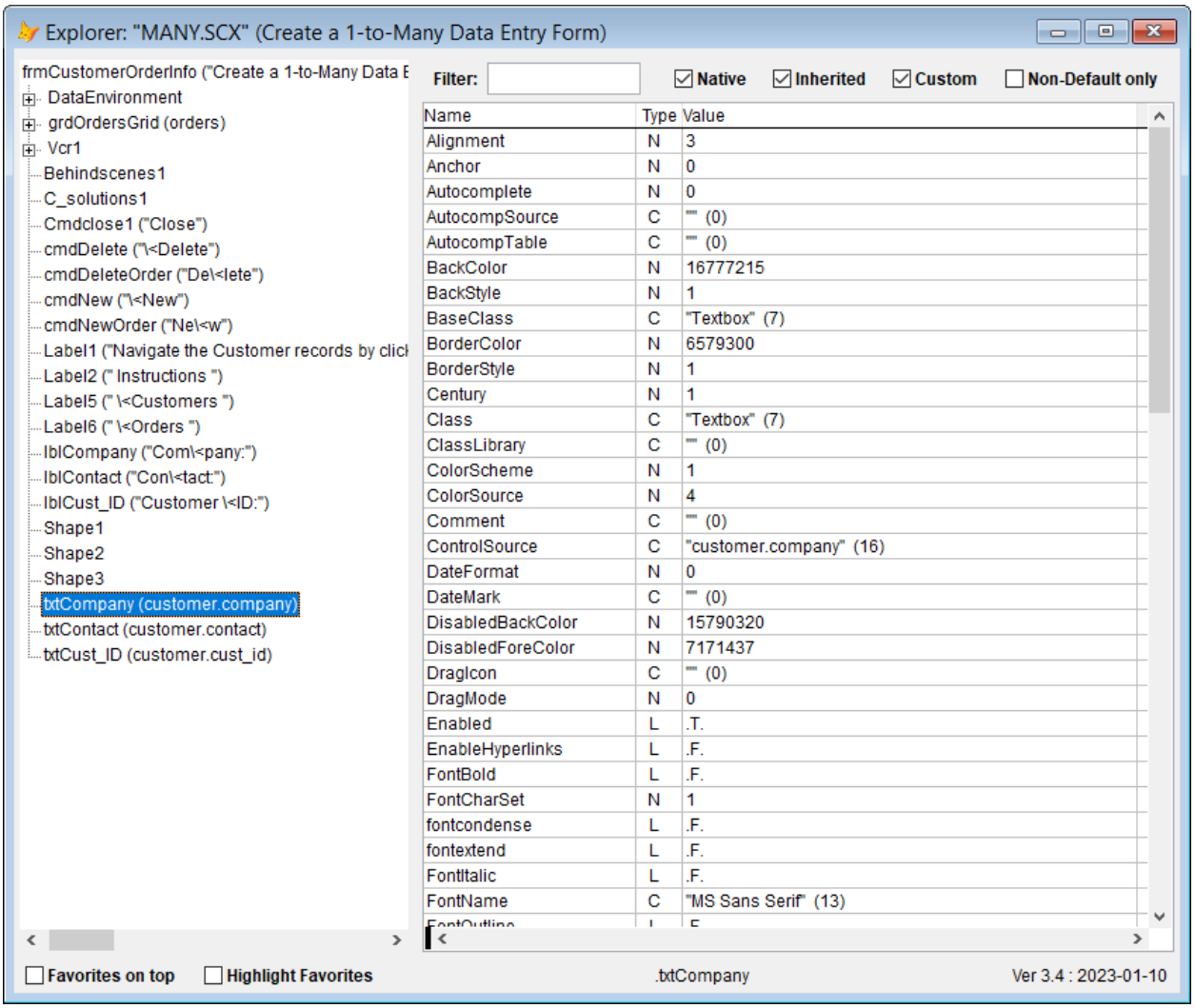

Figure 16. The Object Explorer lets you drill into object hierarchies and even modify property values.

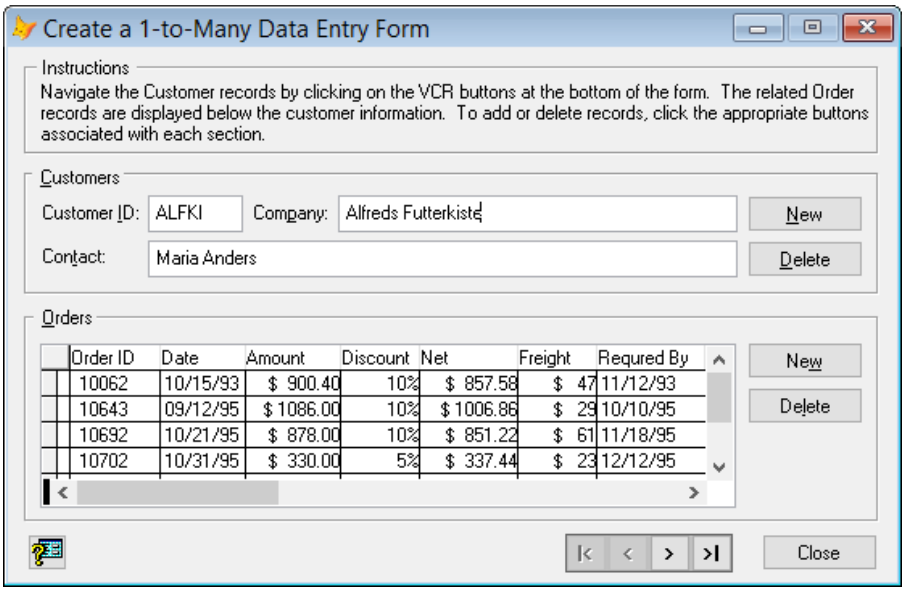

Figure 17. This form (from the VFP Solution Samples) is explored in **Figure 16**. Because focus is on the Company field here, it's initially highlighted in the tool.

As this example demonstrates, Object Explorer was designed primarily to work with visual objects, but you can apply to non-visual objects, including collections. To do so, you can call it directly (from code or the Command Window) and pass a reference to the object of interest, or as with the Object and Collection Inspector, you can position the cursor over a variable that refers to an object or collection and hit your hotkey to open the Object Explorer for that object or collection. **Figure 18** shows the tool exploring the oCountries collection created earlier in this paper. (Note that I've made the tool shorter than its default size for the examples that follow, to make the images fit better in this paper.) In this case, the title bar shows the name (that is, the value of the Name property) of the object being explored.

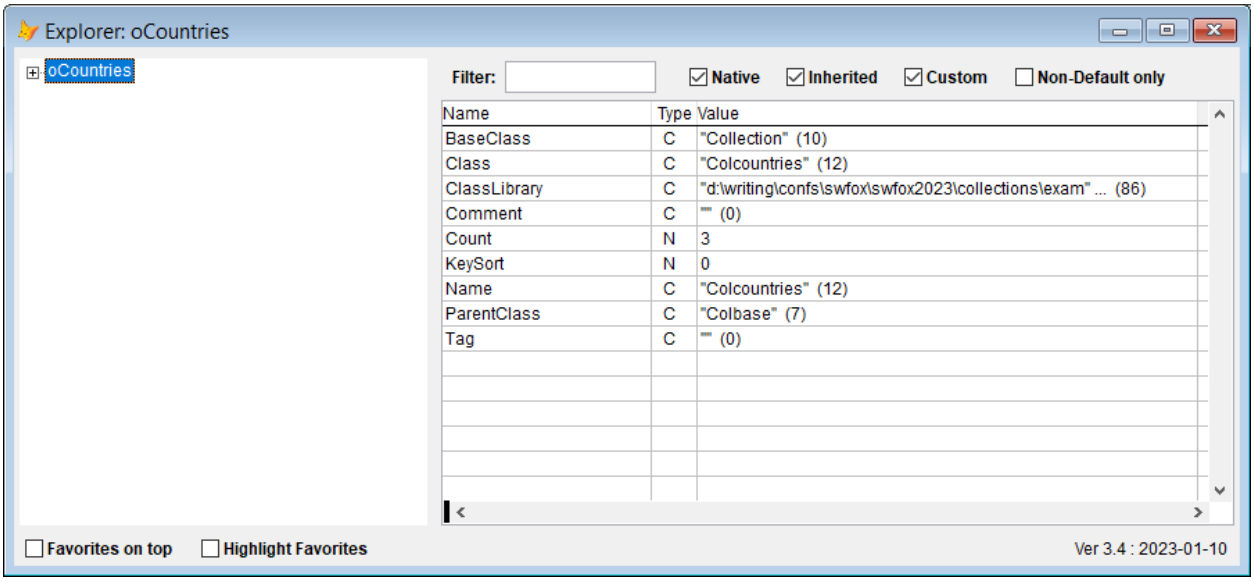

Figure 18. Though the designers were thinking primarily for forms and other visual objects, the Collection Explorer works with non-visual objects.

As in Object and Collection Inspector, the left pane is a treeview. Click an item in the left pane to show its properties in the right pane. In **Figure 19**, I've drilled down into Switzerland's cantons and select "LU" (Lucerne). In the middle of the bottom section of the tool (below the right pane), you can see the path from the object passed in to the currently highlighted node. In this case, it's .Switzerland.oSTATESPROV.LU.

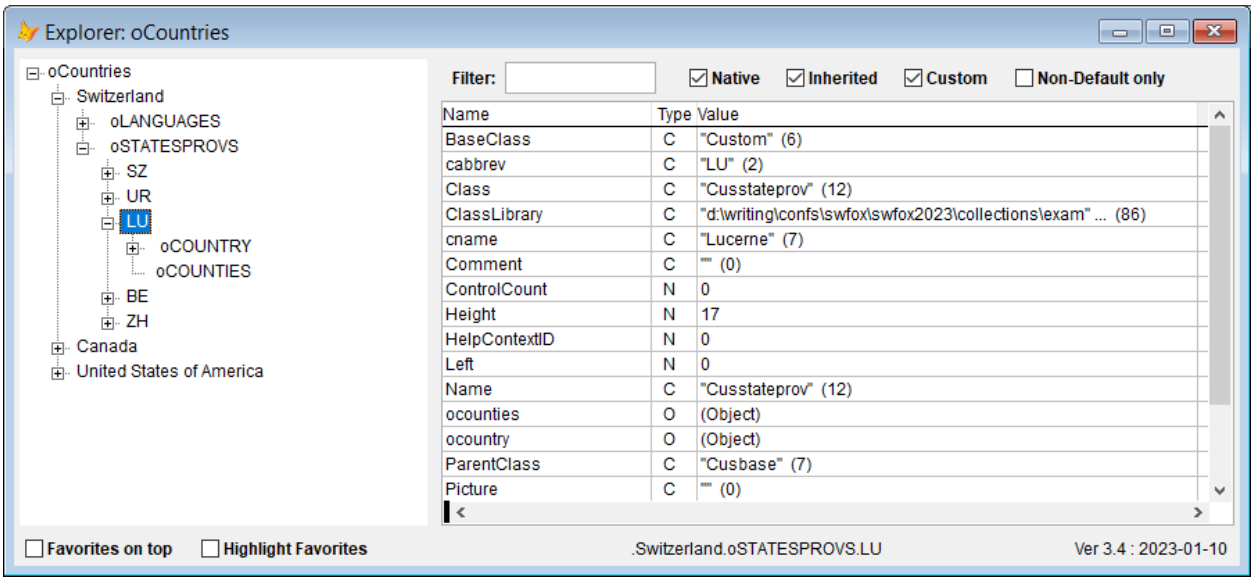

Figure 19. Drill down in the left pane to see the objects in a collection or contained in another object. Click an object in the left pane to see its properties in the right pane.

One small pitfall: Collection Explorer doesn't load the properties of an object until you select it. That means that when you have circular hierarchies, it is possible to keep drilling down indefinitely as in **Figure 20**, where I've drilled down several levels simply to get to

another node that represents Lucerne. You can see part of the much longer path from oCountries (colCountries) to the highlighted node below the right pane.

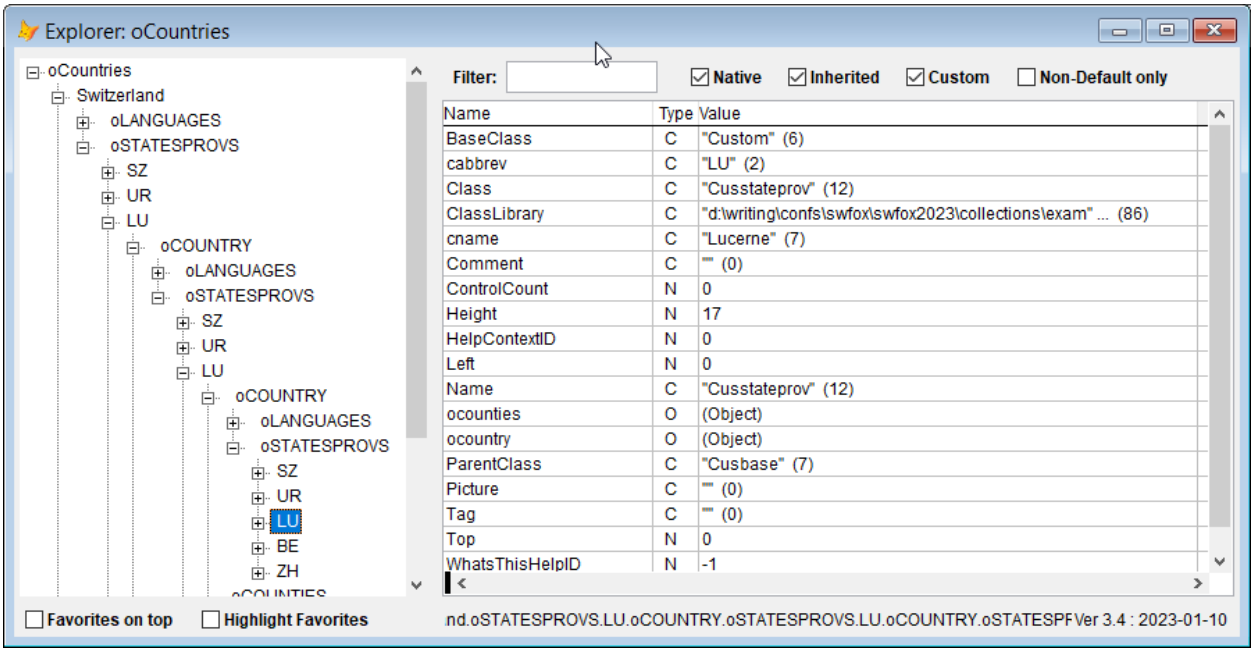

Figure 20. Object Explorer doesn't check for circular hierarchies, so you can keep drilling down forever.

Object Explorer shows scalar members of a collection with the item number and value in the treeview and treating the value as an unnamed single property in the right pane, as in **Figure 21**.

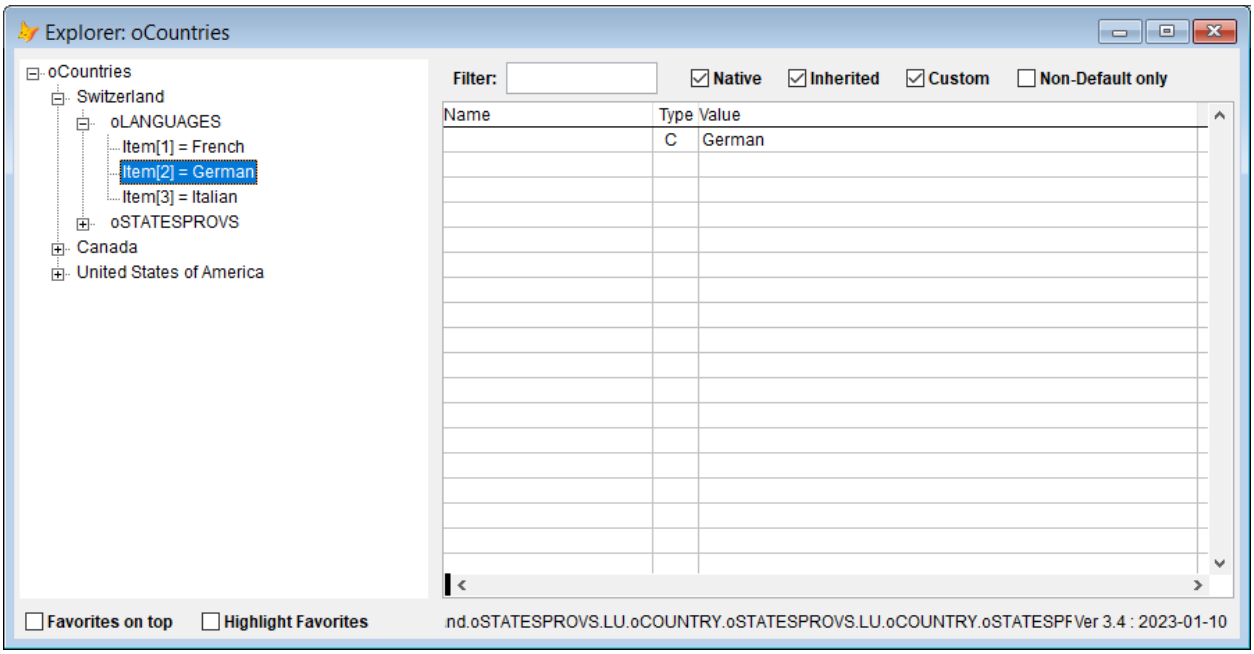

Figure 21. For scalar members of collections, Object Explorer shows the item number and treats the value like a single property.

Like Object and Collection Inspector, Object Explorer lets you narrow down the list of displayed properties in several ways. First, the checkboxes above the right pane let you include or exclude properties based on whether they are native to VFP, inherited from another class, or added in the current class. For example, in **Figure 22**, only inherited properties of the Manitoba object are shown. The final checkbox in that set lets you see only those properties that have been modified in the current object, as in **Figure 23**.

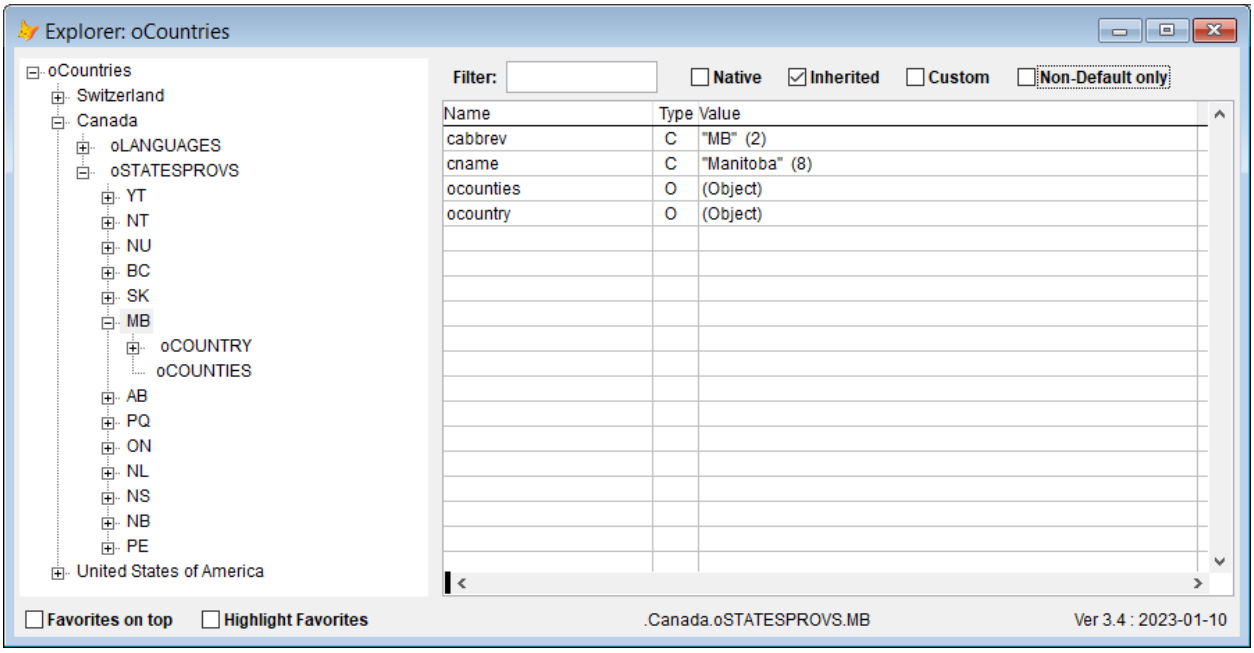

Figure 22. The checkboxes above the right pane let you filter the list of properties based on their origin and status.

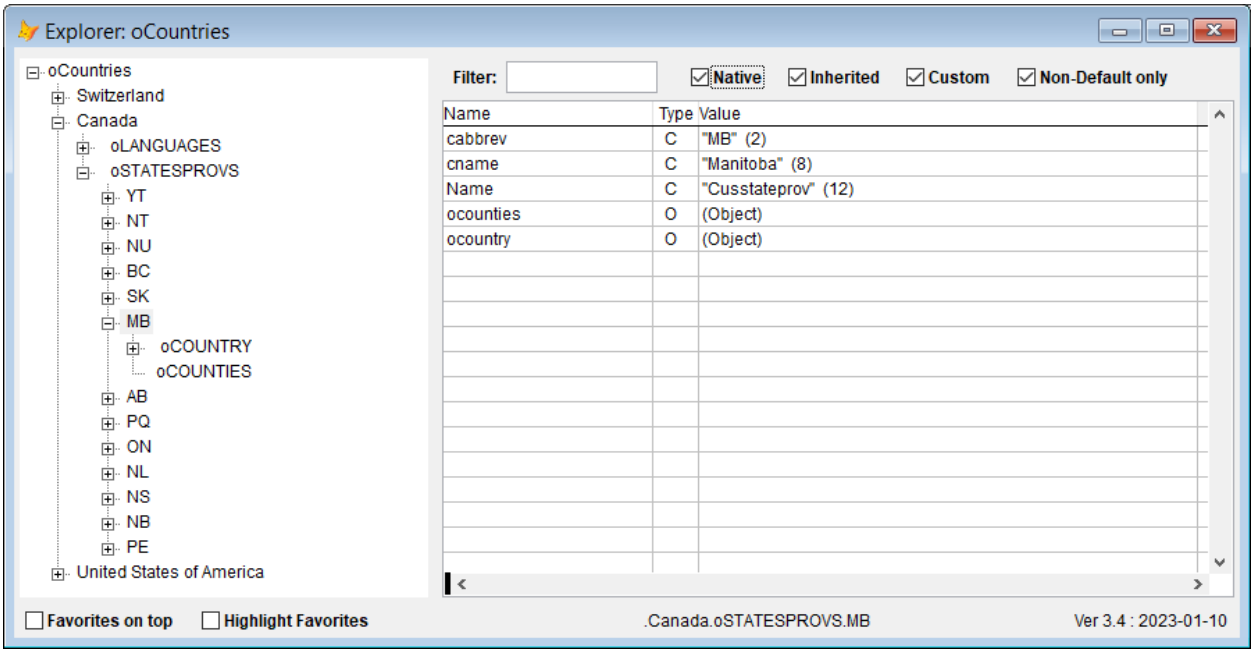

Figure 23. The Non-Default only checkbox indicates whether to filter out properties unchanged in this object.

The filter textbox lets you limit the right pane to only those properties whose names or values contain the specified string. For example, **Figure 24** shows the properties of the United States of America object that have "ST" in their name or value.

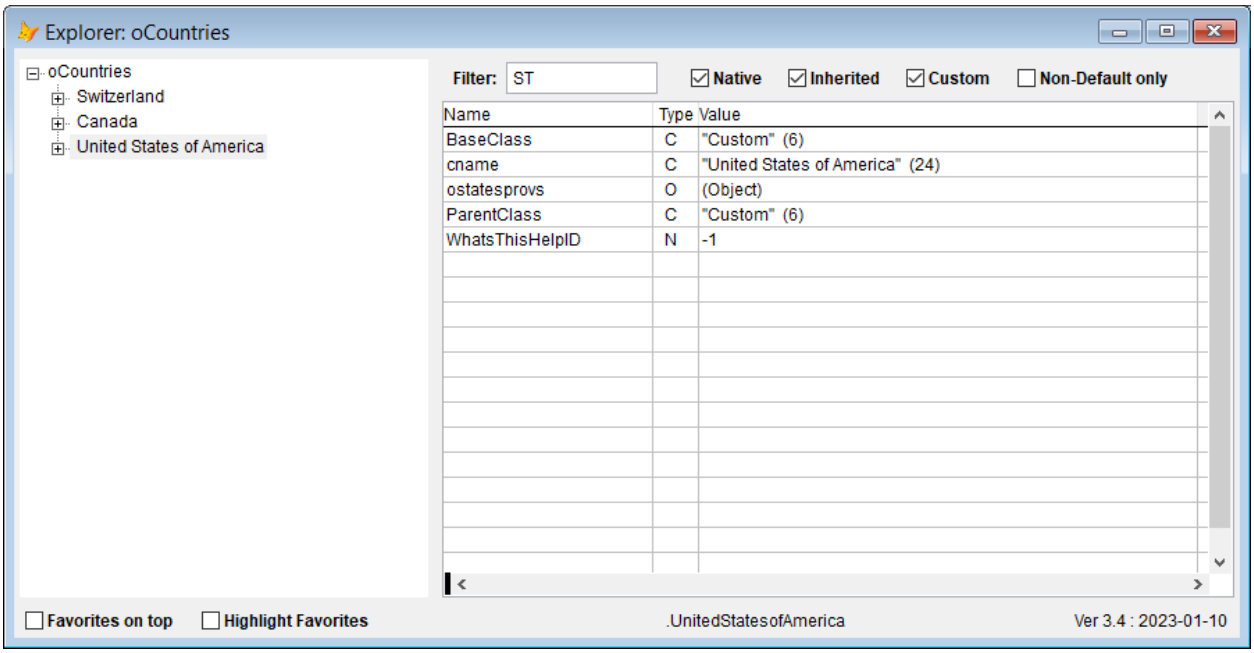

Figure 24. You can filter the right pane to those properties whose names or values include a particular string.

Object Explorer lets you specify a set of favorite properties and gives you a couple of ways to easily find them. The default list of favorites (which appear to be chosen for easy debugging of forms) is shown in **Table 1**.

Table 1. The default list of favorite properties is focused on debugging forms.

BaseClass	Caption	Class
ClassLibrary	ControlSource	DataSessionID
Enabled	Height	Left
ParentClass	RecordSource	RowSource
RowSourceType	Text	Top
Value	Visible	Width

The two checkboxes under the left pane let you make favorites more visible in the right pane. When Favorites on top is checked, properties marked as favorites appear first in the right pane, as in **Figure 25**. When Highlight Favorites is checked, as in **Figure 26**, the properties marked as favorites appear with a bright yellow background. You can check both checkboxes and then you get a block of yellow-highlighted favorites at the top of the list.

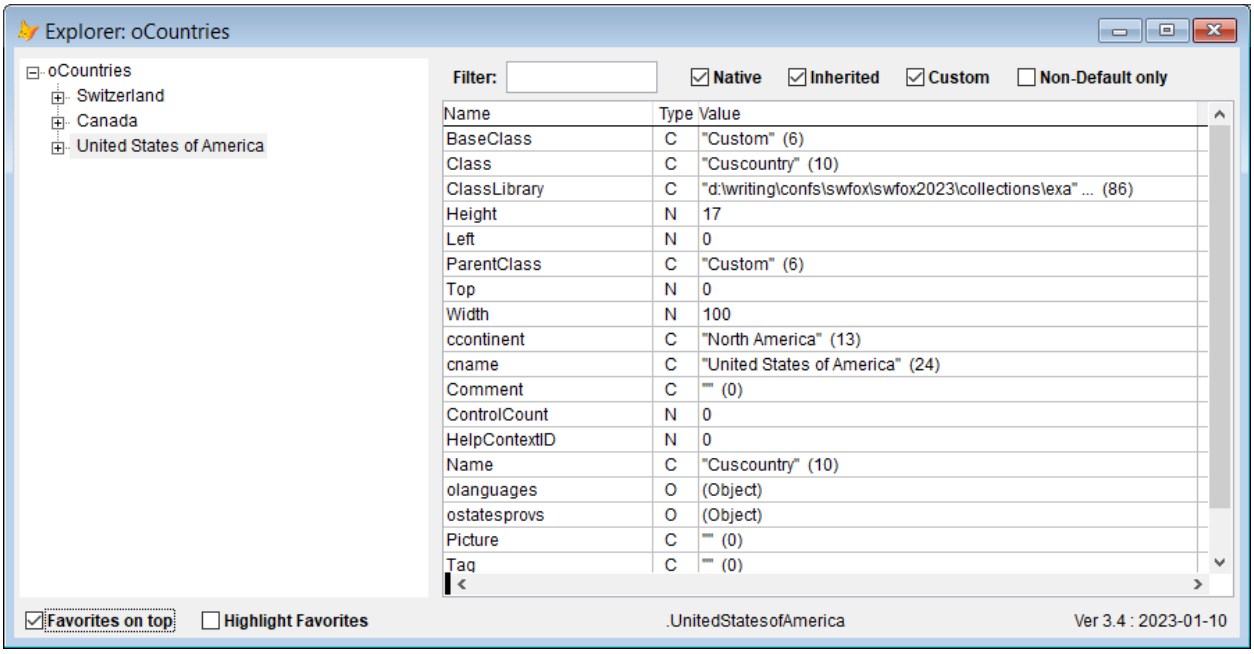

Figure 25. When you check Favorites on top, properties marked as favorites appear first in the right pane.

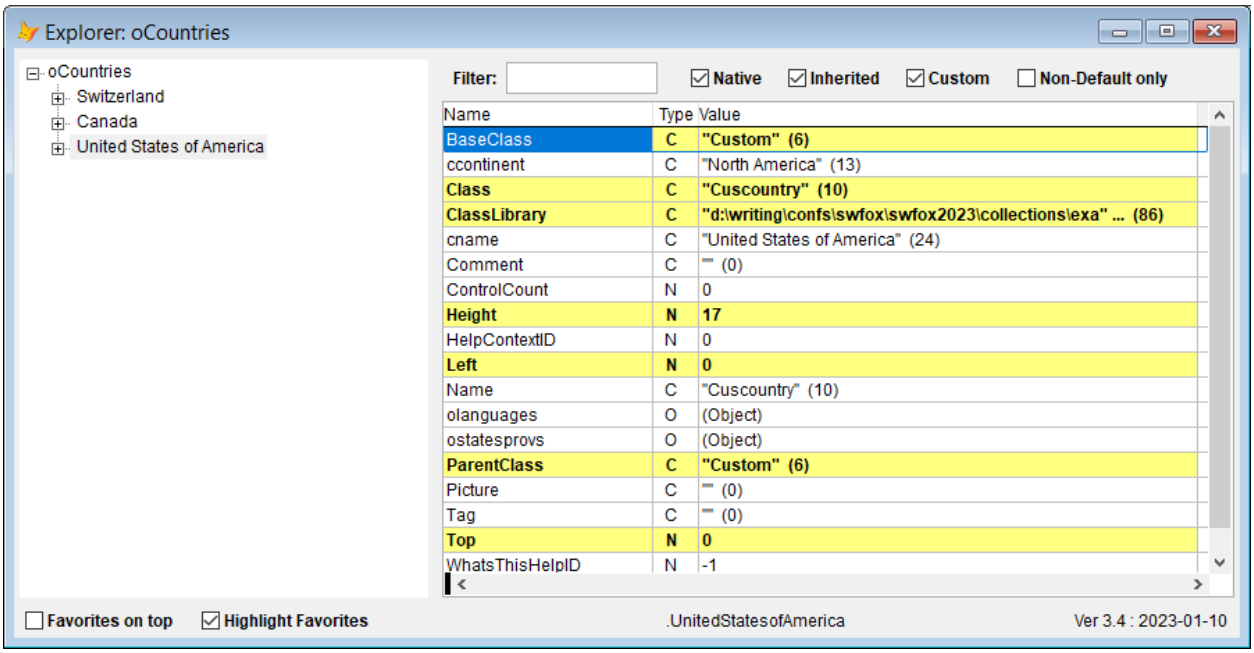

Figure 26. Check Highlight Favorites to make properties marked as favorites stand out in the right pane.

You can add and remove favorites by right-clicking on a property in the right pane and choosing Favorite, as in **Figure 27**. As you'd expect, once you've marked a property as a favorite, it shows up in the same way as other favorites, and if you right-click again, Favorite is checked in the context menu, as in **Figure 28**.

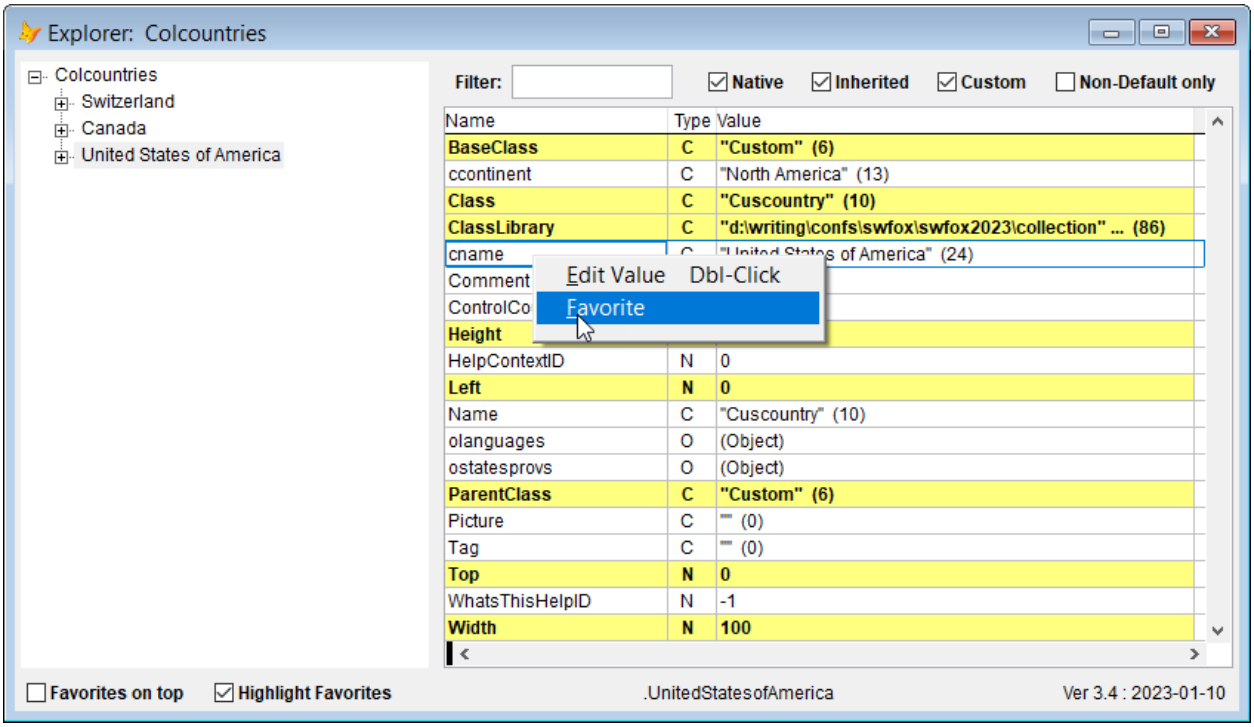

Figure 27. You can customize the list of favorites by right-clicking on properties and choosing Favorite.

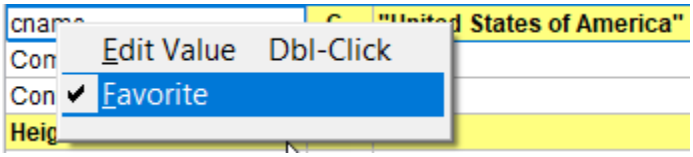

Figure 28. Once you've marked a property as a favorite, it's checked in the context menu.

Your list of favorites is remembered between Object Explorer sessions (and, in fact, between VFP sessions). The list is stored in a table called ExplorerSettings.DBF in VFP's settings folder (the one specified by HOME(7).

As Figure 27 and **Figure 28** indicate, you can change property values in the Object Explorer. In addition to choosing Edit on the context menu, you can double-click on the property or its value in the right pane to pop up a dialog that lets you edit the current value, as in **Figure 29**. To save your changes, hit Enter.

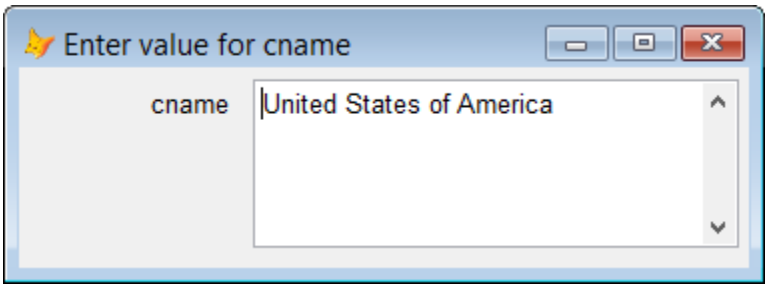

Figure 29. When you right-click on a property and choose Edit or double-click on the property or its value in the right-pane, this dialog opens to allow you to edit the current value.

If you attempt to change the value of a read-only property (like Class), a dialog appears to let you know. Of course, just as when you change values in the Debugger, the changes apply only to this instance of the object. If, for example, you change the position of a control in a form, when you close and reopen the form, the control is back in its original position.

When you right-click on an object in the left pane, a dialog opens to let you specify the name of a public variable in which to place a reference to the selected object. That reference makes it easy to experiment with that object in the Command Window or Debugger or pass it to some other tool. In **Figure 30**, the variable goButton will be created and given a reference to the button Cmdclose1.

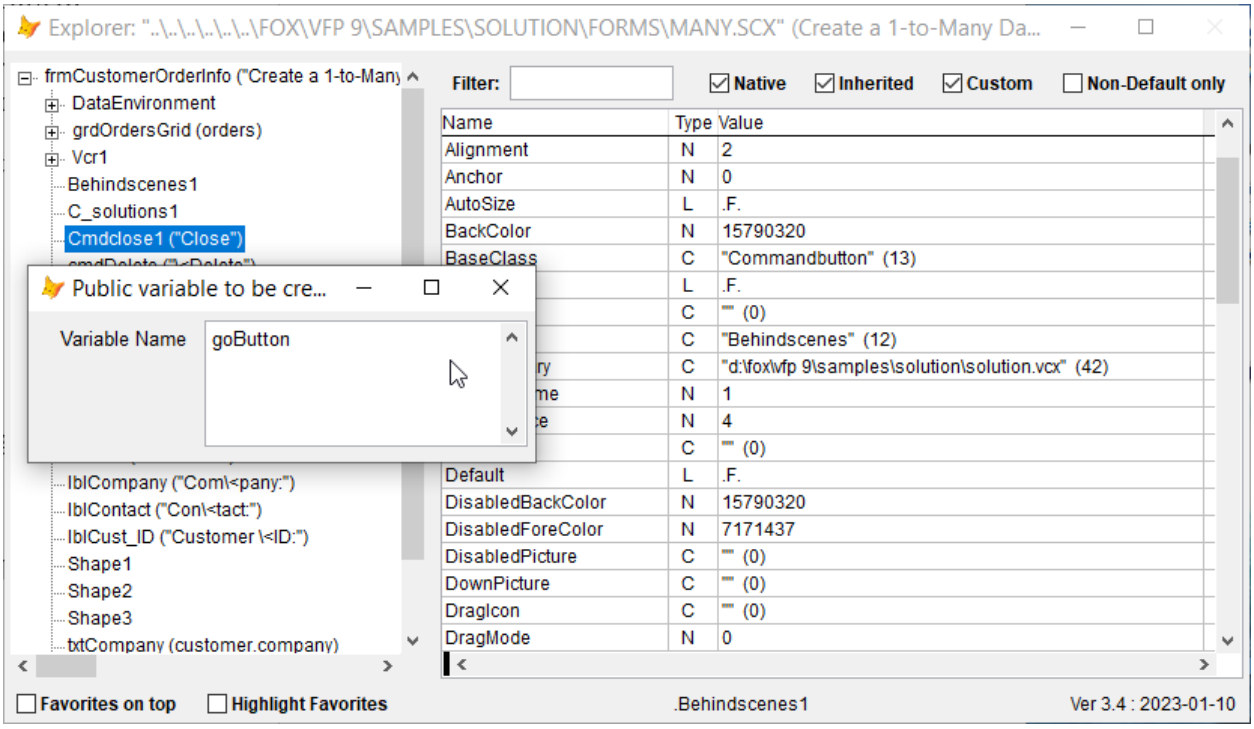

Figure 30. Right-clicking in the left pane lets you assign an object reference to a public variable.

Once you've used this capability in an Object Explorer session, the variable name you specified is remembered for the rest of that Object Explorer session. In addition, if you're using the Object Explorer via Thor, you can specify a default name to use on the Options tab of the Thor Configuration form, as in **Figure 31**. In fact, when you do this, every time you start the Object Explorer, the specified variable is assigned a reference to the top-level object you're exploring. (This makes a lot of sense, given the tool's primary focus is on debugging forms and controls, but also works for non-visual objects.)

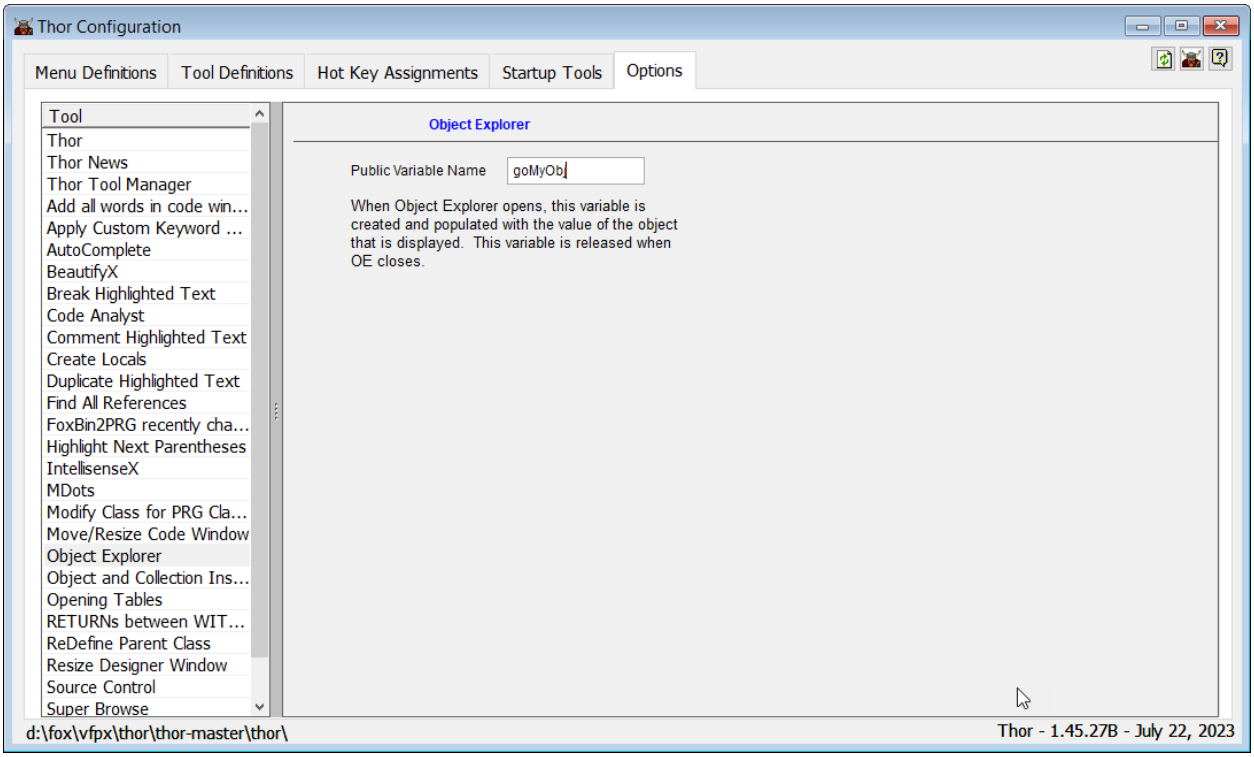

Figure 31. You can specify a default public variable name to use with the Object Explorer in the Thor Configuration form.

Give collections a try

The more I work with collections, the more uses I find for them. In addition to using them in business objects, I use them to pass data around within an application, to hold lists of things that need to be done, and lots of other ways.

Like so much in VFP, you may find collections a little strange when you get started, but over time, I think they'll grow on you.