Collections instead of Arrays

The Collection class provides an easy way to work with groups of objects.

Tamar E. Granor, Ph.D.

Every procedural programming language I used before 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.

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, as is still true for VFP. Although this 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, 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://www.tomorrowssolutionsllc.com/Materials/arrays.html.)

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 offers 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.

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 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 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 took until VFP 8 for us to get 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 actually 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 1 shows code to create a collection of strings representing the names of the first five states alphabetically. The state's postal abbreviation is used as the key for the item. Each item can be accessed by its position (index) in the list or by its key, as the last two lines show.

Listing 1. 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 2.

Listing 2. 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 last 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. Listing 3 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. However, when a member of a collection is an object, you can then access its individual properties (and methods).

Listing 3. 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", 4447100)

oStates.Add(oState, oState.cAbbrev)

oState = CREATEOBJECT("Empty")
ADDPROPERTY(oState, "cName", "Alaska")
ADDPROPERTY(oState, "cAbbrev", "AK")
ADDPROPERTY(oState, "nOrder", 49)
ADDPROPERTY(oState, "nPopulation", 626932)

oStates.Add(oState, oState.cAbbrev)

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

Why collections?

When looking at the example in Listing 3, 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 more clear (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 4 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 4. 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 specified value
* or type.
IF ALEN(aArray, 1) = 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 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 5. Adding a row to an array is a three-step process.

```
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"
DISPLAY MEMORY LIKE aArray
```

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).

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 6 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. However, creating everything on the fly is simpler for an example like this.)

Listing 6. 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.

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

Collections have a few other capabilities worth knowing about. 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 7 demonstrates, using the collection of countries constructed in Listing 6.

Listing 7. The collection class's GetKey method switches between indexes and keys.

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

You can use this ability to avoid errors. If you refer to a member of a collection that doesn't exist, you get an error (error 2061). If you have the key for an item and you want to make sure the item exists, you can use code like Listing 8 to handle it.

Listing 8. GetKey can be part of a strategy to avoid errors when accessing collection members.

```
nIndex = oCountries.GetKey("France")
IF m.nIndex > 0
   oFrance = oCountries["France"]
ELSE
   oFrance = .null.
ENDIF
```

In my older business object code, I have a lot of methods that look more or less like that. My newer code simply wraps the access in TRY-CATCH, as in Listing 9.

Listing 9. TRY-CATCH offers another way to avoid errors when addressing collection members by their keys.

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

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.

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. So VFP 9 adds the FOXOBJECT keyword, which tells FOR EACH to create native objects rather than COM objects. Listing 10 demonstrates.

Listing 10. Use FOR EACH to walk through a collection and access every item. Make sure to add the FOXOBJECT keyword when working with native VFP objects.

```
FOR EACH oCountry IN m.oCountries FOXOBJECT
   ?oCountry.cName, oCountry.cContinent
ENDFOR
```

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.

There is one big negative to working with collections. The VFP Debugger does not provide good tools. You can't drill down into a collection in the either the Locals or the Watch window, 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 window.

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.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of ten books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL . Her latest collaboration is Making Sense of Sedna and SP2, coming out this year. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com