# Use the right loop for the job

## Tamar E. Granor, PhD

I came to the Fox world from Pascal. While I'd worked with several other languages over the years, Pascal was the one that I loved, that I'd taught to dozens of undergraduates and that I'd used for both my Master's thesis and my PhD.

One of the cool things about Pascal was that it had not one, not two, but three loop constructs. WHILE and REPEAT-UNTIL were similar, continuing a loop until some condition was met. The difference between the two was when the condition was checked and whether the loop stopped when the condition failed or when the condition was met. FOR provided a counted loop. What more could a programmer need?

When I started working with FoxBASE++, I found it had only one way to write a loop. DO WHILE worked just like Pascal's WHILE, but there was no counted loop construct, nor any concept of a loop that didn't check its condition until after the first pass. Of course, I could achieve the same results with DO WHILE, but I missed having other options.

As time went by, looping choices in FoxPro improved. While VFP still doesn't include an analogue of REPEAT-UNTIL, there are now no fewer than four different ways to construct a loop. So how do you know which one to use in a given situation? There are some fairly simple rules.

### Looping through tables and cursors

Back in the early days of Xbase, we had to use DO WHILE to loop through a table (there was no such thing as a cursor then), like this:

```
GO TOP
DO WHILE NOT EOF()
   * Do whatever you need to this record
   SKIP
ENDDO
```

This structure worked, but you had to remember to put SKIP at the end of the loop and if you changed work areas within the loop, you had to make sure to change back before you reached SKIP or your code would fail.

The addition of the SCAN command made looping through tables much easier. The basic SCAN loop looks like this:

```
SCAN
   * Do whatever you need to this record
ENDSCAN
```

SCAN has four advantages over DO WHILE NOT EOF(). First, unless you include the optional WHILE clause, it always starts at the top of the table.

Second, the SKIP is built in; you don't need to code it.

Third, at the end of each pass, it automatically returns to the controlling work area, whichever work area was selected when execution reached the SCAN command.

Fourth, in most cases, SCAN is faster than the equivalent DO WHILE loop. In my tests, looping through an unordered table and doing nothing else, SCAN took about 70% of the time of DO WHILE.

Things change somewhat if you use an ordered table. In the same tests, if I SET ORDER TO an index tag, SCAN's advantage changed to about 80% of DO WHILE's time. In fact, the relative times depend on the tag you use. Over the years, I have seen a few cases where using a particular index order, DO WHILE was faster by as much as 20%, but usually SCAN has the advantage.

What if you only want to work with some of the records, that is, you need to filter the data? Both loop constructs can handle this, but there are some caveats.

DO WHILE continues as long as its condition is true. When you add a condition other than NOT EOF(), the loop continues only as long as records meet that condition. If you can't order the data so that all the records you want to process are together, then you need to use an IF statement inside the loop, rather than adding the condition to the loop. For example, using the Northwind Customers table, suppose you want to work with all customers in the UK. This loop won't find them all:

```
GO TOP
DO WHILE NOT EOF() AND Country = "UK"
   * Do something with this record
   SKIP
ENDDO
```

Since there's no tag available for country, to make sure you find each UK customer with DO WHILE, you need code like this:

```
GO TOP
DO WHILE NOT EOF()
   IF Country = "UK"
      * Do something with this record
   ENDIF
   SKIP
ENDDO
```

SCAN offers a better alternative in this case; add the FOR clause, like this:

```
SCAN FOR Country = "UK"
   * Do something with this record
ENDSCAN
```

Sometimes, you can order the data so that all the records you want to process are together. In that case, stay away from SCAN FOR and use SCAN WHILE instead; in general, it will be much faster, since it only visits the matching records. This example visits every record in the Nothwind OrderDetails table for a particular product:

```
SELECT OrderDetails
SEEK m.nProductID
SCAN WHILE ProductID = m.nProductID
   * Do something with this record
ENDSCAN
```

In my tests, SEEK followed by SCAN WHILE is the fastest, but SEEK followed by DO WHILE is nearly as fast.

A final note on looping through tables: Often, there's actually no reason to do processing in a loop. Most of the Xbase commands in VFP accept FOR and WHILE clauses, so that all relevant records can be processed in a single command. Save loops for those times when you need to do more complex processing.

## Counted Loops

More often than looping through a table, I need to write a loop that executes a fixed number of times. DO WHILE is up to the task, with a structure like this:

```
nCount = m.nStart
DO WHILE m.nCount <= m.nEnd
   * Do whatever you need to
   nCount = m.nCount + 1
ENDDO
```

Once again, though, VFP offers a better alternative, the FOR loop. Instead of the code above, use code like this:

```
FOR m.nCount = m.nStart TO m.nEnd
   * Do whatever you need to
ENDFOR
```

As with SCAN, FOR lets you omit the statement that keeps the loop moving; the loop variable is incremented automatically. Also, like SCAN, FOR is faster than DO WHILE. In this case, the difference is more than an order of magnitude. In my tests, using an otherwise empty loop, DO WHILE takes 10 to 13 times longer than FOR.

There are a couple of things to be aware of with FOR. First, the end value is evaluated only once, when you enter the loop. That is, if you use a variable or expression to specify the last value, and do something in the loop that changes the value of the variable or expression, the number of passes doesn't change. For example, if you write this:

```
nStart = 1
nEnd = 200
FOR m.nCount = m.nStart TO m.nEnd
   * Do whatever you need to
   nEnd = 100
ENDFOR
```

the loop still executes 200 times. You can't short-circuit it by changing the end variable; use EXIT instead.

Second, FOR has an optional STEP clause that lets you count by something other than ones. You can specify any positive or negative number; it doesn't even have to be an integer. So you could write code like this:

```
FOR nValue = 0 TO 1 STEP .1
   * nValue will be 0, 0.1, 0.2, etc.
ENDFOR
```

VFP is smart enough that it tests whether you've passed the endpoint, rather than testing whether you've exactly matched it. So you can even write:

```
FOR nValue = 0 TO 5 STEP .3
   * nValue will be 0, 0.3, 0.6, etc.
ENDFOR
```

This loop stops when nValue = 5.1; during the last pass through the loop, nValue = 4.8.

The ability to specify negative numbers means you can work backwards. In that case, be sure to make the start value larger than the end value. For example, you might write a loop like this:

```
FOR nValue = 500 TO 0 STEP -50
   * nValue will be 500, 450, 400, etc.
ENDFOR
```

Unless you need to test additional conditions, there's no reason ever to use DO WHILE for a

counted loop. Even if you have additional conditions to test, you may be better off using IF and EXIT inside the loop.

## What is DO WHILE good for?

If DO WHILE isn't the best choice for looping through tables and cursors or for counted loops, when is it appropriate? When you need to do something until a condition changes, that is, exactly for the cases its name implies.

Most of the DO WHILE loops I write look something like this:

```
lFound = .F.
DO WHILE NOT m.lFound
   * Do something that sets lFound
ENDDO
```

For example, in a class that generates test data, I need to create a unique ID number that's random rather than ordered (to replicate the real world). I use this loop:

```
lNewNum = .F.
DO WHILE NOT m.lNewNum
   nNumber = This.RandInt(10000000, 99999999)
   cNumber = TRANSFORM(m.nNumber)

   * Check whether it exists already
   IF NOT SEEK(m.cNumber, "__StudNums",;
               "cNumber")
      lNewNum = .T.
      INSERT INTO __StudNums ;
             VALUES (m.cNumber)
   ENDIF
ENDDO
```

The RandInt method returns a random integer between the parameters supplied. Then, I search the list of ID numbers already generated. If this one isn't there, I add it and set the lNewNum flag to .T. to end the loop.

## Looping through collections

VFP has one additional looping construct that wasn't needed back in the old days. Both VFP itself and the many Automation servers it can talk to use collections to hold sets of similar items. For example, on a VFP form there's a collection called Objects that contains an object reference to each control on the form. When automating Word, you can talk to its Documents collection or to an individual document's Paragraphs collection. VFP has built-in collections for forms.

Although you can traverse a collection using a FOR loop, VFP also supports the FOR EACH loop, designed specifically for walking through collections. To go through a collection with a FOR loop, you use code like this:

```
FOR nItem = 1 TO ThisForm.Objects.Count
   oObject = This.Objects[m.nItem]
   * Do what you need to with oObject
ENDFOR
```

The analogous FOR EACH loop looks like this:

```
FOR EACH oObject IN ThisForm.Objects FOXOBJECT
   * Do what you need to with oObject
ENDFOR
```

There is one big difference. In my experience, although the two loops process the items in the same order, you can't always count on it. FOR EACH promises only to visit each item in the collection; it doesn't make any guarantees about the order in which they'll be processed. In my tests, FOR EACH is about twice as fast as FOR.

The FOXOBJECT keyword needs some explanation. Prior to VFP 8, there were only a few collections native to VFP, like the form's Controls collection and the grid's Columns collection. Most of the collections you needed to deal with, including some that appeared to be native like the Projects and Files collections, were actually COM objects. As a result, FOR EACH was designed to work with COM objects. By default, the object it hands you each pass through the loop is a COM object.

In VFP 8, the Collection base class was added, giving us the ability to create our own native collections. Suddenly, having FOR EACH provide COM objects caused problems. Those objects didn't behave the way we expected. Not only that, but FOR EACH loops were slow.

The FOXOBJECT keyword was added in VFP 9. When you add it to FOR EACH, the objects you're working with inside the loop are native VFP objects. Using FOXOBJECT, not only do the objects behave as expected, but FOR EACH without FOXOBJECT takes about 10 to 20 times as long as FOR EACH with FOXOBJECT. The bottom line is that when working with a native collection, you should always add FOXOBJECT to FOR EACH.

There is one situation where you must use FOR rather than FOR EACH. That's when you're removing items from the collection inside the loop. Assume oColl is a collection containing some items, where each item has a cKey property, indicating its key in the collection. Consider this code to delete all the items from the collection, one by one:

```
FOR EACH oItem IN oColl FOXOBJECT
   oColl.Remove(m.oItem.cKey)
ENDFOR
```

In fact, only half the items get removed. Internally, VFP must use a pointer of some sort to keep track of its position in the collection. When you remove an item, you mess up the internal pointer.

A FOR loop, running backwards through the

collection, solves the problem:

```
FOR nItem = oColl.Count TO 1 STEP -1
   oItem = oColl[m.nItem]
   oColl.Remove(m.oItem.cKey)
ENDFOR
```

Of course, to remove all items from a collection, you can simply pass -1 to the Remove method, so this loop, as written, is unnecessary. However, the same principle applies to a loop where you're doing some testing to determine whether to remove an item.

A final note: FOR EACH can be used with arrays as well as collections. However, I've never found a reason to do so. With an array, I generally like the guarantee of processing items in order.

## Happy looping

Learning to use the right loop for the situation will make your code faster and more readable. Both of those goals are worth breaking old habits and building new ones.

The Downloads for this article include programs to test the speed differences between DO WHILE and SCAN, DO WHILE and FOR, and FOR and FOR EACH (with and without FOXOBJECT).

*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 nine books including the award winning Hacker's Guide to Visual FoxPro and Microsoft Office Automation with Visual FoxPro. Her most recent books are Taming Visual FoxPro's SQL and What's New in Nine: Visual FoxPro's Latest Hits. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Certified Professional and a Microsoft Support Most Valuable Professional. Tamar speaks frequently about Visual FoxPro at conferences and user groups in North America and Europe, including every FoxPro DevCon since 1993. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*