The Scope of Things

Using the right scope for variables makes your code stronger.

Tamar E. Granor, Ph.D.

When I started using FoxBase+, there were two kinds of variables available: public and private. If you did nothing, any variables you used in code were private. Since that was the most restrictive scope available, it was easy to get into the habit of not declaring variables unless they were public.

VFP 3 changed the rules by introducing local scope for variables. Local is more restrictive than private, but has to be explicitly declared. What do the three scopes mean and how do they work?

Public variables are available everywhere unless you define a private or local variable with the same name at a lower level (in earlier versions of Fox, you couldn't redefine a public variable without releasing it first, but that's no longer true). To define public variables, use the PUBLIC keyword. The variables are created and set to .F.

Private variables are available in the routine where they're created and any routine called by that one (all the way down the call chain) unless something in the calling chain declares a local or private variable with the same name. To declare a private variable, use the PRIVATE keyword. However, unlike PUBLIC and LOCAL, doing so does *not* create the variable; it just reserves the name as private. You still have to actually assign a value to it, in order to create it. Listing 1 (included in this month's downloads as PrivateDoesNotCreate.PRG) demonstrates this. The variable cPrivate, declared private, but not created in the main program, is not the same cPrivate as in the subroutine. When you attempt to display the value of cPrivate after the call, an error is generated.

Listing 1. Declaring a variable PRIVATE doesn't create the variable.

```
* Declaring a variable private doesn't create
* the variable, it just reserves the name as
* private.

PRIVATE cPrivate

DO SubProc

ON ERROR WAIT WINDOW MESSAGE()
* The next line generates an error
?m.cPrivate
ON ERROR

RETURN

PROCEDURE SubProc

* The variable here is private (because it's
* not declared) but it's not the same variable
* as in the main program

cPrivate="abc"
?m.cPrivate

RETURN
```

Local variables are available only in the routine where they're declared. To define a local variable, use the LOCAL keyword. The variable is created and set to .F.

The best practice is to use local variables for everything, unless you can identify a specific need for a broader scope. Most VFP experts recommend using only a single private variable in applications. That variable (often called goApp) holds an object reference to an application object, and is declared private in the application's main program. Defining the variable as private in the main program is sufficient to make it available throughout the application.

The problem with using public or private variables is that they make it too easy for one piece of code to accidentally break another. If one program (or class or form or report) depends on a particular variable that isn't passed as a parameter, and another program (or class or form) changes that variable, the first routine may no longer work.

By using only local variables and requiring all communication between routines to happen either through parameters or through properties of objects passed as parameters, code in one part of an application is protected from other parts of the application. (This is, in essence, what encapsulation is all about).

The application object is typically the exception here, and is often used to communicate between different parts of the application. Even so, it's best if other parts of the application check for the application object's existence and work through its methods, rather than modifying application object properties directly. Limiting non-parameter communication to that provided by the application object also means that you know where to look when problems occur.

Arrays and scope

Like other variables, arrays can be public, private or local. But unlike other variables, arrays have to be declared. To create a public or local array, you declare it like any other variable, just adding the dimensions, such as PUBLIC aPublic[2,4] or LOCAL aLocal[17]. In both cases, there's an optional ARRAY keyword.

To declare an array private, however, the PRIVATE keyword isn't sufficient. Instead, you use either the DIMENSION or the DECLARE keyword, as in Listing 2. Of course, you should choose one keyword or the other and use it every time.

Listing 2. To create a private array, use either DIMENSION or DECLARE.

```
DIMENSION aPrivate[10,3]
DECLARE aAlsoPrivate[100]
```

There's one confusing item. Once you create an array with whatever scope you choose, issuing DIMENSION or DECLARE simply reshapes it; these commands don't change the array's scope. This actually makes sense when you consider what's really happening. If you use DIMENSION or DECLARE for an array that doesn't already exist, it's the same as using any other undeclared variable; it gets created as private. However, if you've already established the array's scope, it keeps that scope.

The same rule applies to the arrays created by VFP's "A" functions. There's a large set of functions (whose names all begin with the letter "A") that retrieves some information and puts it into an array. For example, APrinters() gets the list of available printers, while AFields() gets the list of fields in a table. For all these functions, if the array already exists, it keeps its original scope. If the function creates the array, it's declared private.

As with scalar variables, avoid public arrays entirely, and keep private arrays to a minimum. Declare array variables local.

Passing parameters

There are two issues related to parameters where things have changed over the years: scope and figuring out how many parameters were passed.

In early Fox days, parameters received by a function or procedure were always private. That is, the command PARAMETERS x, y, z creates x, y and z as variables private to the routine. When local scope was added in VFP 3, we also got the LPARAMETERS command, which lets you indicate that the variables created to receive parameters should be scoped as local. Listing 3 shows both forms.

Listing 3. PARAMETERS defines private variables, while LPARAMETERS defines local variables. LPARAMETERS is a better choice.

```
PROCEDURE HasPrivateParams
PARAMETERS nFirst, cSecond

PROCEDURE HasLocalParams
LPARAMETERS nFirst, cSecond
```

You can also define parameters without using either keyword by including them in the procedure, function or method header, as in Listing 4. Parameters declared this way are local.

Listing 4. Parameters listed in the header are local.

```
PROCEDURE ParamsInHeader(cSomething, ;
                         nSomethingElse)
```

Making parameters local is always a better choice than making them private, for the same reasons that local variables are better than private variables.

Counting parameters

Occasionally, a routine needs to know how many parameters were actually passed to it, so that it can behave appropriately. FoxPro has had the PARAMETERS() function since the early days to provide that information. In FoxPro 2.6, the PCOUNT() function was added "for dBase compatibility."

Unlike most of the functions so tagged, however, the dBase version is superior to the Fox version. The value returned by PARAMETERS() is reset every time another routine is called, including when ON KEY LABEL fires. So unless you grab the value as soon as you get into a routine and store it, the result can be wrong. In fact, because an ON KEY LABEL can fire at any time, even storing the return value of PARAMETERS() as the first executable line of a routine can occasionally fail. PCOUNT(), on the other hand, always returns the number of parameters passed to the currently executing routine.

Listing 5 demonstrates the difference between the two functions; this program is included in the downloads as CountingParameters.PRG. When you run this code, you get the output shown in Listing 6.

Listing 5. Use PCOUNT() rather than PARAMETERS() to determine how many parameters were passed.

```
* Demonstrate PARAMETERS() vs. PCOUNT()

Subproc("abc", 123)

RETURN

PROCEDURE Subproc(cParm1, nParm2)
? "Immediately on entry to Subproc"
? "  PARAMETERS() returns ", PARAMETERS()
? "  PCOUNT() returns ", PCOUNT()

Subsubproc()

? "After call to Subsubproc"
? "  PARAMETERS() returns ", PARAMETERS()
? "  PCOUNT() returns ", PCOUNT()
```

```
RETURN

PROCEDURE Subsubproc

* No code needed here to demonstrate
* the point

RETURN
```

Listing 6. PCOUNT() returns the same value no matter where you call it in the routine. The result of PARAMETERS() depends on what else has happened.

```
Immediately on entry to Subproc
  PARAMETERS() returns     2
  PCOUNT() returns     2
After call to Subsubproc
  PARAMETERS() returns     0
  PCOUNT() returns     2
```

Keep it local

If you're careful writing your code, does scope really matter? It does for several reasons.

An application I'm currently working on has a timer to process incoming data. Now and then, we were seeing strange behavior. We tracked it down to having undeclared (thus, private) variables in both the straight-line code and the timer code. When the incoming data was processed, changes to variables in the processing code called by the timer were clobbering same-named variables in the other code. As soon as we made sure that all variables were declared local, the problems went away.

Even if you're not using a timer, in a modern, event-driven application, you can't predict the order in which your code will execute and thus can't be sure that code in one method won't interfere with code in another.

Another reason to aim for local wherever possible is that code you write today may be running for years to come. (Right now, I'm maintaining some applications that were written as much as a dozen years ago). Even if you have clear naming conventions and standards when you write the code, the chances are good that over time, the code quality will decline. Anything you can do up front to prevent problems down the road is a good idea.

The bottom line with scope is that everything that can be local should be local. If you find yourself looking for private or public variables, revisit your design to see where you should be passing parameters or adding things to your application object.

Author Profile

Bio: 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.