FoxTalk
PINNACLE
Solutions for Microsoft® FoxPro® and Visual FoxPro® Developers

Visual Basic for Dataheads

The Basics of Writing Code

Whil Hentzen

This month, I'll cover more details on the Visual Basic 6.0 IDE, tighten up on the terminology for various parts of the IDE, and discuss how code is constructed and scoped.

LAST month, after some hemming and hawing about why I was covering the dreaded "Visual Basic" in *FoxTalk*, I introduced the VB IDE and showed you how to write and compile your first VB program. Now it's time to delve deeper into the tools you'll use every day in VB.

The Properties Window, Property Pages, and Toolbars

If you open the VB Properties Window, you'll see a drop-down list box just like in VFP—it's called the "object box" and, like VFP, contains a list of all of the objects in the currently selected form. However, if you pop it open, you'll notice that all of the objects on the form, as well as the form itself, are listed in alphabetical order—and there's no apparent hierarchy involved. I'll discuss this in more detail in the next section.

The Properties Window works like VFP's: Select an object, and the associated properties and their values show up in the list box below. The two tabs above the list box allow you to choose whether you want to see the properties in alphabetic order or by category.

If you select the Categorized tab, you'll see a third column appear on the left side of the list box—a series of plus and/or minus signs that allow you to expand or contract the list of properties for that category. I'll probably get around to describing a bunch of "tips and tricks" for various control properties later, but here's one that you'll likely want right away. In VFP, you use the "\<" character string in a command button's caption to set the next character to act as a hot key; in VB, you use the ampersand (&) character. Thus, VB would use the text string "&Done" to accomplish the same task as the VFP caption "\<Done".

Here's another difference between VB and VFP. Right-clicking on a form displays a context menu with a Properties menu command, and selecting the Properties menu command will open the Properties Window; but right-clicking on a control in a form and selecting the Properties menu command will bring up a "Property Page." This Property Page contains a tabbed dialog box that contains various physical or visible attributes of the control. More on this later.

Visual Basic's control toolbar is called the "Toolbox," and, as we saw last month, you can customize it by adding non-default Microsoft or third-party controls to it. First, select (or create) the tab (the gray object that you and I would have called a "bar") under which you want to add the control. (Remember that you can add a new tab by right-clicking in the Toolbox and selecting the Add Tab menu command.) Next, right-click under the tab, select the Components menu command, and use the resulting dialog box to select the control you want to show on the Toolbox.

Finally, remember that the Properties Window only shows properties—not methods or events. I'll discuss those shortly when I get to programming.

VB's version of containership

Earlier in this article I mentioned that there's no hierarchical display of objects in the Properties Window object box. This is because VB doesn't have the same type of containership as there is in VFP—and, while confusing, this actually makes life easier in some cases. For instance, if you put a check box on page two of a page frame in VFP, you use the following code to reference its caption:

```
thisform.pageframe1.page2.checkbox.caption
```

In order to reference the same check box's caption in VB, you simply use the following code:

```
check2.caption
```

This tells us a couple of things. First of all, you'll wear down your keyboard less, since you have to do much less typing. Second, every control on a form has to have its own unique name—regardless of where it lives. Thus, while you can have a "checkbox1" on each page of a four-page tabbed dialog box in VFP, you'd have "check1," "check2," "check3," and "check4" in VB. Third, the containership, or lack thereof, is going to have some serious ramifications on programming that we'll see later.

Slinging code

As I just mentioned, when you write code, you don't have to use a fully qualified name for a control—simply the name of the control and its property or method name. Second, in order to get to the code, you can either press the View Code button or just double-click on the form. But here's where it can get confusing. Remember that you can use the icon in the bottom left of the window to display all the code for the form, or just the code for the current object (either the form, or a control on the form). When you double-click on a control that has no code attached to it, you'll get a new module for the Click event of that control already built for you, as shown in Figure 1. Otherwise, double-clicking on a control with code in one of its events will bring up the code for that event.

The code window can contain code for one or more events, for one or more controls. Each "chunk of code" attached to an event has two parts. The first is the declaration of the module, and the second is the termination of the module, as in the following code:

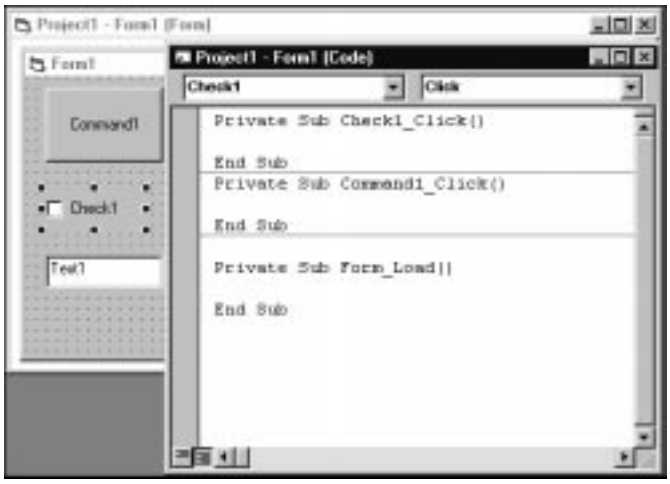

Figure 1. Double-clicking on a control formerly without code opens up a new module for the Click() event of that control.

```
Private Sub Form_load()
<your code goes here>
End Sub
```

Each of these "chunks of code" is referred to as a subprocedure, or a function, or, generically, a code segment. The "Sub" keyword means that this is a subprocedure—much like procedures in Fox. (I'll discuss the difference between procedures and functions shortly.) The "Private" keyword means that this subprocedure scope is the form—it can be called by code elsewhere in this form, but not anywhere else in the project.

The subprocedure's name consists of the name of the control and the event to which the module is attached. Remember last month's warning: Putting code in a module and then changing the controls' name will result in the module being orphaned.

The code window also has a section called "General" (it's the first entry in the object drop-down at the top of the code window). You can use this area to create your own subprocedures and functions, much like form-level methods in Fox.

The parentheses following the name of the control enable you to pass parameters to the code segment. For example, suppose you have a form with a check box and a command button on it. You could use the following code to pass the caption of the control to a form-level function that would display the caption whenever the control was clicked.

```
Private Sub MySub(xmessage)
 MsgBox (xmessage)
End Sub

Private Sub Check1_Click()
MySub (Check1.Caption)
End Sub

Private Sub Command1_Click()
MySub (Command1.Caption)
End Sub
```

In this brief example, you can see that the same basic <groan> programming principles apply—calling a subroutine with a parameter. And since MySub is declared as private, you can't go off and call that very valuable subprocedure from another component in the project.

Procedures vs. functions

The difference between subprocedures and functions in Visual Basic is the same as in VFP—subprocedures don't return values; functions do. You know the rest. Well, actually, you don't know all the rest. You define a function (a user-defined function, of course—VB also comes with its own set of functions that work just like in any other language) like so:

```
Private Function AlphabetizeName(FirstName, LastName)
AlphabetizeName = LastName & " " & FirstName
End Function
```

```
Private Sub Command1_Click()
 MsgBox (AlphabetizeName(Check1.Caption, Check4.Caption))
End Sub
```

The AlphabetizeName function takes two parameters as input, reverses them, and concatenates them with a space between them. The Command1 button takes the captions of two check boxes on the form and passes them to the AlphabetizeName function. The result of the AlphabetizeName function is then sent as a parm to a MsgBox function as the message to display.

The most interesting thing to note about VB functions is how to return values. The AlphabetizeName function has a variable with the same name as the function that receives the value to be returned to the caller. Once you've got that down, the rest is easy.

Next topic, please.

Option Explicit

Visual FoxPro is a "weakly typed" language, which means that you can assign any old value to any old variable, and then change the value of the variable any old time you want. This provides terrific flexibility, but at the same time, it exposes you to untold amounts of danger, which you're well aware of.

Visual Basic has this same capability—weakly typed variables—but you can also force VB to make you define what variables you're going to use, and what data type they'll be. This is called "strong typing" and is done by checking the Require Variable Declaration check box in the Editor tab of the Tools, Options dialog box.

After you do so, every new code window you open up will have the keywords "Option Explicit" automatically entered in the General code segment. All variables in the form will need to be declared before you can use them.

In order to declare a variable, it would be helpful to know what data types Visual Basic has, wouldn't it? They're shown in **Table 1**.

Table 1. Variable types in Visual Basic 6.0.

Type	Description
Boolean	Logical
Currency	Floating point number with four decimal places
Date	Date and/or Time
Double	Large floating point (decimal) number (in the gazillions)
Integer	Small integer
Long	Large integer
Single	Small floating point (decimal) number (in the billions)
String	We know these as character strings

To declare a variable, use the Dim keyword, like so:

```
Dim bIsAlive as Boolean
Dim cNetPay as Currency
Dim dBirth as Date
Dim fNationalDebt as Double
Dim iCounter as Integer
Dim lWorldCitizens as Long
Dim fTaxRate as Single
Dim sName as String
```

You can also use the Private and Public keywords to control the lifetime of the variable; Private variables have scope throughout the routine, form, or module they were declared in, while Public variables are available to any code anywhere in the project in which they're declared.

```
Public sName as String
Private sName as String
```

I'll beat on this for a bit, because strongly typed variables are a significant difference between VB and VFP. Once you have an Option Explicit declaration in your code (and you can type it in manually in the General code segment if you didn't check the Require Variable Declaration check box), you must declare every variable you use.

This prevents two undesirable behaviors from occurring. First, you can't accidentally create a bug by mistyping the name of a variable. In the following code, we're trying to assemble a message, sMessage, from consecutive values of the string sNewValue. However, sMessage is never modified, because the second-to-last line assigns the concatenation of the existing string, sMessage, and the new value, sNewValue, to a mistyped variable, aMessage:

```
sMessage = ""
For iCounter = 1 to 10
 <some code>
 sNewValue = <some function>
 aMessage = sMessage + sNewValue
End For
```

If you included an Option Explicit declaration in your code, you'd have to declare sMessage, iCounter, and sNewValue, and the compiler would detect an undeclared variable, aMessage. Bet you've done that once or twice in Fox, haven't you?

Second, by declaring the data type, you won't accidentally change the data type of a variable. You've all done something like this:

```
dBirth = date()
<some code>
dBirth = 0
```

Strong typing of your variables will prevent this from happening again.

Adding your own methods to forms and applications

Earlier in this article, I showed how you can add your own methods—functions or subprocedures—to a form by simply adding them in the General section of the code window. Even if declared Private, those functions and

subprocedures are now available throughout the form.

However, you'll also want to create functions and subprocedures that are available globally throughout your application, much like common code in an .APP procedure file in Fox. In VB, these types of files are called modules. They're text files with .BAS extensions, and they live in the Project Explorer alongside forms and other components. In order to create a module, open the Project Explorer (you can do so through the View, Project Explorer menu command), right-click in the window, and select Add, Module from the context menus that display (see Figure 2).

You'll be prompted for the module to add—either an existing one or a new one, as shown in Figure 3.

The module will be added to the Module node of the Project Explorer, as shown in Figure 4.

To create a subprocedure that's available in other parts of your project, simply create it like you did in a form:

```
Sub Herman()
 MsgBox ("This is the Herman sub in the first module")
End Sub
```

Then, to call the Herman subprocedure, use code like the following that's in the click event of the Command2 command button on a form in the project:

```
Private Sub Command2_Click()
Call Herman
End Sub
```

Notice that the Call keyword was required this time in order to run the subprocedure. This can vary, as you've seen earlier, depending on how you structure the syntax of the subprocedure and the way arguments, if any, are passed to it. Me? I like to use functions as much as possible so that I can return a value that indicates whether the execution of a procedure was successful or not, so I don't worry about the Call syntax at all.

You can also create a subprocedure in a module that serves as your startup program for your application. In order to do so, name it "Main." Then, if your project also has forms, you'll need to tell Visual Basic that it should run the Main subprocedure first, instead of the first form created in the project. To do so, right-click in the Project

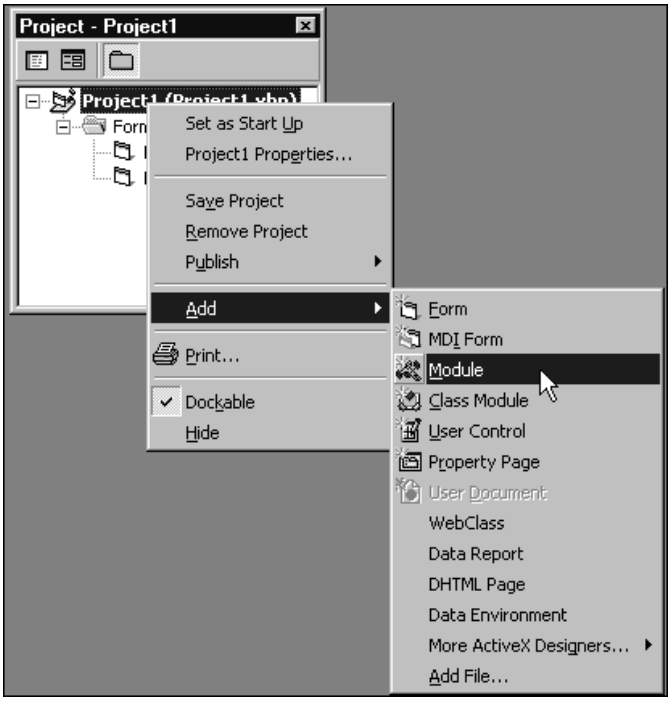

Figure 2. Add a module to a project through the Project Explorer context menu.

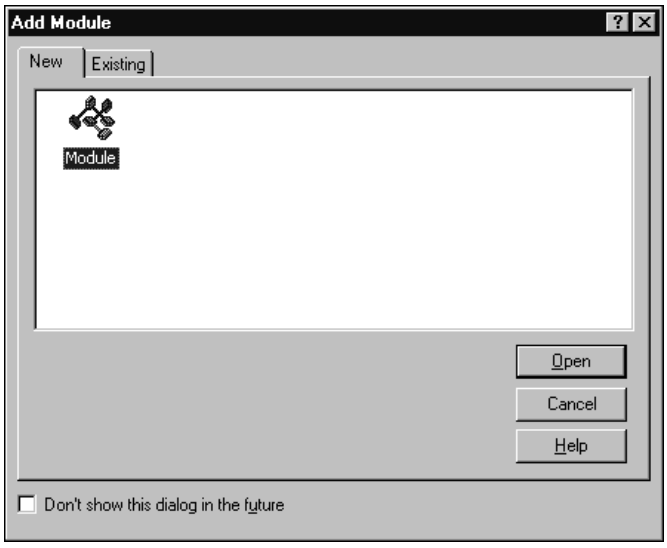

Figure 3. You can choose to add a new module or an existing one to your project.

Figure 4. The Module node of the Project Explorer holds all modules for this project.

Explorer and select the ProjectN Properties menu command (where "ProjectN" is the name of your project). Open the Startup Object drop-down in the General tab, and choose Sub Main (see **Figure 5**).

That's about it for this month—we've covered a lot of fundamentals about building programs and determining where all the pieces go, and now it's time for the next big chunk—commands and functions—in next month's column. Stay tuned! ▲

Stupid VB Tricks

Differences between VB and VFP and inconsistencies in the VB environment that can "gotcha" if you're not careful or observant. Only one this month:

Stupid VB trick #4

If you double-click on a control in order to open the code window for that control, the procedure for the Click() event of that control will be displayed if there wasn't any code attached to that control to begin with. However, if you select that control, right-click to bring up the context menu, and select the View Code menu command, the same behavior doesn't occur.

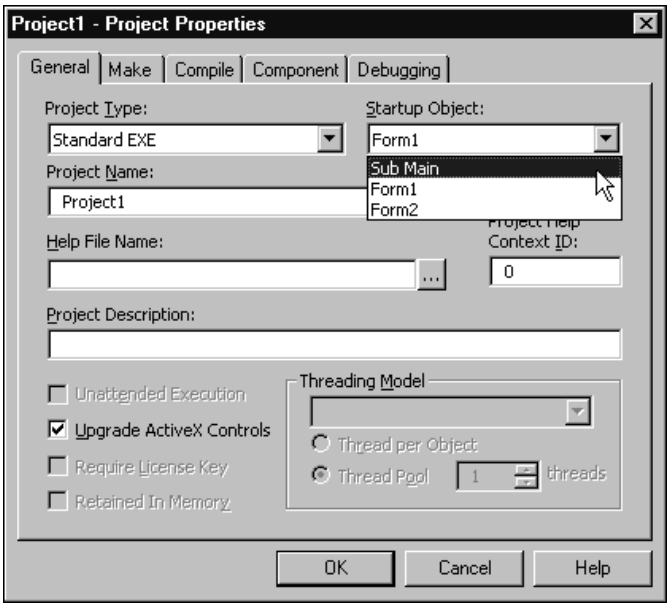

Figure 5. Set the Startup Object in the Project Properties to the Sub Main subprocedure in order to make it run as the first component in your application.