Visual FoxPro
The Fox Hunt

By Whil Hentzen

Last month's column discussed the concept of classes, the terminology, and the underlying mechanism of how they work. We used objects that were visual in nature, such as checkboxes, option buttons (radio buttons), and forms as examples. However, classes are not limited to visual objects and controls such as those we place on forms - there is a whole other type of class - a non-visual class.

As we learned last month, a class has properties and methods. Properties are values - they are similar to the variables we used in procedures and functions in FoxPro 2.x, while methods are subroutines or chunks of code, similar to functions or procedure calls in a function library in FoxPro 2.x. For example, a command button class has, among others, the properties of height, width, fontname, and visible. Each of these properties can be modified, simply by storing a new value to the property. This command button class also has methods, such as click(), valid(), and lostfocus(). You can place your own code in these methods, and that code will be executed when the method is fired. The method can be fired either by an explicit call, such as when you include the call to the method in your own program code, or when the event associated with that method is fired.

Given this foundation, the concept of a non-visual class can be hard to grasp, but it's important to understand how it works and what it might be used for. The sooner you internalize the concept of classes - both visual and non-visual - the better off you'll be when you're developing applications.

You can think of a non-visual class much like a function library, only much better. Just like you can SET PROCEDURE TO a procedure file, and then access the chunks of code inside, you can SET CLASSLIB TO a class library that contains non-visual classes, and then access the methods of the desired class. But you can go further. Since a class contains properties, you can also access - read and write - those properties. Thus, you've got a repository where you can store values at the same time that you can access functions. And the best part is that, if you've designed your classes correctly, you can extend an existing class by subclassing it - something that a regular old function library couldn't even dream of.

Let's go through an example of a non-visual class, so that you have something real that you can get your arms around. The class we're going to create will handle our environment - saving the existing settings of the various SET commands, changing the settings to those that our application requires, and then restoring those original settings when the application closes up. This month, we're going to create a non-visual class, learn how to add properties and methods, and then instantiate the class in preparation for dealing with our ENV class.

In the olden days, our top-level program would look like this:

```
* IT.PRG

*
* "talk" status is a special case
*
if set("talk") == "ON"
  set talk off
  m.gcOldTalk = "ON"
else
  m.gcOldTalk = "OFF"
endif

* (lots of these commands)
m.gcOldDelete = set("DELETE")
m.gcOldExact = set("EXACT")
m.gcOldSafety = set("SAFETY")

* (lots of these commands)
set delete on
set exact off
set safety off

[main part of application]
```

```
* (lots of these commands)
set deleted &gcOldDelete
set exact &gcOldExact
set safety &gcOldSafety

* <eof>
```

Two of the results of this methodology are (1) a whole bunch of global variables that we never use, and (2) if we find another SET command that we have to deal with, we have to manually change the main program of every application in order to incorporate the change. Not killers, but not quite right either.

We're going to create a class called ENV, and create methods in this class that will handle the saving of the original SETtings, the SETting of the environment the way we want it, and the restoration during shutdown. We're also going to store all of these original settings as properties of the class.

Let's look at the actual mechanisms required to create a class, to add methods and properties to a class, to instantiate a class, and how to call methods and reference properties of a class.

In order to create a new class, use the CREATE CLASS command (or select the File, New, Class menu option). Once you're presented with the New Class dialog box, enter the following values:

and press OK. You'll see a new window on the screen called the Class Designer window, and you'll probably see several other new objects, including the Class Designer toolbar and the Properties window. You've also created a file on disk called BASEAPP.VCX that contains a record for the ENV class. (You will probably find it very informative to open up BASEAPP.VCX as a regular table - USE BASEAPP.VCX - and browse the contents. Be careful - don't touch anything - but you'll see the various fields that define the class and class library.)

Since this is a custom class, there isn't much to it. The Red Square-Blue Circle - Green Triangle graphic in the BASEAPP.VCX (ENV) Class Designer window is a clue that this is a non-visual class - we're not manipulating an object that we can see - checkbox or a form, for instance.

The next thing to do is add properties and methods - our own properties and methods - to our ENV class. When you created the ENV class, the Properties window probably appeared; if it didn't, you can bring it forward by right-clicking the mouse in the Class Designer window and selecting the Properties menu, by selecting the View, Properties menu option, or selecting the Properties Window icon from the Class Designer toolbar. You'll see a list of existing properties and methods in the Properties window, and what we're going to do is supplement that list.

Use the Class, New Property and Class, New Method menu options (the Class Designer window or one of it's related windows must be active for the Class menu to be available) to create names for new properties and methods. I ***strongly*** advise you to fill the description edit box in the dialog - this description appears in the bottom of the Properties window and is very useful in remembering what you had in mind when you created something.

Once you've created a property or method, you'll probably want to place a value or some code in it. You can change the value of a property by simply editing it in the Properties window (all new properties are initialized with a value of .F.), but you'll need a new tool to add code to a new method. Double-clicking on the method in the Property window will bring forward the Code window which shows the name of the class in the upper left dropdown listbox and the current method in the upper right dropdown listbox. The editing window is used to enter the actual code for the method.

For the time being, create a method of the ENV class called XXX, and then enter a command like

```
wait window "We are executing the XXX method"
```

in the Code window for the XXX method. Also, create two properties of the ENV class called cMyProperty and nMyProperty, and assign the values "HERMAN" and 12345 (numeric, not character) to the properties, respectively.

The last step in our short journey is to actually create an object from this class, and then access the properties and methods of the object. This is a little confusing, so let's issue the necessary commands, and then discuss what happened. Close the Class Designer window, saving your work in the process. Then, issue the commands

The first command is similar to the SET PROCEDURE TO command that we've all used with procedure files, and tells Visual FoxPro that we're using the BASEAPP class library. The .VCX extension isn't necessary but I put it there to help you remember what BASEAPP really is. The second command is the trickiest one. It instantiates the ENV class (that belongs to the BASEAPP.VCX class library) and creates an "object reference." The ENV class is the definition - the blueprint - for the class, but oEnv is an actual instance - a physical implementation of that blueprint. oEnv is much like a memory variable in that we can access the members of the class (the members are the properties and methods) by referring to it.

We can refer to a method of the ENV class (actually, of an object instantiated from the ENV class, right?) with the third command. This executes the XXX() method, and you'll see the wait window appear after you press Enter. We can also refer to the properties of the oEnv instance by appending their property names to oEnv, like is done in the fourth and fifth commands. The sixth command changes the value of the cMyProp property. The last two commands removes the instance and the class library pointer from memory.

I've found it very handy to keep the debug window open and to place variables like the following in it:

oEnv
oEnv.nMyProp
oEnv.cMyProp

Watch how these expressions are evaluated as you instantiate the oEnv object and then assign and changes property values.

You can see where we're heading - next month, we'll add the methods SaveSets, DoSets, and RestoreSets to our ENV class, and we'll create a number of properties, such as cOldSafety and cOldExact, for this class. Then, after we've instantiated the oEnv object in our startup program, we'll use these methods and properties do handle the various chores that our main program used to do. See you next month!