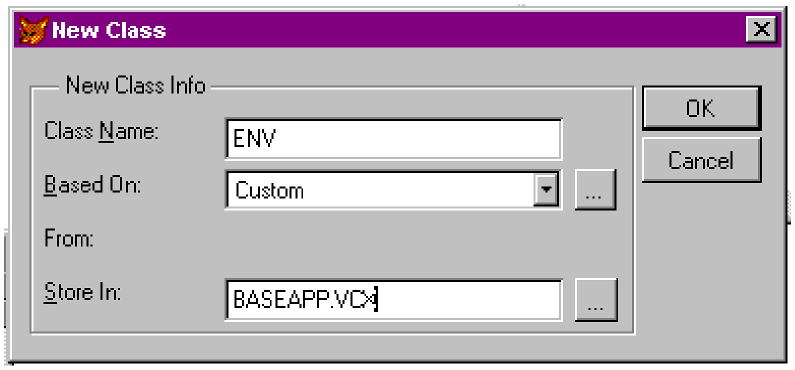
Visual FoxPro
The Fox Hunt

By Whil Hentzen

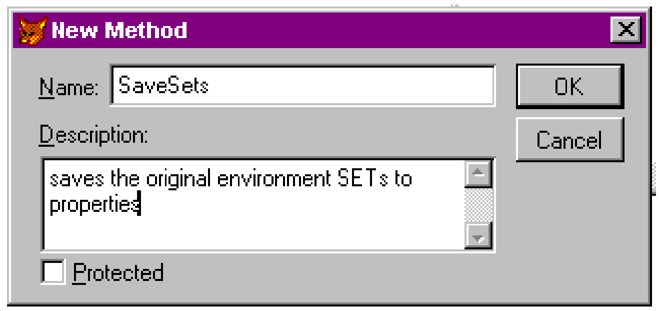
This is the third in a series of columns that discuss Visual FoxPro's classes. First, we went over the concepts and terminology, and last month we discussed the mechanism required to actually create classes, and introduced non-visual classes as an example. This month, we're going to create our environment handling class and learn how to implement it in a Visual FoxPro application

If you haven't already, create a class called ENV, and place it in the class library BASEAPP.



Class Name: ENV
Based On: Custom
From:
Store In: BASEAPP.VCX

We're going to create methods for saving the current environment settings, setting the environment to the way we want, and restoring the original settings back. These methods will be called SaveSets, DoSets, and RestoreSets. In order to add a method, select the Class, New Method menu option to bring forward the New Method dialog. Enter the name and a description (I know, I know, you're tempted to skip the description, because, as we know, "Real programmers don't comment their code." I recommend you reconsider this practice when creating classes. You will thank me many times over if you get into the habit of documenting your classes when you create them.)



Name: SaveSets
Description: saves the original environment SETs to properties
Protected

Finally, create properties for our class. These properties will hold the values of the original environment settings - they'll take the place of the "m.gcOldCent" style memory variables. Use the Class, New Property menu option and create the following properties:

```
cOldCent
cOldClas
cOldDele
cOldEsca
cOldExac
cOldExcl
cOldMult
cOldProc
cOldRepr
cOldSafe
cOldStat
cOldTalk
cOldHelp
cOldReso
cOldOnEr
```

After you created the ENV class, a Class Designer window appeared, much like the Form Designer window that you've undoubtedly already run into. If you right-click on the Class Designer window, you'll get a shortcut menu where you'll see the Code menu option. Selecting this menu option will bring forward the Code Window with all of the methods - those native to the class as well as any that you've created - available through the Procedure dropdown listbox in the upper right corner.

Place the following code in the appropriate methods:

SaveSets:

```
this.cOldCent = set("century")
this.cOldClas = gcOldClas
this.cOldDele = set("delete")
this.cOldEsca = set("escape")
this.cOldExac = set("exact")
this.cOldExcl = set("exclusive")
this.cOldMult = set("multilocks")
this.cOldProc = set("procedure")
this.cOldRepr = set("reprocess")
this.cOldSafe = set("safety")
this.cOldStat = set("status bar")
this.cOldTalk = gcOldTalk
this.cOldHelp = set("help",1)
this.cOldReso = sys(2005)
this.cOldOnEr = on("error")
```

DoSets:

wait wind nowait "Setting up environment..."

```
set century on
set clock status
set deleted on
set escape on
set exact off
set exclusive off
set multilocks on
set reprocess to 5
set safety off
*\\\ additional code to handle Help, Resource and Error
* will eventually go here
```

RestoreSets:

```
luTemp = this.cOldCent
set century &luTemp

luTemp = this.cOldClas
set classlib to &luTemp

luTemp = this.cOldDele
set deleted &luTemp

luTemp = this.cOldEsca
set escape &luTemp

luTemp = this.cOldExac
set exact &luTemp

luTemp = this.cOldExcl
set exclusive &luTemp

luTemp = this.cOldMult
set multilocks &luTemp

luTemp = this.cOldProc
set procedure to &luTemp
```

```
luTemp = this.cOldRepr
set reprocess to (luTemp)

luTemp = this.cOldSafe
set safety &luTemp

luTemp = this.cOldStat
set status bar &luTemp

luTemp = this.cOldHelp
if !empty( luTemp )
 set help to &luTemp
endif

luTemp = this.cOldReso
if !empty( luTemp )
 set resource to &luTemp
endif

luTemp = this.cOldOnEr
if !empty( luTemp )
 on error (luTemp)
endif

luTemp = this.cOldTalk
set talk &luTemp
```

The code in these methods may seem a little odd. What does “This.cOldCent” mean, for example?

When we instantiate our ENV class, we’ll create an object reference to it, named oEnv, much like we can refer to a field in a table by identifying the table name, and then the field that belongs to that table. The object reference is a memory variable, and we use it as a ‘handle’ to refer to the properties and methods. We do much the same thing when we reference a procedure in a procedure file with the syntax DO anyproc IN PROCFILE.

For example, we can use the syntax

```
do oEnv.SaveSets()
```

to run the SaveSets method, and we can use the command

```
m.cWhatIsCentury = oEnv.cOldCent
```

to determine the value of the cOldCent property of the oEnv class.

When we’re executing code in one of the methods of the class, such as when we run the SaveSets method, we can refer to other methods and properties like we did above, but we can also use the “this” keyword as well. “This” in this context means the current object. Thus, when we’re in the SaveSets method, we can refer to the cOldCent property of this class with the syntax

```
this.cOldCent
```

instead of somehow hardcoding the object name in the method.

This has a number of advantages. One primary advantage is that if you use the same class to create several objects (such as if you used a form class to create multiple copies of a form), you would create multiple object references, like oEnv, oTropics, oHandler, and so on. When working inside the class, the code doesn’t need to know the name of the object. It can just reference “this” and that’s that.

Let’s see how we would use our ENV class in an application. Remember that a class is much like a procedure library, and in order to use it, you have to perform two steps. The first step is to open up the class library with a SET CLASSLIB TO command, much like we SET PROCEDURE TO with procedure files. The second step is to instantiate the class and create an object reference to it. This is done with the CREATEOBJECT command. The following code references several methods in the ENV class that we haven’t created yet, and also uses a whole new application class that we haven’t discussed. The point is to show how calls to methods of an instantiated object are made inside a program.

```
* IT.PRG
* this is our main program
```

This methodology has several important ramifications. First, we can access the properties of the ENV class whenever we want to without having to hog a bunch of memory for global memory variables. As we've seen, the following syntax assigns the value of the SomeProp property to the m.cX memory variable.

```
m.cX = oENV.SomeProp
```

This means we can access any property of an object as long as that object is in memory. Similarly, we can access methods of any object in the same way. Second, we can change the behavior by changing the ENV class - we don't have to change our main program, IT.PRG. And, finally, we can subclass ENV if and when we need to.

Next month, we'll discuss some practical uses for these capabilities, and show how to use some native functionality in the object model to do some of our work for us.