Visual FoxPro
The Fox Hunt

By Whil Hentzen

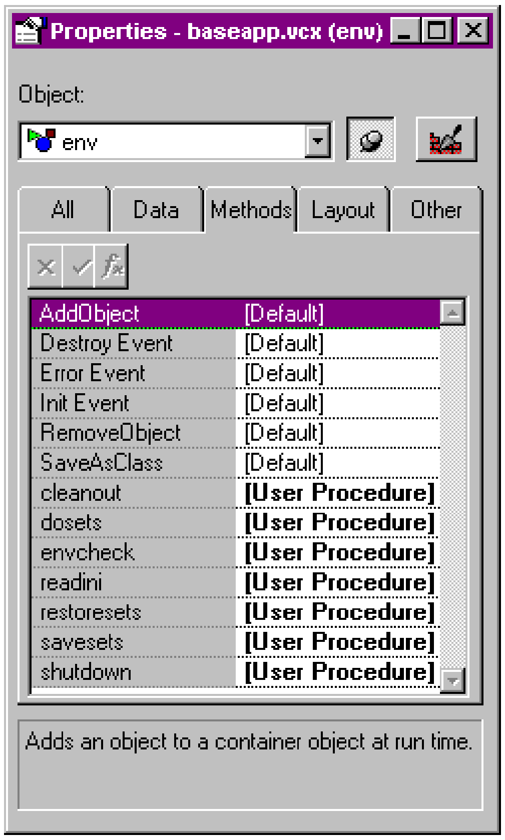
Last month, we converted a procedural startup program, IT.PRG, to one that made calls to the methods of an environment class object, as a starting point to learning object-oriented programming. This month, we're going to bring the subject to closure and leave you with some parting thoughts.

If you look at the TasTraders sample app that ships with Visual FoxPro, you're bound to get a funny feeling in the pit of your stomach. After all, a startup program that looks like

```
PUBLIC gcOldDir, gcOldPath, gcOldClassLib, gcOldEscape
gcOldEscape   = SET('ESCAPE')
gcOldDir       = FULLPATH(CURDIR())
gcOldPath     = SET('PATH')
gcOldClassLib = SET('CLASSLIB')
IF SetPath()
  PUBLIC oApp
  oApp = CREATEOBJECT("TasTrade")
  IF TYPE('oApp') = "O"
    RELEASE gcOldDir, gcOldPath, gcOldClassLib, gcOldTalk, gcOldEscape
    oApp.Do()
  ENDIF
ENDIF
```

in its entirety doesn't look like anything we've seen before. In fact, it still seems a far cry from the startup program we built last month. Some of the syntax might be more comfortable now that we've gone through the creation of an environment object and the use of methods of that object to set up an environment, but we're essentially loading and running our application with a CREATEOBJECT command and a call to the DO method of the instantiated object. This can't possibly be very robust, can it? How does our environment get handled, for example? And where does the rest of the application get handled?

The secret lies in how an object is actually instantiated. As we've seen, we can add our own custom methods to a class, and when we instantiate an object from that class, we can access those methods. However, there's more to the picture. A class by definition has a number of properties, events and methods that are native to it. For example, a non-visual class has the properties BaseClass, ClassLibrary, Comment, Name, and ParentClass. It also has the methods AddObject, Destroy, Init, RemoveObject and SaveAsClass. While most of these don't sound terribly useful right now, the Init and Destroy methods might have sparked some curiosity.



AddObject	[Default]
Destroy Event	[Default]
Error Event	[Default]
Init Event	[Default]
RemoveObject	[Default]
SaveAsClass	[Default]
cleanout	**[User Procedure]**
dosets	**[User Procedure]**
envcheck	**[User Procedure]**
readini	**[User Procedure]**
restoresets	**[User Procedure]**
savesets	**[User Procedure]**
shutdown	**[User Procedure]**

Adds an object to a container object at run time.

When an object is instantiated, the Init event fires, and the code in the corresponding Init method, if there is any, is executed. This happens automatically - no user (or programmer) intervention is necessary. Then, when the object goes out of scope, the Destroy method is executed when the Destroy event fires. Aha! Why should we spend our time writing code like

oEnv.SaveSets

when we could put the SaveSets method in the Init method of the object? And likewise, why not place all the code that needs to be executed upon shutdown in the Destroy method? This way, we take advantage of the native event model inside Visual FoxPro to do our work for us - and we don't have to worry about remembering to do it ourselves!

Now we can see why the code

```
oApp = CREATEOBJECT("TasTrade")
oApp.Do()
```

is really all that is needed to start up our application. When TasTrade is instantiated, its Init method is fired, and that's where the setup code for the program is. And when oApp goes out of scope, the Destroy method takes care of all of the necessary housekeeping. It takes a bit of getting used to, but the first time someone proposed the idea to me, the light bulb went on and the natural elegance made it comfortable immediately.

When we start discussing forms, we'll expand upon this idea of using a class's native methods and properties to our advantage. For example, forms and controls both have methods like Load, Init, Show, and Activiate. Putting our code in the proper place will let Visual FoxPro do much of our work for us.

The other idea I wanted to broach was the use of methods to access properties of an object instead of directly changing it. For example, the oEnv class has a custom property, nTimeBegin, that is used to store the system time as soon as the object is instantiated. (From the discussion above, we can see that we'd use code like

this.nTimeBegin = time()

in the Init() method of oEnv, right? We can use the 'this' prefix because we're in the oEnv object.) Later on, we might want to access this property. We could do so with a line of code like so:

```
m.nSomeReason = oEnv.nTimeBegin
```

(We'd explicitly name the object because we're probably going to be somewhere other than in the oEnv object when we make the call.) After this line of code, we now have a memory variable that has the time that the application was started. This might be useful for tracking multiple logins from the same user or watching for system crashes. However, notice that there is nothing to prevent the programmer from issuing another command, somewhere else, like so:

```
oEnv.nTimeBegin = time() - 1000000
```

Well, if we really needed to depend on the correct value of nTimeBegin, this ability to alter the value would be bad, wouldn't it? We can prevent this from happening by writing a method that is used to access the property indirectly, instead of writing to the property straight.

You may be wondering why this protects the property from being written to directly anyway, and the key word in that sentence is 'protect.' When you create a property, you can specify that this member is 'protected' (it's the small checkbox in the bottom of the dialog). This means that changes to class properties are prevented from outside the class. You use a method of the class - which is another member of the class, and thus, is regarded as being 'inside' the class, to do so instead. Thus, in order to prevent unauthorized changes to a property, mark it as protected and then control its access through the method. By the way, you can also modify a property to be protected or unprotect a previously protected property through the Class, Edit Property/Method menu option.

Well, that's enough for this month. Go back and open up the TasTraders application - it's not as intimidating now, is it? Next month, we'll examine the creation of visual classes, starting with our very own control base classes.