Visual FoxPro
The Fox Hunt

By Whil Hentzen

The description for this column is “Hunting down better ways to develop.” This is kind of a hard charter to fill for a product that’s brand new - after all, we haven’t acquired any habits yet, so we can’t possibly try to improve on them. As a result, we’re going to concentrate on getting starting the right way with Visual FoxPro - how many times have you hit yourself on the forehead, exclaiming, “I wish I knew THAT six months ago!”

Our topic for this month is the use - and non-use - of Visual FoxPro base classes. As with any new piece of software, there’s so much to learn that a natural reaction to the information overload is to try to screen out as much extraneous noise as possible. “Let’s just learn the basics, and we’ll get to the goodies later.” Many people are going to be tempted to look at the object oriented programming capabilities of Visual FoxPro as one of the “goodies” that can be ignored for a while. This is really a mistake, and so we’ll get you started with the use of classes now.

In the olden days (about six months ago), when we created screens, we placed objects on the screen by clicking on the appropriate icon in the Screen Builder and then clicking again somewhere in the screen. Once we were done with identifying prompts, memory variables, and other attributes, we had an object on the screen. A fundamental assumption was that the behavior of this object would never change. We got certain behaviors and characteristics from FoxPro, and they never changed. Where did this information come from? From some C code inside FOXPRO.EXE or FOXPROW.EXE. If we wanted a radio button to be oblong instead of round, we were basically out of luck.

The same thing holds for Visual FoxPro. When you grab a control from the Form Controls toolbar and place it on a form, that control’s behavior and characteristics come straight from VFP.EXE. However, we now have the ability to create our own set of controls, using those Visual FoxPro controls as the default. Then, when we place controls on a form, we use controls from our own set instead of Visual FoxPro’s set. This means that, should we want to change the behaviors or characteristics of a control, we can change that in our own set of controls, and this change will ripple through to all of the controls on all of the forms that are based on this control. This is truly powerful and will aid our productivity in untold ways.

Let’s look at an example. When you place a checkbox on a form, using Visual FoxPro’s base classes, the background for the checkbox in white and the caption is Check1. Similarly, the background for an option button is white, there are two buttons and the captions are Option1 and Option2. If you’ve taken a liking to the grey background of the Windows windows, you’ll find that you’re constantly changing the background color of these controls. You’re also probably getting tired of changing the captions, and adding more option buttons to an option button group.

If you subclass Visual FoxPro’s checkbox, you can change the background color of your checkbox baseclass to grey, and change the caption to something like “X.” Then, whenever you place a checkbox on a grey form using your own checkbox base class, it will automatically be grey and have an “X” caption. Should you decide later that grey backgrounds for forms wasn’t such a good idea, and you want them light yellow, you can change your checkbox base class’s background to light yellow, and all of the checkboxes created from your checkbox base class will automatically become light yellow.

The foregoing discussion has used layman’s terms for concepts that actually have technical terms. Let’s introduce those now. First, the set of controls that come with Visual FoxPro - the command button, page frame, option button, grid, and so on - are referred to as Visual FoxPro’s *base classes*. Each control is referred to as a *class*. Thus, a command button and a checkbox are both classes. When we create a control based on one of Visual FoxPro’s base classes, we are *subclassing* the class. Unfortunately, there is not a single recognized term for the result yet (although sooner or later, we’re bound to come to some sort of agreement.) For the time being, we’ll refer to the classes that we created (subclassed) from Visual FoxPro’s base classes as *our base classes*.

When we place a control on a form using either Visual FoxPro’s base classes or our own base classes, we are *instantiating* a class. “Instantiate” is the verb that goes along with the noun “instance.” Thus, the control on the form, is an *instance* of the class. If we placed two command buttons on the form, we’d have instantiated two instances of the command button class.

I’ve also used the terms behavior and characteristics to describe how an object acts and what it looks like. These are formally known as *events*, *methods* and *properties*. An event is a behavior of an object. For example, when you type a character into a textbox, the character you type appears in the textbox and the cursor moves to the next position. This event - the typing of a character - is the *keypress* event. FoxPro 2.x controls only have a couple of events - when, valid, message, and error. Much like FoxPro 2.x, you can attach code to these events. This code is known as a *method*. There are typically 20 or more events associated with a given control, and, correspondingly, 20 or more methods you can attach to an object, giving you much more granular control over how it operates.

The other side of the coin is the set of attributes of a control, such as the various appearance characteristics. These characteristics, such as height and width, color, font, caption, name (the same as the name of the variable of an object in 2.x), are called *properties*, and can be manipulated programmatically as well as visually.

Finally, remember when we instantiated a checkbox control on a form, using our own checkbox base class, and that checkbox on the form had a grey background, not white? This behavior is called *inheritance*. This is starting to sound like a lot of gobbledy-gook right now, isn't it? Well, the bad new is that there's even more - polymorphism, encapsulation, messaging, we could fill a small dictionary. But stick with it; you'll find that this terminology will become second nature after a while (I'm already trying to CREATE FORM and subclass classes in FoxPro 2.6), and you'll see it's increasingly handy as you try to describe what you're trying to do in the future.

So far, so good. But we've sidestepped around one other important concept - the actual implementation of classes in Visual FoxPro. In other words, what does a "class" look like on disk? Where does a class get it's information on what it is supposed to do? How is inheritance implemented?

When you subclass (create) a class (using the Class Designer), you'll need to specify a name for both the class, which is obvious, and for a *class library*, which may not be as obvious. Classes are stored in a file called a class library, and this file is a DBF - just like screens, reports and projects. It has a VCX extension ("Visual Class"), and the corresponding memo file has a .VCT extension.

The class consists of one or more rows in the class library table. Fields in the class's record refers to Visual FoxPro's base class. When you instantiate a class on a form, you are placing a record in the form's SCX table, just like in FoxPro 2.x. However, this record has a field that points to the class library file (the VCX), and another field that points to the specific class in the class library. Thus, when you change your class, the instantiated object on the form automatically changes.

This has been a brief look at the class mechanism in Visual FoxPro. We've glossed over some important pieces of information, and ignored other aspects that we'll eventually want to know about, but for the time being, we've got a good foundation for carrying forward. In future columns, We'll address these holes, and we'll explore how to implement classes in real world tools that we can use in our applications.