Chapter 10
Using VFP's Object-Oriented Tools

You may be tempted to skip this chapter. You've already learned how to build a menu, create some forms, and drop controls on those forms. You might be thinking, "Skip to the Report Writer section, and I'll know all I need to start building Visual FoxPro applications." Actually, that's not exactly true. You'll have all you need to know to build FoxPro 2.x applications.

Next to SQL, the most wonderful feature of Visual FoxPro is its implementation of object-oriented programming. To skip this chapter would be one of the biggest mistakes of your programming life. In this chapter I'm going to show you what object-oriented programming is, with examples that will make sense to you, a database programmer; how VFP's object-oriented tools work; and how to incorporate them into the forms that you've already learned how to build. Finally, I'll set you up with your own set of base classes that you can use to start building real-world applications—today.

A lumberjack wandered into town after being in the mountains for years and years. First thing on his list was a bath; after that, he stopped in at the general store and asked for a new ax, since the one he had was 20 years old and was getting just plain worn out. The owner asked him if he wouldn't prefer a chainsaw. The lumberjack gave him a quizzical look but reckoned that he'd like to take a look. There it was, with a bright red cover, a shiny steel handle, and hundreds of razor-sharp teeth on the chain. The lumberjack, big as he was, hefted the device in a single hand and swung it around like it was a hand ax. The store owner promised he would cut down 10 trees a day with this thing.

"Ten trees? On my best day I've never cut down more than two."

"Ten trees a day or you can have it for free," pledged the store owner.

"Well, that sounds pretty good," answered the lumberjack. "Let me have it." As he started to walk out the door, the owner stopped him, saying, "Let me show you how to use it." The lumberjack pushed him away, saying, "I think I kin figger this out myself."

A week later the lumberjack staggered back into town, shivering and shaking, clothes torn, blood on his hands and drool hanging from his chin, swearing at the store owner. "I worked my tail off this week and I could never get more than three trees down a day. It's a helluva lot better than my ax, but there's no way a man could cut 10 trees in a day."

"Only three trees?" asked the storeowner. "Hell, my grandmother could cut down 10 trees a day with that baby. Give it here and let me see what's wrong." He started it up and the lumberjack jumped back, alarmed. "What's that noise?"

So far in this book, we've seen the incredible power and flexibility of Visual FoxPro—we can create blindingly fast applications with visually appealing, sophisticated interfaces and user-friendly features galore. I'm sure that more than a few of you are already overwhelmed with the number of tools, dialog boxes, objects, choices, commands, and other minutiae to

remember. The last thing you need to hear is that we have a whole additional level of functionality to learn. However, it's true. We've skipped one feature of VFP almost completely. And it's this capability that will increase your productivity to unheard-of levels.

I'm talking about Visual FoxPro's object-oriented programming capabilities. You can use Visual FoxPro without taking advantage of its object-orientation capabilities, but it's like using a chainsaw without starting it up.

The industrial revolution started when people stopped making items one at a time and used the concept of interchangeable parts to create a large quantity of the same widget. Productivity went up, quality improved, and economies of scale were realized when the manufacturer could crank out 20 of the same item instead of custom building each one. The resulting cost savings meant a greater number of buyers, which meant increased profits, some of which were reinvested in improving the tools. However, many people don't realize that the concept of interchangeable parts wasn't invented in 1820; it had been around for centuries. But it took the ability to smelt metal to precision tolerances so a manufacturer had the ability to reliably reproduce a part to specific measurements. The concept was there but the environment wasn't ready.

The same is true for us in the software industry. The mechanism of creating interchangeable parts in software is object-oriented programming (OOP for short). Instead of writing the same code over and over, we create the mold once and then reuse it over and over for multiple applications. Of course, we've been hearing about OOP for years and years—almost as long as we've been hearing about artificial intelligence. But the promise and the delivery have been two different things. Why?

As with the development of manufacturing during the industrial revolution, the environment hasn't been ready. Now, many of you have the power of a small Cray sitting under your desk in the form of a 256 MB, 10 GB Pentium III with a 21-inch flat-screen monitor and a color graphical user interface. True object-oriented programming requires a lot of horsepower. We now have it.

The basic idea behind the promise of object-oriented programming is that we should be able to create an object—a form, a control on a form, even a function that doesn't have a visual component. Instead of cutting and pasting a copy of that object, we'll make references—pointers—to that object in the actual application. When the object changes, the application's objects change, too. Automagically, as they say.

VFP has this capability. Those who use this capability will be cooking with gas grills, while the rest of the crowd will still be sticking raw meat on a sharp stick into a campfire. The tools that enable us to do OOP are the Class Designer and the Class Browser—and that's what this chapter is about. Before we discuss these tools, however, we should cover what a class is, what OOP means to Visual FoxPro, and so on.

This chapter consists of three parts. The first will be a light introduction to OOP. It's not an exhaustive tutorial on all of the details; rather, you're going to learn enough to discuss classes and the class tools effectively. There are two reasons for this light treatment. First, this section in the book is about the tools, and we're going to cover just enough theory in order to learn how to use the tools. Second, OOP is complex and hard to understand. You have to learn the lingo. It takes more than one pass at it, and simply reading the same chapter twice isn't enough. So I'm going to repeat this information, in greater detail, ad nauseum, perhaps, in the

Sample Application section of this book. (It's a bit like learning a foreign language: You have to hear it over and over before you begin to think in the language.)

Just as development of an application is an iterative process, so is learning about object-oriented programming. You learn a little bit and get comfortable. Then you learn a little bit more, piling that on top of the stuff you just learned and have grown comfortable with. And so on and so forth. It's important to first learn the terminology and theory.

I remember sitting in the lounge late at night at a conference with about 20 other developers back in 1994 or so. One of the stalwarts was listening to several people go back and forth about the upcoming Visual FoxPro 3.0, and asked, "Do you really think we're going to start using 'polymorphism' and 'inheritance' in our daily conversations?" Another in the group replied, "Sure. Twenty years ago, I can remember the same question being asked about 'normalization' or 'tuples'—but those concepts are second nature to us now."

The second part of this chapter will deal with the tools themselves: the Class Designer and the Class Browser. Once you're comfortable with these concepts and techniques, you'll learn the tools you need to use in order to use object-oriented programming in Visual FoxPro.

Finally, I'll build a set of basic, but solid, base classes that you can begin to use and enhance in your own development, and show you how to build forms with them.

A quick introduction to object-oriented programming

The first thing to understand about OOP is that there is no single way to do any of this. If you wander the halls of a conference on object-oriented programming, you'll hear all of the buzzwords and catch phrases—and then find different people using them to mean different things according to context, language, and platform. Once you get past that shock, you'll encounter a variety of people—from the pragmatists, those who have to implement real-world solutions today, to the theorists, whose mission in life is to evangelize about POOP—Proper Object-Oriented Programming. It's much like the Relational Database Management System purists who claim that today, in 1999, there still isn't a "true" relational database system commercially available. All along, however, we've been building normalized data structures, making do with the available tools, and running global businesses with these systems. While we may never reach the Holy Grail of OOP, we can give it our best shot and get 80% of the benefits with 20% of the worry.

Thus, as a result, the discussion that follows won't necessarily win any "POOP" awards or put me in the OOP Hall of Fame. This is my take on OOP as it relates to Visual FoxPro, at this stage in the lifecycle of VFP. I've been using and learning Visual FoxPro OOP over the past five years, and it's starting to sink in. I've even shipped an application or two. This chapter will give you the fundamentals. In 20 minutes, you'll understand what you need to know to take advantage of VFP's OOP capabilities. Once you're comfortable with the contents of this chapter, you may want to investigate Markus Egger's book, *Advanced Object Oriented Programming with Visual FoxPro 6.0* (Hentzenwerke Publishing).

There are a dozen or more fancy buzzwords attached to the OOP model, including class, object, property, method, inheritance, hierarchy, encapsulation, polymorphism, subclass, and so on. In my mind, the key concepts that will bring you immediate benefits are subclassing and inheritance.

The original form and control objects

Visual FoxPro comes with a default form object that is used as a starting point every time you create a new form. It has a gray background, measures about 360 pixels wide by 240 pixels high, has a system-type border, is named "Form1" and has a caption of "Form1." Additionally, VFP comes with about 20 native controls, from command buttons to hyperlinks. Each form and control has default properties and events as well. Each time you create a new form or place a control on a form, it seems like you are making a copy of the default form or control.

Actually, you are not making a copy—you are creating an object that consists of pointers to the original version of that form or control. The original is contained in the code of the VFP6.EXE file and can't be changed. This object is called an *instance* of the form or control, and the process of creating the instance is called *instantiation*. You can, to an extent, think of the original version as the die (or mold) used to create multiple copies of a firing pin for a rifle, and each instance as an individual copy of that firing pin.

When you run your application, the form or control points back to the information in the original version in the .EXE. Because you're always either running the application from within Visual FoxPro or from an executable that requires a Visual FoxPro component, the code that describes the form or control is always available.

Inheriting from the original

Because the instances reference the original version, if you change the original version, each instance will reflect that change. (In fact, you see that behavior often as you use new releases of software. Ever notice that a new version of word processor has unintended effects on existing documents?) In fact, now that we're using the third major release of Visual FoxPro, we can see this ourselves. In Visual FoxPro 3.0, the default form had a white background. In VFP 5 and 6, the default form had a gray background. If you ran a VFP 3 application in VFP 6, all of the white forms would automatically have gray backgrounds—without you lifting a finger.

This behavior is called *inheritance*. The instances of a form inherit properties, such as color, size, and captions, and methods such as Click, Drag, and Valid, from the original version. When you change the original version, as was done when you used VFP 6's executable instead of VFP 3's, the instances inherit those changes automatically. If inheritance existed in the rifle factory described earlier, when you changed the size or shape of the die, every firing pin created from that die would automatically morph to the new size or shape. Furthermore, you could think of a firing pin as having certain events—for example, when the pin strikes another piece of metal, it creates a spark. The firing pin's Strike event initiates the Spark function. Again, if the firing-pin mold had inheritance, you could change the functionality of the die's Strike event to Puncture, and all of the firing pins ever made from this die would then Puncture when they struck another piece of metal.

Creating your own originals

But you're not in a rifle factory. You're sitting in a chair in front of a cathode ray tube with a stuffed Dilbert doll sitting on top of it, and you want to create an order-entry system. Here's the essence of object-oriented programming in Visual FoxPro: You have the capability to create your own versions of these default forms and controls, so that subsequent forms you create (or controls you drop on forms) inherit properties and methods from the original versions you created. Then you can make changes to your original versions, and all of the forms and controls

you've created from those original versions will reflect those changes. You're no longer restricted to accept the defaults that came with the package. (Before I continue, I should mention that it's important to remember that the original versions you create still use Visual FoxPro's default forms and controls as *their* original definitions!)

How about a real-world example or two? When was the last time you created a form that had no objects on it? Pretty long ago, eh? Every form I've ever created in my life has had two buttons on it: a Help button and a Done button. And those two buttons always do the same things: The first brings up a help window where the topic displayed relates to the current form, and the second one cleans up everything and closes the form. Wouldn't it be nice if the next time, and every time after that, when you issued the command:

```
create form <name>
```

the form that appeared already had those two buttons on it, and those two buttons had the behavior you defined? You can do this simply by creating your own default form, dropping two buttons on it, and then telling VFP to use that form as the default instead of Visual FoxPro's default. Furthermore, if you realized that you needed to change the behavior of one of those buttons, you could do that once, in your default form, and every form in the application would then inherit that behavior automatically.

For example, suppose you decided that you wanted the Help button to work one of three ways—for most forms, it would bring up a context-sensitive Help window; for a small percentage of forms, it would bring up a user-defined Help screen; and for the last few, it wouldn't be visible at all. You could create a property of the form that indicated which type of behavior the button should have, and then control the behavior of the button in the default form. From then on, every form that was based on your default form would have those three behaviors available, depending on how the property was set.

You can also create your own default controls. For example, in Chapter 9 I discussed the List Box and Combo Box controls, and indicated that I thought the best way to populate them was to include a custom property of the control that was an array, and then to set the ControlSource of the list box or combo box to the contents of that array. Having to add an array property to every list box and combo box would be a nuisance—if indeed you could even do it. If you define your own List Box and Combo Box controls, however, you can add your own properties and methods, and they'll be available each time you drop instances of those controls on a form.

OOP terminology

Now it's time to introduce the real terminology for these pieces and processes. The definition of one of these objects, be it a form or a control, is called a *class*. When I created a form with a Help button and a Done button on it that I was going to use as the default to create more forms, I created a *form class*. When I created a list box (with some of my own custom properties and methods), which I was going to use as the default when I needed a list box to put on a form, I created a *list box class*.

The default forms and controls that come with Visual FoxPro are definitions for each of those objects, and they are the *Visual FoxPro base classes*. After you create your own forms and controls to use as the defaults instead of Visual FoxPro's base classes, you'd refer to your

creations as *your base classes*. The act of creating a class that references another class is called *subclassing*, so when you created a list box with an array property, you were subclassing Visual FoxPro's List Box control.

When you create a form from a class, you are creating an *instance* of that class—or, as mentioned earlier, *instantiating* the class. The form is the instance, and the class from which the form was instantiated is simply called the form's *parent class*. If you subclass a class, the class higher in the hierarchy is also called the parent class. Some languages, by the way, call the class higher in the hierarchy the *superclass*.

I've just described creating your own form base class with a Help button and a Done button. What if you need a form that isn't supposed to have a Help button or a Done button? You might be thinking that the best solution would be to create a complete set of your own base classes that are no different from Visual FoxPro's base classes, and then create a second set of classes that were based on your base classes. That way, you'd have a form base class that was completely generic, and you could subclass your form base class, creating a form class with a Done button and a Help button. If, at some point, you needed a form that didn't have a Done button or a Help button, you could simply subclass your form base class, and create a second form class that had the attributes or behaviors you desired.

Thus, you'd have a hierarchy that looked like this:

```
VFP form base class – Your form base class – Your Help/Done form subclass
                                           - Your Other form subclass
```

with two subclasses both inheriting from your form base class. The philosophy here is to never directly create instances from your base class; instead, you create instances from subclasses of your base class. Your base class, then, is referred to as an *abstract class*—one that is never directly used. This is a valid approach, and you might feel most comfortable following it.

You can create subclasses of subclasses as many times as you like. This hierarchy of classes is called a *class hierarchy*, which looks much like a set of routines that call subroutines that in turn call other subroutines. Of course, before you get too carried away, dreaming of class hierarchies dozens or hundreds of levels deep, remember that in the real world you have to maintain those classes, and you also have to make your applications perform on computers with limited resources. Digging through a class hierarchy 10 or 15 levels deep will require more horsepower than a class hierarchy only one or two levels deep, and will end up being very difficult to use, as you try to determine the differences in behavior from one level to another.

Inheritance and overriding methods

I've discussed the idea of inheritance several times, but there's one more item to mention before moving on. As you've seen, you can create a class and put code in its methods. Then, once you create an instance of that class, the code that is in the methods of its parent is also available to the instance. Suppose, in the form with the Done button, there was code in the Release method of the form to perform certain housekeeping chores before the form is actually completely closed. You might want to check for data that has been entered in the form but not yet saved, for example.

When you instantiate a form based on that form class, you'll see the Done button, but if you look in the Release method of the form instance, there won't be any visible code. This is to be expected if you think about it for a minute. When you create a form based on Visual

FoxPro's form base class and you open up a method, you don't see any code in it, do you? However, certainly Visual FoxPro code gets executed, right? For example, if you look in the GotFocus event, you don't see code, but when GotFocus is fired, things happen, such as the color of the title bar changing. The code is stored in the base class—in this case, Visual FoxPro's VFP6.EXE.

This is good, although I've heard this described by idiots as a downside to OOP. It means that you're not carrying around a lot of excess baggage, but even more so, it means that because the code being executed is stored only in the base class, changes need to be made only to the base class. If that code also ended up in the instance, changes made to the base class would either have to be manually propagated to every instance, or they wouldn't be reflected in the instances. In other words, no inheritance. I've heard rumors that Visual Basic operates like this, but I can't believe anyone would be so foolish to actually use it, given such a lame feature.

There might be times, however, when you don't want the inherited behavior from the parent class. For instance, in the example of the class with code in the Release method, you might need different behavior than what was provided in the class definition. Or you might just not want the code executed at all. In both cases, this is referred to as *overriding* the method code in the class definition.

Non-visual classes

Now that you're a little comfortable with the concept of classes, I'm going to throw you a curve. As you've seen, a class can be a form, from which you can create multiple instances that all inherit the properties and methods of the class. It can also be a control, and you can drop instances of the class on a form as needed. Finally, a class can also be a group of controls that act together as a unit; just like the class definition of a single control, you could drop an instance of the control class on a form as needed.

So it seems that classes are "things" that you can see, move around, and otherwise manipulate. The next conceptual leap to make is that of non-visual classes, and I've found these are harder to grasp for many people. A class can have a non-visual nature—in other words, there is no object to call from a menu or to move around on a form. The question that immediately comes to mind is "So what is a non-visual class, and, more importantly, what good is it?"

When you began creating applications in the very olden days, there was no such animal as a "form" or a "report"—instead, you used programming commands to draw discrete elements on a computer display console. These programming commands resided in files of two types. The first type was a custom-written program, while the second was a library. The library included programs that could be reused throughout an application, and, indeed, across applications.

In the early 1990s, the idea of "forms" (some development environments called them "screens") and "reports" was born, and then a programmer had two types of elements to work with—programs that still consisted of commands, and forms and reports, which were visual representations of programs. Programmers still used libraries of commonly used programs.

With object orientation, these libraries can be moved into non-visual classes. An entire application, and indeed, multiple applications, can still access these libraries, but by placing them into classes, another advantage accrues: The library can be subclassed, just like a form or control.

This still hasn't quite answered the question, "So what is it?" Remember that a class has properties and methods. Properties, again, are really just variables that are initialized with respect to the class. A form that allows you to search for the name of a company might have a NameToSearchFor property. When the form is started, that property might be an empty string, but it would eventually be populated with the value that a user enters into a text box. Methods, as well, are just functions that belong to the class. When the class is a form or a control, those methods are easy to visualize—Init, Click, Destroy, and so on.

You can create a class that has properties and methods but doesn't have a form or a control to go along with them. What could this be? Think of a function library that has a number of procedures that do date and time calculations. You would have variables that are initialized, a number of subroutines that perform operations and return values, and—hey, these sound a lot like properties and methods, eh? One variable might be the initial date sent to a function; another might be a value that deals with the type of format required. And one function in this library might simply convert any type of input to a common format, while another function might return error messages if a parameter passed to the function was invalid.

A non-visual class could do the same types of things—using properties to store parameters sent to a method in the class, and then various methods of the class to validate incoming parameters, throw error messages, and so on.

Another example of a non-visual class would be one that handles the environment of the machine during application startup. One of the properties might be the default directory at startup, while a method in this non-visual environment class would check for available memory or disk space before loading the rest of the application.

A non-visual class can be subclassed just like any other class, and its methods can be overridden in an instance of the class, just like visual classes.

So a non-visual class can be thought of as a function library. But we can take this considerably further. As you get deeper into the world of Visual FoxPro applications, you'll start to see the term *application class*, as in "The XXX application class provides the framework for all of our custom applications." Your mind starts reeling again. What in the world is an application class?

Many developers have created a core set of programs that provide a generic foundation upon which they can duplicate and add components as required. This foundation provides a series of services, such as file handling, user security and permission levels, error trapping, and so on. All of these generic services should "come along for the ride" for every new application. You can think of this as a skeleton or framework upon which a custom application can be built. This skeleton consists of one or more programs, some of which might be located in a common library.

In the Visual FoxPro world, an application class is often a non-visual class that contains all of the properties and methods (remember, these are just variables and subroutines) that make up this framework. Depending on the complexity of applications and the requirements of the developer, an application class might actually contain both visual components (such as a logon screen or a user-maintenance screen) and non-visual components (such as a routine to handle missing files or to do an environment check).

Where OOP fits in with Visual FoxPro

I hope you're getting a little anxious by now. I know that after I'd heard all the explanations without having gotten my hands on the product yet, I was bursting with questions like:

- "Where is a class stored?"
- "What does a class look like on disk?"
- "How does inheritance actually work in an application?"
- "I'm in Toledo, but I've got an application running on a machine in Madrid. How does my application in Madrid know that I've changed a parent class in Toledo?"

Most of the books that discuss OOP use examples such as blueprints, telephones, light switches, and red circles. As far as theory goes, great. But the first thing I asked was, "A class has to be stored in some sort of file, right? What does that file look like? Is it a text file? Binary gobbledygook? A table? Some new sort of structure? And where is it stored on disk?" And then I wanted to know exactly how VFP made inheritance work. Just as changing the original mold from which firing pins were being manufactured wasn't automatically going to change all of the firing pins that had already been made, I wasn't quite sure how the changes I made to this class file on my machine were automatically going to be reflected in the various applications that relied on this class file. It seemed like an awful lot of magic.

The answer is really quite straightforward, and reading back over my words of five years ago, it now seems pretty obvious—but it sure wasn't then. And I'm sure the implementation in C code had one or two programmers at Microsoft working past 5 p.m. once in awhile.

First, as you know, the representation of a form on disk is a table. The form itself is represented by a couple of records, and then each control on the form is represented by an additional record in the .SCX file. Indeed, if you haven't before, you can open a form's file as a table just like any other .DBF file—you just need to include the extension. The following code will show you the inside of a form:

```
use MYFORM.SCX
browse normal
```

Be careful, though—Visual FoxPro expects certain items in certain places, and if you change or delete things accidentally, you might not be able to open the form in the Form Designer again.

A class is just another table—with the exact same structure as an .SCX table, but with a .VCX extension. Just as a regular form has records that map to the controls on it, and fields in each record that describe the properties and methods of the form or a control on the form, a form class has records that map to the controls on the form, and fields that describe the properties and methods. In addition, other fields in the .SCX table point to .VCX files, and to specific records in the .VCX file. Kinda like a parent-child relationship, eh? And when you change the class—in other words, change the records in the .VCX—the next time the form that references that class is compiled, the changes in the .VCX will be taken into account. Rather simple, isn't it? If you're curious (and careful), you can open a .VCX file just as you can open a form's .SCX file.

Thus, when you ship an application, you need to include the classes—the .VCX files—along with your menus, forms, reports, and other files. Or, if you're building an executable file, the build process will compile the classes used in your application just as it does ordinary programs.

Thus, here's the answer to the question, "How do changes to a class library get reflected in the application?" When you change a class, you must ship the new .VCX or a new .EXE that reflects the new .VCX to the location where the application is running, just as when you updated a procedure library file.

I'd like to discuss the contents of the .VCX and .SCX files in a bit more depth before moving on, using a real example to illustrate where all the pointers go.

Suppose your form base class consists of a Visual FoxPro base form and two command buttons, both from a Visual FoxPro command-button base class, as shown in **Figure 10.1**. The form base class is named frmBase, and the VCX that holds this form class is named BASECLASSES.VCX. If you open BASECLASSES.VCX, you'll see five records. The first is just a placeholder. The next three records represent the three objects that make up this class: a VFP form and two VFP command buttons. If you examined the Class field of these three records, you'd see they reference Visual FoxPro's built-in objects. The last record is another placeholder that you shouldn't worry about. The key point is that this class definition consists of three records.

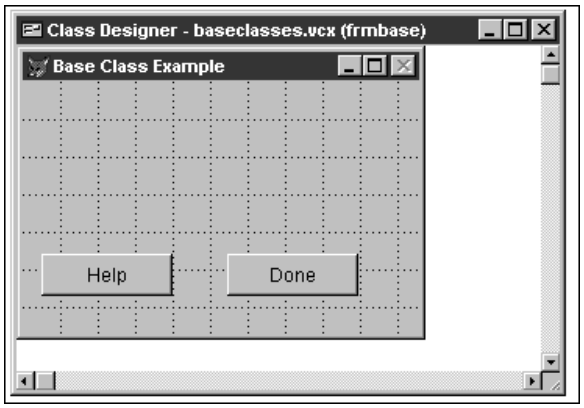

***Figure 10.1**. A sample form base class consisting of a Visual FoxPro form and two Visual FoxPro command buttons.*

Next, suppose you were to instantiate a form from this class (don't worry about exactly how that process works quite yet), and name this form MYSAMPLE.SCX, as shown in **Figure 10.2**. If you opened MYSAMPLE.SCX as a table with the USE command, what would you see? If you had created a regular form, you'd expect to see three records—one for the form and one for each of the command buttons, right? But we created this form from a class, which means that this form is essentially just a pointer to a class. That means you're going to see only one record in the .SCX file. The Class field in that record is going to point to frmBase, and the ClassLoc (class location) field in that record is going to point to BASECLASSES.VCX.

(Actually, the .SCX contains several other records—placeholders similar to those found in the .VCX file as well as another record for a DataEnvironment. But those records aren't germane to our discussion.)

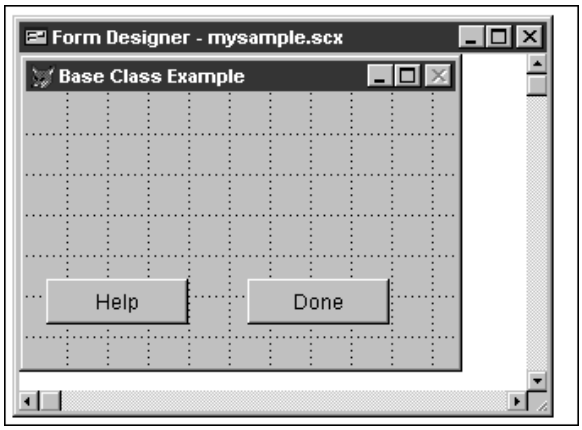

***Figure 10.2**. A sample form instantiated from frmBase.*

Finally, you might be wondering about the relationship between frmBase and BASECLASSES.VCX. A .VCX file is called a class library, and much like a regular library can contain more than one book, a class library can contain more than one class. Thus, frmBase is simply one class that is located in BASECLASSES.VCX. You could create a list-box class, lstBase, and store that in BASECLASSES.VCX as well. Then the .VCX would have four records (in addition to the placeholders I mentioned earlier). The first three would make up the form base class and the fourth would be the list-box class. Because a list-box class contains only one control, it needs only one record.

This class library, BASECLASSES.VCX, and the associated form, MYSAMPLE.SCX, are both included in the source code downloads for this book. You can open them and spelunk through the data to your heart's content.

Quick start to creating classes

You're probably plenty full of theory by now. It's time to actually start creating your own classes. In this section, I'll cover creating classes both from forms and from controls, and then show you how to use both of them together.

Form classes

Creating your own form base class

1. Make sure that an active form base class hasn't already been subclassed from Visual FoxPro's base classes.
2. Look in the Forms tab of the Tools, Options dialog. The Form Set and Form Template Classes text boxes should be empty and the check boxes should be unchecked.
3. Create a form by issuing this command:

   ```
   create form X
   ```

4. Modify the form to reflect how you want all of your forms to look and behave. Here are some ideas:

   - Change the caption to frmBase. This will act as a visual clue that the form you just created has come from your own base class—not from Visual FoxPro. And more importantly, the reverse. It's pretty discouraging to create a form, drop a bunch of controls on it and work with it for a while, only to discover that you're working on a Visual FoxPro base class form and not your own base class form.
   - Drop a couple of Visual FoxPro buttons on the form – such as for Help and Done (or Close, if you prefer.) Note that when you're doing this for real, you'll most likely want to use controls that you've subclassed yourself, not Visual FoxPro's controls.

5. Select the File, Save As Class menu option. The Save As Class dialog opens as shown in **Figure 10.3**. (Note that if you select Save As, you'll just create another form that references Visual FoxPro's base class.)
6. Enter the name of the class—frmBase—in the Name text box, and the name of the class library—BASECLASSES—in the File text box.

   Note that Visual FoxPro will append the .VCX extension to BASECLASSES automatically, because, after all, it knows you are creating a class.

   You can also choose an existing class library by clicking the ellipsis button to the right of the File text box, and then selecting a .VCX from the list that appears or navigating to the .VCX you want.

7. At this point you've created a .VCX file, but you've also still got an .SCX file on the screen. You can get rid of it by closing it and not saving the results.

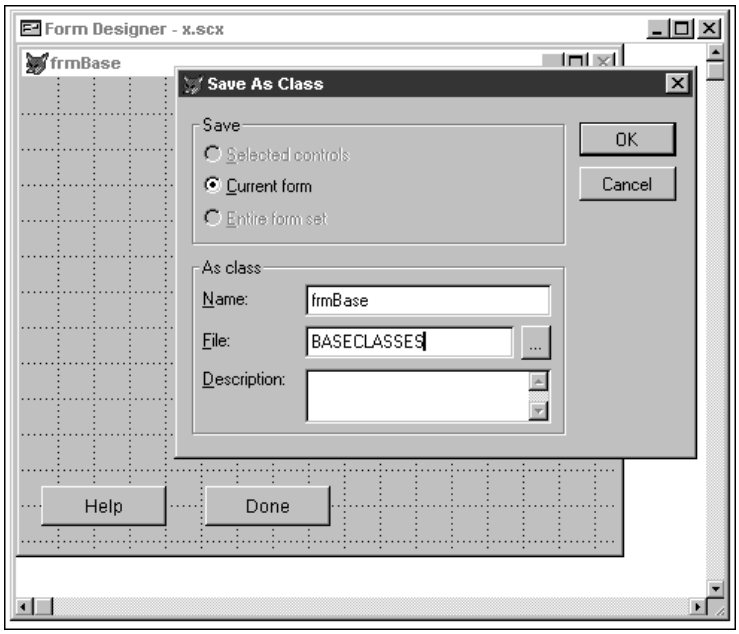

***Figure 10.3**. The Save As Class dialog allows you to convert a form into a class definition.*

Using your own form base class

So now you've got your own form class. How do you use it as the parent class for all forms you build from now on? If you've tried to create another form, watching out for some "create form from user-defined base class" option that you thought might have previously escaped your notice, you'll find nothing new.

To use your new form base class, follow these steps:

1. Open the Forms tab of the Tools, Options dialog. The Form Set and Form Template Classes text boxes should be empty and the check boxes should be unchecked, since you just verified this when you created your form class in the previous section.

2. Click the Form check box in the Template Classes control group. The Form Template dialog will open, prompting you to select a form class.

   This can be a bit tricky the first few times you use it. Because it looks much like a regular File, Open dialog, you might be tempted to select a .VCX file and then immediately click OK. Resist that temptation. Note that the Class Name list box on the right side of the dialog becomes filled with the classes in the .VCX that you chose, as

shown in **Figure 10.4**.

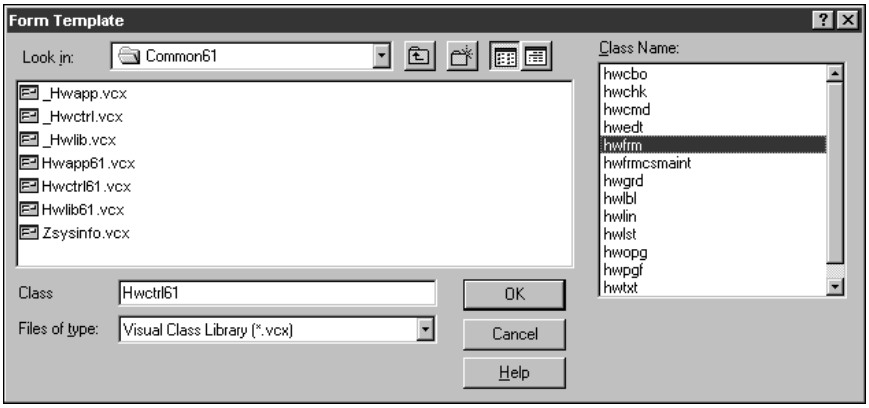

***Figure 10.4**. Be sure to select the appropriate class in the Class Name list box when picking a Form Template class.*

If you just click OK, you'll end up picking the first class in the .VCX—most likely not what you wanted.

Many developers keep all form and control classes in the same .VCX. Yes, the .VCX can get rather large, but because it's likely that you're going to use multiple controls on a form, you might as well keep all of them in the same file so you have to open only one class library.

3. Once you've selected the form class you want, click OK and you'll be returned to the Tools, Options dialog.

4. Click OK if you want this form class to be used as your default form-class template for the current Visual FoxPro session, or click Set As Default if you want this selection to persist past this session.

Creating a form from a Form Class template

Once you've set up a Form Class template, creating forms based on that class is no big deal.

1. Create a new form by issuing this command:

```
create form Y
```

2. You'll see that your new form has the characteristics of the class you used as your Form Template class.

3. Open the Properties window and examine the Class and ClassLibrary properties. You'll see that they have read-only values of the class/class library you specified.

Using a different form base class "on the fly"

As you become more experienced, you'll find that you don't want to use a single form class as the parent for all of your forms. You might have one form class for minor entity maintenance, another for dialogs, a third for query screens, and a fourth for a specific type of data-entry form. It would be pretty darn inconvenient if you had to go through the rigmarole of having to change a Tools, Options setting each time you wanted to create a different type of form. Fortunately, you don't have to.

The command CREATE FORM now has a clause that allows you to specify a class in a specific class library, like so:

```
create form Z as hwFrmQuery from BASECLASSES.VCX
```

This command will create a form named Z and use the hwFrmQuery class in the BASECLASSES class library. Pretty slick, eh?

Creating form subclasses

In the last section, I suggested that you would likely end up with multiple form classes in your class library. However, for the sake of expediency, I took a shortcut when providing examples: each of the classes I named was created from the Visual FoxPro form base class. While technically you could that, it would be a bad idea. Bad! Bad! Bad!

Rather, you would want to create a form class on which all of your other form classes would be based. You might not even use that first form class to create actual forms, but only to serve as the parent class of the form classes that you *will* use to create actual forms.

This might feel a little vague, so an example may help. My form base class is named hwFrm, but I never create a form from it, like so:

```
create form ZZZ as hwFrm from BASECLASSES.VCX
```

Instead, I create more form classes using hwFrm as the class definition, not Visual FoxPro's form base class. These classes would have names like hwFrmDialog, hwFrmQuery, hwFrmMaint, hwFrmMaintME, hwFrmMaintCommon, and so on. If you look carefully, you'll see that you can infer a three-level hierarchy here:

```
hwFrm - hwFrmDialog
      - hwFrmQuery
      - hwFrmMaint - hwFrmMaintME
                   - hwFrmMaintCommon
```

The three form subclasses—hwFrmDialog, hwFrmQuery, and hwFrmMaint—all have hwFrm as their parent. Then, hwFrmMaintME and hwFrmMaintCommon both have hwFrmMaint as their parent. The subclasses hwFrm and hwFrmMaint are not used for creating forms directly; they're used simply as definitions for further classes. If you remember the definitions above, these are both abstract classes.

So how do you go about creating hwFrmDialog from hwFrm? Follow these steps:

1. Identify the class upon which you want to base a subclass, such as hwFrm in BASECLASSES.VCX.

2. Issue the CREATE CLASS command (or select the File, New, Class menu option). The New Class dialog appears, as shown in the background in **Figure 10.5**.
3. Enter the name of the new class, such as hwFrmDialog, as shown in the New Class dialog in Figure 10.5.
4. Click the ellipsis button next to the Based On combo box.
5. The Open dialog will appear, as shown in Figure 10.5.

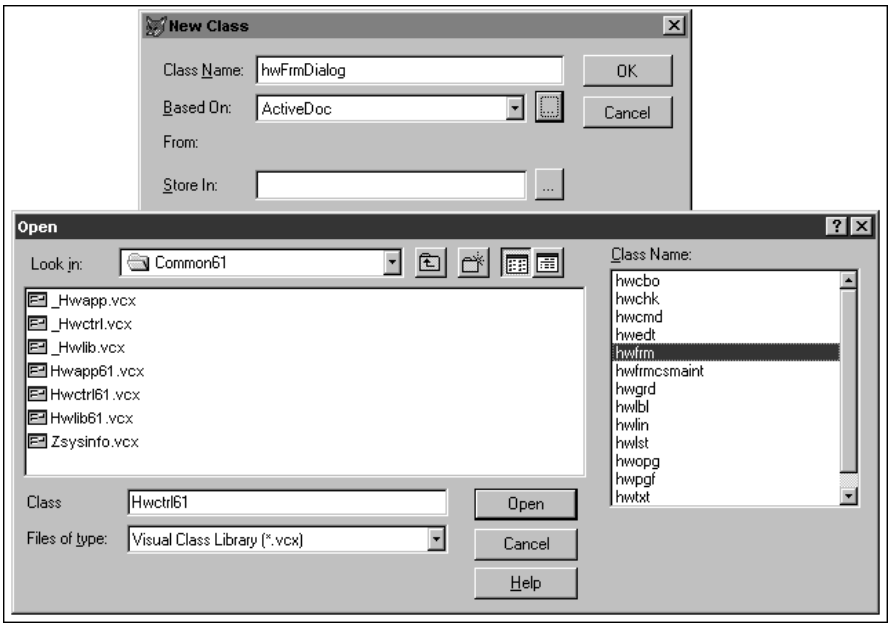

***Figure 10.5**. Selecting a class upon which to base another class.*

6. Select the class upon which you want to base your new class. In Figure 10.5, hwFrmDialog will be based on the hwFrm class in the HWCTRL61.VCX class library.
7. Click Open.
8. Choose (or create) the class library in which the new class will be stored. I suggest that you store your new class in the same class library that holds the parent class.

   Note that you *could* choose a different class library to store hwFrmDialog. You would want to have a specific reason to do so. The downside of using a second class library to store hwFrmDialog is that you'd then have to open two class libraries in order to use a single class—why make it that hard on yourself? On the other hand, what if you wanted to distribute hwFrmDialog to someone else, without sending them every class

you ever created (that also happened to be in that class library)? This is one of the toughest design decisions to make: how granular to make your class libraries.

Registering a form class with Visual FoxPro

As you know, you can select controls from Visual FoxPro's base class by grabbing them from the Form Controls toolbar. You can create your own toolbar that contains buttons for your own classes and grab those classes to create forms and place controls on forms. There are two ways to do this.

Use the Add button from the View Classes menu

1. The View Classes button on the standard Form Controls toolbar looks like three books and has a very small arrow in the lower right corner. Click it.
2. A context menu with three commands appears. Select the first command: Add.
3. A standard Open dialog opens, prompting you to select a class library file. Select the class library of your choice.
4. The contents of the Form Controls toolbar will now change. The first two buttons, Select Objects and View Classes, and the last two buttons, Builder Lock and Button Lock, will stay the same. In between, however, the Visual FoxPro base class control buttons are replaced by buttons representing the classes in the class library you chose.
5. When you move your mouse over a toolbar button, a tool tip displays the name you gave that class.
6. When you click the View Classes button again, you'll see that your class library name has been added to the context menu.

   You can continue adding class libraries to this context menu by following the previous steps, and thus be able to switch between class libraries quickly and easily.

Use the Controls tab of the Tools, Options dialog

1. Open the Tools menu and select Options, then Controls. See **Figure 10.6**.
2. Select the Visual class libraries option button.
3. Click the Add button.
4. Select the name of the class library (remember, it's a .VCX).
5. Click OK to register this library for this session, or select Set as Default to register this class library for future sessions.

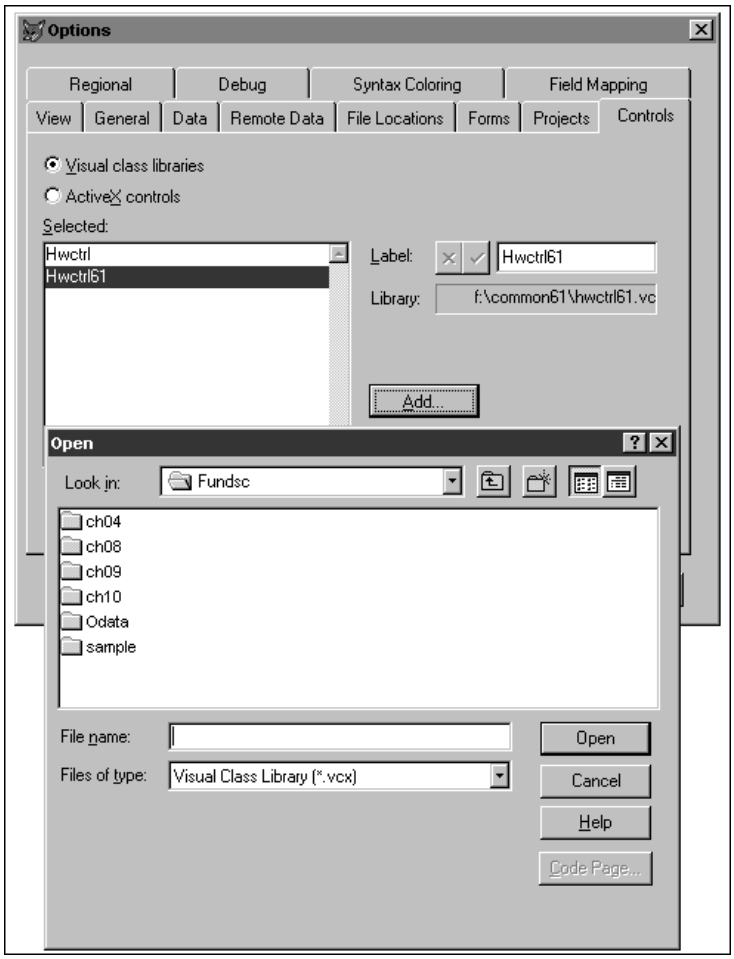

***Figure 10.6**. You can register a class library using the Controls tab of the Tools, Options dialog.*

Attaching a custom icon to a registered class

If you have several forms in the same class library, as you ought to, you'll see that your toolbar now has four or five (or a couple dozen!!!) buttons—each with the exact same icon. You don't have to settle for this boring panoply of computerdom, however. You can attach a different toolbar icon of your own choosing to every class in the library. Here's how:

1. Bring up the Class Designer with this class by issuing the MODIFY CLASS command or opening the File menu and selecting Open, Class menu. The Class Designer appears.
2. Select the Class, Class Info menu option.

3. Select an icon for this class by clicking the ellipsis button next to the Toolbar Icon label. You can draw your own, or if you'd rather leverage the work of others, you can find a slew of sample icons in the <location of VFP>\MISC\BMPS directory.

Modifying a form class

Once you've created a form class, you might find you want to modify it. You can simply use the MODIFY CLASS command, like so:

```
modify class hwFrm of <path>\MYBASECLASSES.VCX
```

I find, however, that as I get older, I want to type less and less. If you're inclined along the same lines, you can open a class by using MODIFY CLASS without any further parameters. Doing so will bring up the Open dialog pictured in Figure 10.5. Locate the .VCX that contains the class you want to modify, and then be sure to select the class itself in the Class Name list box on the right.

Control classes

Creating your own control base class

Now that you've got the hang of creating your own form classes, you'll want to create your own control classes. The procedure is similar to that of creating form classes:

1. Issue the command CREATE CLASS or select the File, New, Class menu option. The New Class dialog appears.
2. Type the name of the class you are creating.

   Again, remember that naming conventions will make your life easier down the road. Consider using the first three characters of the naming convention for controls on forms, such as cbo for Combo Box and cmd for Command Button. I personally use names like hwCmd, hwChk, and so on (where "hw" stands for Hentzenwerke).
3. Select the object you are using as the parent class for your new class.
4. Select the name of the class library into which this class will be placed, or enter a new class name by typing in the File text box.

   Remember, classes are contained in class libraries, and class libraries are files with .VCX extensions. The name you select depends on what else you're going to put in the library. For now, use a name like BASECLASSES.VCX.
5. Click OK. The Class Designer appears with a copy of the control in the window. Because you are creating only the control, resizing the window will change the default size of the control, so be careful.
6. Make modifications to the control as desired.

   For example, you might want to change the background color of certain controls such as check boxes and option button groups so that they match the background color of

the form base class you created in the previous section. If you don't, the background of the control will show up against the form—the programming version of wearing a colored t-shirt or bra underneath a white dress shirt or blouse.

You might also want to change the default caption of the object. I change the caption of all of my base-class controls to conform to the Visual FoxPro conventions so that when I drop the control on a form, it's obvious that the control is one of mine instead of a VFP base class. For example, the default caption for my label class is "lbl", and for my check-box class it's "chk."

7. Save the class.
8. Repeat the previous steps for every control you want to add to your base class. Be sure to specify the same class library name.

Creating control subclasses from your base class

Just as you created subclasses from your form base class, you will most likely want to create subclasses of your control base class. This mechanism works the same way as subclassing your base form. For example, you might want two types of text boxes—an editable text box and a read-only text box. I would name mine like so:

```
hwTxtEditable
hwTxtReadOnly
```

Actually, I'd abbreviate the names because when I drop a control class on a form, I name the instance by appending a specific identifier to the class name. Thus, a pair of text boxes for a person's name would look like this:

```
hwTxtEdNameFirst
hwTxtEdNameLast
```

If they were read-only text boxes, they'd look like this:

```
hwTxtRONameFirst
hwTxtRONameLast
```

Registering your controls base class

You can register your controls base class the same way you registered your form base class. Remember to use the Set as Default button in the Controls dialog (Tools, Options, Controls) in order to keep the registrations available from session to session.

Placing controls on a form from your base class or a subclass

Now that you've got your controls base class, and probably several subclasses of your form base class as well, you're going to want to create controls on forms from those classes. This is done a little differently than with forms—and fortunately it's easier, because you'll be creating more controls than forms. As with all things FoxPro, there is more than one way to do this.

Use the Project Manager

1. Add the control class to the Project Manager:
   - Open the Project Manager.
   - Select the Classes tab or the Class Libraries icon in the All tab.
   - Click the Add button.
   - Select the class library that contains the class or subclass from which you want to create controls. The class library is added to the Project Manager.

     Note: You can select the outline control (the plus sign to the left of the library icon) to expand the class library and see the classes contained in it.
2. Create a blank form or modify an existing form.
3. Drag the class (not the class library) from the Project Manager to the form. The control will be placed on the form.

Select the class from the Form Control toolbar (after registering the class)

You must register the class from which you want to create controls. (See the section "Registering a control class with Visual FoxPro.") Then follow these steps:

1. Create a blank form or modify an existing form.
2. Select the View Classes icon on the Form Control toolbar and select the menu option for the class library that contains the class from which you want to create controls. The buttons for that class appear on the toolbar.
3. Click the button for the control class from which you want to create a control. The button appears depressed.
4. Move the mouse and click on the Form Designer. An instance of the control class will be placed on the form.

Control and container classes

Now that you have your control base class, and probably several subclasses of your base class as well, you might want to create a class that consists of several controls but without using a form to "hold them together."

How to make this happen? You can do this one of two ways.

The first is to create a container class. A container is an object that contains other objects, and those other objects can be addressed at runtime. For example, a page frame and a grid are both containers. A grid contains one or more columns, and you can address both the grid as well as each individual column in the grid. (And because a column is also a container, you can address the objects in the column—the header and the text box, for example.)

An example of a container would be a custom control that replaced a combo box. You could put a text box and a command button in a container, and allow the user to either enter a

value in the text box or click the command button to "open up" a pick list of one sort or another.

The second way is to create a Control class. Note that I am capitalizing the word "Control" here so you don't mistake this with the term "control class" that I have been using for the last umpteen pages.

The objects in a Control class can only be accessed individually while in the Class Designer—they are not available when a form is being designed or at runtime. You can think of a spinner or a list box as being a control with multiple components. A spinner has a text-box component, an up-arrow component, and a down-arrow component. You can access only the entire control—you can't change the color of the up arrow to be different from the color of the text box, nor can you access a method of the up arrow; you access a method of the Spinner control.

An example of a Control class that you would make yourself would be a class that you want to distribute to others but that you don't want modified. Just as you can't get to the internals of the native Visual FoxPro spinner control—you can access only the interface provided to you—you might not want the user of your class to access anything but the interface you've provided.

The steps to create a Control or container class are essentially the same as creating a form: First you create the container and then you add controls to it. Remember that the controls you add to a Control class can only be modified here—and can't be changed once you've instantiated the control on a form!

1. Issue the command CREATE CLASS or open the File menu and select New, Class. The New Class dialog appears.
2. Type the name of the class you are creating. For fear of being so redundant that you tune me out, remember that naming conventions will make your life easier down the road. The naming conventions for Control and container classes use the first three characters of ctl and cnt.
3. Select either a Control or container class in the Based on combo box.
4. Select the name of the class library into which this class will be placed, or enter a new class name by typing in the File text box.
5. The Class Designer appears with an empty box in the Designer window. This is the holder (I hesitate to use the word "container" for fear of confusion) for all the objects you will place in the class.
6. Add the controls and modify them as desired.
7. Save the class.

Working with Visual FoxPro's object orientation

Learning how to use the tools can take you only so far. Once you start actually building your own classes and start using those classes in real-world applications, you're going to run into a

variety of issues that aren't directly addressed in the directions for the tool. Here are some situations you're likely to run into.

Naming classes

There are several schools of thought with regard to naming classes.

My preference is to name classes from the generic to the specific, as you saw with the sample forms I described in an earlier section. The first few characters act as a common identifier so that my classes don't conflict with someone else's—if three developers named their classes "BaseForm" and kept them in a library named "BASECLASSES.VCX", it would be rather difficult to use them in concert.

The second batch of characters identifies the type of object, based on Microsoft's standard naming conventions, where "frm" stands for "form," "lbl" stands for "label," and so on. The type of class comes next. For example, for forms, you might have Dialog, Help, Maintenance, Data Entry, Query, and other types of forms. You might even break these down further, with multiple types of Maintenance or Query forms. There are no hard and fast rules about making decisions regarding these types of breakdowns, but you probably want to think along the lines of behavior and functionality, rather than appearance, when doing so.

The other popular naming convention goes the other way. I first encountered this convention when working with David Frankenbach on a Visual FoxPro 3.0 book on object orientation. His preference, coming from experience with other object-oriented languages like Visual C++ and used by the Microsoft Foundation Classes for Visual C++, was to prefix the specialization to the general class, instead of adding the specialization to the end.

Thus, he would name a hierarchy of text box subclasses like so:

```
TextBox (Visual FoxPro base class)
cTextBox (His custom base class)
ReadOnlyTextBox (A subclass of his base class that is read only)
```

Furthermore, David uses the three-character abbreviations to indicate that a class is functional, not abstract. Thus:

```
ReadOnlyTextBox
```

would be an abstract class while:

```
CmdReadOnlyTextBox
```

would be a functional class that you could drop on a form.

I'm not arguing that one is better than the other; but because I'm comfortable with my style, I'll use it throughout this book. If you prefer the other style, please feel free to use it. Or create your own convention if you like. Just be sure to use it regularly and consistently.

Adding your own properties and methods

One capability of Visual FoxPro's toolset that escapes most people is the ability to add properties and methods to a class or a form. Remember that a property is simply a variable that is carried along with the object (the class or the form), and it isn't visible when that object

hasn't been instantiated. You're familiar with the idea of declaring variables "local" or "private" within a subroutine—this does the same thing, only better.

Furthermore, a method is a segment of code that can be accessed only when the object is instantiated. The analogy to procedural programming here is not having access to a function in a function library until you've SET PROC TO.

This tying of properties and methods to an object is called *encapsulation* and, when done properly, forms the basis for object-oriented programming. Instead of having an application made up of module after module of spaghetti code, you'd simply have an object send a message to another object, referencing that second object's properties and methods. For example, when the Customer object (perhaps a form) needed to know the total sales for a customer, it would send a message to the Sales object, requesting that the Sales object run the TotalSales method, using the ThisCustomer property to do so. Yes, if you're used to procedural programming, this might sound a little freaky, but once you get the hang of it, you'll likely agree that it's much cleaner and less error-prone than the Old Way.

To take advantage of these properties and methods, we need to be able to add them ourselves. The process is extremely simple: Just select the Class, New Property or Class, New Method menu option (there are similar menu options under the Form menu when you're working with the Form Designer), and enter the name of the property or class as shown in **Figure 10.7**.

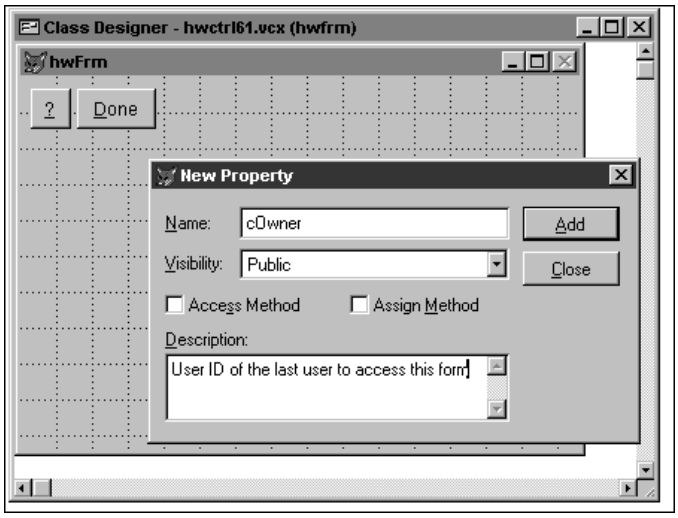

***Figure 10.7**. Adding a new property to a form class.*

Once you've added a property or method, it will appear at the bottom of the Properties window. You can initialize a property to a value or data type in the Properties window as shown in **Figure 10.8**, and you can add code to the method via the Code window. Notice that the description for the property or method appears in the bottom of the Properties window. You did enter a description, didn't you?

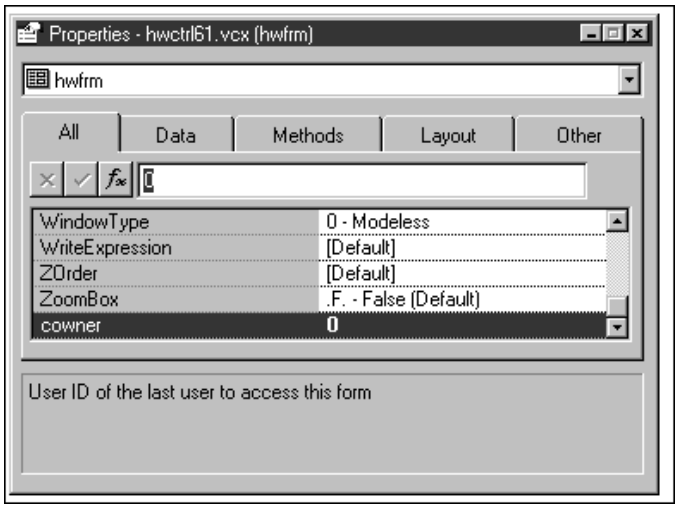

***Figure 10.8**. You can initialize a custom-added property in the Properties window.*

You can edit or remove a property or method from the class by using the Edit Property/Method menu option under the Class (or Form) menu pad.

You can mark a property or method as Public, Protected, or Hidden in both the New and Edit dialogs. Public is probably obvious—the property or method can be accessed by anything else. A hidden property or method is at the other end of the scale—it can't be accessed by anything except methods in the same class. A protected property or method is halfway in between—it can be accessed by subclasses of the class, but not by object instances or other objects.

Perhaps I should put that into plainer language. Suppose you have a form class, frmMaintME, that you use for creating simple maintenance forms. This form has an iidLastDeleted property, which contains the primary key of the last record that was deleted, and a WarnIfNotSaved method, which can be called, for example, if the user has attempted to navigate to another record without saving the changes to the current record. This method would display a dialog telling the user that they have modified a record but haven't saved it—and would they like to save it now?

If this property and method were public, they could be called from anywhere in the application, as long as the form instantiated from this form class was open. For example, you might create a form from this form class and add a custom command button to it. However, before allowing the user to click the button, you might want to test to make sure that changes have all been saved. You could call the WarnIfNotSaved method from the beginning of the command button's Click method.

You could even access those items from other forms. Generally it isn't practical to do so, but here's an example. Suppose, after deleting a record, you were going to ask the user whether or not to perform an operation with other records that included the ID of the deleted record. If you had to call another form to do so, you could reference the ID of the deleted record by accessing the property of the form from the second form.

Another example would be if you had stored a fully qualified file name in a property. You wouldn't want the user of the class to be able to enter garbage in this property, so you wouldn't mark it as public. Instead, you could mark it as protected. Then you would create a method of the class to enforce validation and other rules on the value that the user was entering; and only when the value passed would you allow the property to be assigned that value. Both the class as well as any subclasses of the class could address the property and work with it. The method would be accessed by the class user, not the property.

However, suppose further that this file name could actually be of two sorts—either a DOS-style filename like C:\WINDOWS\SYSTEM\MYAPP.DLL, or a UNC name like \\MAXIMILLION\DATA\SALES\REVENUES.DBF. You might not want any old subclass to be able to access this property, because the subclass might not know which type of file name it is. Instead, you could set this property to be hidden, and thus it would be hidden not only from forms instantiated from the class, but also from subclasses of the class. Those subclasses would still have to call methods of the class that were allowed to address the property.

This mechanism has a similarity to Control classes, in that you want to provide a very controlled interface for the user of the class; you don't want them calling any old method of the class, nor do you want them reading (or writing to) any old property. Instead, you make most of your properties and methods hidden or protected, and allow the user of the class to use just a few specific properties and methods that have been left public.

In this way, you can provide a class for others to use, and they can access the public and protected members according to certain rules. You've protected the internal workings of the class by making the appropriate properties and methods protected or hidden.

This is all getting a bit theoretical, so I'll stop now. It's good to be aware, however, that you have control over access to properties and methods of classes in a manner similar to the scoping (public, private and local) you can provide for variables.

The property and method hierarchy

Now that you're comfortable with the idea of classes as well as how to use them, it's time to talk about the ramifications of this inheritance concept. As I've said earlier, I believe that the tag team of subclassing and inheritance is the key productive feature of Visual FoxPro's object-oriented capabilities.

Overriding property inheritance

The definition of inheritance is that a subclass references the properties and methods of its parent class, and that parent class references the properties and methods of its parent, and so on up the line until the base class is reached. As a result, if the Visual FoxPro form base class has a white background color, every class subclassed from that base form will also have a white background, and so on down the line.

However, as we've also seen, if you create your form base class with a purple background color, every subclass created from your base class will also have a purple background color. This is called *breaking inheritance*, and your base form's background color is said to have overridden the parent's background color property. (I'm using a property here as an example, but the same is true for methods, as you'll see in a minute.) If you continue to subclass several levels down, and, at level four you change the background color from purple to red, you've broken inheritance again, and all classes created from that red subclass will then have a red

background color. Furthermore, of course, all forms created from the class with the red background color will have a red background. The important thing to know, now, is that if you change the background color of your base form from purple to green, the classes and forms that have red background colors will not change to green. Once you have overridden a property or method, you don't get it back automatically.

Remember that this is all property and method specific. Overriding one property in a class does not have any effect on any of the other properties or methods.

Overriding method inheritance

Methods behave the same way, but there is additional functionality to be aware of due to the nature of program code vs. values. An object can't have more than one value of a property—for example, if a background color is blue, it can't also be yellow (no, the two properties wouldn't combine to create green). However, it is possible to "add to" a method.

Just as with properties, methods are inherited. Suppose you've created a command button with a Click method that contains the following code:

```
beep()
```

Thus, every time the command button is pressed, the computer's speaker beeps. Annoying, but suitable for this discussion. Now, when you subclass the command button, the subclass will also have a Click method, but if you open the subclass' Click method in the Code window, there won't be anything there. However, when you click on the command button, it will beep, because it inherits the Click method of its parent.

You'll want to be able to do two things. The first is to override the Click method code of the parent and replace it with another piece of code, such as the following:

```
wait window "You have pressed this Command Button"
```

You can see that the Click method in the parent has been overridden by the Click method of this class. If you subclass this class, that level will also execute the Wait Window code because we have broken inheritance, just as with the color of the forms above.

Preventing a parent's method from firing

The next thing you're going to want to do is override the parent's method but not do anything in its place. Here's a nice trick. To do this, you needn't invent any complex code: Anything contained in the method of the current class will cause the parent's event to be ignored. As a result, you can use a simple comment, such as

```
* override
```

in the Click method of the form or subclass.

Preventing an event from firing

Watch carefully, because this next point is subtle. The last several sections have all addressed the code in a method. As I discussed earlier, there is a difference between events and methods.

There are times when you might want the behavior of an event to be suppressed. (Are you picturing the peasant in *Monty Python and the Holy Grail*, hollering, "He's repressing me…"?)

For example, you can create your own pick list with a grid that features incremental search—as the user types a character, the highlight in the pick list repositions itself to the nearest entry containing those characters. The grid would have a text box in the column that's being searched. As you know, the default behavior of the KeyPress event of a text box is to display the character typed in the text box. You don't actually want the character typed to be displayed, so you would want that behavior—displaying the character—to be overridden.

Use the NODEFAULT keyword in the event's method to do this. For example, you could write code to trap the character being typed, perhaps storing it to a property of the pick list, and then using NODEFAULT to prevent the character from being displayed, like so:

```
* they pressed an alphanumeric key
 nodefault
 * append the last key pressed to the keyboard buffer property
 this.cStuffInKBBuffer = this.cStuffInKBBuffer + chr(nKeyCode)
 * look for the complete string
 if not seek(upper(this.cStuffInKBBuffer))
  go this.nCurRecNum
 endif
```

The actual code would be more complex because you would want to test for the time elapsed between keypresses, but you get the idea.

Calling a parent's method in addition to the local method

Another task you're likely want to perform is to call the parent's method in addition to the code in the current method. In other words, this time you're mixing the blue and the yellow colors, and want green. You can reference the code of the parent class using the :: operator (referred to as the *scope resolution operator*) or the DODEFAULT() keyword. The DODEFAULT() keyword is usually preferred because you don't have to know the name of the parent class you're referencing.

Suppose the parent class is named cmdDrill and you've subclassed it with a name of cmdDrillLocation. The click event of cmdDrill has the following code:

```
beep()
```

If you simply put code in the Click method of the cmdDrillLocation subclass, you would end up overriding any code in the cmdDrill class's Click method. To call the BEEP code in the cmdDrill class as well as the code in the Click event of the cmdDrillLocation class, you would use the following code:

```
Dodefault()
<code in the cmdDrillLocation class>
```

or

```
cmdDrill::click()
<code in the cmdDrillLocation class>
```

Both of these code snippets would call the parent class's Click method first, and then execute the Click method of the current class. You could put a call to dodefault() at the beginning or end of a method, depending on when you wanted that behavior to be executed. Even better, you could put it in the middle of a method, and have the other code in the method act as a "wrapper" for the code in the parent class's method.

Suggested base class modifications

Isn't it irritating to read lines like "Make modifications to your own form base class as desired," as I did earlier in this chapter? It's the post-college equivalent of "This exercise is left to the reader." When you were a student, you actually believed the author—but it really meant that the author had no clue as to the answer.

Similarly, wouldn't it be much more useful if I spent a couple more pages and described some concrete modifications for your own base classes? I know all the beta testers would have loved to have a series of examples when they first broke open their copy of VFP 3.0.

So here's a quick list of suggestions of properties and methods you might want to set or add to your classes. This is just to get you started—obviously you'll want to expand this list according to your own needs, wants, and style.

Control	Property/Method	Description
Everything	About (custom method)	Add to each class and use for your own extended documentation.
	Release() `Release This`	
Forms	AutoCenter	Set to .T. so the form will display in the center of the screen.
	Caption	Change to "frm" so you know when you accidentally created a form with a VFP base class, because it will have a caption of "Form1."
	MaxButton	Set to .F. so the user can't maximize the form and then wonder why the form is full-screen but the controls are still jammed in the upper-left corner of the form.

Control	Property/Method	Description
Forms *(continued)*	Load()	See **Listing 10.1** at the end of this table.

	Destroy()
	
	`*Hide the form so it`
	`*appears to go away`
	`*faster.`
	
	`This.Hide()`	

Labels	AutoSize = .T. BackStyle = Transparent Caption = "lbl"	
Command Buttons	Caption = "cmd"	
Page Frames	ActivePage = 0 PageCount = 0 TabStyle = Nonjustified	
All data-related controls—text boxes, check boxes, combos, lists, etc.	AnyChange (custom method)	Add the following code to the InterActiveChange and ProgrammaticChange events: `This.AnyChange()` Why? So you can centralize code that should be called if either the InterActiveChange event or the ProgrammaticChange event is called. You can still put code that is specific to either InterActiveChange or ProgrammaticChange in the respective method, but this technique saves a lot of redundant code.
	Valid()	See **Listing 10.2** at the end of this table.
	Validation (custom method)	This method is called from the Valid() method.
Text and Edit Boxes	IntegralHeight = .T. SelectOnEntry = .T. Valid()	
Check Boxes	AutoSize = .T. BackStyle = Transparent Caption = "chk" Valid()	
Option Groups	OptionGroup AutoSize = .T. BackStyle = Transparent ButtonCount = 12 Valid()	Option groups have buttons that are not already subclassed and can't be subclassed. So ignore the Option Button class. I create an Option group with a dozen option buttons that are already preconfigured. Then I set Enabled and Visible to False and resize the option group. That way, the buttons have the properties already. Option button properties: AutoSize = .T. Caption = "1" through "12"
List Boxes	IntegralHeight = .T.	List boxes don't get a Valid() because the Valid() fires each time you move the highlight in the list box.

Control	Property/Method	Description
List and Combo Boxes	aItems[1] (custom property) RowSourceType = 5 - Array RowSource = this.aItems ItemTips = .T. SelectOnEntry = .T. Init() This.aItems[1] = ""	Enter "aItems[1]" in the property name so that Visual FoxPro knows it's an array. You might want to turn ItemTips to True, so that the entire row is displayed when the combo is opened. You might want to turn SelectOnEntry to True so that an item in the combo is automatically selected when the user tabs into the control.
Spinners	SelectOnEntry = .T.	
Grids	AllowHeaderSizing = .F. AllowRowSizing = .F. DeleteMark = .F. RecordMark = .F. SplitBar = .F.	

***Listing 10.1**. The Form controls' Load() method code from the above table.*

```
* Set some environmental things the way we want.
set talk off
if oApp.lCentIsOn
 set century on
else
 set century off
endif
set deleted on
if upper(oApp.cMethod) = "DEV"
 set escape on
else
 set escape off
endif
set exact off
set exclusive off
set fullpath on
set near off
set safety off
if oApp.cMethod = "DEV"
 set strictdate to 2
else
 set strictdate to 1
endif
set unique off
```

***Listing 10.2**. The data-related controls' Valid() method code from the above table.*

```
*\\\ except list boxes
*\\\ compliments of Doug Hennig
* If the Valid method is fired because the user clicked on a button
* with the Cancel property set to .T. or if the button has an lCancel
* property (which is part of the SFCommandButton base class) and it's
* .T., don't bother doing the rest of the validation.
```

```
local oObject
oObject = sys(1270)
if lastkey() = 27 or (type('oObject.lCancel') = 'L' and ;
  oObject.lCancel)
  return .T.
endif lastkey() = 27 ...

* Do the custom validation (this allows the developer to put custom
* validation code into the Validation method rather than having to
* use code like the following in the Valid method:

* dodefault()
* custom code here
* nodefault

return This.Validation()
```

Up close with the Class Designer

The Class Designer has the same interface and nearly the same functionality as the Form Designer. You can open it by using the MODIFY CLASS command or after you've defined a class with the New Class dialog that results from the CREATE CLASS command.

The differences from the Form Designer have to do with the fact that a class is a definition of an object and can't be "run" itself, any more than the mold for a firing pin can be placed into a rifle and used to fire the cartridge. Thus, the Run icon is disabled when the Class Designer window is active, and the shortcut menu does not have a Run menu option.

The Class menu pad appears on the menu bar when the Class Designer is the active window. The first four menu options—New Property, New Method, Edit Property/Method, and Include File—are the same as those found on the Form menu that appears when the Form Designer is the active window.

The fifth menu option, Class Info, opens the Class Info dialog. This dialog is used to attach information and icons to the class.

The Toolbar Icon text box and ellipsis button allow you to select an icon that will be displayed on the class's toolbar button when you register this class under the Tools, Options menu option, and beside the name of the class when it is indented under the class library name in the Project Manager. The Container Icon text box and ellipsis button allow you to select an icon that will be displayed in the Class Browser.

The OLE Public check box is used to specify that the class can be used to generate a custom Automation server when it is built through the Project Manager.

The Members tab of the Class Info dialog simply has a list box that displays all members of the class. A "member" is a generic term that encompasses both properties and methods. This list box isn't all that useful, actually, since (1) it doesn't identify whether a member is a property or a method (yeah, it *should* be obvious, but if you're spelunking through someone else's class, it may well not be), and (2) it doesn't really tell you much. The Edit Property/Method dialog is much more useful in my opinion. Click the Modify button on this tab to get to the Edit dialog.

Up close with the Class Browser

The Class Browser is a tool that helps in managing class libraries. Because multiple classes can be contained in the same class library, even to the extent of multiple levels of classes, it can be difficult to keep straight where in the class hierarchy a specific class is located. Furthermore, because classes reference each other through file name references stored in the class library, it can be very difficult to perform simple maintenance tasks such as renaming a class or moving it from one location to another. The Class Browser makes short work of many of these tasks.

Starting the Class Browser

You can select Class Browser from the Tools menu or call it manually using the following line of code:

```
do home() + "browser.app"
```

If you do so, you will be prompted for the name of the class library with which you want to work.

You can also store the name of the Class Browser application, BROWSER.APP, to the _BROWSER system memory variable (in the File Locations tab of the Tools, Options dialog), and then issue the command:

```
do (_browser)
```

The Class Browser takes an optional parameter. If you pass it the name of a class library, you will not be prompted for the class library:

```
do home() + "browser.app" with "baseform"
```

Using the Class Browser

The Class Browser is a single form that displays all classes in a class library. It has a number of functions that allow you to view and manipulate those classes in a variety of ways. See **Figure 10.9**.

The list box in the upper-left quadrant of the Class Browser shows the hierarchy of the classes in the class library you are browsing. To view a single class type, you can filter the list on the left by using the Type drop-down combo box.

The right panel contains a list of all exposed members of the selected class. You can right-click and select to display Protected, Hidden, and Empty members as well. The Class Browser will remember the settings you've selected from session to session.

Here is a list of toolbar buttons, from left to right:

- Component Gallery—Opens the Visual FoxPro Component Gallery, which will be discussed in Chapter 17.
- Open—Opens a class library (and closes the current one, if one is open). If you right-click on the button, you'll see a list of the class libraries you most recently opened.

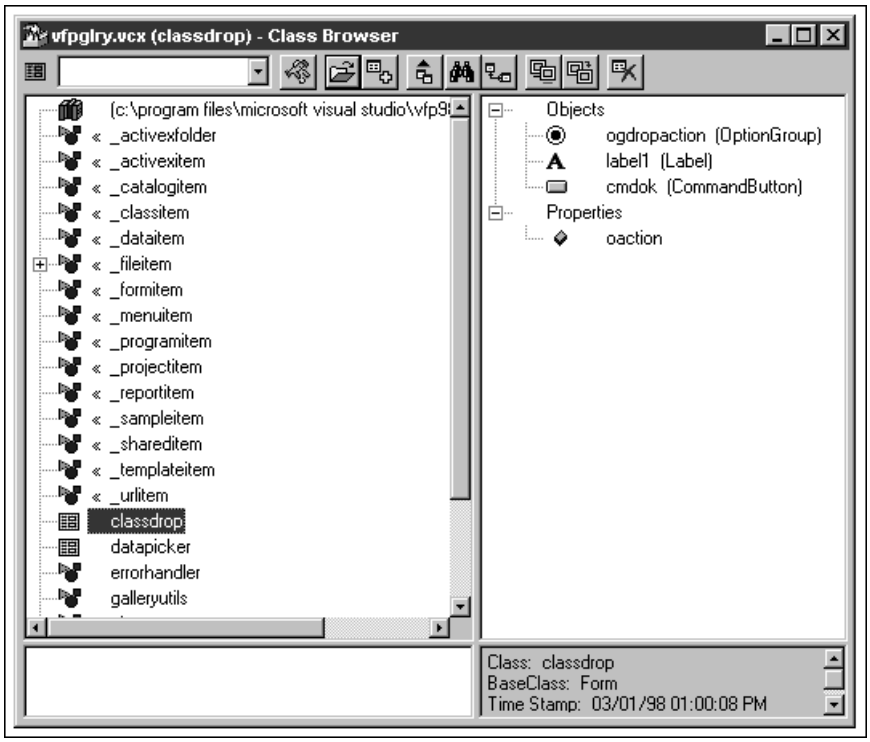

***Figure 10.9**. The Visual FoxPro 6.0 Class Browser.*

- View Additional File—Opens another class library in the Class Browser. This is particularly handy if your inheritance hierarchy spans multiple class libraries. Again, right-clicking displays a list of libraries you recently opened.
- View Class Code—Generates a file called VIEWCODE.PRG, which is the programmatic equivalent of the class.
- Search—Allows you to search through a class or class library for a string of text. Note that because you can search for text in the descriptions as well, it behooves you to document your classes well!
- New Class—Creates a new class in the same fashion as the CREATE CLASS command. However, the Based On and Store In values are already filled in with defaults pointing to the current class and library in the Class Browser.
- Rename—Allows you to rename a class or properties and methods. Renaming a class can be very dangerous because it can break inheritance. If you rename a parent of a class, you might not be able to instantiate that class anymore—nor will you be able to even open the class to modify it. You might as well throw it away. Renaming properties and methods can be equally dangerous, because source code in other methods that referenced the old name will no longer work.

- Redefine—Allows you to change the parent class of a class. Suppose you had two text-box classes, txtEditable and txtReadOnly. You then subclassed txtReadOnly to create a specialized data-entry text box, and only after you were just about done did you realize that you had subclassed the wrong class—you actually meant to subclass txtEditable. (How useful would a ReadOnly data-entry text box be?) Redefine allows you to fix this type of mistake quickly and easily. You need to make sure that the target class has the same structure as the source class—if it doesn't, you can lose code and otherwise mess up your classes.
- Clean Up—Essentially performs a "pack" on a .VCX file. When you modify a class library, the records themselves in the .VCX aren't modified; they're deleted and new copies are appended. As you know, the deleted records are still hanging around—they're just flagged as deleted. As a result, a .VCX undergoing a lot of maintenance can get very large. This function will remove all of the deleted records permanently.

The icon to the left of the combo box represents the class selected in the list box. You can use this icon to place an object of that class on a form—yes, yet another way to add controls to a form! Drag the icon onto the Visual FoxPro screen or onto an appropriate container (like a form).

You can also modify a class from the Class Browser by double-clicking on the class in the list box.

Chevrons next to a class in the list box indicate that the parent class for that class is not in the same class library and isn't currently open in the Class Browser.

The Class Browser was designed with an open architecture so that others could extend it as needed. One method is to write a separate application and register it with the Class Browser. These add-ins can be displayed by clicking the Add-ins button.

There is a world of hidden functionality in the Class Browser. Markus Egger has documented just about all of it in detail in his book, *Advanced Object-Oriented Programming with Visual FoxPro 6.0*. If you're at all interested in using the Class Browser, I suggest you refer to this book.