

Introduction to Object Oriented Programming

Session: CBER1

*Craig Berntson
3M Health Information Systems
3757 South 700 West #21
Salt Lake City, UT 84119
Voice: 801-699-8782
www.craigberntson.com
Email: craig@craigberntson.com*

Overview

In this session, you will learn the basics of Object Oriented Programming (OOP). You will learn the terminology and see how Visual FoxPro supports OO techniques. You will also learn about the Object Browser, a tool used to maintain class libraries. Finally, the session will cover the Fox Foundation Classes, a group of classes that ship with Visual FoxPro that you can use in your applications.

Terminology

Before you can begin using OOP, it's important to know the terminology used. Many of the terms are simply names for things you may already know. Others may be new to you.

Class

The basic and most important concept you will deal with is a class. A class is a template that specifies the attributes and behavior of something. A class models a real world thing, such as an invoice. When discussing an invoice class, you are not talking about a specific invoice, but rather a description of an invoice. You don't even include specific data items in a class.

Attribute

The attributes are the items that make up the thing being modeled. Visual FoxPro calls these properties. In general OOP terms, an attribute is called an instance variable. An invoice would have attributes such as date, invoice number, billing address, shipping address, and line items. A class never includes specific information about an invoice. A class definition would not have information (attributes) such as the specific date of the invoice, only an attribute that a date would be included.

Behavior

Behaviors are things the object does. In VFP terminology, a behavior is a method. An invoice object would have behaviors such as print, calculate shipping charges, and add or delete a line item. A class may have code behind the behavior that is run when the method is called.

Behaviors can also include things that a class does automatically. These are called events. A command button has a click event that automatically fires when you click on the button. However, if you want custom code to run when the user clicks on the button, you do not put code in the event, but rather in the event method. The properties, events and methods of a class are collectively called PEMs.

Polymorphism

What sounds like a big, confusing word, is actually quite simple. Polymorphism means that two classes can have a behavior with the same name, but two different things can happen. For example, a command button and an option group both have click events, but different code can run in each one.

Encapsulation

No, this isn't what is done to an astronaut before being launched into space. When a class is encapsulated, it includes everything it needs to do its job. For example, an invoice class would include a behavior to calculate tax. Instead of an external program to print an invoice, the invoice class would include a print behavior.

Subclass

This is not a course my son took before joining the Navy. A subclass is a class based on another class. When you create the subclass, it may inherit all the attributes and behaviors of its parent

class. (OK, you caught me sneaking in some additional terminology here. Be patient and I'll explain the other terms.)

So, why should you subclass? Simply put, because it makes your job easier and the application better. Look at the UML diagram in Figure 1.

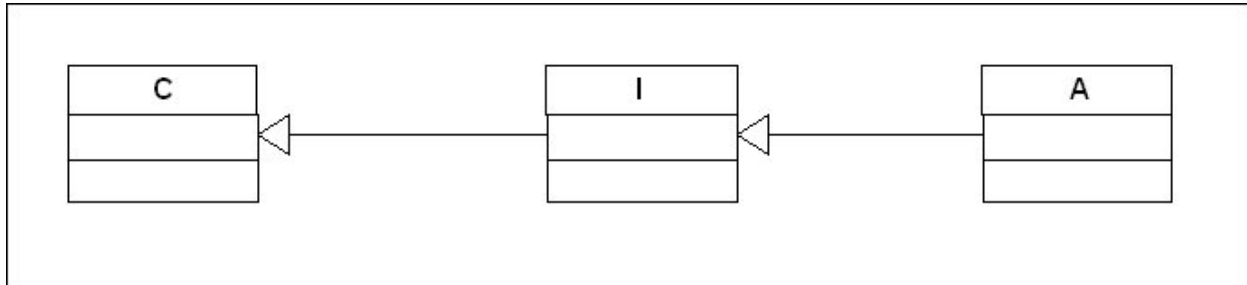


Figure 1. A UML diagram showing subclasses and inheritance.

This diagram shows three classes, C, I, and A. From looking at the diagram, you can see that C is the parent class and I is derived from or is a subclass of C. You can also say that C is the parent class of I. Furthermore, A is a subclass of I. VFP ships with several base classes (Yes, you're correct. More terminology that I haven't explained.) In a well-designed application, you won't use the VFP base classes directly, but rather you would subclass them (level C in the diagram). For the sake of this diagram, the C level is your corporate level classes. For example, you have particular attributes or behaviors that you want to include in every command button. Make these changes at the C level.

You can then subclass the corporate level class into an intermediate or I level. These may be classes that you always use for a particular client. Any customization you make at the C level may be available at the I level subclass.

You may further subclass to the A or application specific level. One great benefit of doing things this way is that any changes you make at the C level only need to be tested and debugged at that level. You don't need to test this at the I or A level or each time you use the C level class.

Need another reason why you should subclass? What if you have an application that consists of several hundred data entry forms? One day, your boss comes to you and says that you need to change the background color of every form to bright pink. If you have just used the VFP base classes, you will need to open and modify every single form. However, if each form is based on a form subclass, you only need to change the subclass and recompile. The hundreds of forms will inherit the new background color automatically.

Inheritance

A subclass gets the attributes and methods of its parent class. You may also hear about multiple inheritance. This means that a subclass can have more than one parent class. VFP does not support multiple inheritance. This may actually be a good thing as multiple inheritance introduces additional problems. For example, if the Error methods of the two parent forms handle errors in a different way. Which error handling code should the subclass use?

Object

An object is a specific instance of a class. Earlier, when I discussed an invoice class, I said that the class was a description of an invoice. The object is the actual invoice. So, you can have an object that is invoice number 34534. When you create an invoice object, you instantiate, or create it.

That pretty much covers the terminology. It wasn't too bad, was it? I will introduce some additional terms as we go along, but you now have the basics.

Doing it with Class

Visual FoxPro includes a number of base classes. These are divided into visual and non-visual classes. Visual classes create objects that are visible on the screen. Non-visual classes are not visible. All subclasses you create will be created from one of the base classes. Table 1 lists the VFP base classes.

Table 1. A List of Visual FoxPro baseclasses.

Visual base class	Visual subclassing?	Non-visual base class	Visual subclassing?
Checkbox	Yes	Collection	Yes
Column	No	Cursor	Yes
Combobox	Yes	CursorAdapter	Yes
CommandButton	Yes	Custom	Yes
CommandGroup	Yes	Empty	No
Container	Yes	Exception	No
Control	Yes	DataEnvironment	Yes
EditBox	Yes	OleBoundControl	Yes
Form	Yes	OleControl	Yes
FormSet	Yes	ProjectHook	Yes
Grid	Yes	Relation	Yes
Header	No	ReportListener	Yes
Hyperlink	Yes	Session	No
Image	Yes	Timer	Yes
Label	Yes	XMLAdapter	Yes
Line	Yes	XMLTable	Yes
Listbox	Yes	XMLField	Yes
OptionButton	Yes		
OptionGroup	Yes		
OleBoundControl	Yes		
OleControl	Yes		
Page	Yes		
PageFrame	Yes		
Separator	Yes		
Shape	Yes		
Spinner	Yes		
Textbox	Yes		
Toolbar	Yes		

Visual Subclassing

Visual subclassing does not mean that you create a subclass of a visual class, but rather you use the Visual Class Designer to create the subclass. Not all the base classes can be visually subclassed. See Table 1 to determine which classes can be visually subclassed.

Subclasses that have been visually subclassed are stored in a Visual Class Library. This is a group of files that hold the definitions for the subclass. Visual class libraries are simply DBF files with a VCX and VCT extensions.

You will usually group subclasses into a Visual Class Library by usage. For example, you may have several subclasses that deal with file handling that are placed in FileHandling.vcx

The following steps walk you through visually creating a subclass.

1. Type CREATE CLASS in the Command Window and press Enter. The New Class dialog (Figure 2) is displayed.

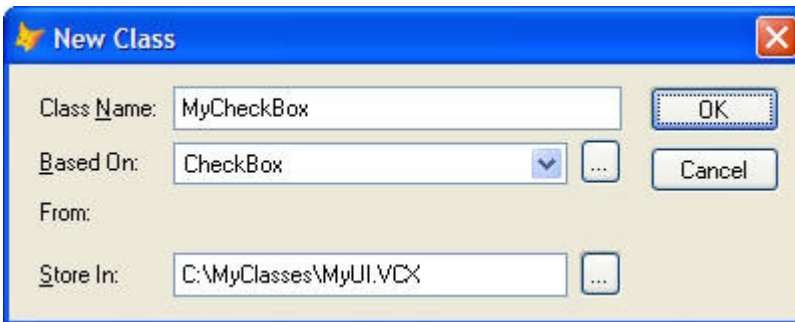


Figure 2. The New Class dialog is where you enter a new class and assign it to a class library.

2. Enter the Class Name. The name cannot have any spaces and must begin with a letter or underscore.
3. Select the Parent Class on which the subclass will be based. The VFP base classes are listed in the drop down or you can click the browse button to select a class from a Visual Class Library.
4. Enter the name of the Visual Class Library where the new subclass will be stored.
5. Click OK. The new subclass will be displayed in the Class Designer (Figure 3).

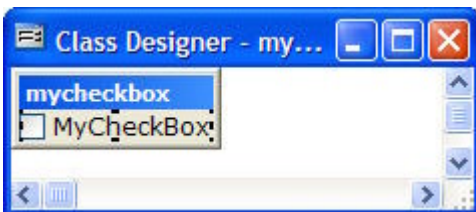


Figure 3. The Class Designer is used to create and modify a subclass.

6. Using the Property Sheet, change the FontName to Tahoma, then FontSize to 8, AutoSize to .T., Alignment to 0, and Caption to MyCheckbox.
7. Close the designer and save the changes.

Non-visual Subclassing

All VFP base classes, except the Empty class, can be non-visually subclassed. The class definition is stored in a PRG file. Here's the same subclass that was created visually.

```
DEFINE CLASS MyCheckbox AS checkbox
    Height = 15
    Width = 92
    FontName = "Tahoma"
    FontSize = 8
    AutoSize = .T.
    Alignment = 0
    Caption = "MyCheckbox"
    Name = "mycheckbox"
ENDDDEFINE
```

PEMs

PEMs are the properties, events, and methods of a class. By default, there are several PEMs for each class. You can add your own custom properties and methods to a subclass. The new property or method will show up in the Property Sheet for the class. You cannot add custom Events to a subclass. The following steps show how to add a custom property to a subclass.

1. Modify an existing subclass or create a new subclass from another class in the Class Designer.
2. Select Class | New Property from the menu. The New Property dialog (Figure 4) is displayed.

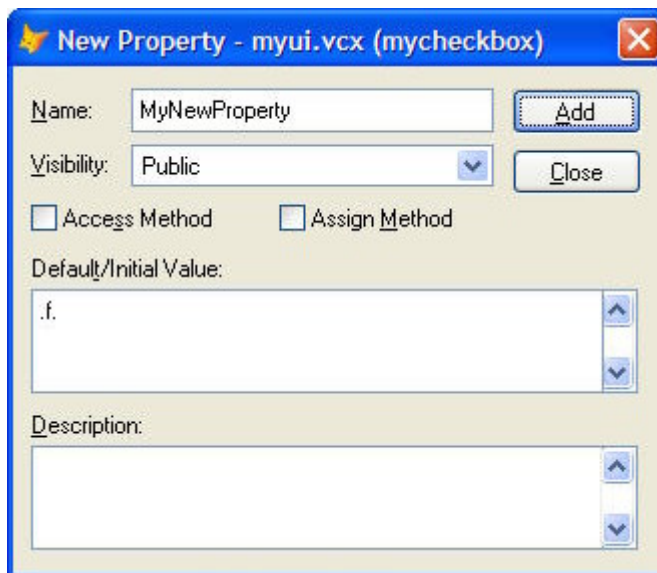


Figure 4. Use the New Property dialog to add a new property to a subclass.

3. Enter the name for the new Property
4. Select Visibility. (Note: Visibility and Access and Assign Methods are discussed later in this section.)
5. Check Access Method or Assign Method if you want to add them to the new Property.
6. Enter the Default or Initial Value for the Property. If you don't change this, the Value for the new Property will be False.

7. Enter a Description for the Property
8. Click Add to add the property.
9. Click Close to dismiss the New Property dialog.

The next steps show how to add a custom method to a class.

1. Modify an existing subclass or create a new subclass from another class in the Class Designer.
2. Select Class | Method from the menu. The New Method dialog (Figure 5) is displayed.

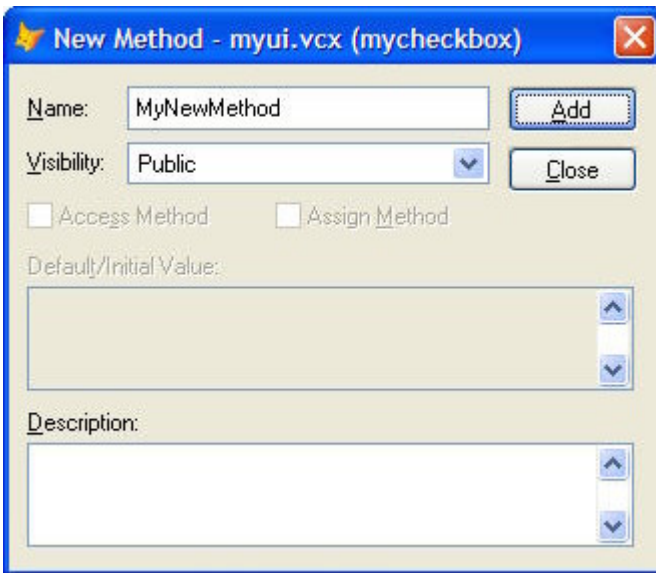


Figure 5. Use the New Method dialog to add a new method to a subclass.

3. Enter the name for the new Method.
4. Select the Visibility. (Note: Visibility is discussed later in this section.)
5. Enter a Description for the Method.
6. Click Add to add the Method.
7. Click Close to dismiss the New Method dialog.

Visibility

Each property and method can be given specific visibility. That is, you can determine how a PEM is seen outside a class. There are three possible visibility settings.

- Public – Visible to the class, subclass, and the outside world
- Protected – Visible to the class and subclass. Hidden from the outside world.
- Hidden – Visible to the class. Hidden from subclasses and the outside world.

You change the visibility based on the usage of the PEM. For example, you may have some properties and methods that are needed for the class to work, but you don't want a subclass or code outside the class to have access to them. In this case, you make the property hidden.

Thought must go into the visibility setting as data hiding is one of the most important concepts of Object Oriented Programming. Data hiding means that the data and methods internal to an object are hidden from the outside. That way, the data can't be modified by an outside function or method. Data hiding can even go so far as to say that properties are only accessed by method calls, rather than referring directly to the property.

You set the visibility of a method or property when you create it or from the Edit Property/Method dialog (Figure 6) of the class designer.

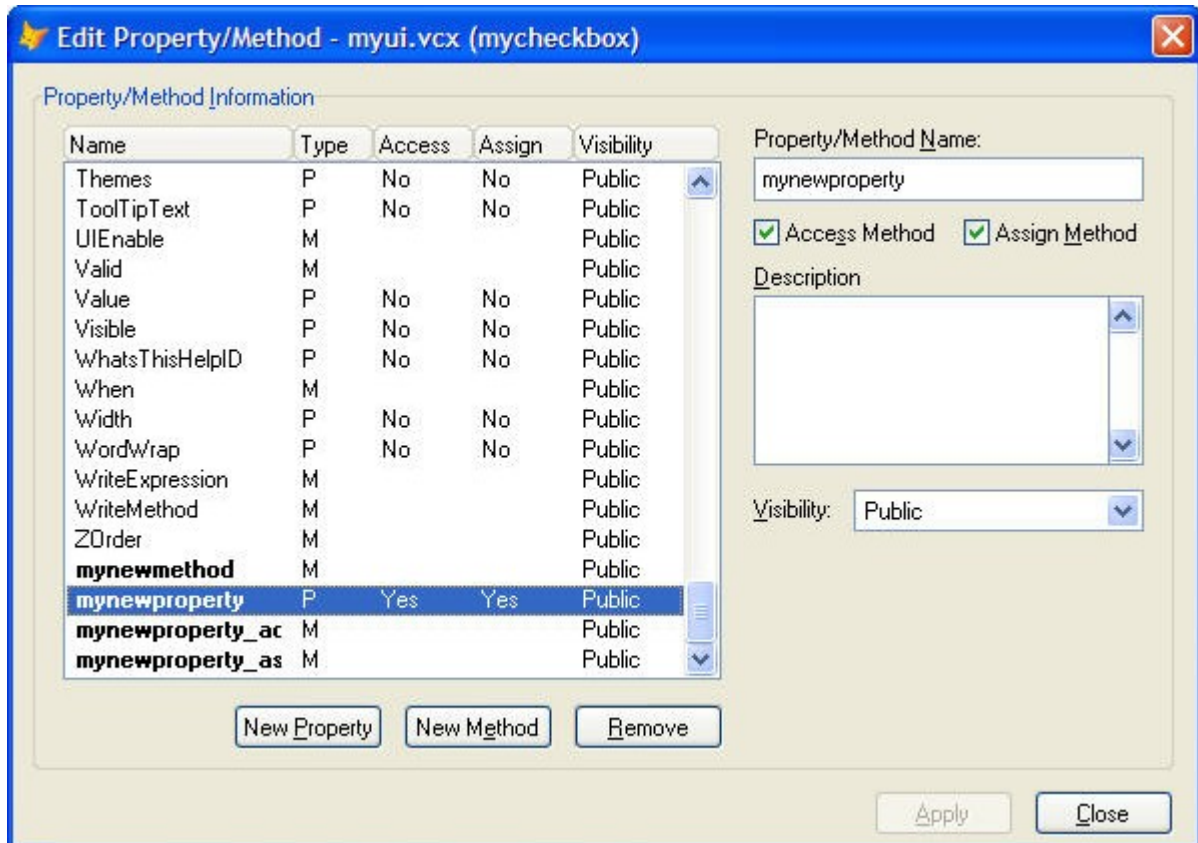


Figure 6. The Edit Property/Method dialog is used to change information about a property or method.

You can also set the visibility and create access and assign methods when you create the class in code. If you don't specify visibility, the new property or method will default to Public visibility. Note there are some exceptions on specific classes, such as the Session class.

Access and Assign

In addition to visibility, properties also have Access and Assign methods. In OOP parlance, these are called Set and Get. The Access method contains code that fires whenever the property is read or accessed. The Assign method is run whenever the value of the property is changed. The methods show up in the Property Sheet for the class and can be edited like any other method.

The following sample shows how to create the above class in code.

```
DEFINE CLASS mycheckbox AS checkbox
    Height = 16
    Width = 97
    FontName = "Verdana"
    AutoSize = .T.
    Alignment = 0
    Caption = "MyCheckBox"
    Name = "MyCheckBox"
    MyNewProp = .F.

    PROCEDURE MyNewMethod
        MESSAGEBOX("Hello from a custom class")
    ENDPROC

    PROCEDURE MyNewProp_Access
        *To do: Modify this routine for the Access method
        RETURN THIS.MyNewProp
    ENDPROC

    PROCEDURE MyNewProp_Assign
        LPARAMETERS vNewVal
        *To do: Modify this routine for the Assign method
        THIS.MyNewProp = m.vNewVal
    ENDPROC
ENDDDEFINE
```

The code in the Access and Assign methods is the default code automatically added by Visual FoxPro when the class was created in the Visual Class Designer.

Using classes

Now that you know how to create a class, you need to know how to use it. Visual classes are the easiest to use. Simply drag them from the toolbar or toolbox onto the designer.

To use a non-visual class, you must create it in code using `CREATEOBJECT()` or `NEWOBJECT()`. These two commands are similar. `NEWOBJECT()` lets you specify the file where the class is stored. You can also add an object to an existing object using `ADDOBJECT()`.

Whenever you instantiate an object, the class constructor is run. The constructor does all the things that need to happen to create the object. This would be things like run the `Load()` and `Init()` methods and set the initial values for the properties.

When you finish using an object and no longer need it in memory, the object's destructor is called. This runs all the code that is needed to destroy the object. This includes the object's `Unload()` and `Destroy()` methods.

When you create an object, you may not know what it is called. If an object needs to reference itself, you use the `This` keyword. When an object is on a form and you need to reference the form, use `ThisForm`. However, an object could also be placed on a container or a page. To reference them, use `This.Parent`. Don't confuse an object's parent with its parent class. When you call a method, you are said to "send the object a message".

When you put code in a subclass, by default, the code in the method of the parent class is not run. In this case, the subclass overrides the parent class. To run the code in the parent class, use the `DODEFAULT()` command.

Sometimes you don't want the VFP default functionality to happen. In this case, use the `NODEFAULT` command. For example, you can use `NODEFAULT` in the `KeyPress` method of a textbox to stop the key stroke from being added to the `Textbox.Value`.

It is also important that the correct object contains a specific property or hold the code for a specific method. This is called assigning or delegating responsibility. For example, you may have a data entry form with a Save button. A developer will frequently put the actual save code in the `Click()` method of the button. However, this is probably not the correct spot. The responsibility for saving the data should go on the form. So, the `Click()` method code would look something like:

```
ThisForm.SaveData()
```

The reason for doing things this way is that the form holds the data and has responsibility for updating the database. It also makes the Save button more generic, increasing reuse. Assigning responsibility to the correct object is an important OOP concept.

Composite Classes

One of the great benefits of creating classes is that it increases reuse. In other words, you don't have to create and test the same code multiple times. You can also combine classes to make another class. A class that is made up of other classes is called a composite class.

For example, how many times have you had to create a form that collected contact information such as name, address, and phone number? You could build this form every time or create a composite class. Start with a Container class and add labels and textboxes as needed (Figure 7). When the container is functioning exactly how you want, save it in a class library. To use the class, simply drag and drop it onto a form.

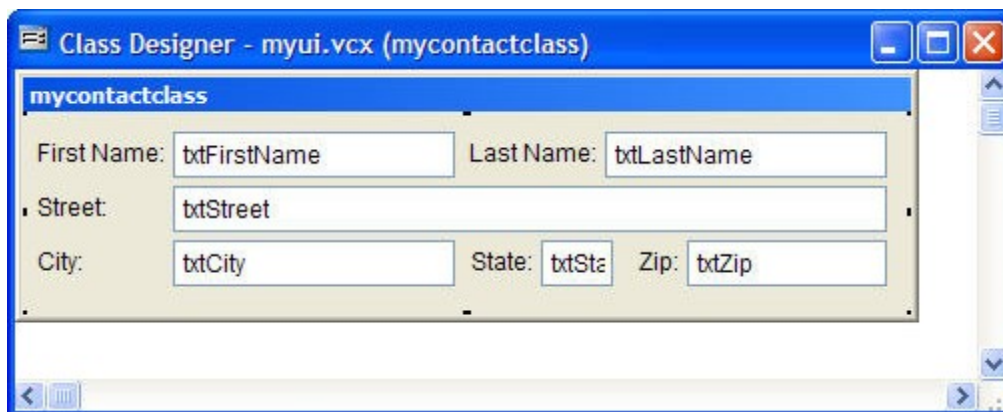


Figure 7. You can create composite classes such as this contact information container.

This all sounds very easy. However, it comes with some drawbacks. What happens if you create an application that needs a phone number or multiple lines for the street address or the country? You may find that the drawbacks outweigh the benefits. I'm not saying the composite classes are bad; I've used composite classes with great success. You just need to be aware of the pros and cons associated with using them.

Interface Inheritance

Earlier, I discussed inheritance and said that a subclass gets the attributes and methods of its parent class...and that's exactly what happens. What I didn't tell you is that there are two types of inheritance: implementation inheritance and interface inheritance.

Under implementation inheritance, a subclass not only gets the attributes and methods of the parent, it also gets the exact behavior of the parent class. For example, if you have a command button that displays a message when the Click() method is run, when you subclass that command button, the new class will, by default, inherit that code and display the same message. Implementation inheritance is what we're most accustomed to in VFP.

There is, however, another type of inheritance called interface inheritance that is used with COM objects. Under interface inheritance, the subclass only gets the method names and parameters. The code behind the method is not inherited. You need to write your own code to run in the subclass. The easiest way to do interface inheritance is to create a stub program using Visual FoxPro's Object Browser. The following steps show how to do this.

1. Open a new, blank program file (.PRG).
2. From the VFP menu, select Tools | Object Browser or click the Object Browser button on the Standard Toolbar. The Object Browser will open (Figure 8).

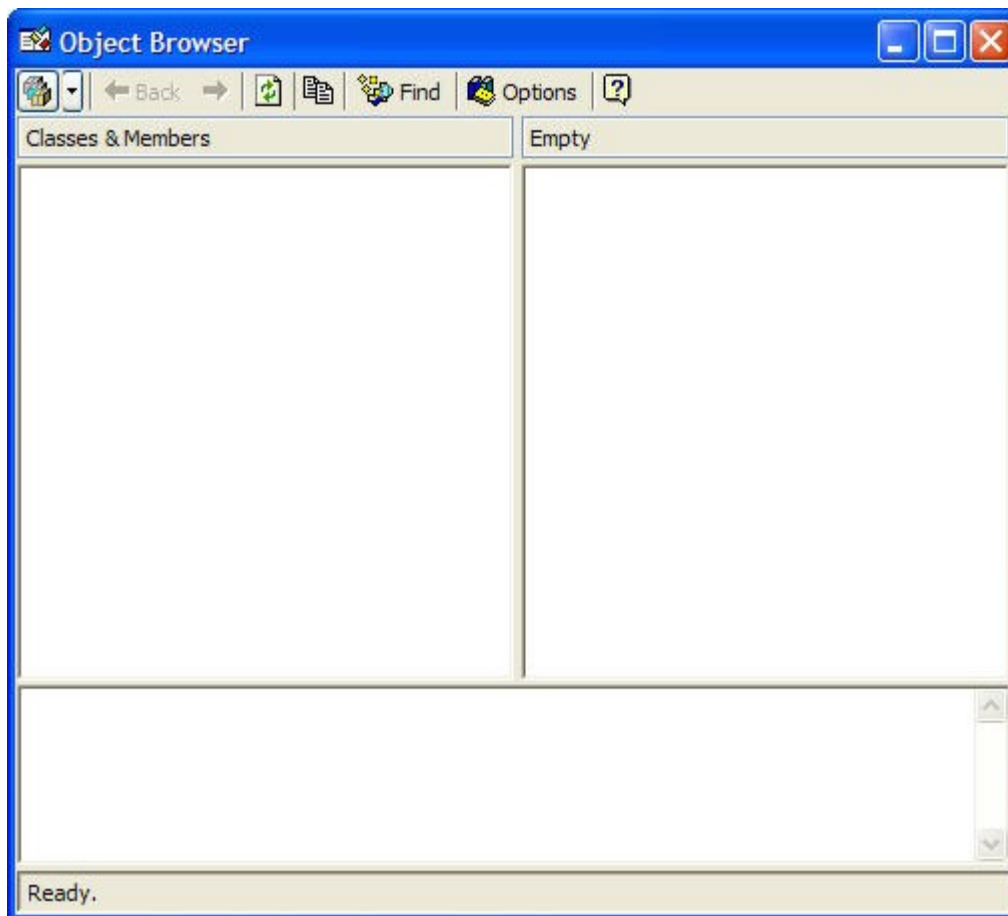


Figure 8. The Object Browser is used to find information about a COM component.

3. Click the Open Type Library button on the Object Browser. The Object Browser will begin reading Type Library information from the register, then display the Open dialog (Figure 9).

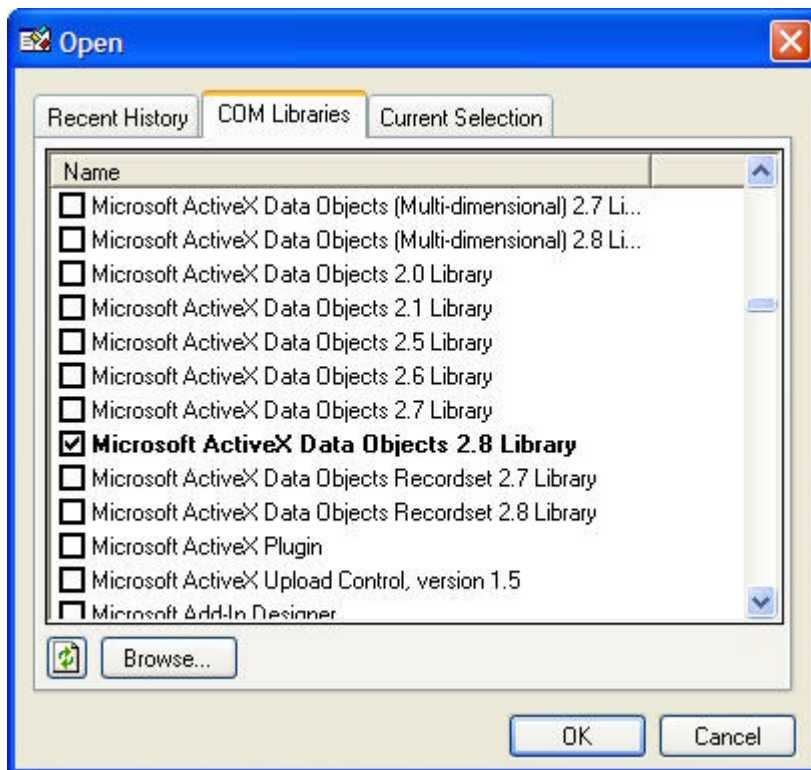


Figure 9. The Open dialog is where you select the type libraries to display in the Object Browser.

4. Select the COM Libraries page, check Microsoft ActiveX Data Objects and click OK. (Your version may be different than the one shown in the example). The Open dialog will close and return you to the Object Browser.
5. Expand the ADODB entry on the Classes & Members tree view.
6. Expand the Interfaces entry.
7. Scroll down to where you see RecordSetEvents under Interfaces. (Figure 10).

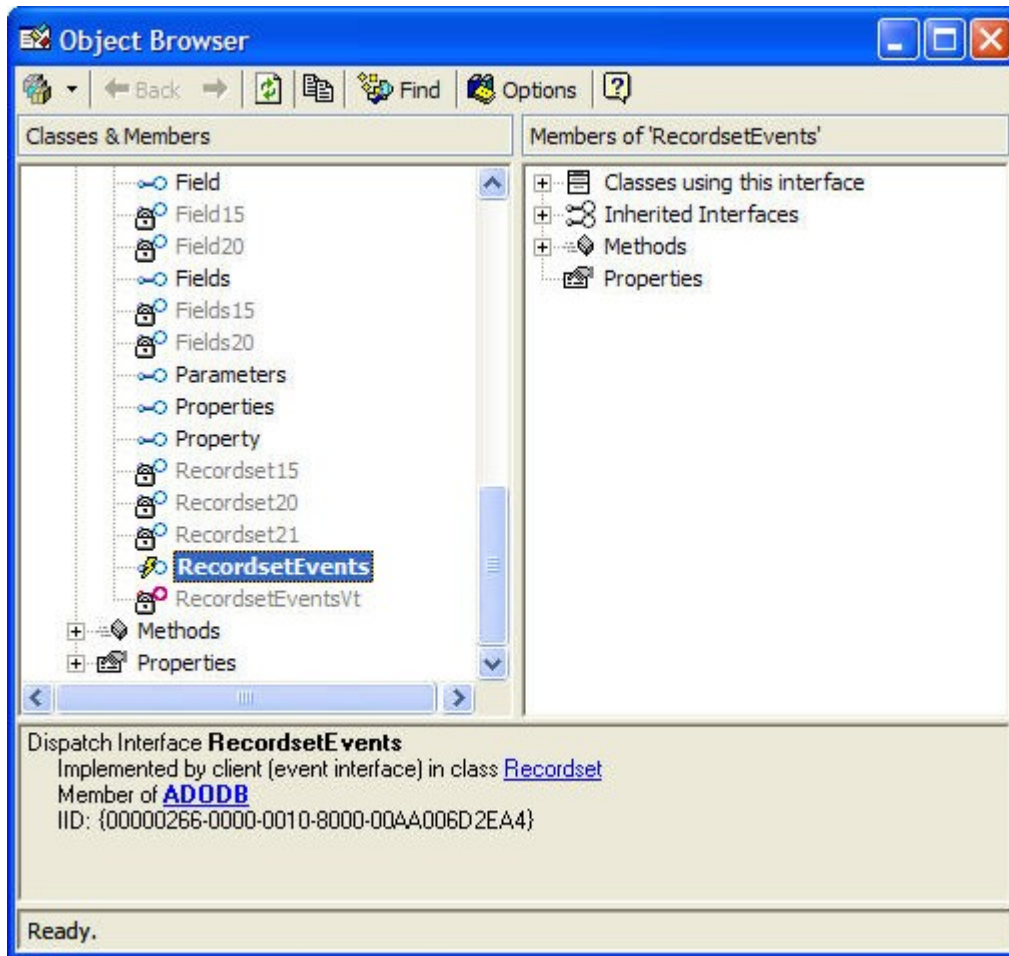


Figure 10. After selecting a Type Library, information about the selected COM component is displayed in the Object Browser.

8. Drag RecordSetEvents and drop it onto the open program window. The stub program will be created. However, one rule of using Interface Inheritance is that you must define every interface, so you cannot remove code from your program.

```
x=NEWOBJECT("myclass")

DEFINE CLASS myclass AS SESSION OLEPUBLIC

    IMPLEMENTS RecordsetEvents IN "c:\program files\common
files\system\ado\msado15.dll"

    PROCEDURE RecordsetEvents_WillChangeField(cFields AS NUMBER, FIELDS AS
VARIANT, adStatus AS VARIANT @, pRecordset AS VARIANT) AS VOID
        * add user code here
    ENDPROC

    PROCEDURE RecordsetEvents_FieldChangeComplete(cFields AS NUMBER, FIELDS AS
VARIANT, pError AS VARIANT, adStatus AS VARIANT @, pRecordset AS VARIANT) AS
VOID
        * add user code here
    ENDPROC
```

```
PROCEDURE RecordsetEvents_WillChangeRecord(adReason AS VARIANT, cRecords
AS NUMBER, adStatus AS VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_RecordChangeComplete(adReason AS VARIANT,
cRecords AS NUMBER, pError AS VARIANT, adStatus AS VARIANT @, pRecordset AS
VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_WillChangeRecordset(adReason AS VARIANT,
adStatus AS VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_RecordsetChangeComplete(adReason AS VARIANT,
pError AS VARIANT, adStatus AS VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_WillMove(adReason AS VARIANT, adStatus AS
VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_MoveComplete(adReason AS VARIANT, pError AS
VARIANT, adStatus AS VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_EndOfRecordset(fMoreData AS LOGICAL @, adStatus
AS VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_FetchProgress(Progress AS NUMBER, MaxProgress AS
NUMBER, adStatus AS VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC

PROCEDURE RecordsetEvents_FetchComplete(pError AS VARIANT, adStatus AS
VARIANT @, pRecordset AS VARIANT) AS VOID
* add user code here
ENDPROC
ENDDEFINE
```

The first thing you notice when looking at the code is that many procedures are defined, but there is not code there. The new class is using Interface inheritance. There are some other parts of the code that you may be unfamiliar to you.

The OLEPUBLIC keyword tells Visual FoxPro that this is a COM object. IMPLEMENTS means that the new method implements or uses the RecordSetEvents interface of the ADO COM

component. Each procedure, along with the help string, is defined in the COM component's type library. That's where Visual FoxPro got this information.

Now you can implement the events. The following code, taken from the Visual FoxPro help file, shows how you can implement the events. The only change I made was to use a valid VFP OLE DB connection string.

```
LOCAL oEvents
LOCAL oRS AS adodb.recordset
LOCAL oConn AS adodb.CONNECTION

oEvents = NEWOBJECT("myclass")
oConn = NEWOBJECT("adodb.connection")

oConn.Provider="MSDASQL"
oConn.ConnectionString="Provider=vfpoledb.1;Data Source=" ;
+ ADDBS(HOME()) + "Samples\Northwind\Northwind.DBC"
oConn.OPEN
oRS = oConn.Execute("select * from customer")
? EVENTHANDLER(oRS, oEvents)
?
? PADR(oRS.FIELDS(0).VALUE,20)
? EVENTHANDLER(oRS, oEvents, .T.)
oRS.MoveNext
? PADR(oRS.FIELDS(0).VALUE,20)
oRS.MoveNext
CLEAR ALL
RETURN

DEFINE CLASS myclass AS SESSION
    IMPLEMENTS RecordsetEvents IN "adodb.recordset"

    PROCEDURE Recordsetevents_WillChangeField(cFields AS NUMBER @, FIELDS AS
    VARIANT @, adStatus AS VARIANT @, pRecordset AS VARIANT @) AS VARIANT
    ? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

    PROCEDURE Recordsetevents_FieldChangeComplete(cFields AS NUMBER @, FIELDS
    AS VARIANT @, pError AS VARIANT @, adStatus AS VARIANT @, pRecordset AS
    VARIANT @) AS VARIANT
    ? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

    PROCEDURE Recordsetevents_WillChangeRecord(adReason AS VARIANT @, cRecords
    AS NUMBER @, adStatus AS VARIANT @, pRecordset AS VARIANT @) AS VARIANT
    ? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

    PROCEDURE Recordsetevents_RecordChangeComplete(adReason AS VARIANT @,
    cRecords AS NUMBER @, pError AS VARIANT @, adStatus AS VARIANT @, pRecordset
    AS VARIANT @) AS VARIANT
    ? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

    PROCEDURE Recordsetevents_WillChangeRecordset(adReason AS VARIANT @,
    adStatus AS VARIANT @, pRecordset AS VARIANT @) AS VARIANT
    ? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())
    ?adReason,adStatus,pRecordset.recordcount
```

```
PROCEDURE Recordsetevents_RecordsetChangeComplete(adReason AS VARIANT @,
pError AS VARIANT @, adStatus AS VARIANT @, pRecordset AS VARIANT @) AS
VARIANT
? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

PROCEDURE Recordsetevents_WillMove(adReason AS VARIANT @, adStatus AS
VARIANT @, pRecordset AS VARIANT @) AS VARIANT
? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

PROCEDURE Recordsetevents_MoveComplete(adReason AS VARIANT @, pError AS
VARIANT @, adStatus AS VARIANT @, pRecordset AS VARIANT @) AS VARIANT
? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

PROCEDURE Recordsetevents_EndOfRecordset(fMoreData AS LOGICAL @, adStatus
AS VARIANT @, pRecordset AS VARIANT @) AS VARIANT
? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

PROCEDURE Recordsetevents_FetchProgress(Progress AS NUMBER @, MaxProgress
AS NUMBER @, adStatus AS VARIANT @, pRecordset AS VARIANT @) AS VARIANT
? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())

PROCEDURE Recordsetevents_FetchComplete(pError AS VARIANT @, adStatus AS
VARIANT @, pRecordset AS VARIANT @) AS VARIANT
? " "+PROGRAM() + ' ' + TRANSFORM(DATETIME())
ENDDDEFINE
```

The Class Browser

By now, you may be thinking that creating, maintaining, and managing classes sounds like a lot of work. Luckily, Visual FoxPro ships with the Class Browser (Figure 11) to help you. Using the Class Browser, you can add, modify, copy, remove, rename, and redefine classes and more.

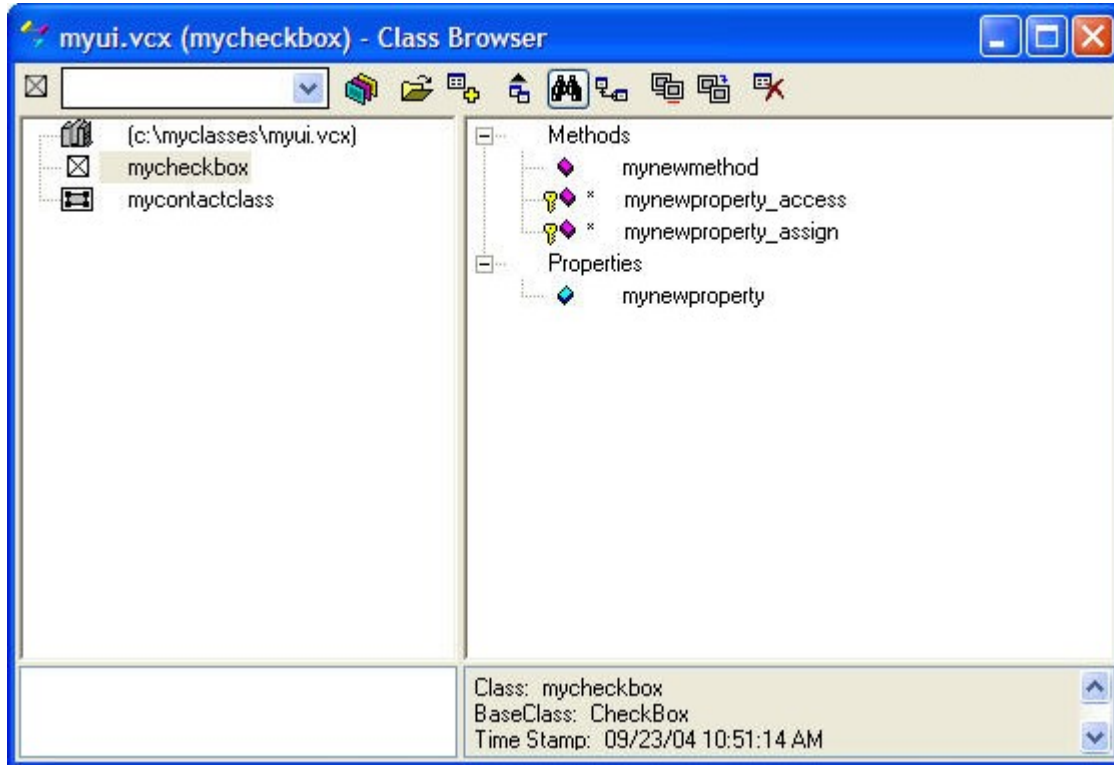


Figure 11. The Class Brower is used to manage Visual FoxPro classes.

The Class Browser toolbar has many of the functions needed to manage classes. From left to right, the controls are:

- Icon: The icon for the selected control in the tree view. If you drag the icon onto the desktop, the control will be instantiated and becomes a live object that you can test.
- Class Type: Filters the items in the tree view to the selected class type.
- Component Gallery: Toggles the tool between Component Gallery and Class Browser.
- Open: Opens a specific class library, form, or program file.
- View Additional File: Opens another class library, form, or program file in the same tree view.
- View Class Code: Displays the selected class as code. Note that you may not be able to run the code.
- Find: Opens a Find dialog to locate a specific item.
- New Class: Creates a new class.
- Rename: Renames the current class.
- Redefine: Redefines the parent class of the selected class.
- Cleanup Class Library: PACKs the current class library.

When you select a class from the tree view, its PEMs are displayed in the right-hand panel. There is also additional functionality available when you right-click on a class.

Fox Foundation Classes

Finally, Visual FoxPro ships with a number of predefined classes called the Fox Foundation Classes (FFC), located in the HOME() + “FFC” folder. These classes cover a wide variety of categories:

- Application
- Automation
- Buttons
- Data Navigation
- Data Query
- Data/Time
- Dialogs
- Internet
- Menus
- Multimedia
- Output
- Text Formatting
- User Controls
- Utilities

You use the classes from the FFC from either the Component Gallery or the Toolbox. The following steps walk you through exploring and using one class in the FFC.

1. Select Tools | Component Gallery from the Visual FoxPro Menu. The Component Gallery will open.
2. Expand the Foundation Classes item from the tree view.
3. Select Dialogs from the tree view. The right-hand panel will display the classes in the selected catalog (Figure 12).

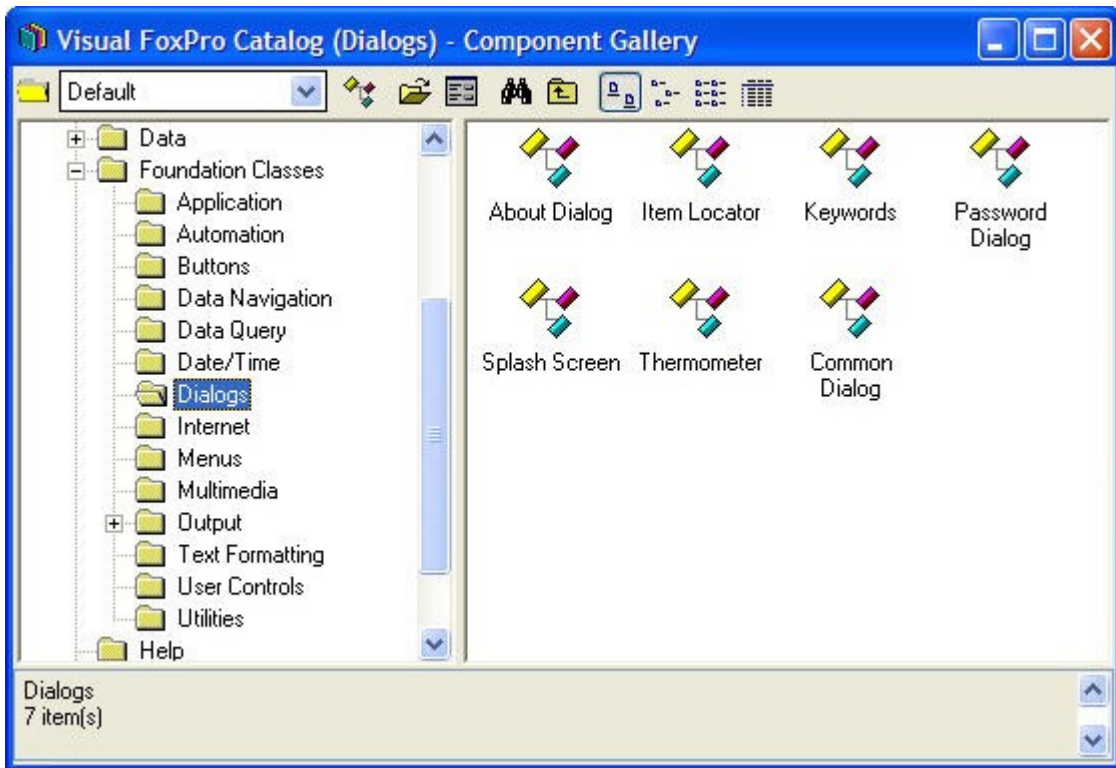


Figure 12. The Visual FoxPro Component Gallery is used to categorize files and information, such as the Fox Foundation Classes

4. Right-click on About Dialog and select Run from the Context menu. The About Custom Application dialog is displayed (Figure 13).

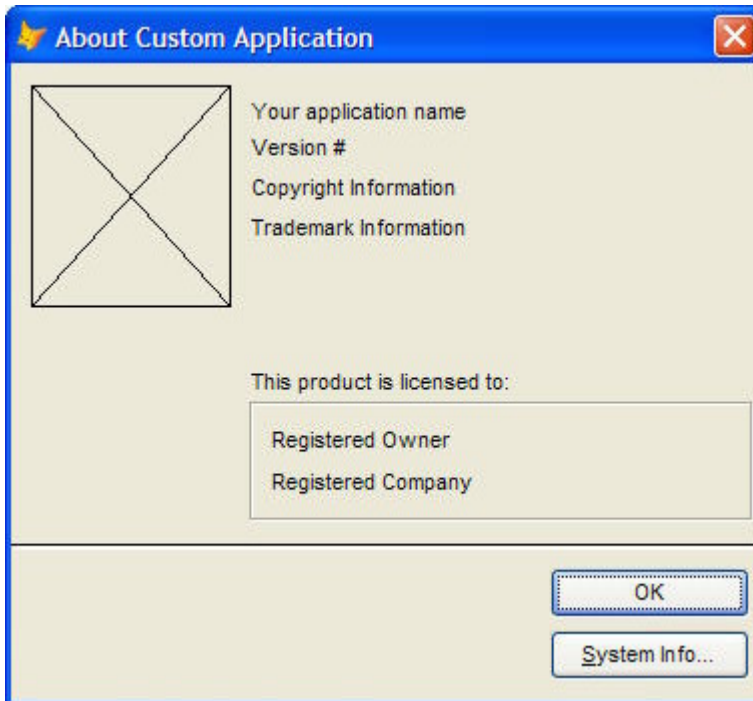


Figure 13. The About dialog is a class in the FFC that you can use in your own applications.

5. Click OK to close the dialog.
6. Right-click on About Dialog and select Modify from the Context menu. The About dialog class is opened in the Class Designer. This is good for exploring the class to determine how it works, but I recommend that you don't make modifications here as they will be overwritten when Microsoft updates the Fox Foundation Library.
7. Right-click on About dialog and select Add to Project from the Context menu. A list of Visual Class Libraries in the current project is displayed. Select a VCX and the About dialog will be added. This new class is a subclass of the About dialog class. You can make changes here to customize the About dialog subclass any way you want.

The Component Gallery has much more functionality. You can use it to categorize any file that you want. It will even watch folders where you are adding or deleting files. I encourage you to explore the Fox Foundation Classes and Component Gallery as these tools can make you more productive and improve your applications.

Additional Resources

Here are some additional resources from Microsoft and third parties:

- Creating Add-Ins for the Visual FoxPro Class Browser
(http://msdn.microsoft.com/archive/en-us/dnarfoxgen/html/msdn_addins.asp)
- Developing Reusable Objects
(http://msdn.microsoft.com/archive/en-us/dnarfoxgen/html/msdn_griver2.asp)
- Explanation and Examples of Non-Visual Classes
(http://msdn.microsoft.com/archive/en-us/dnarfoxgen/html/msdn_nonvis.asp)
- Managing Classes with Visual FoxPro
(http://msdn.microsoft.com/archive/en-us/dnarfoxgen/html/msdn_Manclass.asp)
- The Visual FoxPro Object Model
(http://msdn.microsoft.com/archive/en-us/dnarfoxgen/html/msdn_vfp_objm.asp)
- The Visual FoxPro 6.0 Class Browser
(http://msdn.microsoft.com/library/en-us/dnfoxgen/html/sb_browser.asp)
- The Microsoft Visual FoxPro 6.0 Component Gallery
(<http://msdn.microsoft.com/library/en-us/dnfoxgen/html/vfpgallery.asp>)
- Inside the Visual FoxPro 8.0 Toolbox
(http://msdn.microsoft.com/library/en-us/dnfoxgen8/html/vfp8_toolbox.asp)
- *Advanced Object Oriented Programming with Visual FoxPro 6.0* by Markus Egger, ISBN 0-96550-938-9, Hentzenwerke Publishing
(<http://www.hentzenwerke.com/catalog/aoopvfp.htm>)
Free sample chapter downloads
(<http://www.hentzenwerke.com/samplechapters/zsamplechapters.htm>)
- *Effective Techniques for Application Development with Visual FoxPro 6.0* by Jim Booth and Stephen A Sawyer, ISBN, 0-96550-937-0, Hentzenwerke Publishing
(<http://www.hentzenwerke.com/catalog/efftech.htm>)

Summary

Object Oriented Programming can greatly increase your productivity and the quality of the software you produce. However, a basic understanding of the terminology and the OOP tools provided by Visual FoxPro, such as the Class Designer, Class Browser, and Fox Foundation Classes, is required. While being a great tool, OOP can also be your worst enemy if it isn't used properly. You should now be on your way to understanding OOP and successfully using it in your application development.

Craig Berntson is a Microsoft Most Valuable Professional (MVP) for Visual FoxPro, a Microsoft Certified Solution Developer, and President of the Salt Lake City Fox User Group. He is the author of "CrysDev: A Developer's Guide to Integrating Crystal Reports", ISBN 1-930919-38-7, available from Hentzenwerke Publishing. He has also written for FoxTalk and the Visual FoxPro User Group (VFUG) newsletter. He has spoken at Advisor DevCon, Essential Fox, the Great Lakes Great Database Workshop, Southwest Fox, Microsoft DevDays and user groups around the country. Currently, Craig is a Senior Software Engineer at 3M Health Information Systems in Salt Lake City. You can reach him at craig@craigberntson.com, read his blog, FoxBlog, at www.craigberntson.com/blog/blogger.asp or visit his website, www.craigberntson.com.

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.