Lab 7: Creating ActiveX Code Components

For background information on this lab, click each of these topics:

Objectives

After completing this lab, you will be able to:

- Create an ActiveX code component.
- Declare methods and properties within a code component.
- Test the component from a second instance of Visual Basic.
- Compile and register the component.
- Create an object model.
- Build and test an ActiveX code component.

Prerequisites

Before working on this lab, you should be familiar with the following concepts:

- Automation.
- The contents of this chapter.

Lab Setup

To complete this lab, you need the following setup:

- Visual Basic version 5.0 or later

To see a demonstration of the completed lab solution, click this icon.

Estimated time to complete this lab: **90 minutes**

Note There are project and solution files associated with each lab. If you installed the labs during Setup, these files are in the folder <*Install Folder*>\Labs on your hard disk. If you did not install the labs during Setup, you can find them in the \Labs folder of the *Mastering Microsoft Visual Basic 5* CD-ROM.

Exercises

The following exercises provide practice working with the concepts and techniques covered in Chapter 7.

Exercise 1: Creating a Code Component

In this exercise, you will create a code component that exposes one object.

Exercise 2: Debugging and Error Handling

In this exercise, you will raise a custom error from the **CreditCard** object, and trap that error in your client application.

Exercise 3: Adding Component Information and Help

In this exercise, you will define procedure attributes that provide descriptive information and context-sensitive Help for the properties and methods of the **CreditCard** object.

Exercise 4: Defining and Using Events

In this exercise, you will use events to provide status information to the client application while the **Approve** method is processing.

Exercise 5: Compiling and Registering the Component

In this exercise, you will compile the code component and client application, and test the compiled versions.

Exercise 6: (Optional) Creating an Asynchronous Method

In this exercise, you will create a method that executes asynchronously.

Exercise 1: Creating a Code Component

In this exercise, you will create a code component that exposes one object. The **CreditCard** object will be used to simulate the validation of a credit card purchase.

This table shows the interfaces you will implement.

Name	Type
CardNumber	Property
ExpireDate	Property
PurchaseAmount	Property
Approve	Method

▶ **Create a code component**

1. Create a new project.
2. When prompted for a project type, click **ActiveX DLL**, and then click **Open**.

   The class module and project properties are set automatically when you choose an ActiveX DLL project.
3. Set the properties for the project, as shown in the following table.

Property	Setting
Project Name	Lab
Project Description	CreditCard Object
Unattended Execution	Selected

4. Set the **Name** property of the class module to CreditCard.
5. Save the class module as Cc.cls, and the project as Cc.vbp in the folder <*Install Folder*>\Labs\Lab07.

▶ **Create properties and methods**

1. In the class module, use public variables to create two properties with the information in the following table.

Property name	Data type
CardNumber	Integer
ExpireDate	Date

   For information about creating properties and methods, see Creating Properties.
2. Using property procedures, create a property named **PurchaseAmount**.
   a. Create a private module-level variable to store the value of the purchase amount.
   b. In the **Property Get** procedure, return the value of the private variable.
   c. In the **Property Let** procedure, test to see whether the return value of the private variable is greater than zero. If it is greater than zero, set the private variable to the purchase amount. If it is less than zero, set the private variable to zero.

For more information about property procedures, see Using Property Procedures to Create Properties.

3. Using a **Public** function, create a method named **Approve** that returns a **Boolean** value.

   If the credit card is approved, the method should return a value of **True**, and if it is declined, a value of **False**.

   For this exercise, use the following logic to validate the credit card.

If...	Then...
PurchaseAmount < 1000 And ExpireDate > Now()	Approve = **True**
PurchaseAmount >= 1000 And ExpireDate <= Now()	Approve = **False**

4. Make **CardNumber** the default property for the CreditCard class.
5. Save the project.
6. Compile the project as Cc.dll.

   This .dll file will enable **Project Compatibility** automatically in the **Project Properties** dialog box. This will ensure that as you make changes to the project, the GUID will remain the same, and references from your client application will not be lost.

   For information about version control, see Version Compatibility.

▶ Create a client application

1. To add a new project to the CreditCard project, click **Add Project** on the **File** menu.
2. When prompted for a project type, click Standard EXE, and then click **Open**.

   This project will be used as the client that tests the component.
3. In the References dialog box, add a reference to the CreditCard project.
4. Dimension a module-level variable named cc as type Lab.CreditCard.
5. Add a **CommandButton** control to the form.
6. In the Click event procedure for the **CommandButton** control, add the following code:

```
set cc = New Lab.CreditCard
cc.ExpireDate = "1/1/99"
cc.PurchaseAmount = 50
cc.CardNumber = 1234
MsgBox cc.Approve
```

7. In the **Project Group** window, right-click Project1, and then click **Set as Start Up**.

   This command sets Form1 as the startup form for the project.
8. Press F8 to run the client application in Step mode, and step through the code.

   When you set the **PurchaseAmount** property and call the **Approve** method, you are actually stepping into the CreditCard project.
9. Try changing the value of **PurchaseAmount** to 2000. Does the **Approve** method return **False**?

   To see the code for the CreditCard class module if your code is not working properly, click this icon

```
Option Explicit
Public CardNumber As Integer
Public ExpireDate As Date
Private msngAmount As Single
Public Property Get PurchaseAmount() As Single
    PurchaseAmount = msngAmount
```

```
End Property
Public Property Let PurchaseAmount(ByVal sngAmount As Single)
    If sngAmount > 0 Then
        msngAmount = sngAmount 'save in private module variable
    Else 'purchase less than zero is invalid
        msngAmount = 0
    End If
End Property
Public Function Approve() As Boolean
    'dummy logic for approving credit card
    If msngAmount < 1000 And ExpireDate > Now() Then
        Approve = True
    Else
        Approve = False
    End If
End Function
```

Exercise 2: Debugging and Error Handling

In this exercise, you will raise a custom error from the **CreditCard** object and trap that error in your client application.

▶ **Add error-handling code**

1. Open the CreditCard class module.
2. Select the **PurchaseAmount Property Let** procedure.
3. Rewrite the **Property Let** procedure, so that if the value of the amount the user has entered is negative, a run-time error will be raised with the following arguments.

Argument	Value
Number	vbObjectError + 1000
Source	Lab.CreditCard
Description	Purchase amount must be greater than zero.

4. In the **General** tab of the **Options** dialog box, set the **Error Trapping** option to **Break in Class Module**.
5. Change the client code to pass a purchase amount of –100.
6. Run the client application.

   Note that the run-time error occurs in the class module.
7. In the **Options** dialog box, set the **Error Trapping** option to **Break on Unhandled Errors**.
8. Run the client application.

   Note that this time, the run-time error occurs in the client application.
9. Save the CreditCard project.

▶ **Error-handling in the client**

1. In the client application, add an error handler to the Click event for the **CommandButton** procedure.
2. In the error handler, test for the error generated in the class module.
3. If the error is found, display a message box telling the user that the purchase amount must be a positive value.

   For more information about error handling, see Raising Run-Time Errors.

   To see the error-handling code to use if your code is not working properly, click this icon.

```
Private Sub Command1_Click()
    On Error GoTo HandleError
    Set cc = New Lab.CreditCard
    cc.ExpireDate = "1/1/99"
    cc.PurchaseAmount = 50
    cc.CardNumber = 1234
    MsgBox "Approval = " & cc.Approve
    Exit Sub

HandleError:
    Select Case Err.Number
    Case vbObjectError + 1000
        MsgBox "object error"
    Case Else
        MsgBox "Unknown error: " & Err.Number & " " & Err.Description
    End Select
End Sub
```

Exercise 3: Adding Component Information and Help

In this exercise, you will define procedure attributes that provide descriptive information and context-sensitive Help for the properties and methods of the **CreditCard** object.

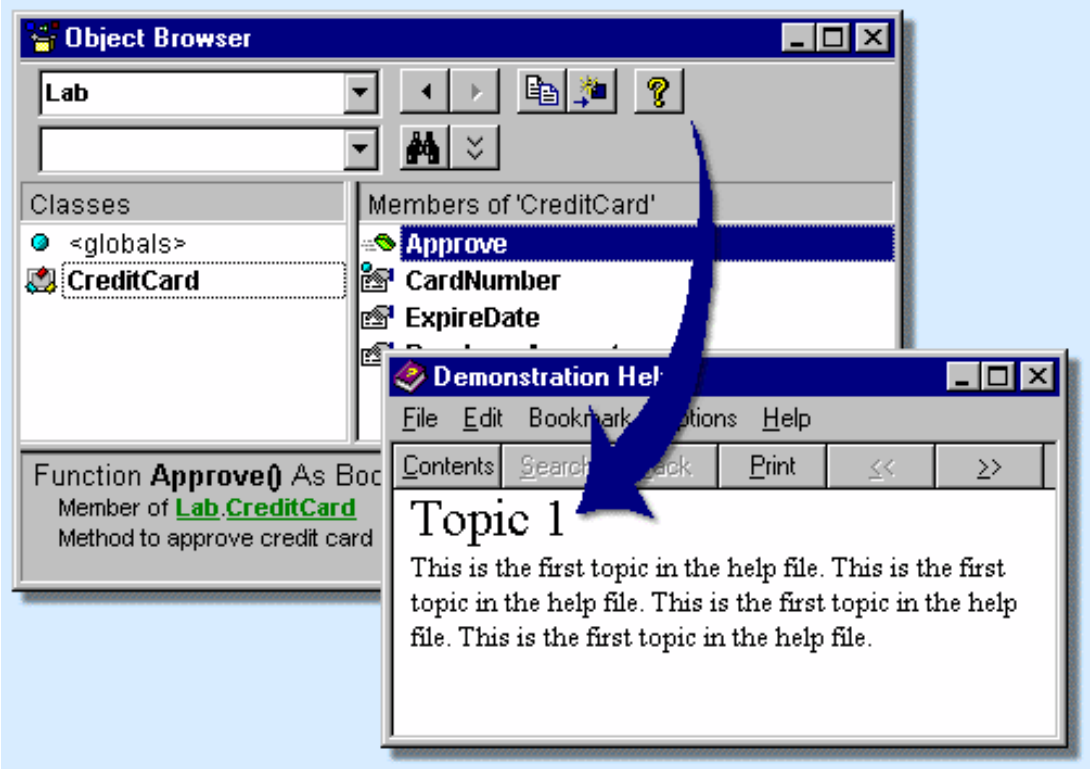
▶ **Set procedure attributes**

1. In the **Project Properties** dialog box, set **Help File Name** to Lab.hlp.

   The file Lab.hlp is located in the folder <*Install Folder*>\Labs\Lab07. It is a Help file with three sample topics.
2. Open the CreditCard class module.
3. In the **Procedure Attributes** dialog box, add a description for the **Approve** method and **PurchaseAmount** properties.
4. In the **Procedure Attributes** dialog box, set **Help Context ID** for the **Approve** method to 1 and the **Help Context ID** for the **PurchaseAmount** property to 2.
5. Save the project.

▶ **Test procedure attributes**

1. Open the client project.
2. Display the **Object Browser** dialog box, and select the **Lab Library**.
3. Check to see that the descriptions for the **Approve** method and **PurchaseAmount** property are displayed correctly.
4. Test the Help information for the **Approve** method and **PurchaseAmount** property as shown in the following illustration.



Exercise 4: Defining and Using Events

In this exercise, you will use events to provide status information to the client application while the **Approve** method is processing.

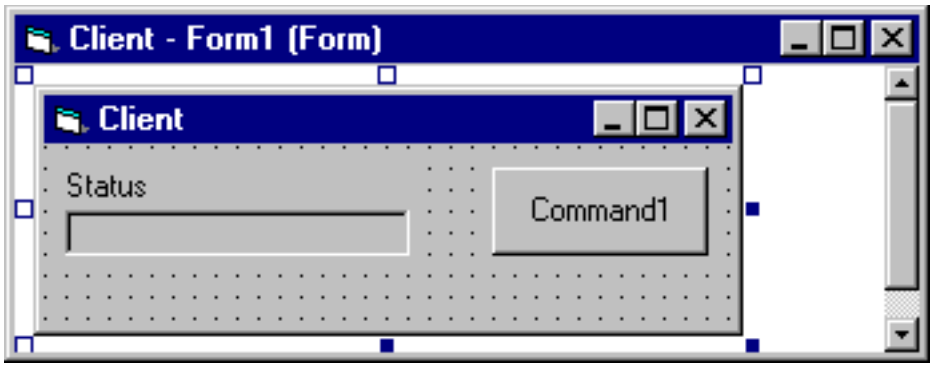
▶ Define and use the Status event

1. In the CreditCard class module, add the following code to the beginning of the **Approve** procedure:

```
Dim sngEndTime As Single
RaiseEvent Status("Dialing bank...")
'Simulate delay dialing bank.
sngEndTime = Timer + 2
Do While Timer < sngEndTime
    DoEvents    ' Yield to other processes.
Loop
RaiseEvent Status("Processing card...")
'Simulate delay processing card.
sngEndTime = Timer + 2
Do While Timer < sngEndTime
    DoEvents    ' Yield to other processes.
Loop
```

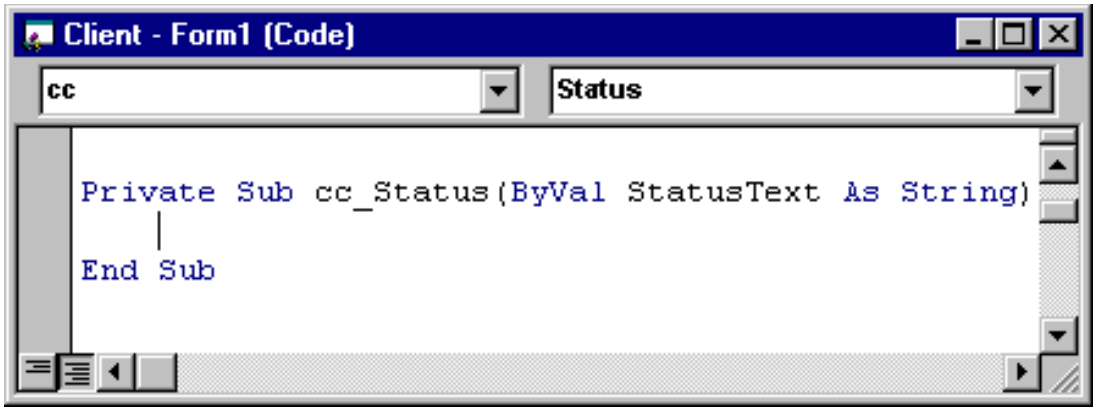
   This code adds two delays of two seconds each to simulate the time necessary to dial the bank and process the credit card information. Before each delay, an event is generated to inform the client application of the current status.

2. In the CreditCard class module, declare the Status event with one string argument.
3. In the client application, add a label to hold status information, as shown in the following illustration.



4. In the client application, add the **WithEvents** keyword to the cc variable declaration.

   This adds a new object with one new procedure named Status to the code window, as shown in the following illustration.



5. In the Status event, display the status text in the **Label** control.
6. Save both projects, and test the application.

   Each of the status messages should be displayed for two seconds in the label.

   To see the code, click this icon.

Client Event Code

```
Dim WithEvents cc As Lab.CreditCard
Private Sub cc_Status(ByVal StatusText As String)
    Label2.Caption = StatusText
End Sub
```

CreditCard Class Event Code

```
Public Event Status(ByVal StatusText As String)
Public Function Approve() As Boolean
    Dim sngEndTime As Single

    RaiseEvent Status("Dialing bank...")
    'simulate delay dialing bank
    sngEndTime = Timer + 2
    Do While Timer < sngEndTime
        DoEvents    ' Yield to other processes.
    Loop
```

```
    RaiseEvent Status("Processing card...")
    'simulate delay processing card
    sngEndTime = Timer + 2
    Do While Timer < sngEndTime
        DoEvents    ' Yield to other processes.
    Loop

    'dummy logic for approving credit card
    If msngAmount < 1000 And ExpireDate > Now() Then
        Approve = True
    Else
        Approve = False
    End If
End Function
```

Exercise 5: Compiling and Registering the Component

In this exercise, you will compile the code component and client application, and then test the compiled versions.

▶ **Compile the component and test the application**

1. Compile the credit card project as a .dll file.
2. Compile the client project as an .exe file.
3. Close Visual Basic.
4. In the folder <*Install Folder*>/Labs/Lab07, run the file Client.exe.
5. Test the client application.

Exercise 6: (Optional) Creating an Asynchronous Method

In this exercise, you will create a method that executes asynchronously.

▶ **Create the initial project**

1. Create a new ActiveX EXE project.
2. Set the project name to AsyncSrv.
3. Set the class module name to Async.
4. Add a form to the project and name it frmTimer.

▶ **Implement a Timer control**

1. Add a **Timer** control to the frmTimer form.
2. At the form level, add a public variable called **CallBack** of type **Async**.
3. Add code to the Timer event procedure in the **Timer** control to disable the **Timer** control and call the **DoWork** method in the **Callback** object. For example:

```
Private Sub Timer1_Timer()
    Timer1.Enabled = False
    'Start the real processing
    CallBack.DoWork
End Sub
```

▶ **Implement the Async class**

1. Add a class level variable called **CurrForm** of type **Form**.
2. Add a method called **AsyncMethod** that does the following:
   a. Initializes the CurrForm variable with a new instance of the frmTimer form.
   b. Sets the Callback variable of the form to point to the **Async** class.

c. Enables the timer on the form.

```
Public Sub AsyncMethod()
'Create instance of form so it is unique to this class.
    Set CurrForm = New Form1
'Pass current class to form instance.
    Set CurrForm.CallBack = Me
'Enable timer to start async processing
'and return immediate control to client
    CurrForm.Timer1.Enabled = True
End Sub
```

3. Add a **Friend** method called **DoWork**.
4. Add a **Single** variable to **DoWork** called **sgnEndTime**.
5. Destroy the instance of CurrForm.
6. Add the following code to simulate a long, asynchronous task:

```
sngEndTime = Timer + 5
    Do While Timer < sngEndTime
        DoEvents    ' Yield to other processes.
    Loop
```

7. Add an event called Complete that passes the string "Hello, World" back to the client once the task is complete.

   The following example shows the completed code for the DoWork procedure:

```
Public Event Complete(Result As String)
Friend Sub DoWork()
    Dim sngEndTime As Single
    'Get rid of form instance.
    Set CurrForm = Nothing

    'Delay to simulate a long procedure.
    sngEndTime = Timer + 5
    Do While Timer < sngEndTime
        DoEvents    ' Yield to other processes.
    Loop

    'Raise event to client to signify end of processing.
    RaiseEvent Complete("Hello, world")
End Sub
```

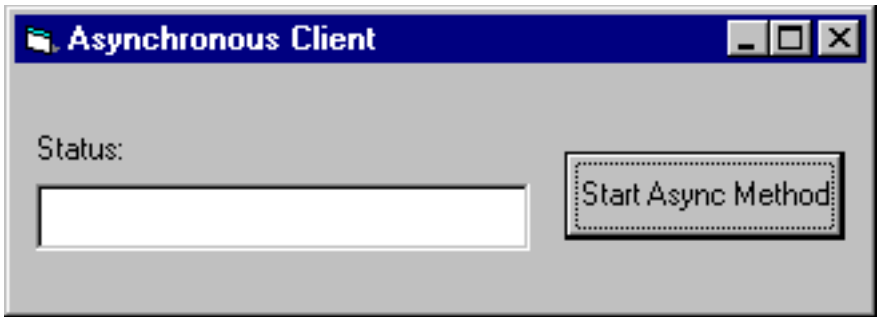
▶ Build the project

1. Save all of the project files as Asyncsrv in the folder <*Install Folder*>\Labs\Lab07.
2. Compile the project as an ActiveX EXE project.

   For more information about asynchronous methods, see Making Asynchronous Calls with Events.

▶ Create an asynchronous client

1. Start a second instance of Visual Basic.
2. In the second instance, create a new Standard EXE project.
3. In the new project, add controls to the form as shown in the following illustration.



4. Set the **Enabled** property of the text box to **False**.
5. Create a reference to the AsyncSrv project.
6. Add a form level variable called **mAsync** to hold the **Async** object created above.

   Be sure the client can receive events from the object.
7. In the Load event for the form, initialize the **mAsync** variable with a new instance of the **Async** object.
8. In the Click event of the **Start Async Method** button, set the text in the text box to indicate that the component is processing, and then call the **AsyncMethod** method of the component.
9. In the Complete event of the **Async** object, update the text in the text box to indicate that the asynchronous task is complete and display the resulting text. The following example shows the completed code for the client:

```
Dim WithEvents mAsync As AsyncSrv.Async
Private Sub cmdStartAsyncMethod_Click()
    Text1 = "Processing..."
    mAsync.AsyncMethod
End Sub
Private Sub mAsync_Complete(Result As String)
    Text1 = "Complete. Result = " & Result
End Sub
Private Sub Form_Load()
    Set mAsync = New AsyncSrv.Async
End Sub
```

10. Test the asynchronous method.