Inheriting Windows Forms

Now we're ready for the really fun stuff. The ability to have multiple levels of inheritance for our training form will literally revolutionize training development. Many training creators today use templates to try to speed up the process and enforce commonality. But in almost all cases, changing the template has no effect on the screens that are already completed. Inheritance is much different, as you will see.

Visual Inheritance: Tailor-made for Training Applications

We use the term *Visual Inheritance* to describe what is possible with Windows Forms, namely that the user interface and the methods, properties, and events can be inherited. With Web Forms, the code part (methods, properties, and events) can be inherited but not the user interface. On both the Windows and Web side, you can use visual inheritance with controls, but we'll wait for that topic until a later chapter.

The reason this capability is so exciting is that you can create both your user interface and functionality in layers. When you need a new type of training form, you just inherit from the most appropriate layer and go from there. You can then change any layer and any *children* will all be updated the minute you rebuild.

The Mechanics

To use the IDE to inherit from an existing form, go to the Project menu → Add Inherited Form as shown in Figure 54. You then give the form a name as shown in Figure 55. To avoid having to change the properties of the form later, it is a good idea to make it something understandable (*QuestionForm* in our example) rather than the default *Form2*. The next step is to use the *Inheritance Picker* to choose from which form to inherit. This can be a form in the current project or a form contained in an assembly (use the browse button). In the latter case, the browse dialog defaults to .dll files but you can select .exe files as well. Figure 56 shows this situation with forms listed in the current project (*LMTraining* namespace) and another assembly (*DatabaseSample1* namespace). We then select the form from which we want to inherit. We choose *NavigationForm* since we are implementing the hierarchy shown in Figure 1.

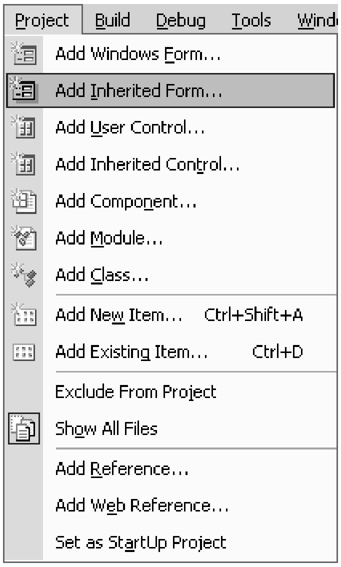
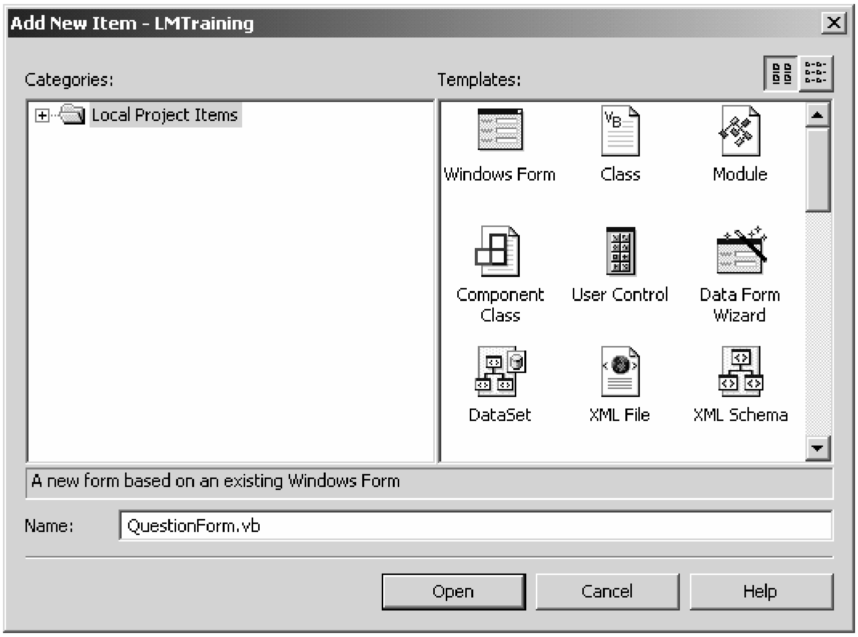


Figure 54. Inheriting a Form

The IDE then adds *QuestionForm* to the Solution Explorer and displays its Design view in the main window. It looks exactly like *NavigationForm* (Figure 2) of course since that is what we inherited. If we switch to the Code view, it looks like this:

```
Public Class QuestionForm
  Inherits LMTraining.NavigationForm

' Windows Form Designer generated code here

End Class
```

This tells us that *another* way we could have done all that is just added a normal form and go to the line directly below the class declaration and type in the *Inherits* part. Note that IntelliSense will help us with both the namespace and the form name.



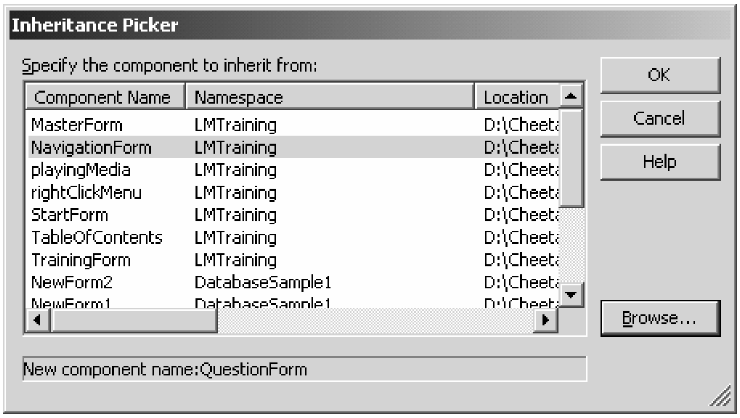
A new form based on an existing Windows Form

Name: QuestionForm.vb

Figure 55. Naming an Inherited Form

Inheritance Example 1

Our first example is the *DatabaseSample1* application (part of the *Database Connectivity* solution of the downloadable sample applications). We'll cover the database aspects of this later in the book, so for now let's concentrate on its use of inheritance. This application has a login form and then a series of training forms. These training forms can be grouped into two categories. The first set of forms has a unique form for each record in the database. This is to demonstrate the situation where you want to move content around the screen or perhaps even have the content displayed in different types of objects. The second set of forms demonstrates "binding" to a series of records. This is for the situation



Specify the component to inherit from:

Component Name	Namespace	Location
MasterForm	LMTraining	D:\Cheet
NavigationForm	LMTraining	D:\Cheet
playingMedia	LMTraining	D:\Cheet
rightClickMenu	LMTraining	D:\Cheet
StartForm	LMTraining	D:\Cheet
TableOfContents	LMTraining	D:\Cheet
TrainingForm	LMTraining	D:\Cheet
NewForm2	DatabaseSample1	D:\Cheet
NewForm1	DatabaseSample1	D:\Cheet

New component name:QuestionForm

Figure 56. Inheritance Picker

where a series of forms has the same "look and feel." Whenever we need a form with this design, we just use the same one over and over. To allow for multiple forms in each category, we have two "master"[1] forms from which we inherit.

MasterNewForm

In the *MasterNewForm* shown in Figure 57, we define the elements of the user interface we want to be common to all the *children* forms. These include:

- *BackColor* property ("Wheat")
- *FormBorderStyle* property ("Fixed3D[2]")

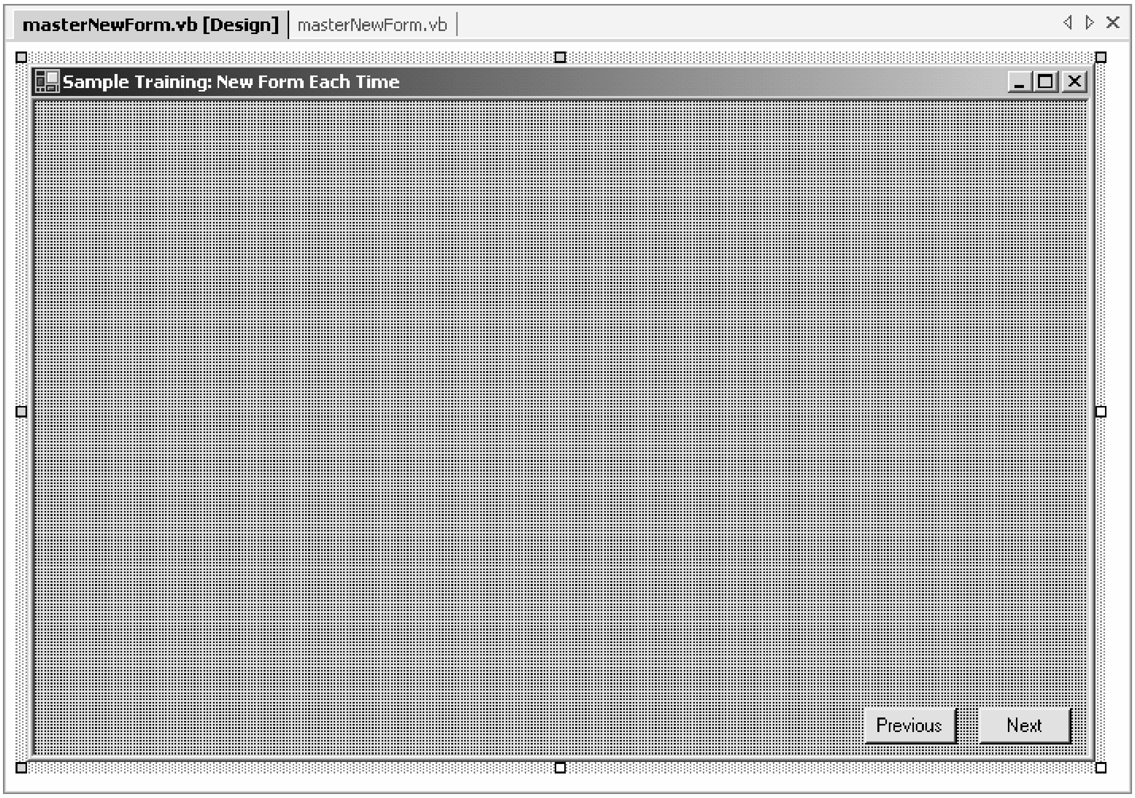

masterNewForm.vb [Design]
masterNewForm.vb
Sample Training: New Form Each Time
Previous
Next

Figure 57. Master New Form

[1] Or, in object-oriented programming terminology, "base."

[2] This is an important setting, particularly if you use a graphic as a *BackgroundImage*. The reason is that .NET automatically *tiles* your image when you make the size of your form bigger than the image size. This is not a good thing if you use a "sculpted" image created in PhotoShop like that of the "Sample Windows Training" application. A Fixed3D style along with the *MaximizeBox* set to False keeps the user from making your form bigger (and causing this to happen).

- *Text* (caption) property ("Sample Training: New Form Each Time")
- *Size* (width and height) property (672, 440)
- *StartPosition* property ("CenterScreen")
- *MaximizeBox* property (False)
- *nextButton* button control
- *previousButton* button control

On the code side, we implement as much of the functionality here as possible so that we don't have duplicate code in forms that inherit from this one. Listing 26 shows the code with the exception of *Implementation Methods*, which we leave for later in the book.

```
Option Explicit On
Option Strict On
Imports System.Data.OleDb
Imports System.ComponentModel
Imports System.Reflection

Public Class masterNewForm
  Inherits System.Windows.Forms.Form

  ' Windows Form Designer generated code here

  ' Variables
  Private c_nextFormName As String
  Private c_previousFormName As String

  ' Base Events
  Private Sub masterNewForm_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    Call initializeNewForm()
  End Sub

  Private Sub masterNewForm_Closing(ByVal sender As Object, ByVal e As _
    System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
    If g_allowClose = True Then
      Call closeApplication()
    End If
  End Sub

  Private Sub nextButton_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles nextButton.Click
    Dim nextFormName As String = Me.NextForm

    If nextFormName <> "" Then
      Call ReferenceAndShowNextForm(nextFormName)
    End If
  End Sub

  Private Sub previousButton_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles previousButton.Click
    Dim previousFormName As String = Me.PreviousForm

    If previousFormName <> "" Then
```

```
      Call ReferenceAndShowNextForm(previousFormName)
    End If
  End Sub

  ' Implementation Methods
  Private Sub initializeNewForm()
  ' Code shown later
  End Sub

Private Sub ReferenceAndShowNextForm(ByVal nextFormName As String)
' Code shown later
  End Sub

  ' Properties
  <Description("The name of 'next' form to open."), Category("Extended")> _
    Public Property NextForm() As String
    Get
      Return c_nextFormName
    End Get
    Set(ByVal Value As String)
      c_nextFormName = Value
    End Set
  End Property

  <Description("The name of 'previous' form to open"), Category("Extended")> _
    Public Property PreviousForm() As String
    Get
      Return c_previousFormName
    End Get
    Set(ByVal Value As String)
      c_previousFormName = Value
    End Set
  End Property
End Class
```

Listing 26. MasterNewForm Code

We start with our familiar Option and Imports lines at the top of the class. The private variables are used to implement the *NextForm* and *PreviousForm* properties. The *nextButton_Click* and *previousButton_Click* subroutines use these properties to go forwards and backwards. We cover this in detail in our next inheritance example as well as in the *Hyperlinks and Navigation* chapter. The key point here is that we implement the properties, the objects (buttons in this case), event handlers (the two Click subroutines), and the *implementation* methods all as part of this class. They are then inherited by all the *children* of this class. The same goes for the *Load* event and its corresponding *initializeNewForm* subroutine.

The other interesting thing about the code in Listing 26 is how we deal with closing the form. We want to call *closeApplication* when the user closes the application but not every time a form closes, so we add logic to distinguish between these two events. The *module* for this project is shown in Listing 20. In it, we started the application, showed a "startup form," and declared the public *g_allowClose* variable. When we are navigating to another form, we set *g_allowClose* to False. The rest of the time we ensure it is True. We then handle the *Closing* event and close the application (implementation in Listing 20) if *g_allowClose* is True. This closes the application when the user clicks the "x" at the top of the window but leaves it open when she navigates to the next page. Without this logic, we could get a situation where the application is still running but there are no visible forms, or a situation where we shut down the entire application when the user is trying to just navigate to another form[3].

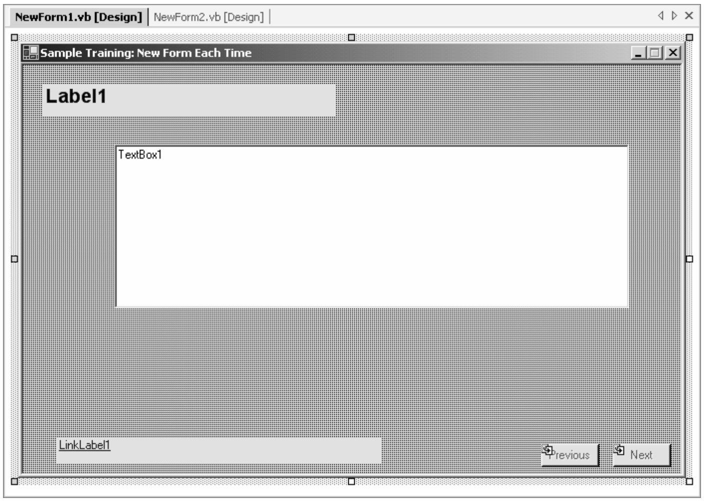
We can *inherit* from this master *NewForm* as shown in Figure 58. The code simply has the inheritance line plus any functionality specific to the inheriting form.

```
Public Class NewForm1
  Inherits _
  DatabaseSample1.masterNewForm
```

Notice that the inherited user interface elements (the buttons to the bottom right) have an arrow on their upper left to show that they are inherited. The control marked "Label1" is the same on both forms (and could thus be placed on the master if we decided to keep it for all forms), but the other (non-inherited) objects are of different types, locations, and sizes.

MasterBindForm

In our other master form, we set the same list of properties as we did on the first form. So why didn't we inherit both from a third form? The answer is that only the *FormBorderStyle*, *Size*, *StartPosition*, and *MaximizeBox* properties have identical values. And most of the implementation code differs. The only common code needed between the two master forms is also needed by the login form, which looks completely different. So we put that code in the module instead. We certainly *could* have had another inheritance layer, however; we do this in the next example. Both the MasterBindForm and its single [4] inheriting form (class) are shown in



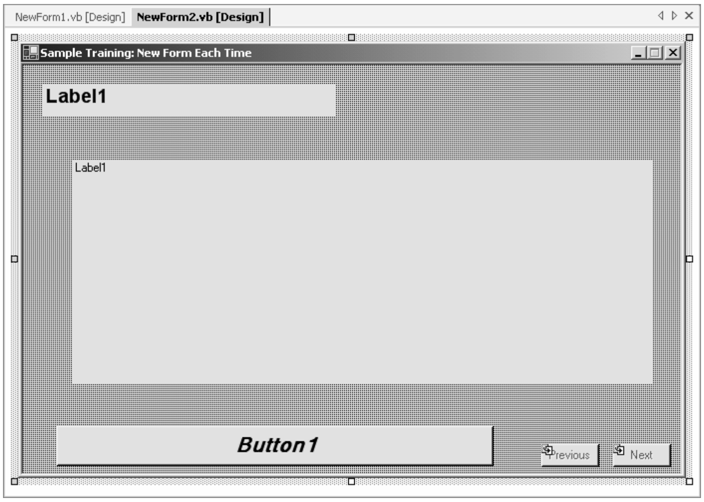


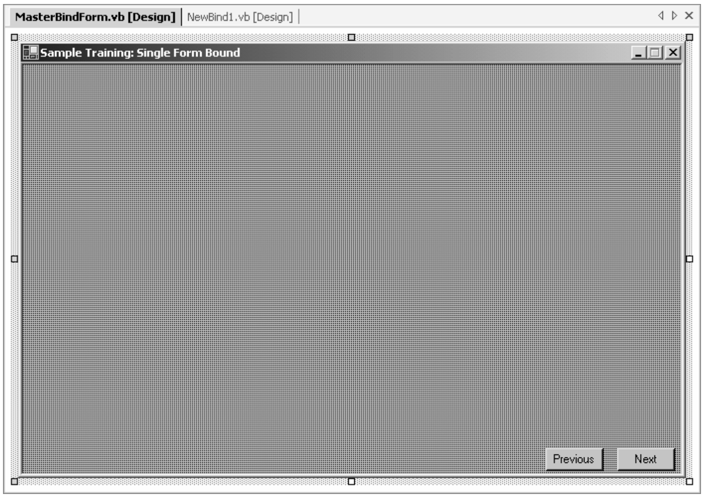
Figure 58. MasterNewForm Class

[3] Remember that we must explicitly call *Application.Exit* because we started it via *Application.Run* in the Main subroutine.

[4] I only did a single binding form in this example project, but it would be easy to generate other looks and feels from the *MasterBindForm*.

Figure 59. Notice how we have a different background color and text for the *MasterBindForm* than we had for the *MasterNewForm*. The *nextButton* and *previousButton* objects, while in the same position and so forth, have totally different implementations. The form's code is similar to that in Listing 26, so we won't show it here. The key differences are:

1. There are no *NextForm* or *PreviousForm* properties. Instead, this information is stored in the database.
2. The "Implementation Methods" vary between the two forms.


MasterBindForm.vb [Design]
NewBind1.vb [Design]
Sample Training: Single Form Bound
Previous
Next

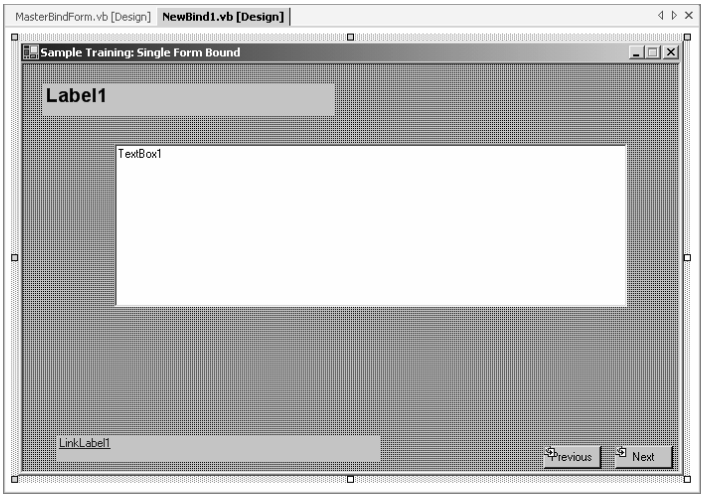

MasterBindForm.vb [Design]
NewBind1.vb [Design]
Sample Training: Single Form Bound
Label1
TextBox1
LinkLabel1
Previous
Next

Figure 59. MasterBindForm and Its Instance

Inheritance Example 2

In our second example ("Sample Windows Training"), we implement the user interface and core capabilities that are part of our commercial *Learning & Mastering ToolBook...* series (Figure 51 show an example screen from this training). How could inheritance be applied to this situation? The first step was to outline the various potential classes to figure out how many inheritance levels we would need. The criterion for a level was when the interface diverged <u>and</u> multiple forms would use this diverged interface. Figure 1 shows the basic design. I'll go through each level in turn, explaining why I chose to make it a level and the functionality included in that level. I encourage you to download and start up the project so you can follow along. Figure 22 shows an enlightening *Class View* of this sample.

CommonCode Module

The module in this example has one key function. It provides a Sub *Main* that the application calls on startup[5]. We also declare and initialize a global[6] variable that defines the path from which we will read our graphics. As we have seen previously, we then create an instance of our *StartForm* class and show it. We *run* the application so that it will stay open until we manually shut it down. See Listing 27 for a complete listing of this code.

```
Option Explicit On
Option Strict On

Module CommonCode
  Public graphicsPath As String

  Sub Main()
    graphicsPath = Application.StartupPath & "\graphics\"
    Dim startUpForm As New LMTraining.StartForm()
    startUpForm.Show()
    Application.Run() ' must close manually
  End Sub
End Module
```

Listing 27. Sample Windows Training Module Code

Master Form

As its name implies, all other training[7] forms inherit from *Master Form* (Figure 60). We define these “visual” properties to be different from the default:

- *FormBorderStyle* (“Fixed3D”)
- *Text* (“VBTrain.Net: Using VB.NET to Make Training”)
- *Size* (806, 598)
- *StartPosition* (“CenterScreen”)

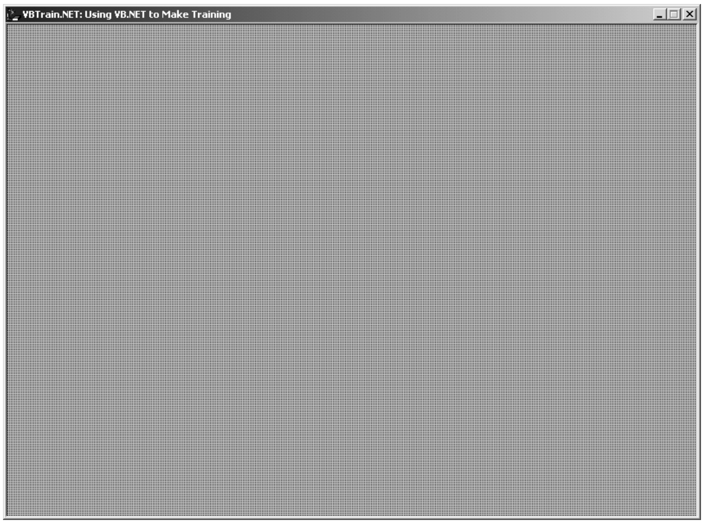


Figure 60. Master Form

[5] Since we set *Sub Main* as the project’s startup object. Notice that this allows us to easily implement a “return to where you left off” bookmarking function simply by reading a database or other storage for the user’s last form. If we assigned a particular form as the startup object, that form would of course always be the first one shown.

[6] Public variables of a module are effectively global variables since any object in the project can access it.

[7] There may be other forms in a real project that don’t inherit from this form. These might include forms to logon, change your settings, search for text or keywords, run simulations, etc.

- *KeyPreview*[8] (True)
- *Icon* (set to the icon file that we want to use)
- *MaximizeBox* (False)

Note that we don't define the *BackColor* property since all forms have a background image. We don't define the BackgroundImage property either, however, since that is set further down the hierarchy. There are also no controls or other user interface elements at this level. The code for *Master Form* contains common functionality as show below. Note that Figure 29 shows how this code can be collapsed into various "regions."

```
Option Explicit On
Option Strict On
Imports System.ComponentModel
Imports System.Reflection

Public Class MasterForm
  Inherits System.Windows.Forms.Form

' Windows Form Designer generated code here

#Region "Variables"
  Private c_stopCloseMessage As Boolean
  Private c_nextFormName As String
  Private c_previousFormName As String
#End Region

#Region "Base Events"
  Private Sub MasterForm_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    c_stopCloseMessage = False
  End Sub

  Private Sub MasterForm_Closing(ByVal sender As Object, ByVal e As _
    System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
    If c_stopCloseMessage = False Then
      If queryExit() = True Then
        Application.Exit()
      Else
        e.Cancel = True
      End If
    End If
  End Sub

  Private Sub MasterForm_KeyDown(ByVal sender As Object, ByVal e As _
    System.Windows.Forms.KeyEventArgs) Handles MyBase.KeyDown
    Select Case e.KeyCode.ToString
      Case "Next"
        Call goNextPage(Me, Me.NextForm)
      Case "Prior"
        Call goPreviousPage(Me, Me.PreviousForm)
```

[8] Setting this to true allows the form to receive any *KeyPress*, *KeyDown*, and *KeyUp* events in addition to whatever control might have the focus. We use this so that the "Page Down" key goes forward, the "Page Up" key goes backwards, and "F1" brings up help.

```
      Case "F1"
        Call ShowHelp()
    End Select
  End Sub
#End Region
```

Listing 28. Master Form Class Definition, Variables, and Base Events Code

We inherit from *System.Windows.Forms.Form* in this case since this is the "highest level" form. All our other training forms will inherit from this form or its children. The variables are private as we only use them internally. The *c_stopCloseMessage*[9] variable is used to determine whether the user is closing the application or just navigating to the next page. The two "formName" variables store the value of the *NextForm* and *PreviousForm* properties (See Listing 30). The only thing we do upon loading the form is set our *c_stopCloseMessage* variable to False. We then handle the *Closing* event (generated both when the user clicks the "x" on the window and when navigating to another form) to see if *c_stopCloseMessage* is still False. If so, the user didn't click a navigation button (see the *ReferenceAndShowNextForm* method in Listing 29) and hence is trying to close the application. We call our *queryExit* function to ask if they are sure. If so, we exit the application. If not, we set the *Cancel* property of the *e* parameter (here of type *System.ComponentModel.CancelEventArgs*) to prevent the form from closing. The last "Base Event" that we handle is *KeyDown*. This event is sent each time the user presses a key on the keyboard[10]. If the user presses "Page Down," we call our *goNextPage* method. Similarly, we call *goPreviousPage* when she presses "Page Up." Finally, we call *ShowHelp* for F1. The implementation of each of these is shown below. As always, I used the drop-down list boxes shown in Figure 27 and Figure 28 to create the shells of these event handlers.

```
#Region "Implementation Methods"
  Protected Sub goNextPage(ByRef formObj As Form, Optional ByVal nextFormName As _
    String = "")
    If nextFormName <> "" Then
      Call ReferenceAndShowNextForm(formObj, nextFormName)
    End If
  End Sub

  Protected Sub goPreviousPage(ByRef formObj As Form, Optional ByVal prevFormName As _
    String = "")
    If prevFormName <> "" Then
      Call ReferenceAndShowNextForm(formObj, prevFormName)
    End If
  End Sub

  Private Sub ReferenceAndShowNextForm(ByRef formObj As Form, ByVal nextFormName As _
    String)
    Dim nextForm As Object
    Dim nextFormReference As String = String.Concat(Application.ProductName.ToString, _
      ".", nextFormName)

    If nextFormName <> "" Then
      Dim nextFormType As Type = Type.GetType(nextFormReference)
      nextForm = System.Activator.CreateInstance(nextFormType)
```

[9] The c_ is a naming convention that indicates the variable is class-level only.

[10] This is a good example of how the *e* parameter comes in useful. Here, *e* is of type *System.Windows.Forms.KeyEventArgs*. It has a *KeyCode* property that we can read to find out which key the user pressed.

```
      c_stopCloseMessage = True ' so not prompted if want to exit
      nextForm.GetType().InvokeMember("Show", BindingFlags.InvokeMethod, Nothing, _
        nextForm, New Object() {})
      formObj.Close()
    End If
  End Sub

  Private Sub ShowHelp()
    MessageBox.Show(String.Concat("Show online help for form '", Me.Name, "'."))
  End Sub

  Protected Function findObjectByName(ByRef formObj As Form, Optional ByVal objectName _
    As String = "help") As Object
    Dim obj As Control
    Dim returnObj As Control

    returnObj = Nothing
    For Each obj In formObj.Controls
      If obj.Name = objectName Then
        returnObj = obj
        Exit For
      End If
    Next
    Return returnObj
  End Function

  Protected Sub ExitTraining()
    If queryExit() = True Then
      Application.Exit()
    End If
  End Sub

  Protected Function queryExit() As Boolean
    Dim answer As System.Windows.Forms.DialogResult
    Dim dialogReturn As Boolean

    answer = MessageBox.Show("Are you sure that you want to exit?", "Exit?", _
      MessageBoxButtons.YesNo, MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)
    If answer = dialogResult.Yes Then
      dialogReturn = True
    Else
      dialogReturn = False
    End If
    Return dialogReturn
  End Function
#End Region
```

Listing 29. Master Form Implementation Methods

The first thing to note about the implementation methods is that a number of them are marked as *Protected* rather than *Private*. As explained in Table 6, this allows forms inheriting from this class to call the methods directly. For example, this allows us to have one button on the *StartForm* (next section) call the *goNextPage* method and another call the *ExitTraining* method. Let's look at each method in turn. We use the *goNextPage* and *goPreviousPage* methods, as the names imply, to move forward or backward in the training. Notice that the parameters, *nextFormName* or *prevFormName*, are optional so that we can call the methods without generating an error if there is not a next or previous form. Both methods then call a third method (which is *Private* since it doesn't need to be called directly from an inheriting class) called *ReferenceAndShowNextForm*. This uses our old friend, *Reflection*[11], to generate a reference to and then show a form based on simply its name. Since this is such an important capability when writing training, let's look at the key lines in detail:

```
Dim nextFormReference As String = String.Concat(Application.ProductName.ToString, ".", _
  nextFormName)
```

This line creates the complete namespace reference to the form, *LMTraing.StartForm* for example.

```
Dim nextFormType As Type = Type.GetType(nextFormReference)
```

This line creates the *Type* object that we need to use for Reflection. Notice that we pass in the complete namespace reference.

```
nextForm = System.Activator.CreateInstance(nextFormType)
```

This uses the *Activator* class to create an instance of the form. In non-technical terms, it finds the form in the assembly, verifies that it is a valid object, and references it.

```
nextForm.GetType().InvokeMember("Show", BindingFlags.InvokeMethod, Nothing, nextForm, _
  New Object() {})
```

This is similar to the syntax that we saw in Listing 24. We call the *InvokeMember* method to call the *Show* method of the form. The *BindingFlags.InvokeMethod* flag tells .NET to search for only methods in the assembly. *Nothing* is the normal "default binder," *nextForm* refers to the object that we want to act on (*show* in this case), and *New Object() {}* is an optional array of parameters.

Another (easier) implementation would be:

```
Dim formID As Form = CType(nextForm, Form)
formID.Show()
```

We then just close the current form[12].

The *findObjectByName* function is a general-use function for determining an object reference based on its name. We include it here so that it is available for all training forms in the project[13].

The *ExitTraining* and *queryExit* methods work together. The first is similar to the code in the *Closing* handler but allows it to be called from a button or menu within other forms. The *queryExit* function demonstrates how to ask the user for input, in this case whether he really wants to exit. The *answer* variable is a good example of an *Enum* (enumeration). There are various possible answers to the message box (Yes, No, Cancel, etc.) and this set of values is defined as an Enum, *System.Windows.Forms.DialogResult*. We then declare a variable of this enum type and use it as the return value of the MessageBox.Show method. This allows us to use the *If answer = dialogResult.Yes* syntax.

[11] This method is why we have the *Imports System.Reflection* line at the top of the class.

[12] Showing the new form first eliminates a flash that you will see if you close the first form *before* showing the second one.

[13] Like including it in a system book in ToolBook.

```
#Region "Properties"
  <Description("The name of 'next' form to open."), _
    Category("Extended"), Bindable(True)> _
    Public Property NextForm() As String
    Get
      Return c_nextFormName
    End Get
    Set(ByVal Value As String)
      c_nextFormName = Value
    End Set
  End Property

  <Description("The name of 'previous' form to open."), _
    Category("Extended"), Bindable(True)> _
    Public Property PreviousForm() As String
    Get
      Return c_previousFormName
    End Get
    Set(ByVal Value As String)
      c_previousFormName = Value
    End Set
  End Property
#End Region
End Class
```

Listing 30. Master Form Properties

We saw some of this code in Listing 25 when we were talking about attributes. The <Description>, <Category>, and <Bindable> are attributes of the properties, determining their descriptive text, category of the Properties window, and whether they can be bound to a database respectively. These property definitions in the MasterForm class make the *NextForm* and *PreviousForm* properties available in the Properties window (and via code if desired) for each inheriting training form in the project. Notice how we store the value of the properties in the private variables declared in Listing 28.

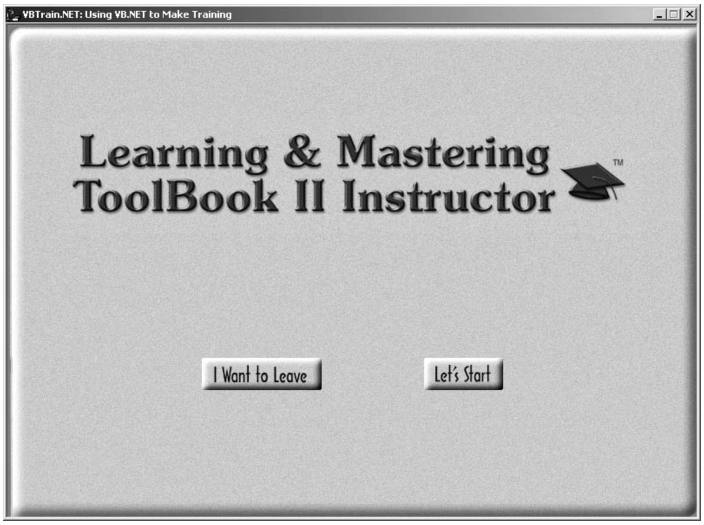

VBTrain.NET: Using VB.NET to Make Training
Learning & Mastering ToolBook II Instructor™
I Want to Leave
Let's Start

Figure 61. Start Form

Start Form

The starting form of the training inherits directly from *MasterForm* because it is a unique user interface not repeated elsewhere in the training (Figure 61). Here are properties and controls that we add at this level:

- *BackgroundImage* property (set to *background no cutouts.bmp* but then stored inside the assembly)
- *NextForm*[14] property ("TableOfContents")
- PictureBox control displaying the "Learning & Mastering…" graphic
- Two *platteGraphicalButtonLarge* controls[15]: one for starting the training and the other for exiting

Notice that any properties (*Size*, *KeyPreview*, etc.) that we don't change take on the value they have in the *MasterForm*. The only significant code in *StartForm* is for the graphical buttons as shown in Listing 31.

```
Option Explicit On
Option Strict On

Public Class StartForm
  Inherits LMTraining.MasterForm

' Windows Form Designer generated code here

#Region "Base Events"
  Private Sub leaveButton_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles leaveButton.Click
    Call ExitTraining()
  End Sub

  Private Sub startButton_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles startButton.Click
    goNextPage(Me, Me.NextForm)
  End Sub
#End Region
End Class
```

Listing 31. Start Form Code

The key line here is the *Inherits LMTraining.MasterForm*. This gives us both the visual side and the code side of *MasterForm* as we have been discussing. The *leaveButton_Click* subroutine calls the *ExitTraining* method while the *startButton_Click* subroutine calls the *goNextPage* method (See Listing 29 for both methods). Notice how we send *Me* as the first parameter (so the method knows *which* form to operate on) and *Me.NextForm* as the second parameter (the name of the form to navigate to).

Navigation Form

The rest of the forms in the training have buttons for moving forwards and backwards. So we create another layer and name it *NavigationForm*. We showed this form back in Figure 2 and include the "Component Tray" (which holds components without a user interface) and the Menu Editor in Figure 62. We inherit from *MasterForm* and keep all the form properties the same. We don't add a

[14] We defined this property in *MasterForm*.

[15] I used a custom control here so that these would be graphical buttons. This means that they highlight when you move your mouse over them and depress when you click them. As we will see later, we do this by swapping out graphics.

BackgroundImage in this case since there are different images that we need to use on forms that inherit *NavigationForm* (and thus we don't want to assign this property here). We do add the following controls, however:

- Three *graphicalButtonSmall* custom controls[16], one for going forward, one for going backwards, and one for displaying a popup menu
- A *platteProgressBar* custom control[17] to display the student's percentage completed as she goes through the training
- A *ToolTip* control that allows us to add tooltips[18] to the *graphicalButtonSmall* controls
- A *ContextMenu* control that allows us to create and display a popup menu

As on the previous form, we use a custom control to give us graphical buttons that highlight and depress. We cover how to build these controls later in the book. When we add the *ToolTip* control to the form, every other control (i.e., the navigation buttons) on the form gets a "ToolTip on ToolTip1" property in which we can enter the text that we want displayed. When we click on the *ContextMenu* control, we get a nice editor as shown in Figure 62, where we can enter the various menu items (including cascaded menus if desired) right in the editor. When we select a menu item, the Properties window has additional properties like "Checked" and "Shortcut." The code for *NavigationForm* is shown below.

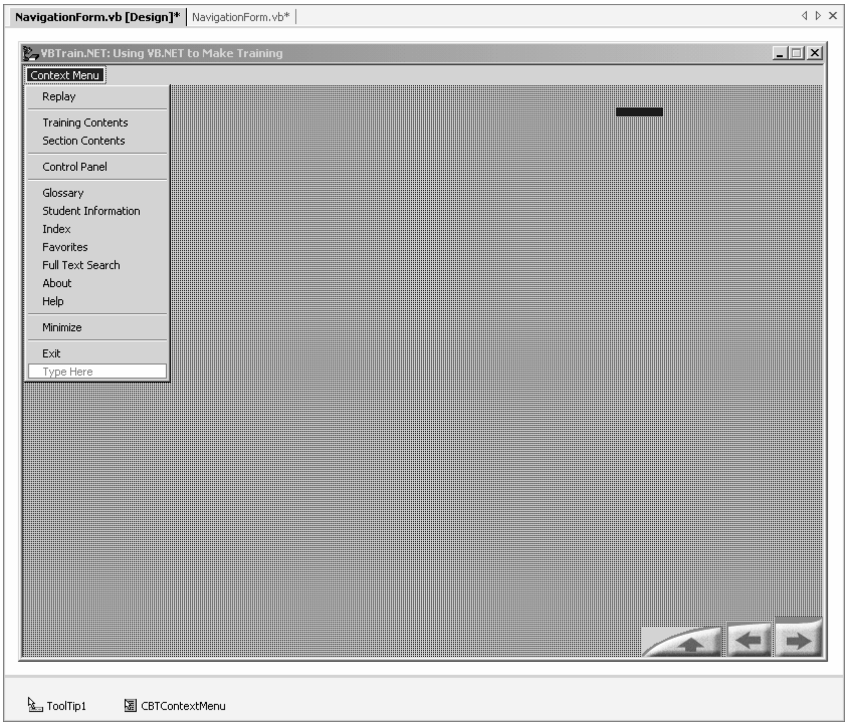


Figure 62. Navigation Form within the IDE

```
Option Explicit On
Option Strict On
```

[16] This is a special control where we store (and allow the developer to set as properties) the up, down, hilite, and disabled graphic of each button. The control then swaps the graphics on the *MouseEnter* and *MouseDown* events to make the button work correctly.

[17] This is the rectangle at the upper right of Figure 62.

[18] The yellow "labels" that you see when you hover over a Toolbar button or other items in most Windows applications.

```
Imports System.ComponentModel

Public Class NavigationForm
  Inherits LMTraining.MasterForm

' Windows Form Designer generated code here

#Region "Variables"
  Private c_numSteps As Integer = 0
  Private c_nextPageDisabled As Boolean = False
  Private c_previousPageDisabled As Boolean = False
#End Region

#Region "Base Events"
  Private Sub nextPage_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles nextPage.Click
    Call goNextPage(Me, Me.NextForm)
  End Sub

  Private Sub previousPage_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles previousPage.Click
    Call goPreviousPage(Me, Me.PreviousForm)
  End Sub

  Private Sub exitMenu_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles exitMenu.Click
    Call ExitTraining()
  End Sub

  Private Sub CBTMenu_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles CBTMenu.Click
    CBTContextMenu.Show(Me, Me.PointToClient(Me.MousePosition))
  End Sub

  Private Sub NavigationForm_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    If Me.NextForm = "" OrElse Me.nextPageButtonDisabled = True Then
      nextPage.Enabled = False
    Else
      nextPage.Enabled = True
    End If
    If Me.PreviousForm = "" OrElse Me.previousPageButtonDisabled = True Then
      previousPage.Enabled = False
    Else
      previousPage.Enabled = True
    End If
  End Sub
#End Region
```

Listing 32. Navigation Form Class, Variables, and Base Events Code

After the familiar *Option* statements, we import *System.ComponentModel*, which we use for our property definitions. We inherit from *MasterForm* as discussed above. Our variables store the values of the three properties described below. The *nextPage_Click*, *previousPage_Click*, and *exitMenu_Click*[19] handlers

[19] Notice how this is the handler of the menu item from our *ContextMenu*. We create subroutine shells with menu items just like we do for other objects.

call the corresponding methods located in *MasterForm*. Notice how the *startButton* on the *StartForm* (Listing 31) calls the same *goNextPage* method. See how inheritance gives us code reuse?

The *CBTMenu_Click* handler calls the *Show* method of our *CBTContextMenu* object[20]. The *Me* refers to the object with which the menu is associated. The second parameter is the position at which to show the menu. Notice how we need the *PointToClient* method to properly convert the mouse coordinates.

In response to the *Load* event, we determine whether to enable the "forward" and "back" buttons. Notice how we look both at the *NextForm* property (defined in *MasterForm*) and the *nextPageDisabled* property (defined here). Notice how we refer to the property (*nextPageDisabled*) rather than the private variable (*c_nextPageDisabled*). We do this to make the code more readable and to allow inheriting forms to override this property if desired.

```
#Region "Properties"
  <Description("Set this to the number of interactive steps on the training form."), _
    Category("Extended"), DefaultValue(0), Bindable(True)> _
    Public Property numSteps() As Integer
    Get
      Return c_numSteps
    End Get
    Set(ByVal Value As Integer)
      c_numSteps = Value
    End Set
  End Property

  <Description("Set this to TRUE if you want the next page button disabled."), _
    Category("Extended"), DefaultValue(False), Bindable(True)> _
    Public Property nextPageButtonDisabled() As Boolean
    Get
      Return c_nextPageDisabled
    End Get
    Set(ByVal Value As Boolean)
      c_nextPageDisabled = Value
    End Set
  End Property

  <Description("Set this to TRUE if you want the previous page button disabled."), _
    Category("Extended"), DefaultValue(False), Bindable(True)> _
    Public Property previousPageButtonDisabled() As Boolean
    Get
      Return c_previousPageDisabled
    End Get
    Set(ByVal Value As Boolean)
      c_previousPageDisabled = Value
    End Set
  End Property
#End Region
```

Listing 33. Navigation Form Properties Code

[20] Buttons, TextBoxes, and some other controls have a *ContextMenu* property that you can set to a ContextMenu object such as *CBTContextMenu* as in this example. .NET will then display the menu when you right-click on the object. We needed to write code in this case, though, since we wanted to display the menu on the normal click.

The property definitions are probably fairly familiar by now. We define the "number of steps" to be completed on the form as well as whether the "forward" and "back" buttons should be disabled. Notice how we add a *DefaultValue* attribute. This allows the developer to right-click on the property value and select "reset."[21] Any value that is <u>not</u> the default value will appear bolded in the Properties window. The default value is distinct from the initial value of the property, which is controlled by the initialization of the private variables (Listing 32).

Table of Contents

We've already looked at the *TableOfContents* form in detail (Figure 49, Listing 17, Listing 18, and Listing 19) as part of our discussion on handling events. We inherit it directly from *NavigationForm* since we need the navigation buttons and progress bar but don't need the training or question elements that we will add to later levels. Here are properties and controls that we add to make up this form:

- *BackgroundImage* property (set to *backHOME.bmp* and then stored inside the assembly)
- *NextForm* property ("playingMedia")
- *PreviousForm* property ("")
- Two *platteTransparentPictureBox* custom controls[22]
- Six *platteBitmapButtonFromFile* custom controls[23]
- A *plattePicture* custom control[24]

Note that setting the *PreviousForm* property to "" has the effect of disabling the "backwards" button so they can't go back to the starting form.

Training Form

For our actual training, we want another set of user interface buttons, another progress bar showing completion within a section, and a title box. We also want all forms of this type to have the same *BackgroundImage*, so we can add that at this level as well. We inherit the rest of the user interface from *NavigationForm* with the result shown in Figure 63. Here are the properties and controls that we add:

- *BackgroundImage* property (set to *back7c.bmp* and then stored inside the assembly)
- Another *platteProgressBar* custom control (section completion)

[21] Accomplishing this with other data types often requires more than just a *DefaultValue* as we will see later.

[22] This is yet another custom control that displays a graphic with a transparent (chromakey) area. We use this to show both the cutout around the progress bar and the cutout around the light.

[23] This control is similar to the *graphicalButtonSmall* custom control that we used on the *NavigationForm*. However, that implementation only works with smaller graphics since it uses *ImageList* controls to store them. For larger graphics, we read them directly from their files, hence the control name. More on this in the Graphics chapter.

[24] This is where we show the various light graphics. As we discuss later in the chapter on graphics, we have to pick the control we inherit from carefully so that other objects will be transparent to a graphic shown in this control rather than just to the *BackgroundImage* of the form itself.

- Four more *graphicalButtonSmall* custom controls (the buttons to the lower left of the form)
- A *Label* control (to display the title)[25]
- Another *ToolTip* control (for showing tooltips on the four *graphicalButtonSmall* controls

Listing 34 and Listing 35 show the implementing code for *TrainingForm*.

```
Option Explicit On
Option Strict On
Imports System.ComponentModel

Public Class TrainingForm
  Inherits LMTraining.NavigationForm

' Windows Form Designer generated code here

#Region "Variables"
  Private c_expertInfoEnabled As Boolean = True
  Private c_codeTipsEnabled As Boolean = True
  Private c_showMeEnabled As Boolean = True
  Private c_letMeTryEnabled As Boolean = True
  Private c_formTitle As String = "Put Title Here"
#End Region

#Region "Base Events"
  Private Sub TrainingForm_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    expertInfo.Enabled = Me.expertInfoEnabled
    codeTips.Enabled = Me.codeTipsEnabled
    showMe.Enabled = Me.showMeEnabled
    letMeTry.Enabled = Me.letMeTryEnabled
    pageTitle.Text = Me.formTitle
  End Sub

  Private Sub expertInfo_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles expertInfo.Click
    MessageBox.Show(System.String.Concat("Expert Info for form ", Me.Name))
  End Sub

  Private Sub codeTips_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles codeTips.Click
    MessageBox.Show(System.String.Concat("Code Tips for form ", Me.Name))
  End Sub

  Private Sub showMe_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles showMe.Click
    MessageBox.Show(System.String.Concat("Show Me for form ", Me.Name))
  End Sub
```

[25] Notice that we set the *BackColor* of the Label to *Color.Transparent* (the first choice under the “Web” tab of the Color dialog), so that we see the background graphic showing through.

```
  Private Sub letMeTry_Click(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles letMeTry.Click
    MessageBox.Show(System.String.Concat("Let Me Try for form ", Me.Name))
  End Sub
#End Region
```

Listing 34. Training Form Class Definition, Variables, and Base Events Code

The variables hold the "enabled" state of the new interface buttons as well as the title of the form. As with *NavigationForm*, we set the *Enabled* of each of the buttons based on the corresponding form property. We use a slightly different syntax here since it is more likely that these buttons will be disabled on various individual forms. We also set the *Text* of the *pageTitle* Label control based on the *FormTitle* property. Each of the interface buttons then has its own *Click* handler. We just show a message box at the moment, but in actual implementation we would display information, simulations, or animations based on the actual form being displayed. That's why *Me.Name* is in the message. This shows that we are able to grab the name of the actual *instance* rather than the name of this class.

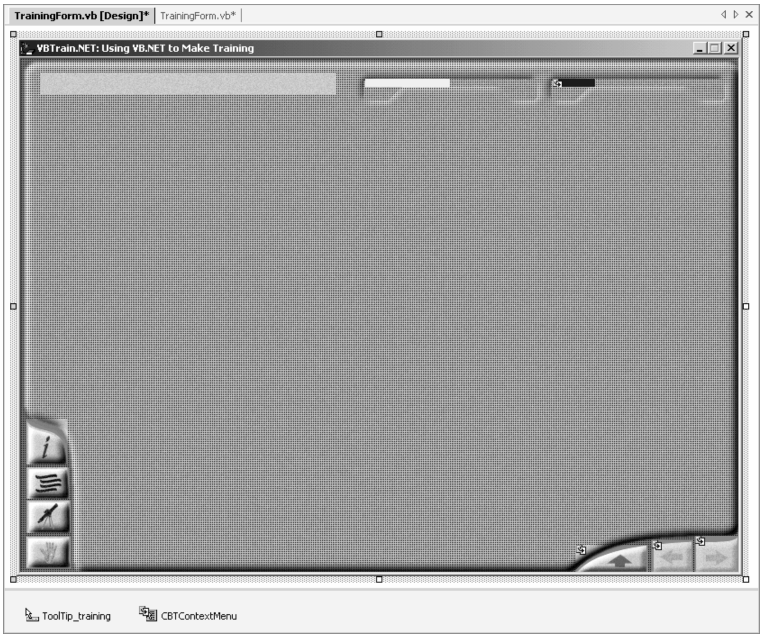

TrainingForm.vb [Design]*
TrainingForm.vb*
VBTrain.NET: Using VB.NET to Make Training
ToolTip_training
CBTContextMenu

Figure 63. Training Form

```
#Region "Properties"
  <Description("Set this to True to enable the 'Expert Info' button on this form."), _
    Category("Extended"), DefaultValue(True), Bindable(True)> _
    Public Property expertInfoEnabled() As Boolean
    Get
      Return c_expertInfoEnabled
    End Get
    Set(ByVal Value As Boolean)
      c_expertInfoEnabled = Value
    End Set
  End Property

  <Description("Set this to True to enable the 'Code Tips' button on this form."), _
    Category("Extended"), DefaultValue(True), Bindable(True)> _
    Public Property codeTipsEnabled() As Boolean
    Get
      Return c_codeTipsEnabled
    End Get
    Set(ByVal Value As Boolean)
      c_codeTipsEnabled = Value
    End Set
  End Property

  <Description("Set this to True to enable the 'Show Me' button on this form."), _
    Category("Extended"), DefaultValue(True), Bindable(True)> _
    Public Property showMeEnabled() As Boolean
    Get
      Return c_showMeEnabled
    End Get
    Set(ByVal Value As Boolean)
      c_showMeEnabled = Value
    End Set
  End Property

  <Description("Set this to True to enable the 'Let Me Try' button on this form."), _
    Category("Extended"), DefaultValue(True), Bindable(True)> _
    Public Property letMeTryEnabled() As Boolean
    Get
      Return c_letMeTryEnabled
    End Get
    Set(ByVal Value As Boolean)
      c_letMeTryEnabled = Value
    End Set
  End Property

  <Description("Enter the descriptive title of this form."), _
    Category("Extended"), DefaultValue("Put Title Here"), Bindable(True)> _
    Public Property FormTitle() As String
    Get
      Return c_formTitle
    End Get
    Set(ByVal Value As String)
      c_formTitle = Value
    End Set
  End Property
#End Region
```

Listing 35. Training Form Properties Code

These property definitions are similar to what we've seen before. Notice that we again initialize the corresponding private variables and then have a *DefaultValue* that allows developers to select "Reset" to set each property back to its default.

Individual Training Forms

All the forms in the training that need this user interface can now inherit

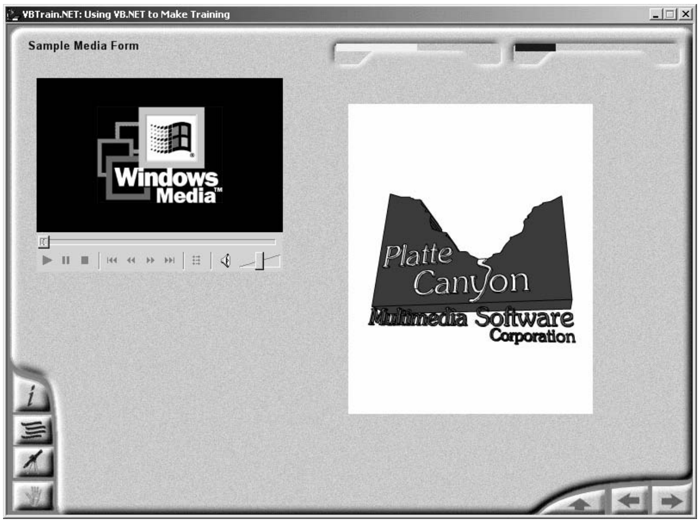

VBTrain.NET: Using VB.NET to Make Training
Sample Media Form
Windows Media
Platte Canyon Multimedia Software Corporation

Figure 64. Sample Media Form (Training Form Instance)

from *TrainingForm*. We saw one example back in Figure 3. We show another one in Figure 64. In this latter case, we set each of the user interface buttons to be enabled and set a *NextForm* and *PreviousForm* property value. The form itself has remarkably little code, as it inherits all the hard parts!

```
Option Explicit On
Option Strict On

Public Class playingMedia
  Inherits LMTraining.TrainingForm

' Windows Form Designer Code

#Region "Base Events"
  Private Sub playingMedia_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles MyBase.Load
    AxShockwaveFlash1.Movie = String.Concat(graphicsPath, "icbt.swf")
    AxMediaPlayer1.FileName = String.Concat(graphicsPath, "radar.avi")
  End Sub
#End Region
End Class
```

Listing 36. Sample Media Form Code

The most important point to make once again is how this form inherits from *TrainingForm* (which in turn inherits from *NavigationForm* which in turn inherits from *MasterForm*). This might be a good time to glance at Figure 1 once again. The only other code is to initialize our Windows Media Player and Flash Player[26] with the runtime location of their media files. Notice how we use the public *graphicsPath* variable that we defined when loading the application (Listing 27).

Question Form

The last part of our training example deals with review questions. We want our trusty navigation buttons but not the other user interface buttons that we added in *TrainingForm*. We therefore want to inherit from *NavigationForm* and our

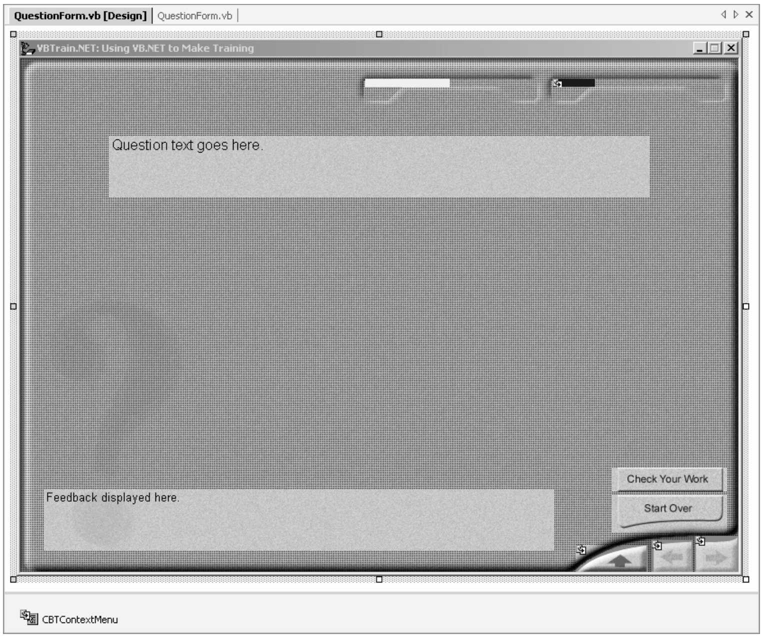


Figure 65. Question Form

[26] The *AxShockwaveFlash1* and *AxMediaPlayer1* syntax reflects that these are ActiveX/COM components. We will cover this in detail in the Media chapter.

question-specific elements to make the *QuestionForm* class shown in Figure 65.

To do this, we add the following properties and controls:

- *BackgroundImage* property (set to *quiz backdrop 2.bmp* and then stored inside the assembly)
- Another *platteProgressBar* custom control (section completion)
- Two *platteBitmapButtonFromFile* custom controls for the “Check Your Work” and “Start Over” graphical buttons
- Two *Label* controls (one to display the question text and the other to display feedback)

Listing 37shows the implementing code for *QuestionForm*.

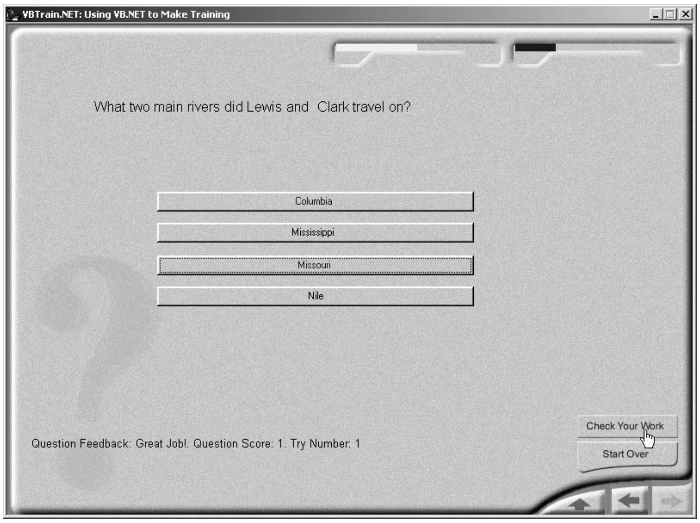
```
Option Explicit On
Option Strict On

Public Class QuestionForm
  Inherits LMTraining.NavigationForm

' Windows Form Designer Code

#Region "Events"
  Protected Event CheckQuestion(ByVal formID As Form)
  Protected Event ResetQuestion(ByVal formID As Form)
#End Region

#Region "Base Events"
  Private Sub QuestionForm_Load(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles MyBase.Load

    questionFeedback.Text = "" ' So no feedback initially
  End Sub

  Private Sub checkWorkBtn_Click(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles checkWorkBtn.Click

    RaiseEvent CheckQuestion(Me)
  End Sub

  Private Sub resetQuestionBtn_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles resetQuestionBtn.Click

    RaiseEvent ResetQuestion(Me)
  End Sub
#End Region
End Class
```

Listing 37. Question Form Code

The inheritance and *Load* part of the code is straightforward. We simply clear the feedback when the user enters the question. The design diverges at this point, however. Rather than variables and properties, we have events. The reason is that we don’t know anything at this level about the actual question objects that will be used. So rather than try to call methods of these unknown objects, we define and raise events that the inheriting forms can then handle to perform the desired actions. So when the user clicks the “Check Your Work” button, we raise the *CheckQuestion* event and include a reference to the current form as a parameter. Similarly with the "Start Over" button, we raise the *ResetQuestion* event. The inheriting form then can handle these events if desired to actually perform the appropriate action(s).

Individual Question Forms

All the review questions in the training now inherit from *QuestionForm*. To implement the individual forms, we add a question object and anything else (screen captures for example) that we might need. Figure 66 shows an example. In addition to a *PreviousForm* property for navigation, we added a *MultipleChoiceButtons* custom object[27] to actually handle the question task. Here's the form's code:



What two main rivers did Lewis and Clark travel on?

Question Feedback: Great Job!. Question Score: 1. Try Number: 1

Figure 66. Individual Question Form

```
Option Explicit On
Option Strict On

Public Class Question1
  Inherits LMTraining.QuestionForm

' Windows Form Designer Code

#Region "Base Events"
  Private Sub Question1_Load(ByVal sender As System.Object, ByVal e As _
    System.EventArgs) Handles MyBase.Load

    questionText.Text = MultipleChoiceButtons1.QuestionText
  End Sub

  Private Sub Question1_CheckQuestion(ByVal formID As System.Windows.Forms.Form) _
    Handles MyBase.CheckQuestion

    Call MultipleChoiceButtons1.Score()
  End Sub

  Private Sub Question1_ResetQuestion(ByVal formID As System.Windows.Forms.Form) _
    Handles MyBase.ResetQuestion

    MultipleChoiceButtons1.Reset()
    questionFeedback.Text = ""
  End Sub
#End Region
```

[27] More on this object in the *Interactions and Questions* chapter.

```
#Region "Question Events"
  Private Sub MultipleChoiceButtons1_AnswerSelected(ByVal AnswerText As String, ByVal _
    AnswerFeedback As String, ByVal AnswerCorrect As Boolean) Handles _
    MultipleChoiceButtons1.AnswerSelected

    questionFeedback.Text = String.Concat("You answered: ", AnswerText, ". Feedback: ", _
      AnswerFeedback)
  End Sub

  Private Sub MultipleChoiceButtons1_QuestionScored(ByVal AnswerText As String, ByVal _
    AnswerFeedback As String, ByVal QFeedback As String, ByVal QuestionScore As _
    Decimal, ByVal TryNum As Integer) Handles MultipleChoiceButtons1.QuestionScored

    questionFeedback.Text = String.Concat("Question Feedback: ", QFeedback, _
      ". Question Score: ", QuestionScore.ToString, ". Try Number: ", TryNum.ToString)
  End Sub
#End Region
End Class
```

Listing 38. Individual Question Form Code

When we load the form, we set the *Text* of the *questionText* Label (inherited from *QuestionForm*) to the *QuestionText* property of the *MultipleChoiceButtons1* question object. We then handle the events raised by *QuestionForm*. Recall that clicking the “Check Your Work” button (as has just happened in Figure 66) raises the *CheckQuestion* event. We handle this in the form and in turn call the *Score* method of the question object. See Listing 10 for some of the implementation code for this method. Note that another type of question object might have a different method to call. Similarly, we call the question object’s *Reset* method in response to the *ResetQuestion* event. Notice that we also clear the *Text* of the *questionFeedback* Label since the question object has no idea that this Label exists. At the end of the code, we handle events of the question object itself. The *AnswerSelected* event is sent each time the user clicks on an answer. We list the answer and the feedback for that answer in the feedback object. The *QuestionScored* event is sent in response to our calling the question object’s *Score* method. We show the desired information once more in our feedback object.