Chapter 19 – Building Your Own Controls & References

# Objectives

* Creating a Control
* Creating a .NET DLL
* Creating a Combination DLL

This chapter centers around showing you how to create your own .NET framework controls and Dynamic Link Libraries (DLLs). By now it should be obvious to you that much of the power of Visual Basic comes from the ability to reuse existing controls to build up new applications – you get spoiled with the drag and drop richness of the controls that are available. We will examine how to create these reusable entities in this chapter.

# Creating a Control

As you've undoubtedly noticed, the controls that are available in VB allow you to quickly add functionality with little work. Think about how tough it would be to write a program that acts like WordPad without having the RichTextBox control available. We even take for granted the functionality provided by controls as simple as Command buttons, TextBoxes and ListBoxes. How thrilled would be with writing Graphical User Interface applications if these little controls didn’t exist?

Many VB programmers will spend a good deal of time looking for controls that provide the functionality they need before even starting to write an application. For example, if you needed to write a program that could perform image processing, would you want to have to code everything from scratch using the graphics tools that we examined in the graphics and multimedia chapter? Instead, you would be better off searching the Internet for a control that's already been written for you to use – just drop it into your project and you can start working on the core functionality of what your application is supposed to do instead of working about a task like image processing.

There will be times though, when you might not be able to find a control that does what you need. When that time comes, it is very nice to know how to create controls. You can place all your effort into creating a reusable control so that you never need to write that particular code again.

Building controls can range anywhere from the trivial to the complex depending on the purpose of the control. Remember that the idea behind a control is that it completely encapsulates and delivers some functionality. The TextBox control allows a user to input and edit text and also allows a programmer to retrieve the user's text into an application. As you might suspect, there are a few older books floating around that address the topic of control building. We will look at creating a couple of simple controls in this section.

We need to pause a minute and talk a bit about how a control gets created, instantiated, used and destroyed. Since we have been using controls right along, I think it merits some discussion to understand a little behind the scenes of what happens. Recall that there are two different times when we can work with our controls: (1) at design time (when we drag and drop them onto our form), and (2) at runtime (when our application is executing). Just as the two times are different in their purpose, what happens to the control in each is different as well.

## Design Time

When a COM/ActiveX control is added to a form at design time, the following events take place:

1. An instance of the control is created.
2. Any constituent controls are created. A constituent control is a control that is placed inside of another control. The other type of control is user-drawn, in which the user directly creates the control’s GUI.
3. The UserControl is created and all constituent controls are placed on the UserControl.
4. The UserControl’s Initialize event is fired.
5. The UserControl is placed where it belongs on its container. If it is the first instance of that control type on a form, the UserControl’s InitProperties event is fired to initialize the control’s properties to their default values. If the form already has an instance of the UserControl, then the ReadProperties event is fired instead of InitProperties.
6. The Extender object becomes available, which contains properties handled by the control’s container and not the control itself.
7. The UserControl’s Resize event fires to size the control. Resize will be used to resize each constituent control as well. User-drawn controls do not use the Resize event.
8. The Show and Paint events get fired so that the control can be seen. User-drawn controls use just the Paint event.
9. The Public events in the UserControl are now available for use. The user can now set the control’s properties through the property window in the IDE.

## Just Prior to Run Time in the IDE

The following events occur if you are in the Visual Studio IDE and decide to run the program:

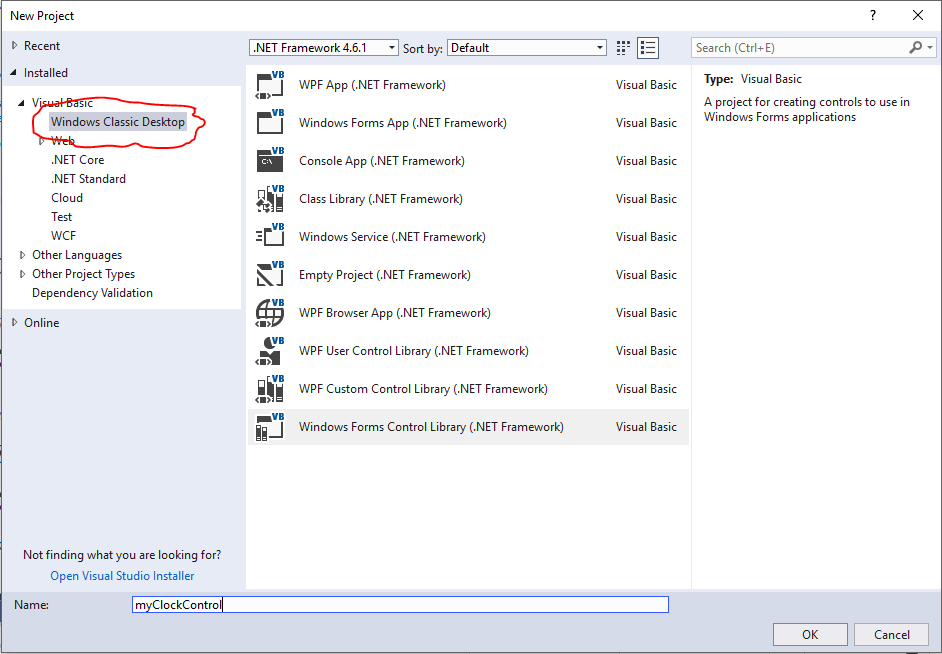
1. Running the program in the IDE closes the form and all its controls. The UserControl’s WriteProperties event is fired to save the design time property settings into the form object that is in memory (and to disk if the user saves the form).
2. The control is removed from the container it belongs in.
3. The UserControl’s Terminate event is fired.
4. The UserControl object and all its constituent controls get destroyed.

## Run Time:

The following events occur when the program begins execution:

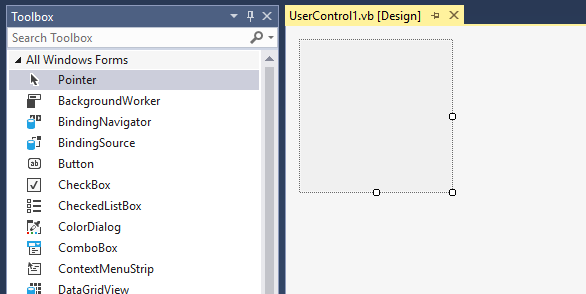
1. The UserControl’s Initialize event is fired and all the constituent controls are created.
2. The UserControl’s ReadProperties event is fired to retrieve the saved property values from the form object in memory or from disk if the form was saved.
3. The UserControl gets placed on the run-time instance of the container. The Extender object is now available.
4. The UserControl fires the Resize and/or Paint events to display itself.

Again, enough talk – let’s build a control. We will begin by starting a new VB Project. You will not select the typical Windows Forms Application as the project type – rather you will pick Windows Forms Control Library. You may need to click on the Windows Classic Desktop tab in the left-hand project type pane. The following screenshot shows the selection of creating a Windows Control Library project – I named the project myClockControl. We will create a clock control in this first example:



Once the project is opened, you will notice that the project screen looks familiar. You are provided with a form-like canvas which is where you will construct your control. Your control can use other controls (we talked about constituent controls above). Most controls that have a visual appearance are often built up upon combinations of other simpler controls.

The next screenshot shows the UserControl1.vb control canvas:

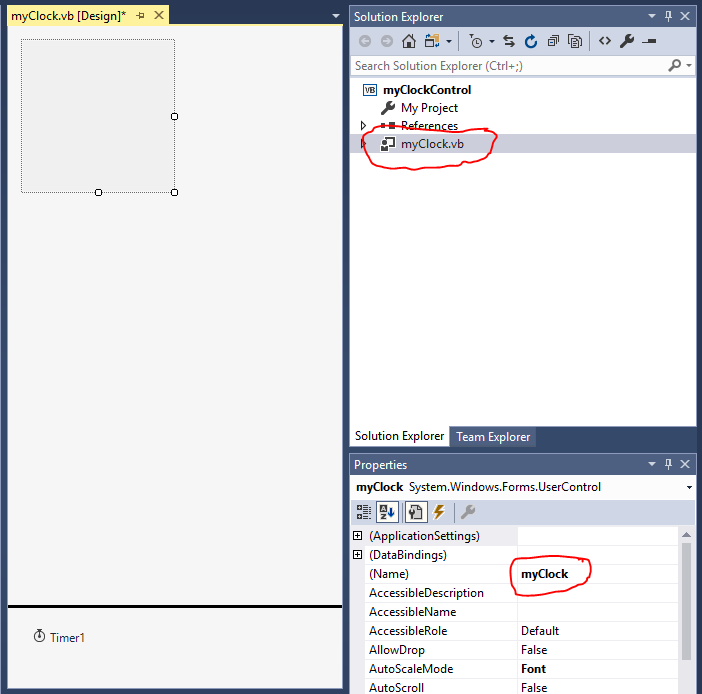


By default, the control will be built in UserControl1.vb, which really is just a simple container – it acts like a borderless form.

As we mentioned earlier, we are going to build a digital clock control. The control will automatically match its background color to that of the container that it is placed in. The developer will be able to set the foreground color to whatever color he/she chooses. Finally, we want the clock to update its display every 1 second (1000 ms).

We will begin by dragging and dropping a Timer control onto the UserControl surface. We need the timer so that every 1000 milliseconds it triggers to remind us to update our display. Set the Timer’s properties up as follows: Name – Timer1 (default name), Enabled – True and Interval – 1000. In addition, change the name of UserControl1 to myClock in both the control’s name and the filename. This last step is quite important as you’ll see in a bit:

Here's a screenshot of the completed control from the design perspective:



There certainly wasn't much to look at in the control itself. That's because the functionality happens in the code that sits behind the control. Add the following code to complete the clock control:

'Chapter 19 - myClockControl

Public Class myClock

'Create the Font and Brush that we will use to print the time

'and also set the default Foreground color to Black

Dim ArialFont As New Font("Arial", 12, FontStyle.Regular)

Dim myBrush As New SolidBrush(Color.Black)

Dim CurrentForeColor As Color = Color.Black

Overrides Property ForeColor() As Color

'We need to set ForeColor up as a property to allow the user

'to get and set the foreground color of the control. Remember

'that we aren't interested in allowing the user to set the

'background color -- it's supposed to match the container that

'it's in, so we will just inherit from whatever container is

'hosting our control. (We can override it if we want using

'the shadows keyword.)

Get

Return (CurrentForeColor)

End Get

Set(ByVal Value As Color)

CurrentForeColor = Value

Me.Invalidate()

End Set

End Property

Private Sub Timer1\_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick

'Every time the timer ticks, refresh the control's context,

'which in effect will force the time to be updated since

'Refresh forces Paint to be called

Me.Refresh()

End Sub

Protected Overrides Sub OnPaint(e As PaintEventArgs)

'Everytime we are told to Paint, draw the time on the control's

'surface...

'e.Graphics is the graphics context for the control

Dim myGfx As Graphics = e.Graphics

'Set the brush to whatever the current foreground color is

myBrush.Color = CurrentForeColor

'Print the time out

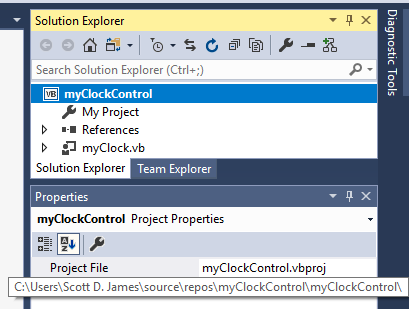
myGfx.DrawString(DateTime.Now.ToLongTimeString, ArialFont, myBrush,

10, 10)

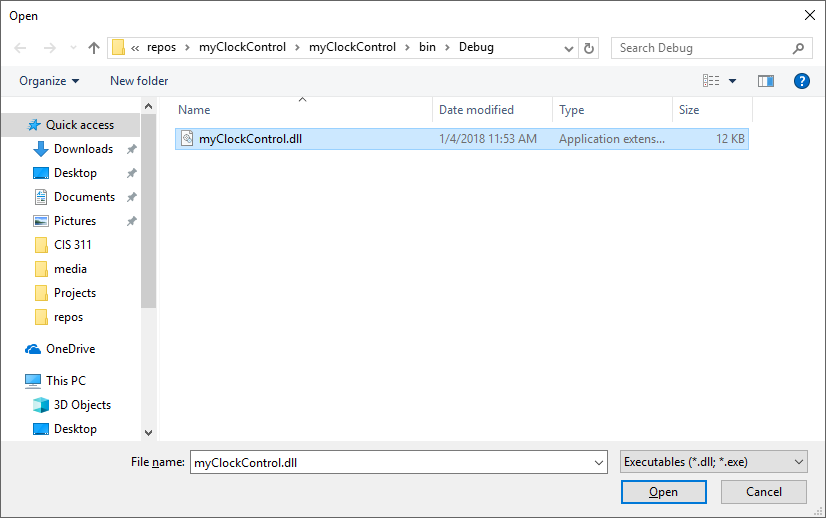
End Sub

End Class

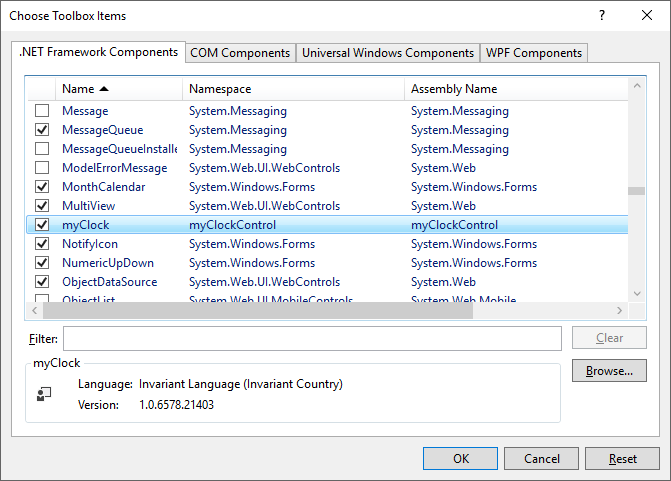
We are just about ready to use our clock control now. First, we must build the control into a control .DLL file so that other projects can use our control. You do this by clicking the Build🡪Rebuild menu option. Once the .DLL has been built, you are going to need to know where it was created. You can find out where the file went by clicking on the myClockControl icon in the upper Solution Explorer window and then hover the mouse over the Project Folder property in the lower properties window to see the path:



Now comes the fun part – let’s test the control out. Save your clock control project and then start a new normal Windows Forms project. After your new project is open, customize the Toolbox (Tools🡪Choose Toolbox Items) for .NET components. Press the Browse button and point to the directory where your .DLL was written (you should have been able to determine this from the Property Pages window in the last step). You will then have to move into the "bin\debug" directory under the directory you are currently sitting in. The next screenshot shows locating the control's DLL file:

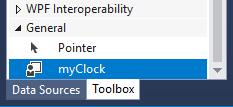


Once you have the control selected, click the Open button. You will then be taken back to the Choose Toolbox Items dialog and you’ll see that the myClock control appears and is checked:

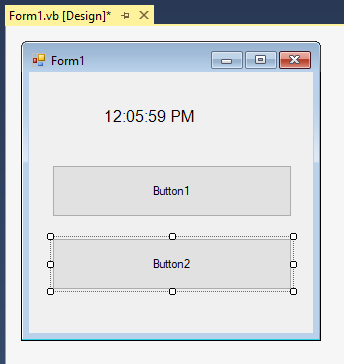


Click OK to close this dialog up.

You will be returned to the form design surface. If you now look toward the bottom of your toolbox, you’ll see the myClock control is there!



Drag a copy of your clock control to your form and also add two command buttons (just use the default names for all of the controls). Your completed form should look like the following screenshot:



Did you notice that the clock started working even in design mode? All you needed to do was drag and drop it on your form and the control went live. That's some pretty slick functionality…

Here's the source code that you need to add to handle the two buttons. When you click on Button1, it will change the clock's foreground color from black to red. When you click on Button2, it will change the form's background color to white and you'll see that the clock automatically changes its background color to match.

'Chapter 19 - Program 1

Public Class Form1

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'Click button 1 to change the clock's foreground color

MyClock1.ForeColor = Color.Red

End Sub

Private Sub Button2\_Click(sender As Object, e As EventArgs)

Handles Button2.Click

'Click button 2 to change to the form's background color and

'see that the clock's background color automatically changes

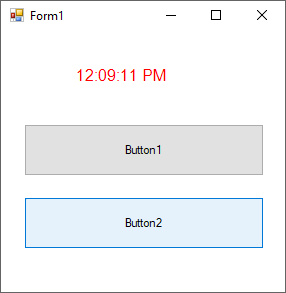
'to match

Me.BackColor = Color.White

End Sub

End Class

Run your program and you can see that the control is doing its thing! The following screenshot shows the program after I have clicked both command buttons:



Building that last control wasn't too tough. Let's go ahead and build another control that's a bit more complex: this one will provide a gradient fill background for the portion of a form where you place the control. You will even be able to add other controls onto the gradient control since it will act as a container.

Using the control will be simple…you can drag and drop the control and you will get the control's default appearance. In addition, you will be able to specify the StartColor, EndColor and GradientMode if you want to change the looks of the control from the default.

Let's get going! Create a new Windows Forms Control Library project again. Select the name of myGradientControl for the project. Once the project is open, change the name of the control from UserControl1 to GradientCtl. Everything in this control project is handled programmatically, so there is nothing to drag and drop on your control. Here’s the source code for the project:

'Chapter 19 - myGradientControl

Imports System.Drawing.Drawing2D

Public Class GradientCtl

'Create the default Starting color, the default Ending color and

'the default GradientMode -- SVSU colors!

Dim theStartColor As Color = Color.Red

Dim theEndColor As Color = Color.White

Dim theGradientMode As LinearGradientMode = LinearGradientMode.ForwardDiagonal

'Allow the user to change the start color of our gradient control

Property StartColor() As Color

Get

Return (theStartColor)

End Get

Set(ByVal Value As Color)

theStartColor = Value

'Force a redraw when this color changes

Me.Invalidate()

End Set

End Property

'Allow the user to set the end color of the gradient

Property EndColor() As Color

Get

Return (theEndColor)

End Get

Set(ByVal Value As Color)

theEndColor = Value

'Force a redraw when this color changes

Me.Invalidate()

End Set

End Property

'Allow the user to specify the gradient mode to use

Property GradientMode() As LinearGradientMode

Get

Return (theGradientMode)

End Get

Set(ByVal Value As LinearGradientMode)

theGradientMode = Value

'Force a redraw when this gradient changes

Me.Invalidate()

End Set

End Property

Protected Overrides Sub OnPaint(e As PaintEventArgs)

'Create a gradient brush as big as the client area,

'with specified start/end color and gradient mode

Dim myBrush As New LinearGradientBrush(Me.ClientRectangle,

theStartColor, theEndColor, theGradientMode)

'Paint the background

e.Graphics.FillRectangle(myBrush, Me.ClientRectangle)

'Get rid of the brush

myBrush.Dispose()

'Now let the base control do its thing...

MyBase.OnPaint(e)

End Sub

Private Sub GradientCtl\_Resize(sender As Object, e As EventArgs)

Handles Me.Resize

'If the gradient control gets resized, we need to redraw it...

Me.Invalidate()

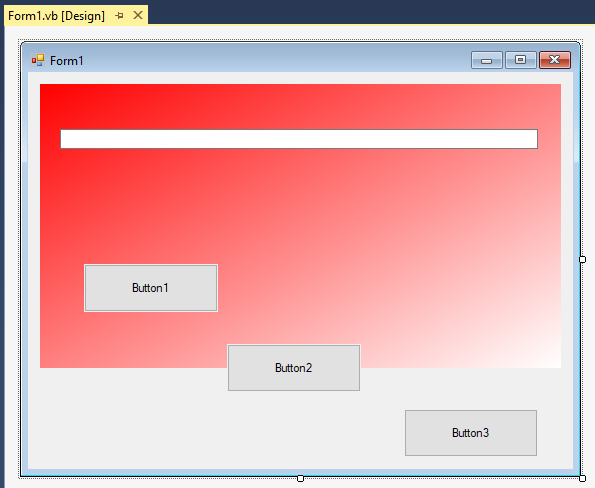
End Sub

End Class

That's it for the gradient control's code. Again, you will need to save the project and build (Build🡪Rebuild) the control into a DLL file. You will probably want to look at the control's property pages so that you know where the DLL file was written.

As we did last time, start another new Windows Forms application. Go through the process of customizing your toolbox and add the .NET component myGradientControl control to your toolbox. You will need to browse to the directory where the DLL file was written; remember that it will be in the "bin/debug" directory under where your myGradientControl project was stored.

I created the following form to test the gradient control. All that is on the form is a copy of the gradient control, a TextBox and three Command buttons. I used the default names for each of the controls. Here’s what the form looks like:



Notice that the TextBox and Button1 are positioned completely inside of the gradient control. Button2 is positioned half on and half off the gradient control. Button3 is placed outside of the gradient control. Each one of the buttons successively changes some attribute on the gradient control. Button1 will change the starting Red color to the AliceBlue color. Button2 will change the ending White color to a DarkOliveGreen color. Finally, Button3 will change the gradient style from a forward diagonal style to a vertical style.

Here's the source code behind the application:

'Chapter 19 - Program 2

Public Class Form1

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'Change the Start color from Red to AliceBlue

TextBox1.Text = "VB Programming Rocks!"

GradientCtl1.StartColor = Color.AliceBlue

End Sub

Private Sub Button2\_Click(sender As Object, e As EventArgs)

Handles Button2.Click

'Change the End color from White to DarkOliveGreen

GradientCtl1.EndColor = Color.DarkOliveGreen

End Sub

Private Sub Button3\_Click(sender As Object, e As EventArgs)

Handles Button3.Click

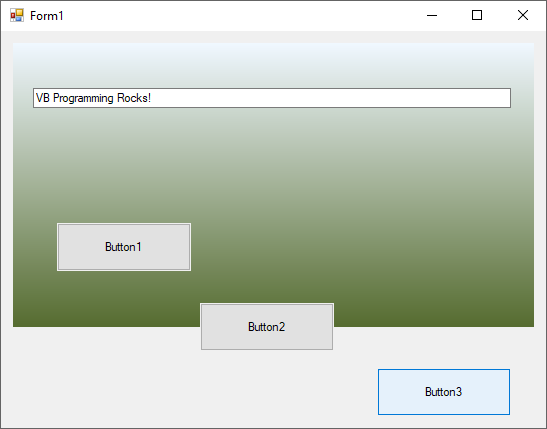
'Change the Linear mode from ForwardDiagonal to Vertical

GradientCtl1.GradientMode = Drawing.Drawing2D.LinearGradientMode.Vertical

End Sub

End Class

The next screenshot shows the executing application after all three of the command buttons have been pressed. Notice that the starting and ending colors as well as the gradient style have all been changed. The background color of the controls embedded in the gradient control are not affected in any way:

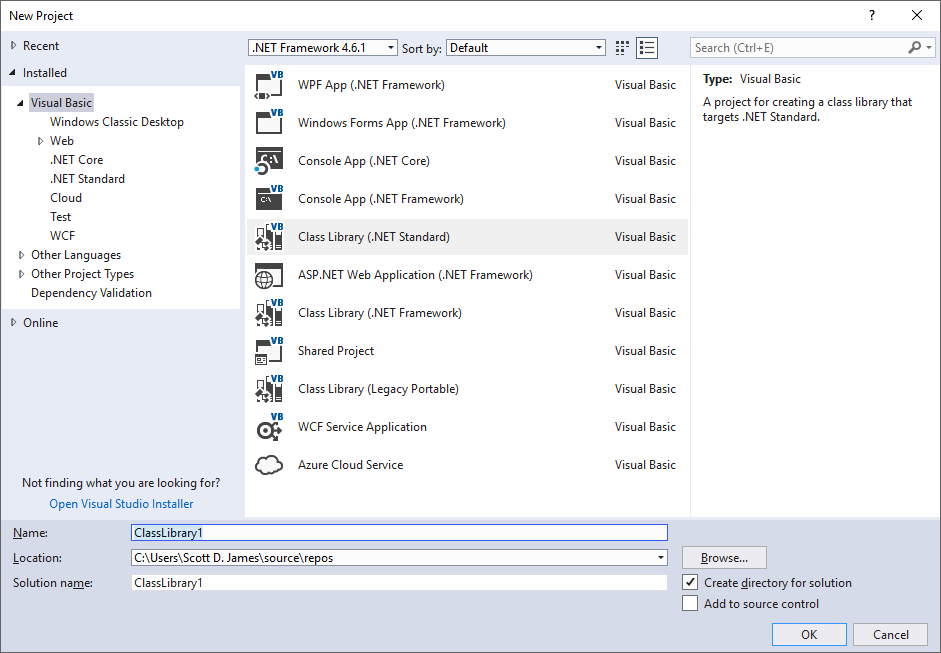


# Creating a .NET DLL

You may be thinking that we already created DLLs in the last section. After all, when we added our .NET framework controls to our form we looked for the .DLL files that contained them. The truth is we were close indeed, however there is a fundamental difference between a control and a pure DLL file.

The primary difference is that DLL files typically are non-visual code libraries that get created for use by other applications. The DLL contains executable code, but is not executable by itself. In fact, DLLs tend to be what we use to access the object model of other applications. These are the Project🡪Add Reference that we added to our projects to allow us to use and communicate with object models that we didn't create.

Let's look at creating a simple DLL. Let's say that we want to build a DLL that contains a spell checker. Rather than creating the spell checker from scratch, we will use Microsoft Word’s spell checking function. In effect, we will be creating a DLL with a simple interface to perform spell checking. Obviously to write this application you must have Microsoft Word on your computer. Start VB up and create a new Class Library project (I just used the default name of ClassLibrary1):



There will only be one class module inside of the project – I changed the name of the class and its filename to clsWordCheck. There aren't any forms or anything else. Also make sure that you add a reference to the Microsoft Word Object Library since we are cheating on the spell checking piece and using Word's instead of writing our own.

Add the following code to the clsWordCheck code module:

'Chapter 19 - ClassLibrary1

Imports Microsoft.Office.Interop

Public Class clsWordCheck

'Create a Word Instance in our DLL so that we can access

'Word's spell checker

Private aWordInst As Word.Application

Public Sub New()

'When our user creates an instance of this class, we will

'start up an instance of word but it will never be visible

aWordInst = New Word.Application()

End Sub

Public Function CheckWord(ByVal strWord As String) As Boolean

'This is the only function that our DLL will provide...CheckWord

'You send in a word, it passes the word to Microsoft Word, which

'will perform the spell checking. Word sends you back a boolean

'value letting you know whether or not the word was found in Word's

'dictionary -- we will simply pass this value back to our user.

'If the value is true, it means the word was spelled correctly; if

'the value is false, it means the word was incorrectly spelled.

CheckWord = aWordInst.CheckSpelling(strWord)

End Function

Protected Overrides Sub Finalize()

'Clean up our word instance when this class instance is getting

'destroyed.

aWordInst = Nothing

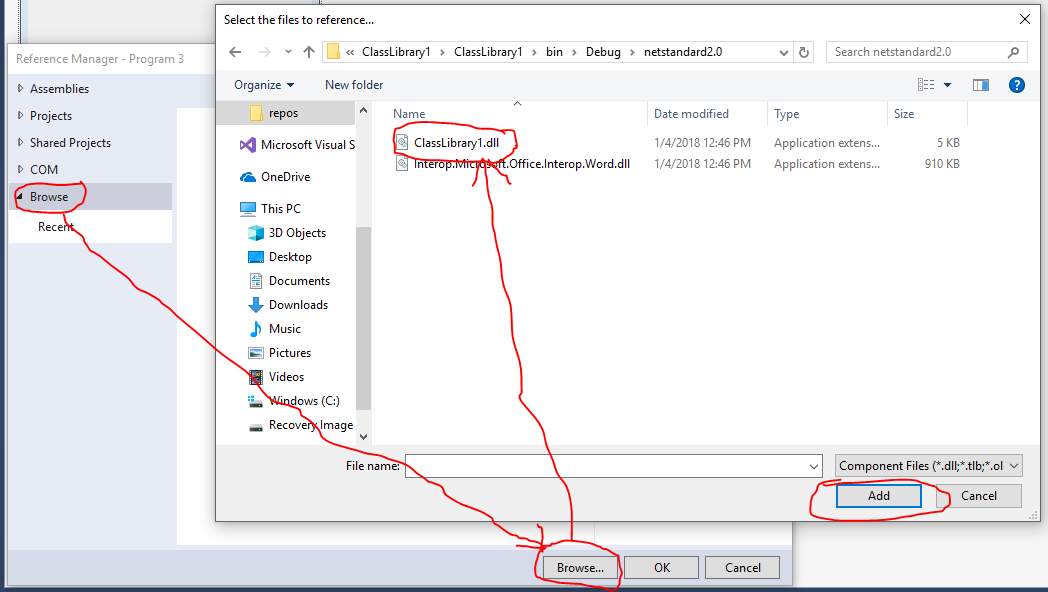
End Sub

End Class

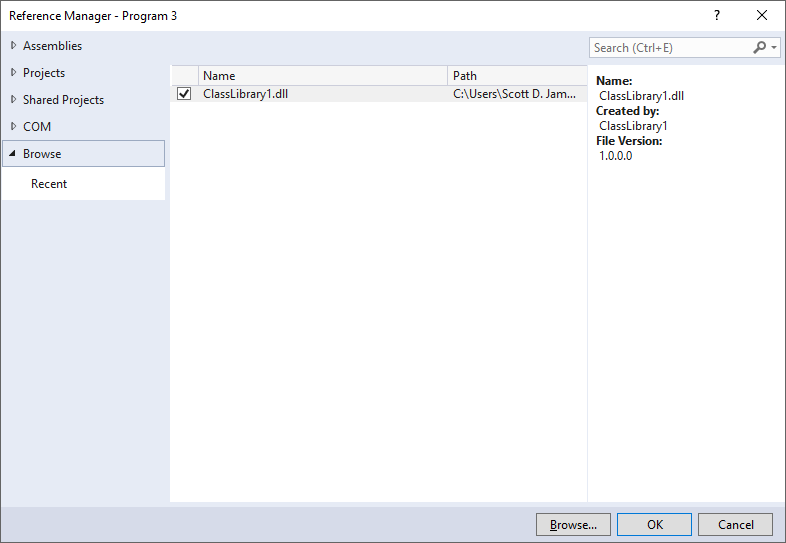
That certainly was simple – all a user needs to do is call CheckWord with a string containing a word to be spell checked. In return, the user will receive a Boolean value indicating whether the word was correctly spelled.

Again, go ahead and save your project. Then build (menu option Build🡪Rebuild) this project and you will find that you have a file called ClassLibrary1.dll. You may want to look at the DLL’s property pages so that you know where it was written.

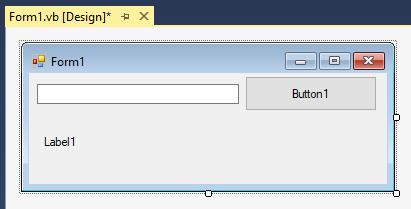
Next, create a new Windows Forms project to test out the DLL that we just created. Add a reference to the ClassLibrary1.dll file from within the Project🡪Add Reference menu option. Once the Reference Manager dialog appears, click on Browse in the left-hand pane and then press the lower Browse button. You’ll get an OpenFileDialog that you can use to navigate to where your ClassLibary1.dll file is located. Click on the ClassLibaray1.dll and select Add. The next screenshot shows this process:



Once you’ve clicked Add, you’ll be taken back to the Reference Manager screen. You should see that the ClassLibrary1.dll is in the list and is checked. Click the OK button to return to your form design surface:



Add the following to your application’s form: a TextBox for a word to be input, a Label to tell if the word was spelled correctly and a Command button that will carry out the spell checking. All controls use their default names. Here’s what my completed form looks like:



Here’s the code for the application:

'Chapter 19 - Program 3

Public Class Form1

'Make sure that you added a reference to our spell checking class

'DLL (ClassLibrary1) -- we will need to create a new instance of

'the class inside of this application

Dim aCheckRef As New ClassLibrary1.clsWordCheck

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'Whenever the user clicks on the command button, grab what has

'been typed into the textbox and pass it into the DLLs CheckWord

'method -- the method will return a boolean value telling whether

'or not the word was correctly spelled. Once we get this value

'back, update the label and let the user know.

If aCheckRef.CheckWord(TextBox1.Text) Then

Label1.Text = TextBox1.Text & " is correct"

Else

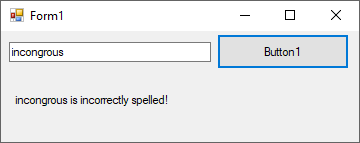
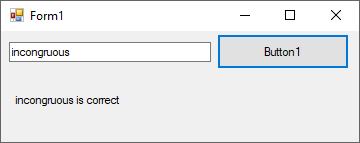
Label1.Text = TextBox1.Text & " is incorrectly spelled!"

End If

End Sub

End Class

Here’s a couple screenshots of the application at work:

Now you may have thought that we would need to add a reference to Microsoft Word in our application besides the reference to our ClassLibrary1. We didn't need to do that because our class library did that for us. We only call the class library, the class library in turn calls Word. Therefore, all that we need to worry about is that we can correctly call our own library.

Notice how we were completely able to encapsulate Word inside of the class library. If you handed someone a copy of your application, that user wouldn't have a clue that he/she was using Word to perform the spell checking. Of course, your user would find out rather quickly if Word wasn't loaded on the computer where your class library tried to use Word’s object model.

# Creating a Combination DLL

A combination DLL is one that has both a GUI that can be accessed as well as the class library object models that underlie it. This is very similar to what could be done with COM automation, but not quite the same thing (in fact I made up the term Combination DLL since I don't quite know what else to call them). In pre-.NET, Microsoft used to call this type of project an ActiveX EXE. Word and Excel are examples of what we could create in an ActiveX EXE – you could run them as standalone applications by clicking on their icons or you could access their object models through COM references.

For whatever reason, Microsoft did not migrate this type of project to .NET – so I will guide you along to creating something as close to an ActiveX EXE as possible. What we will do is rebuild our spell checker project as a DLL that contains both a GUI and a class library. We can then just call the piece that we want: either the GUI or the object model.

Let’s go ahead and start a new Windows Forms project up as the project type. I called my project Program 4 and I added a class to it. The class will be responsible for performing the spell checking using Word just like in the standalone DLL file version that we created earlier. Make sure that you add a reference (Project🡪Add Reference) to the Microsoft Word Object Library. Here's the code that is in the class clsWordChecker:

'Chapter 19 - Program 4

'This is our reference to the Word object model

Imports Microsoft.Office.Interop

'This reference allows us to essentially create a COM model

Imports System.Runtime.InteropServices

'This next statement says that we are exporting the object model

'of this class out so that it can be used as a COM reference.

'Notice that short of this one line, the class looks identical to

'what we built before

<ComVisible(True)>

Public Class clsWordChecker

'Create a Word instance for use in our class

Private aWordInst As Word.Application

'When the user instantiates our class, start up a copy of Word

Public Sub New()

aWordInst = New Word.Application()

End Sub

'The only method that our class will provide

Public Function CheckWord(ByVal strWord As String) As Boolean

CheckWord = aWordInst.CheckSpelling(strWord)

End Function

'Get rid of the Word instance when our class instance is

'destroyed

Protected Overrides Sub Finalize()

aWordInst = Nothing

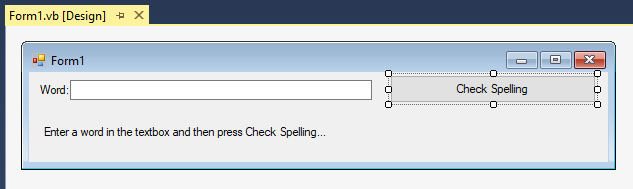
End Sub

End Class

You'll notice that I had to add what VB calls a decorator (in the < >) to indicate that I wanted this class to be exposed for COM use. Other than that, the code looks identical to what you wrote in the standalone spell checking DLL example.

Now that we have the object model part built, let's create a default form for our users to work with. If they don't want to use our object model, they can access this default form and it will make the appropriate calls to the object model to perform the spell checking. This is paramount to just running Word: you get the user interface and you don't have to worry at all about the object model that's running behind it.

Now go to the form in your project and add a TextBox (txtWordToCheck), Command button (cmdCheckSpelling) and Label (lblResults). This is the form where we will do spell checking if asked by our user. Remember, they will have the option of using the form that we provide, or they could design their own form and just call our spell checking class' object model that we exposed. Here is what my finished form looks like:



Here's the code that you need to place behind the form:

'Chapter 19 - Program 4

Imports System.ComponentModel

Public Class Form1

'Create an instance of our spell checking class. Remember that

'it is in the same project as this form so there we do not need

'to add it via Project-->Add Reference.

Dim aCheckRef As New clsWordChecker()

Private Sub cmdCheckSpelling\_Click(sender As Object, e As EventArgs)

Handles cmdCheckSpelling.Click

'Call our spell checking class' CheckWord method and then

'print the results.

If aCheckRef.CheckWord(txtWordToCheck.Text) Then

lblResults.Text = txtWordToCheck.Text & " is okay"

Else

lblResults.Text = txtWordToCheck.Text & " is incorrectly spelled!"

End If

End Sub

Private Sub Form1\_Closing(sender As Object, e As CancelEventArgs)

Handles Me.Closing

'If we find that the form is getting closed, get rid of our

'spell checking class instance.

aCheckRef = Nothing

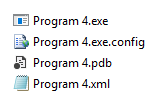
End Sub

End Class

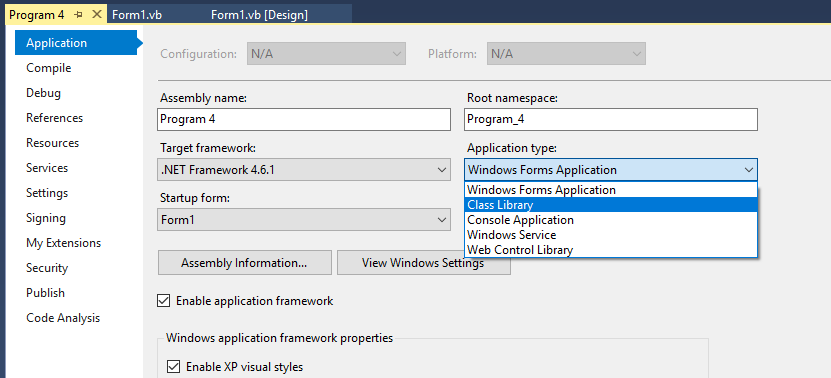
You really should be okay with the form's code since it's pretty much identical with what we did with the program that tested the spell checking DLL that we created in the last section. This time we are embedding both a GUI and the class library in the same project. If you go ahead and run your project, you will see that it starts executing and that you are presented with the default GUI form:



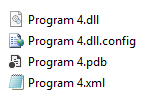
There's a bit of a problem now though. You'll notice that you were able to run this project. You cannot run a .DLL, which means that this isn't a .DLL yet. Furthermore, you can't include a reference to an executable because all references must be .DLL files. Go ahead and click on Build🡪Rebuild and then examine the directory where your files are written. You’ll see there is no .DLL file, only an .EXE:



What we need to do is to compile this project as a DLL. To do this, we need to change the default output object type. Right click on the project icon in the Solution Explorer pane and select Properties. From the Application tab, you will change the Application type to Class Library:

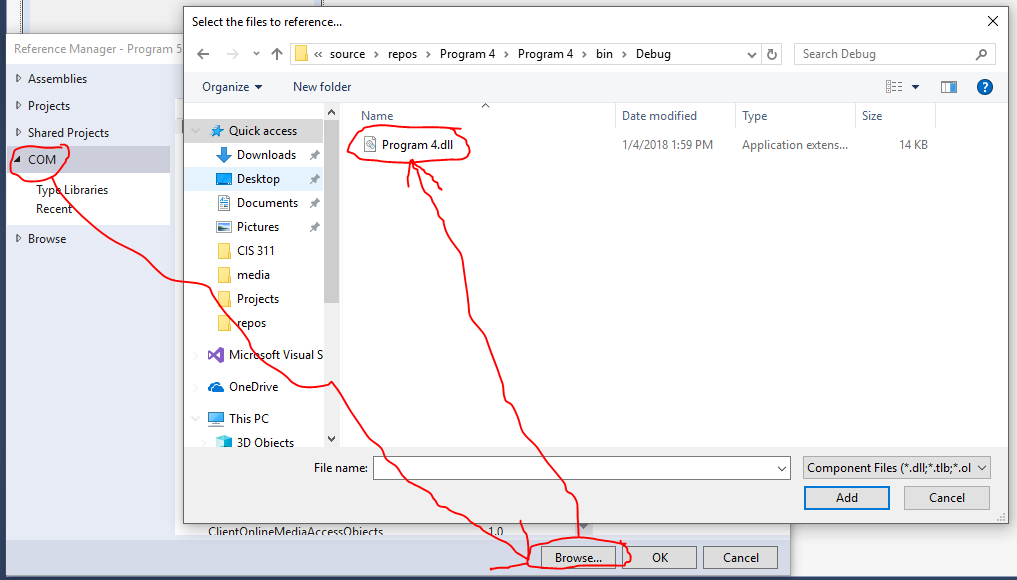


Once you’ve made the change, rebuild (Build🡪Rebuild menu option) the solution to generate the .DLL file. If we look at the directory where our project files are written, we now see that a .DLL is there:

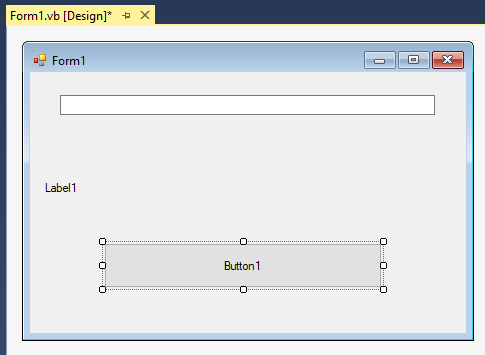


The good news is that we're done with this piece, so save everything and close the solution down. We have successfully generated a .DLL file that contains an object model and a GUI form in it. The bad news is that we can no longer simply run that .DLL file to get the default form to come up. This is where the combination DLL is different from pre.-NET ActiveX EXE projects.

Now start a new Forms application solution up. Using Project🡪Add Reference, add either a .NET reference or a COM reference to the .DLL that you created in the last step. You can select either type of reference since we specifically marked the class to be COM visible, making it accessible to either reference type. Here's a screenshot of selecting the file from the Reference Manager, in my case it was called Program 4.dll:



Next, create a quick and dirty form for doing spell checking from within your new application. We will be using both the default form in the DLL as well as our own to show that both pieces work. Here's a finished copy of my form. I just dragged a TextBox, Label and Command button to the form leaving the default names in place:



Now let's add the code that will run behind our form. Notice that there is nothing special in it:

'Chapter 19 - Program 5

Public Class Form1

'Create an instance of the spell checking class

Dim aCheckRef As New Program\_4.clsWordChecker()

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

If aCheckRef.CheckWord(TextBox1.Text) Then

Label1.Text = "OK"

Else

Label1.Text = "Not OK!"

End If

End Sub

End Class

We want to start the application up using the default form located in the DLL. Once the default form is closed, we will use our local application's form to show that we can also access the object model in the DLL. In order to do this, we will add some code to the Application Events Startup routine. If you don’t remember how to do this, right click on the project icon in the Solution Explorer pane and select Properties. Click on the View Application Events button to get to the code window.

Here's the code for this part of the application:

'Chapter 19 - Program 5

Imports Microsoft.VisualBasic.ApplicationServices

Namespace My

Partial Friend Class MyApplication

Private Sub MyApplication\_Startup(sender As Object, e As StartupEventArgs)

Handles Me.Startup

'We can access the default form in the DLL -- pretty cool, eh?

Dim myDLLForm As New Program\_4.Form1

'We can also use our own form.

Dim myFormFromThisApp As New Form1

'Show the DLL's form first

myDLLForm.ShowDialog()

'Once the DLL's form closes, then use our form and just

'the object model from the DLL

myFormFromThisApp.ShowDialog()

'Now remember as soon as this sub exits, our

'default form will show again since that's how things

'are normally supposed to operate

End Sub

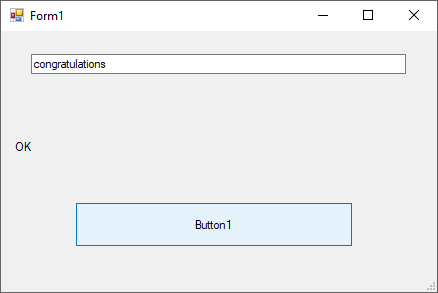
End Class

End Namespace

Now run the application – at first you will see the default GUI that is contained within the DLL:



After you close the DLL’s GUI form, you will see the local application’s form that we created. Remember that this form is just using the object model’s functionality from the class in the DLL:



This is some powerful programming! We have achieved a very high degree of reusability. You can now perform some industrial strength control and application development and make things as automation accessible as those found in Microsoft's Word and Excel’s applications.

You should now understand how to bundle functionality together into DLL code libraries that can be deployed and consumed by numerous applications – this is much of the glue that is expected behind Windows systems development techniques.