Programmer to Programmer ™

Beginning VB.NET 2nd Editon

Richard Blair, Jonathan Crossland, Matthew Reynolds, Thearon Willis

wrox
Programmer to Programmer

Free access to technical support, code downloads and peer discussion groups. Join at:

www.beginningdotnet.com

BEGINNERS' PORTAL

Summary of Contents

7

Building Windows Applications

When Microsoft first released Visual Basic, developers fell in love with it because it made building the user interface components of an application very simple. Instead of having to write thousands of lines of code to display windows – the very staple of a Windows application – developers could simply "draw" the window on the screen.

In Visual Basic, a window is known as a **form**. With .NET, this form design capability has been brought to all of the managed languages as **Windows Forms**. We've been using these forms over the course of the previous six chapters, but we haven't really given that much thought to them – focusing instead on the code that we've written inside them.

In this chapter, we'll look in detail at Windows Forms and show you how you can use Visual Basic .NET to put together fully featured Windows applications. In particular, we will look at:

- Adding features such as buttons, textboxes, and radio buttons
- Creating a simple toolbar and code buttons to respond to events
- Creating additional forms in a Windows Forms application
- Deployment of a Windows application

> **Note that, on occasion, you'll hear developers refer to Windows Forms as WinForms.**

Responding to Events

Building a user interface using Windows Forms is all about responding to **events** (such as `Clicks`), so programming for Windows is commonly known as **event-driven programming**. As you know, to build a form, we "paint" controls onto a blank window called the Designer using the mouse. Each of these controls is able to tell us when an event happens, for example if we run our program and click a button that's been painted onto a form, that button will say, "Hey, I've been clicked!" and give us an opportunity to run some code that we provide to respond to that event.

Button Events

A good way to illustrate the event "philosophy" is to wire up a button. An example would be the `Click` event that is **fired** whenever the button is clicked. We have more events than just the `Click` event, although, in day-to-day practice it's unlikely we'll use more than a handful of these.

Try It Out – Using Button Events

1. Start Visual Studio .NET and select File | New | Project from the menu. Create a Windows Application project called Hello World 2 and click OK.

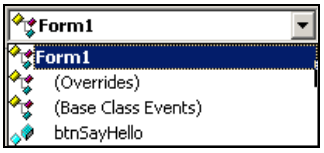
2. Scroll up the Properties window until you find the Text property for Form1. Change it to Hello, world! 2.0.

3. From the Toolbox, select the Button control, and then drag and drop a button onto the form. Change its Text property to Hello, world! and its Name property to btnSayHello:

4. Double-click on the button and add the following code:

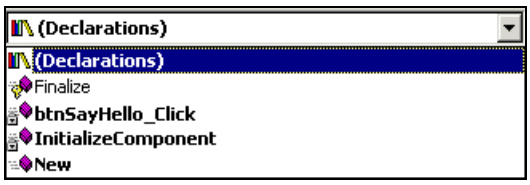
```
Private Sub btnSayHello_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnSayHello.Click

    ' say hello!
    MsgBox("Hello, world!")

End Sub
```

5. Drop down the left list at the top of the code window. You'll see this:



Notice that the last three items are indented slightly. This tells us that (Overrides), (Base Class Events), and btnSayHello are all related to Form1. `btnSayHello` is a **member** of `Form1`. As we add more members to the form, they will appear in this list.

6. Before we go on, let's take a quick look at the right list. Drop it down, and you'll see this:



The contents of the right list change depending on the item selected from the left list. The right list lets us navigate through items related to whatever we've selected in the left. In this case, its main job is to show us the methods and properties that we've added to the class.

The (Declarations) entry takes us to the top of the class where we can change the definition of the class and add member variables.

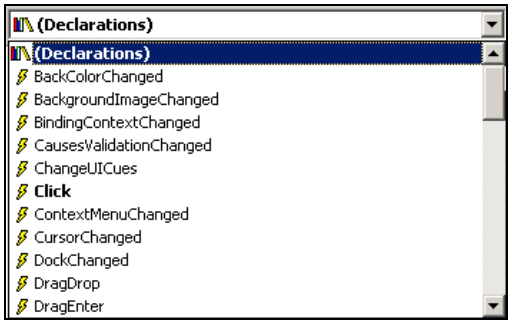
The Finalize option is quite interesting – whenever an entry in the right list appears in faint (as opposed to **bold**) text, the method doesn't exist. However, selecting the object will make Visual Basic .NET create a definition for the function we selected. In this case, if we select Finalize, Visual Basic .NET will create a new method called Finalize and add it to the class.

You'll notice that Visual Basic .NET adds a little icon to the left of everything it displays in these lists. These can tell you what the item in the list actually is. A small purple square represents a method, a small blue square represents a member, four-books stacked together represents a library, and three squares joined together with lines represents a class.

Visual Studio may also decorate these icons with other icons to indicate the way they are defined. For example, next to Finalize you'll see a small key, which tells us the method is **protected**. The padlock icon tells us the item is **private**.

It's not really important to memorize all of these now, but Visual Basic .NET is fairly consistent with its representations, so if you do learn them over time they will help you understand what's going on.

7. Select btnSayHello from the left list. Now, drop down the right list again:



Since the left list is set to btnSayHello, now the right list exclusively shows items related to that control. In this case, we've got a huge list of events. One of those events, Click, is shown in bold because we've provided a definition for this. If you select Click, you'll be taken to the method in `Form1` that provides an event handler for this method.

8. We'll now add another event handler to the button control. With btnSayHello still selected in the left list, select MouseEnter from the right list. A new event handler method will be created, and we need to add some code to it:

```
Private Sub btnSayHello_MouseEnter(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles btnSayHello.MouseEnter

    ' change the text...
    btnSayHello.Text = "The mouse is here!"

End Sub
```

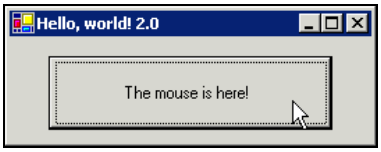
The `MouseEnter` event will be fired whenever the mouse pointer "enters" the control, in other words crosses its boundary.

9. To complete this exercise, we'll need another event handler. Select btnSayHello from the left list and select MouseLeave from the right list. Again, a new event will be created, so add this code:

```
Private Sub btnSayHello_MouseLeave(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles btnSayHello.MouseLeave

    ' change the text...
    btnSayHello.Text = "The mouse has gone!"

End Sub
```

The `MouseLeave` event will be fired whenever the mouse pointer moves back outside of the control.

10. Run the project. Move the mouse over and away from the control and you'll see the text change:



How It Works

Most of the controls that you use will have a dazzling array of events, although in day-to-day programming only a few of them will be consistently useful. For the button control, the most useful is usually the `Click` event.

Visual Basic .NET knows enough about the control to automatically create the default event handlers for us. This makes our life a lot easier and saves on typing!

When we created our `MouseEnter` event and added our custom code, here's what we had:

```
Private Sub btnSayHello_MouseEnter(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles btnSayHello.MouseEnter
```

```
    ' change the text...
    btnSayHello.Text = "The mouse is here!"

End Sub
```

You'll notice that at the end of the method definition is the `Handles` keyword. This ties the method definition into the `btnSayHello.MouseEnter` subroutine. When the button fires this event, our code will be executed.

Although previously we've only changed the button's `Text` property at design time using the Properties window, here you can see that we can change it at run time too.

> *As a quick reminder here, **design time** is the term used to define the period of time that you're actually writing the program, in other words, working with the Designer or adding code. **Run time** is the term used to define the period of time when the program is running.*

Likewise, the `MouseLeave` event works in a very similar way:

```
Private Sub btnSayHello_MouseLeave(ByVal sender As Object, _
         ByVal e As System.EventArgs) Handles btnSayHello.MouseLeave

    ' change the text...
    btnSayHello.Text = "The mouse has gone!"

End Sub
```

A Simple Application

.NET comes with a comprehensive set of controls that we can use in our projects. For the most part, we'll be able to build all of our applications using just these controls, but in Chapter 14 we look at how we can create our own.

For now, let's take a look at how we can use some of these controls to put together a basic application. In the following Try It Out we'll build a basic Windows application that lets the user enter a bunch of text into a form. We'll count the number of letters in the block of text that they enter, and the number of words.

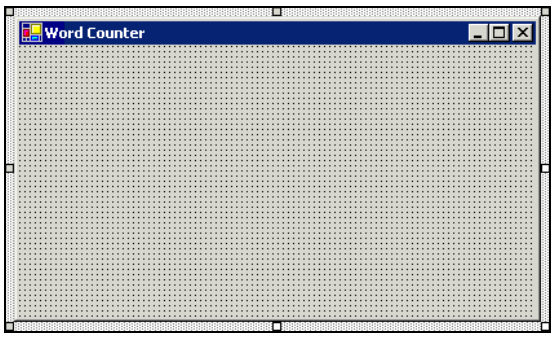
Building the Form

The first job is to start a new project and build a form, so let's do that now.

Try It Out – Building the Form

1. Select File | New | Project from the Visual Basic .NET menu and create a new Windows Application project. Enter the project name as Word Counter and click OK.

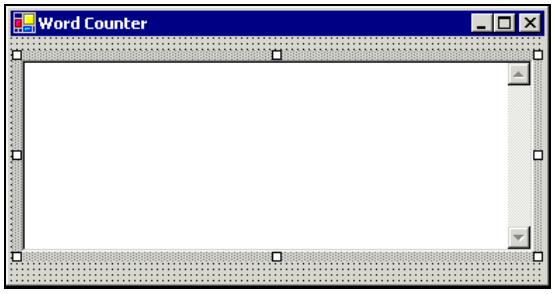
2. Stretch the form until it looks like this and use the Properties window to change the form's Text property to Word Counter:



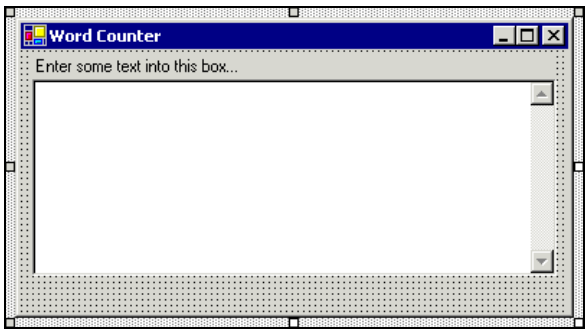
3. From the Toolbox, select the TextBox control and paint it onto the form. Now change the properties of the textbox as shown in the following table:

Property	Value
ScrollBars	Vertical
Text	*Leave blank*
Multiline	True
Name	txtWords

4. Note that when we changed the Multiline property, our six gray sizing handles around the control turned white. This means that we can now change the height of the textbox. Stretch it until it occupies a lot more of the form:



5. To tell the user what they should do with the form, we'll add a label. Select the Label control from the Toolbox, and drag and drop it just above the textbox. Change the Text property to Enter some text into this box…:



Strictly speaking, unless we're going to need to talk to the control from our Visual Basic .NET code, we don't need to change its Name property. With the textbox, we will need to use its properties and methods to make the application work. However, the label is just there for esthetics, so we don't need to change the name from Label1. (This depends on how fussy you are – some developers give every control a specific name, others only give controls that really need them a name.)

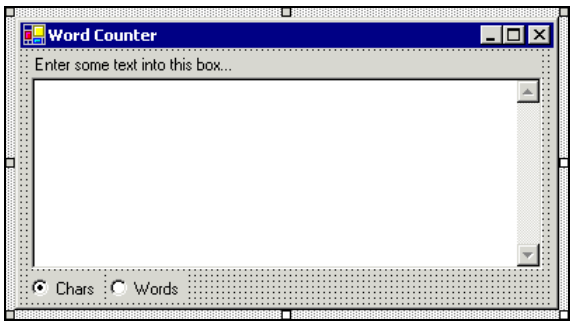
> **It's worth noting that if you are going to refer to a control from Visual Basic .NET code, it's bad practice *not* to give it a name, in other words you should never end up with a line like `Button1.Text`. Developers should be able to work out what the control represents based on its name even if they've never seen your code before.**

6. Our application is going to be capable of counting either the characters the user entered, or the number of words. In order to allow the user to select which they would prefer to use, we will use two radio buttons. Draw two RadioButton controls onto the form. We'll need to refer to the radio buttons from our Visual Basic .NET code, so change the following properties:

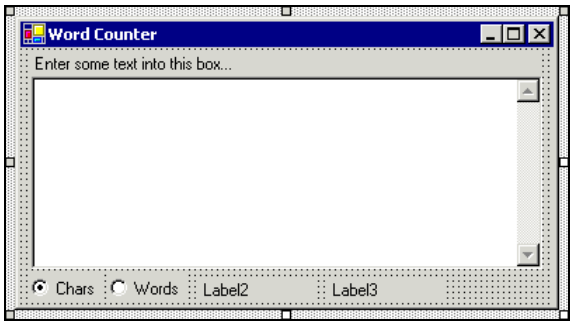
First radio button	
Property	**Value**
Name	radCountChars
Text	Chars
Checked	True

Second radio button	
Property	**Value**
Name	radCountWords
Text	Words

Your form should now look like this:

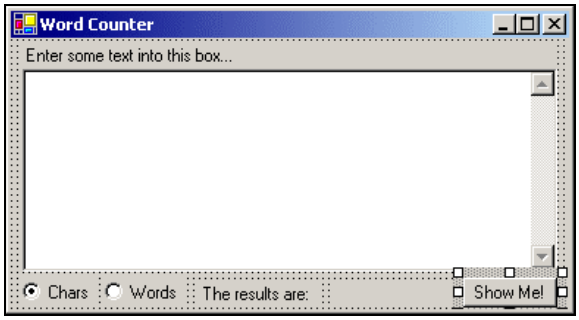


7. As the user types, we'll take the words that they enter and count up the words or characters as appropriate. We'll want to pass our results on to the user, so add two new Label controls like this:



8. The first Label control (marked Label2) is just for esthetics, so change its Text property to The results are:. The second Label control will report the results, so we need to give it a name. Delete the value for its Text property (in other words make it blank) and enter the Name property as lblResults.

9. We also need a Button control that will pop up a message box displaying the results, so add a button control to the form. We don't strictly need this because the user can read the results on the form, but it illustrates a couple of important points. Change the Text property to Show Me! and the Name property to btnShowMe:



10. Finally, now that we have our form exactly how we want it, let's keep it that way. Make sure you select one of the controls and not the actual form, and then select Format | Lock Controls from the menu. This will set the Locked property of each of the controls to True and prevent them from accidentally being moved, resized, or deleted.

Counting Characters

With our form designed, let's build some event handlers to count the number of characters in a block of text that the user types.

Try It Out – Counting Characters

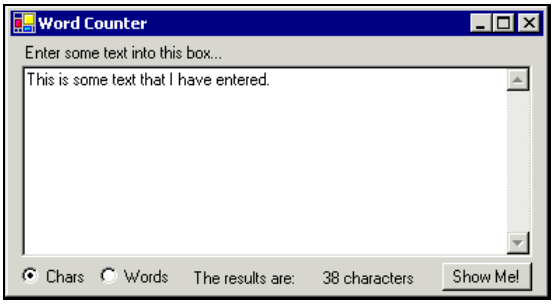
1. Since our application will be able to count both words and characters, we'll build separate methods for each. In this *Try It Out*, we'll write the code to count characters. Add this code to the bottom of `Form1` just before the `End Class` statement:

```
' CountCharacters - count the characters in a block of text...
Public Function CountCharacters(ByVal text As String) As Integer

    ' return the number of characters...
    Return text.Length

End Function
```

2. Now we need to build an event handler for the textbox. Double-click on the textbox control and Visual Studio .NET will create a default handler for `TextChanged`. Add this code to it:

```
Private Sub txtWords_TextChanged(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles txtWords.TextChanged

    ' count the number of characters...
    Dim numChars As Integer = CountCharacters(txtWords.Text)

    ' report the results...
    lblResults.Text = numChars & " characters"

End Sub
```

3. Run the project. Enter some text into the textbox and you'll see something like this:



How It Works

Notice that whenever you type a character into the textbox, the label at the bottom of the form reports the current number of characters. That's because the `TextChanged` event is fired whenever, not surprisingly, the user changes the text in the box. This will happen when new text is entered, changes are made to existing text, and when old text is deleted. We're "listening" for this event and whenever we "hear" it (or rather receive it), we call `CountCharacters` and pass in the block of text. As the user types text into `txtWords`, the `Text` property will be updated to reflect the text that has been entered. We can get the value for this property (in other words the block of text), and pass it to `CountCharacters`:

```
' count the number of characters...
Dim numChars As Integer = CountCharacters(txtWords.Text)
```

In return, it will pass back an integer representing the number of characters:

```
' return the number of characters...
Return text.Length
```

Once we have the number of characters, we need to update the `lblResults` control:

```
' report the results...
lblResults.Text = numChars & " characters"
```

Counting Words

Although on the surface, building a Visual Basic .NET application is actually very easy, building an elegant solution to a problem requires a combination of thought, experience, and a bit of luck.

Take our application – when the Words radio button is checked, we want to count the number of words, whereas when Chars is checked we want to count the number of characters. This has three implications. Firstly, when we respond to the `TextChanged` event we need to call a different method that counts the words, rather than our existing method for counting characters. This isn't too difficult. Secondly, whenever we select a different radio button, we need to change the text in the results from "characters" to "words" or back again. In a similar way, whenever the Show Me! button is pressed, we need to take the same result, but rather than displaying it in the label control, we need to use a message box.

Let's now add some more event handlers and, when we've finished we'll examine the logic behind the technique we've used.

Try It Out – Counting Words

1. The first thing we want to do is add a method after `CountCharacters` that will count the number of words in a block of text:

```
' CountCharacters - count the characters in a block of text...
Public Function CountCharacters(ByVal text As String) As Integer

    ' return the number of characters...
    Return Text.Length
```

```
End Function
```

```
 ' CountWords - count the number of words in a block of text...
Public Function CountWords(ByVal text As String) As Integer

    ' is the text box empty?
    If txtWords.Text = "" Then Return 0

    ' split...
    Dim words() As String = text.Split(" ".ToCharArray())
    Return words.Length

End Function
```

2. Now, add this method:

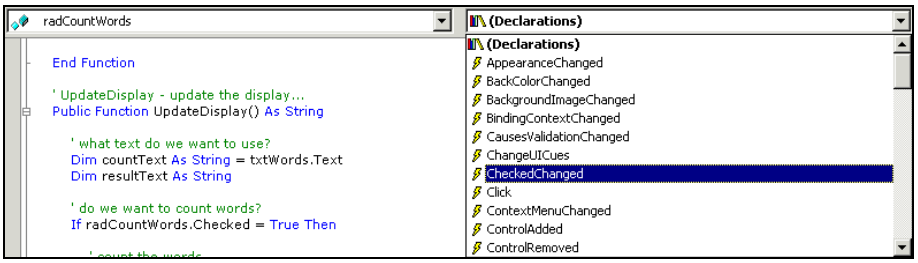
```
 ' UpdateDisplay - update the display...
Public Function UpdateDisplay() As String

    ' what text do we want to use?
    Dim countText As String = txtWords.Text
    Dim resultText As String

    ' do we want to count words?
    If radCountWords.Checked = True Then

        ' count the words...
        Dim numWords As Integer = CountWords(countText)

        ' return the text...
        resultText = numWords & " words"

    Else

        ' count the chars...
        Dim numChars As Integer = CountCharacters(countText)

        ' return the text...
        resultText = numChars & " characters"

    End If

    ' update the display...
    lblResults.Text = resultText

End Function
```

This method will deal with the hassle of getting the text from the textbox and updating the display. It also understands whether it's supposed to find the number of words or number of characters by looking at the `Checked` property on the `radCountWords` radio button.

3. Now, instead of calling `CountCharacters` from within our `TextChanged` handler, we want to call `UpdateDisplay`. Make this change:

```
Private Sub txtWords_TextChanged(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles txtWords.TextChanged

    ' something's changed... update the display...
    UpdateDisplay()

End Sub
```

4. Finally, we want the display to alter when we change the radio button from Chars to Words and vice versa. To add the `CheckedChanged` event, select radCountWords from the left drop-down at the top of the code window and CheckedChanged from the right one:



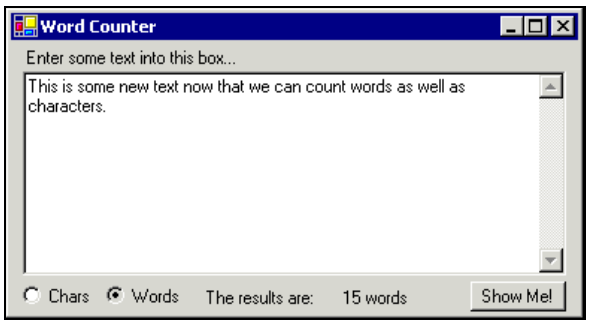
5. Add this code to the new event handler:

```
Private Sub radCountWords_CheckedChanged(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles radCountWords.CheckedChanged

        ' something's changed... update the display...
        UpdateDisplay()

End Sub
```

6. Repeat this step for `radCountChars`:

```
Private Sub radCountChars_CheckedChanged(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles radCountChars.CheckedChanged

        ' something's changed... update the display...
        UpdateDisplay()

End Sub
```

7. Run the project. Enter some text in the box and check Words. Notice how the display changes:



How It Works

Before we look at the technique we used to put the form together, we'll take a quick look at `CountWords`:

```
' CountWords - count the number of words in a block of text...
Public Function CountWords(ByVal text As String) As Integer

    ' is the text box empty?
    If txtWords.Text = "" Then Return 0

  ' split...
  Dim words() As String = text.Split(" ".ToCharArray())
  Return words.Length

End Function
```

We start by checking to see if the textbox is empty; if no text has been entered we immediately return a value of `0`.

The `Split` method of the `String` class is used to take a string and turn it into an array of string objects. Here, the parameter we've passed it is equivalent to the "space" character and so we're effectively telling `Split` to break up the string based on a space. This means that `Split` will return to us an array containing each of the words in the string. We return the length of this array, in other words the number of words, back to the caller.

> **Note that because this code uses a *single* space character to split the text into words, you'll get unexpected behavior if you separate your words with more than one space character or use the *Return* button to start a new line.**

One of our golden rules of programming is that we never write more code than we absolutely have to. In particular, when you find yourself in a position where you're going to write the same piece of code twice, try to find a way that means you only have to write it once. In our example, we have to change the value displayed in `lblResults` from two different places. The most sensible way to do this is to split off the code that updates the label into a separate function. We can then easily set up the `TextChanged` and `CheckedChanged` event handlers to call this method. The upshot of this is that we only have to write the tricky "get the text, find the results, and display them" routine once. This technique also creates code that is easier to change in the future and easier to debug when a problem is found.

You'll find as you build applications that this technique of breaking out the code for an event handler is something you'll do quite often.

The Show Me! Button

To finish off this exercise, we need to wire up the Show Me! button. All we're going to do with this button is display a message box containing the same text that's displayed on `lblResults`.

Try It Out – Wiring up the Show Me! Button

1. From the code window for `Form1`, create a `Click` event handler for the button by selecting btnShowMe from the left drop-down list and then selecting Click from the right one. Then add this code:

```
Private Sub btnShowMe_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles btnShowMe.Click

    ' display the text contained in the label...
    MessageBox.Show(lblResults.Text, "Word Counter")

End Sub
```

2. Run the project. If you type something into the textbox and click Show Me! the same value will be displayed in the message box as appears in the results label control.

How It Works

In this case, all we're doing is selecting the `Text` property from the label control and passing it to `MessageBox.Show`.

Complex Applications

Normal applications generally have a number of common elements. Among these are toolbars and status bars. Putting together an application that has these features is a fairly trivial task in Visual Basic .NET.

In this next section, we'll build an application that allows us to make changes to text entered into a textbox, such as changing its color, and making it all uppercase or lowercase.

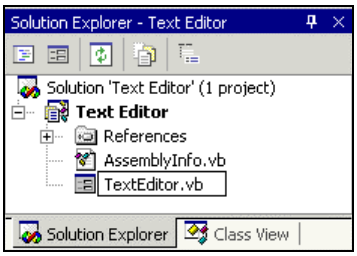
The Project

Our first step on the road to building our application is to create a new project.

Try It Out – Creating the Text Editor Project

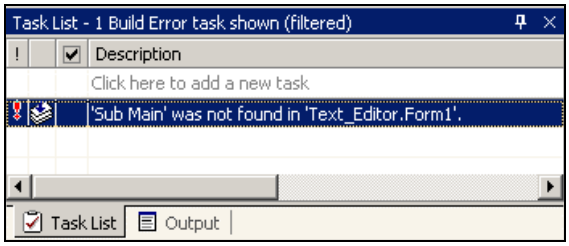
1. Create a new Windows Application project and call it Text Editor.

2. Most of the time, `Form1` isn't a very appropriate name for a form as it's not very descriptive. Right-click on the form in the Solution Explorer, select Rename, and change its name to TextEditor.vb. Then press *Enter* to save the changes:

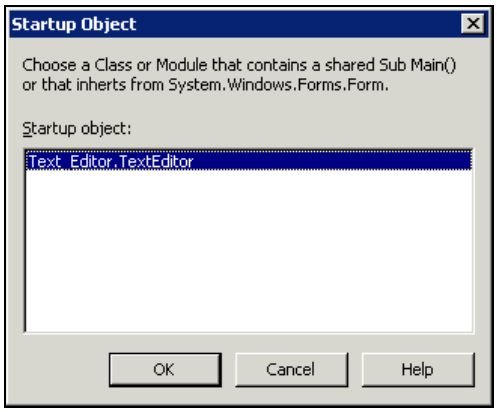


3. That's only half the battle – although we've renamed the form, we haven't actually renamed the class that contains the form's implementation. To do this, select the form in the Designer and change its Name property to TextEditor.

4. Select View | Other Windows | Task List from the menu to bring up the Task List window. It will display an error message saying that `Form1` could not be found, which makes sense because we just renamed it:



5. Double-click on the error message and you'll be prompted to select a new startup class. Select Text_Editor.TextEditor and click OK:



6. In the screenshots, I'm going to show the design window as quite small in order to save paper! You should explicitly set the size of the form by going to the Properties window of the form and setting the Size property to 600, 460.

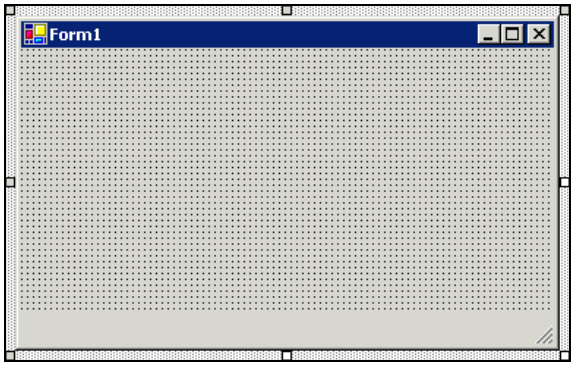
Now let's move on and start building the fun part of the application.

The Status Bar

Perhaps the most boring part of our project is the **status bar**, so we'll tackle this first. This is a panel that sits at the bottom of an application window and tells the user what's going on.

Try It Out – Adding a Status Bar

1. Open the Designer for the `TextEditor` form, and select the StatusBar control from the Toolbox. Draw it anywhere on the form. You will find that it automatically glues itself to the bottom edge of the form and you'll only be able to change the height portion of its Size property. Make the Height property something sensible (in other words a little larger than the text). Set its Name property to statusBar and delete the value for the Text property. You'll get something like this:



2. Open the code editor for the form and add the following code:

```
' StatusText - set the text on the status bar...
Public Property StatusText() As String
    Get
        Return statusBar.Text
    End Get
    Set(ByVal Value As String)
        statusBar.Text = Value
    End Set
End Property
```

There's no need to run the project at this point, so let's just talk about what we've done here.

How It Works

.NET has some neat features for making form design easier. One thing that was always laborious in previous versions of Visual Basic and Visual C++ was to create a form that would automatically adjust itself when the user changed its size.

In .NET, controls have the capability to **dock** themselves to edges of the form. By default, the status bar control sets itself to dock to the bottom of the form, but this can be changed to other areas of the form. So, when we resize the form, either at design time or at run time, the status bar stays where it's put.

So why have we built a `StatusText` property to get and set the text on the status bar? Well, this comes back to abstraction. Ideally, we want to make sure that anyone using this class doesn't have to worry about how we've implemented the status bar. We might want to replace the .NET-supplied status bar with another control, and if this happened anyone wanting to use our `TextEditor` class in their own applications (or a developer wanting to add more functionality to this application later on), would have to change their code to make sure it continued to work properly.

That's why we've defined this property as being `Public`. This means that anyone creating an instance of `TextEditor` in order to use its functionality in their own application can change the status bar if they want. If we didn't want them to be able to change the text themselves, relying instead on other methods and properties on the form to change the text on their behalf, we'd mark it as `Private`.

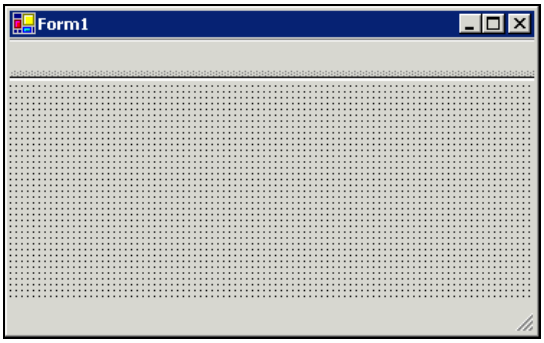
As we work through this example, we'll define things as `Public` and `Private`. From this you'll be able to infer what functionality a developer using our `TextEditor` class might have available.

The Toolbar

To make up for the fact that building the status bar was so dull, we'll add the toolbar in this next *Try It Out.*

Try It Out – Adding the Toolbar

1. Adding the toolbar is a similar deal to adding the status bar. Select the ToolBar control from the Toolbox and draw it anywhere on the form. It will automatically dock at the top of the form:



2. Note that the toolbar control doesn't display any sizing handles when it's selected. This is because the Framework controls the size and position of the toolbars itself, and doesn't expect you do to it! To see if it is selected, look in the Properties window to make sure that the list at the top does indeed refer to the toolbar. If it doesn't, click on the toolbar to make the selection.

3. Change the Name property of the toolbar to toolbar.

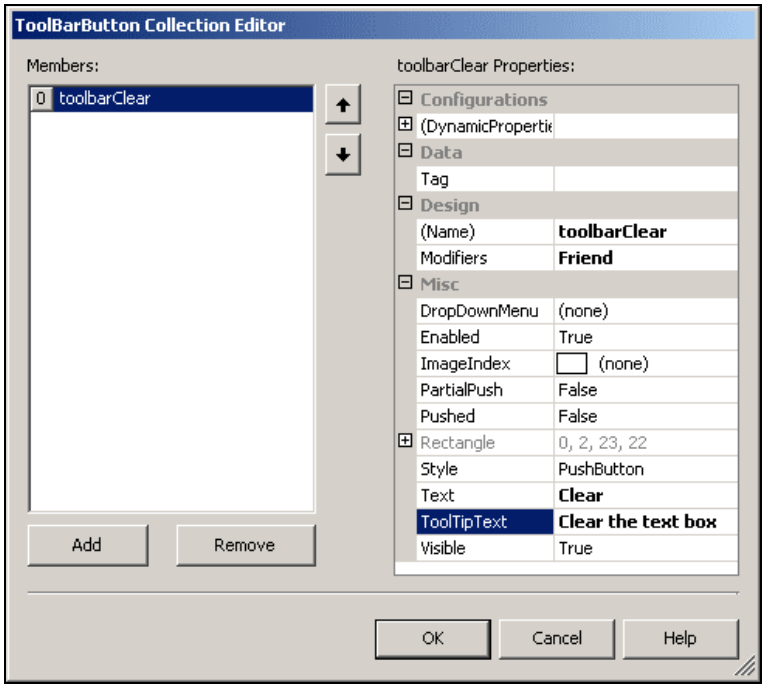
4. To add buttons to the toolbar, we'll use a built-in editor. Find the Buttons property, select it, and left-click on the ellipsis (…) to the right of (Collection).

5. We're going to add six buttons to the toolbar – Clear, Red, Blue, Uppercase, Lowercase, and About. Let's add the first one by clicking the Add button in the ToolBarButton Collection Editor.

6. The Collection Editor displays a properties palette much like the one that we're used to using. For each button, we need to change its text, give it an icon, give it a name, and provide some explanatory tooltip text. We'll add the icons in the next *Try It Out*, so just change these properties of ToolBarButton1:

Property	Value
Name	toolbarClear
Text	Clear
ToolTipText	Clear the text box

The Collection Editor should look like this:



7. Click the Add button again to add the Red button. Set the following properties:

Property	Value
Name	toolbarRed
Text	Red
ToolTipText	Make the text red

8. Click the Add button again to add the Blue button. Set these properties:

Property	Value
Name	toolbarBlue
Text	Blue
ToolTipText	Make the text blue

9. Click the Add button again to add the Uppercase button. Set these properties:

Property	Value
Name	toolbarUppercase
Text	Uppercase
ToolTipText	Make the text uppercase

10. Click the Add button again to add the Lowercase button. Set these properties:

Property	Value
Name	toolbarLowercase
Text	Lowercase
ToolTipText	Make the text lowercase

11. Now we want to add a **separator** to make a space between the Lowercase button and the About button. Click Add and change the Style property to Separator. You don't need to change the name of the control, unless you particularly want to.

12. Click the Add button one last time to add the About button. Set these properties:

Property	Value
Name	toolbarHelpAbout
Text	About
ToolTipText	Display the About box

13. Finally, click the OK button to save the toolbar.

How It Works

Much like the status bar control, the toolbar control also docks to a particular position on the form. In this case, it docks itself to the top edge.

The seven controls (six buttons and one separator) that we added to the toolbar actually appear as full members of the `TextEditor` class, but it's unlikely in this application that we'll need to access them directly. Later, we'll see how we can respond to the `Click` event on the toolbar itself to determine when the button has been pressed.

The `ToolTipText` property enables .NET to display a tooltip above the button whenever the user hovers their mouse over it. We don't need to worry about actually creating or showing a tooltip – .NET does this for us.

At the moment our toolbar is looking pretty boring, so we'll add some images.

Adding Images to the Toolbar

Now that we have added our buttons to the toolbar, we need to add some images to demonstrate the purpose of each button.

Try It Out – Finding Toolbar Pictures in the .NET Framework Samples

1. The first thing you need to do is find the `Microsoft Visual Studio.NET` folder. This will usually be under `c:\Program Files\`, but you may have to dig around to find it if you changed the default installation location.

2. In the `Common7\Graphics\bitmaps` folder beneath it, you'll find numerous pictures that we can use in our toolbar.

3. In my project I shall be using `NEW.BMP` for the Clear button, `DIAMOND.BMP` for the Red and Blue buttons (with the color changed for the Blue button), `BLD.BMP` for the Uppercase button, `JST.BMP` for Lowercase button, and `HELP.BMP` for the About button, but you can use whatever you like. `DIAMOND.BMP` has been renamed to `Red.bmp` and `Blue.bmp`.

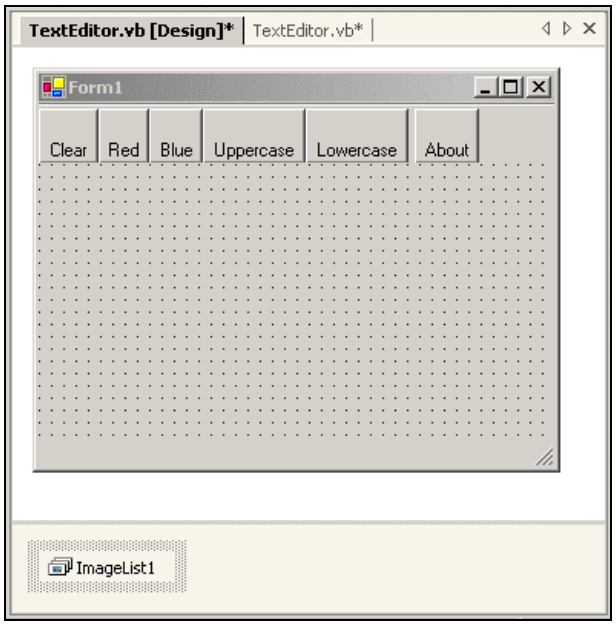
   Browse the folders for the figures you want, and copy them over to a new `Figures` folder in your `Text Editor` project folder.

Icons can be created using Visual Studio .NET by simply right-clicking on a project, and selecting Add | Add New Item. Choose Icon File from the Add New Item window, and give the icon a meaningful name. Select Open, and you will be presented with an icon editor that can be used to create your own icons.

Try It Out – Adding the Toolbar Pictures

1. Now we need to add the toolbar pictures to the project. The first step in doing this is to create an image list that we can add the pictures to. Open the Form Designer and locate the ImageList control in the Toolbox. Add it to your form.

2. You'll notice that the image list doesn't appear on the form, but appears in a new region at the bottom of the designer. ImageList controls don't have a user interface at run time, and this region is where controls that don't have a UI appear:

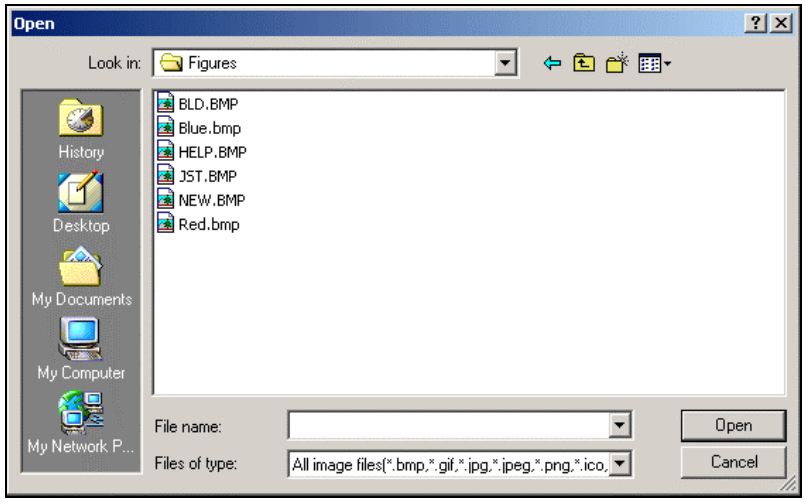


3. Change the Name property of the ImageList control to imgIstToolbar.

4. Select its Images property. Again, you'll find an ellipsis button (…) next to (Collection). Click it.

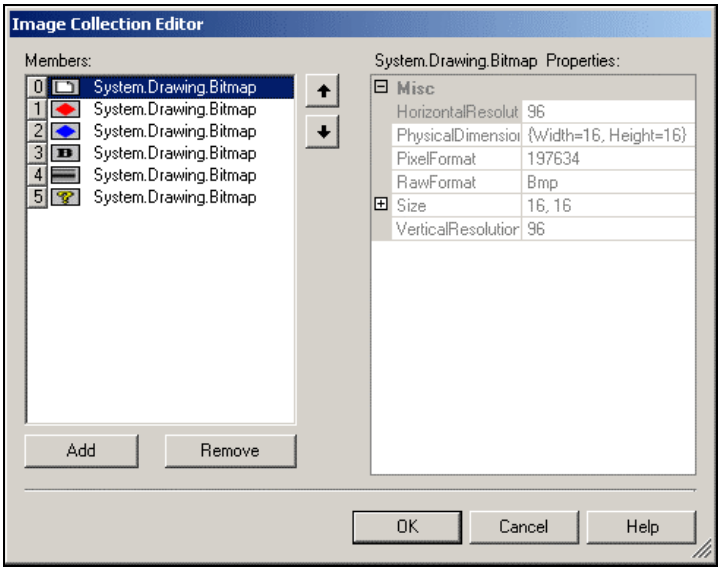
5. Another Collection Editor window appears, but this time we're adding new images to the image list. Click Add.

6. The File Open dialog will appear. Browse to the correct folder using the Look in box:



7. The folder should appear. Select NEW.BMP and click Open.

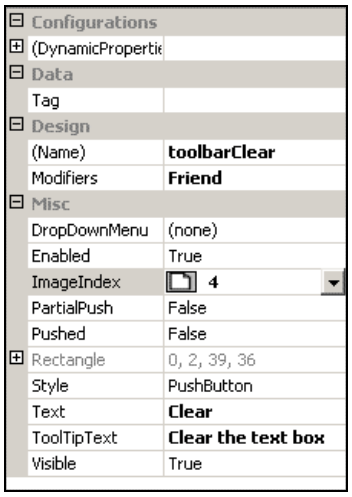
8. Go through the same process to add the images for the Red, Blue, Uppercase, Lowercase, and About buttons (in that order). Finally, click OK to dismiss the Collection Editor:



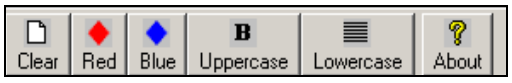
9. Select the Toolbar control and find its ImageList property. Select imgIstToolbar from the list. This action ties the ImageList control to the toolbar. We can now point each button to an image in the image list.

10. Find the Buttons property and click the ellipsis button to open the Collection Editor.

11. toolbarClear should be selected. Drop down the ImageIndex property and select 0, which corresponds to the icon we've chosen for the Clear button:



12. Repeat the process with the other buttons. When you've finished, you should have something like this:



Finally, we can start working on the code!

Creating an Edit Box

The first thing we want to do is create a textbox that can be used to edit the text entered.

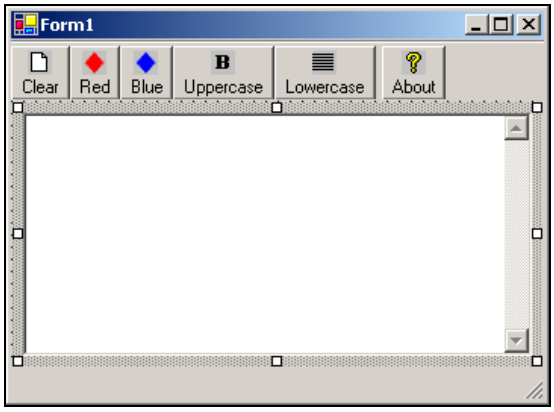
Try it Out – Creating an Edit Box

1. Open the Designer for `TextEditor` and draw a TextBox control into the center between the bottom of the toolbar and the top of the status bar.

2. Change these properties of the TextBox control:

Property	Value
ScrollBars	Vertical
Text	*Blank*
Multiline	True
Name	txtEdit

Your form should now look like this:



3. As we know, when the form changes size, the toolbar and status bar will stay "locked" in their position. To make sure that the textbox itself stretches with the form, set the Anchor property to Top, Bottom, Left, Right:

Clearing the Edit Box

In the following *Try It Out*, we're going to create a property called `EditText` that will get or set the text we're going to be editing. Then, clearing the edit box will simply be a matter of blanking out the `EditText` property.

Try It Out – Clearing txtEdit

1. Add this code to `TextEditor`:

```
' EditText - gets or sets the text that we're editing...
Public Property EditText() As String
    Get
        Return txtEdit.Text
    End Get
    Set(ByVal Value As String)
        txtEdit.Text = Value
    End Set
End Property
```

As previously when we created a property to abstract away the action of setting the status bar text, we've created this property to give developers using the `TextEditor` form the ability to get or set the text of the document irrespective of how we actually implement the editor.

2. We can now build `ClearEditBox`, the method that actually clears our textbox. Add the following code:

```
 ' ClearEditBox - empties txtEdit...
Public Sub ClearEditBox()
```

```
    ' reset the EditText property...
        EditText = ""

        ' reset the font color
        txtEdit.ForeColor = System.Drawing.Color.Black

        ' reset the status bar...
        StatusText = "Clear text box"

    End Sub
```

3. Now select txtEdit from the left drop-down list and TextChanged from the right list at the top of the code editor. Add this code:

```
Private Sub txtEdit_TextChanged(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles txtEdit.TextChanged

        ' reset the status bar...
        StatusText = "Ready"

    End Sub
```

How It Works

The first thing we want to do is clear out our textbox. In the next *Try It Out*, we'll see how we can call `ClearEditBox` from the toolbar.

```
' ClearEditBox - empties txtEdit...
Public Sub ClearEditBox()

    ' reset the EditText property...
    EditText = ""

    ' reset the font color
    txtEdit.ForeColor = System.Drawing.Color.Black

    ' reset the status bar...
    StatusText = "Clear text box"

End Sub
```

All this function does is set the `EditText` property to `""`, set the `ForeColor` property of the textbox (which is the color of the actual text) to black, and place the text Clear text box in the status bar.

As we mentioned, `EditText` abstracts the action of getting and setting the text in the box away from our actual implementation. This will make it easier for another developer down the line to use our `TextEditor` form class in their own application:

```
' EditText - gets or sets the text that we're editing...
Public Property EditText() As String
    Get
```

```
        Return txtEdit.Text
    End Get
    Set(ByVal Value As String)
        txtEdit.Text = Value
    End Set
End Property
```

As we type, the `TextChanged` event handler will be repeatedly called:

```
Private Sub txtEdit_TextChanged(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles txtEdit.TextChanged

    ' reset the status bar...
    StatusText = "Ready"

End Sub
```

Changing the status text at this point resets any message that might appear. For example, if the user has to type in a load of text and looks down to see Clear text box, they may be a little concerned. Setting it to Ready is a pretty standard way of saying "doing something", or "waiting". It doesn't mean anything specific.

Responding to Toolbars

The toolbar implementation in .NET is a little disappointing, since to make effective use of it we need to go through a number of hoops. When we look at building application menus in Chapter 9, you'll notice that menus are far easier to build.

Try It Out – Responding to Toolbar Clicks

1. In the code window, select toolbarClear from the left dropdown and then drop down the right list. You'll see this:

   If we follow the convention that we have been used to, it's pretty logical to assume that the toolbar button control itself is capable of firing an event like `Click` that we could respond to. Unfortunately, this isn't the case and we actually have to respond to a `ButtonClick` event on the `ToolBar` control itself.

2. Select toolbar from the left dropdown, and then select ButtonClick from the right one. You'll see something like this:

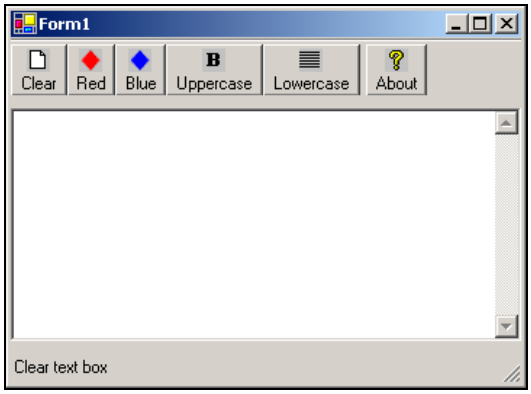
```
Private Sub toolbar_ButtonClick(ByVal sender As System.Object, _
        ByVal e As System.Windows.Forms.ToolBarButtonClickEventArgs) _
        Handles toolbar.ButtonClick

End Sub
```

The second parameter passed to this event handler is a `ToolBarButtonClickEventArgs` object. This object contains a property that refers to the actual button that was clicked. In order to find out what button actually was clicked, we need to try to match it to one of the other controls.

3. Add the following code:

```
Private Sub toolbar_ButtonClick(ByVal sender As System.Object, _
         ByVal e As System.Windows.Forms.ToolBarButtonClickEventArgs) _
         Handles toolbar.ButtonClick

    ' do we want to empty the text box?
    If e.Button Is toolbarClear Then
        ClearEditBox()
    End If

End Sub
```

4. Now run the project. Type some text into our edit box, and click on any of the buttons except Clear – nothing will happen. Now click the Clear button. The box will become blank and the status bar will inform you that it has been cleared:



How It Works

The trick here is to use the `Is` operator. This operator allows matching of objects, so when we say:

```
    If e.Button Is toolbarClear Then
```

…we're actually saying, "Is the button passed through from the `ToolbarButtonClickEventArgs` object the same as the reference we already have in `toolbarClear`?"

When you click any button other than the Clear button, this isn't true. However, when you click the Clear button it is true and so we call the `ClearEditBox` method.

Coding the Red Button

Now let's add the code that will make our Red button turn any text we enter red.

Try It Out – Coding the Red Button

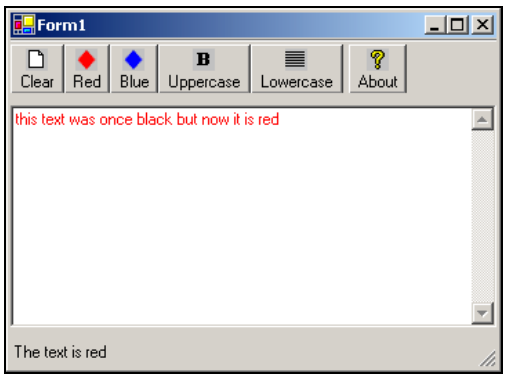
1. The first thing we have to do is create a function that will actually change the text red and update the status bar:

```
Public Sub RedText()

    ' make the text red...
    txtEdit.ForeColor = System.Drawing.Color.Red

    ' reset the status bar...
    StatusText = "The text is red"

End Sub
```

2. Next, change the `toolbar_ButtonClick` method to look like this:

```
Private Sub toolbar_ButtonClick(ByVal sender As Object, _
         ByVal e As System.Windows.Forms.ToolBarButtonClickEventArgs) _
         Handles toolbar.ButtonClick

    ' do we want to empty the text box?
    If e.Button Is toolbarClear Then
        ClearEditBox()
    End If

    ' do we want to make our text red?
    If e.Button Is toolbarRed Then
        RedText()
    End If

End Sub
```

3. Now run the project and enter some text. Click the Red button, and the text's color will change from black to red. Notice that if you carry on typing into the box, the new text will also be red:



4. Click on Clear, to remove the text, and revert the color of the text back to black.

How It Works

This *Try It Out* was really quite simple. All we did was call the `RedText` function from our Red button. `RedText` used `System.Drawing.Color.Red` to set the `ForeColor` property of our textbox to red:

```
' make the text red...
txtEdit.ForeColor = System.Drawing.Color.Red
```

The `ForeColor` remains red until we set it to something else – so clicking the Clear button turns it back to black.

Coding the Blue Button

Let's see how we implement the Blue button. This Try It Out will be almost identical to the previous one.

Try It Out – Coding the Blue Button

1. Add the following `BlueText` function to the `TextEditor` form:

```
Public Sub BlueText()

    ' make the text blue...
    txtEdit.ForeColor = System.Drawing.Color.Blue

    ' reset the status bar...
    StatusText = "The text is blue"

End Sub
```

2. Now add this `If...Then` statement to `toolbar_ButtonClick`:

```
' do we want to make our text red?
If e.Button Is toolbarRed Then
    RedText()
End If
```

```
    ' do we want to make our text blue?
        If e.Button Is toolbarBlue Then
            BlueText()
        End If

    End Sub
```

3. Run the project, and you'll see that the Blue button works in a similar fashion to the Red button, except it turns all our text blue.

Coding the Uppercase and Lowercase Buttons

The code for the Uppercase and Lowercase buttons is very similar, so let's look at them both now.

Try it Out – The Uppercase and Lowercase Buttons

1. Add the following functions to your `TextEditor` form:

```
Public Sub UppercaseText()

    ' make the text uppercase...
    EditText = EditText.ToUpper

    ' update the status bar...
    StatusText = "The text is all uppercase"

End Sub
```

```
Public Sub LowercaseText()

    ' make the text lowercase
    EditText = EditText.ToLower

    ' update the status bar...
    StatusText = "The text is all lowercase"

End Sub
```

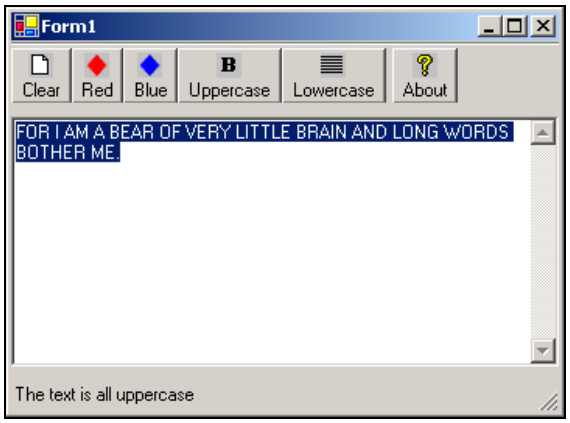
2. Now add this code to `toolbar_ButtonClick` to connect our `UppercaseText` and `LowercaseText` methods to their respective buttons:

```
    ' do we want to make our text blue?
    If e.Button Is toolbarBlue Then
        BlueText()
    End If
```

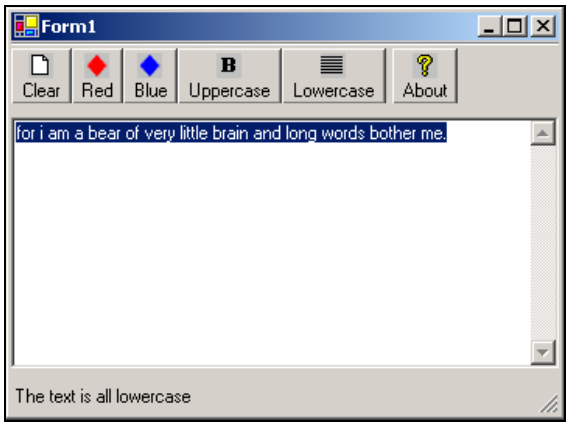
```
    ' do we want to make our text uppercase?
    If e.Button Is toolbarUppercase Then
        UppercaseText()
    End If
```

```
        ' do we want to make our text lowercase?
        If e.Button Is toolbarLowercase Then
            LowercaseText()
        End If

    End Sub
```

3. Run the project and enter some text into the box in a mixture of lowercase and uppercase. Then click the Uppercase button to make it all uppercase:



4. Click on the Lowercase button and all the text becomes lowercase:



How It Works

The code in this *Try It Out* is very simple; we have seen it all before. If the user clicks on the Uppercase button we call `UppercaseText`, which uses the `ToUpper` method to convert all the text held in `EditText` to uppercase text:

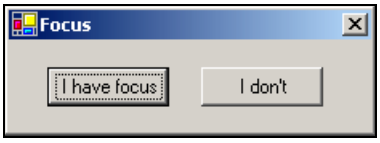
```
' make the text uppercase...
EditText = EditText.ToUpper
```

Likewise, if the user clicks on the Lowercase button we call `LowercaseText`, which uses the `ToLower` method to convert all the text held in `EditText` to lowercase text:

```
' make the text lowercase
EditText = EditText.ToLower
```

Focus

However, there's a problem with our `Text Editor` project. When we change the case using the Uppercase and Lowercase buttons, the entire text in the box is highlighted. This happens because the **focus** has been set to the textbox control. The control that has focus is the control that is currently selected. For example, if you have two buttons on a screen, the code in the event handler for the button that has focus will be executed if you press *Return*:



If there are a number of textboxes on a form, any text you type will be entered into the textbox that currently has the focus.

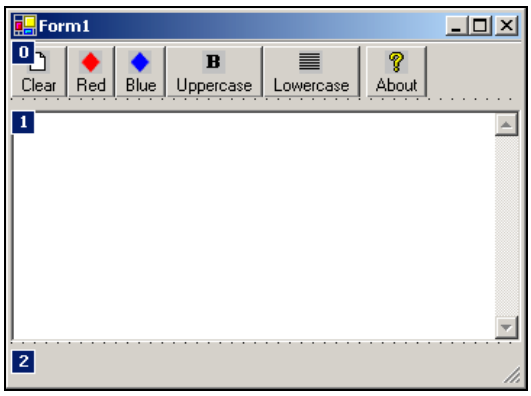
Tab Keys and the TabIndex Property

We can move focus between controls at run time by pressing the *Tab* key. For example, if the user of the previous form pressed the *Tab* key, focus would jump to the I don't button. If the user pressed the *Tab* key again, focus would jump back to I have focus.

The order in which the focus moves between the controls on a form is not arbitrary. As you place controls on a form they are assigned a value for their `TabIndex` property. The first control to be placed on the form has a `TabIndex` of `0`, the second `1`, the third `2`, and so on. This is the same order that the controls will have the focus as you *Tab* through them. If you have placed all your controls on the form, and are not happy with the resulting *Tab* order you can manually change it yourself, by using the Properties window to set the TabIndex properties of the controls.

> **Note that although labels have a `TabIndex` property it is not possible to *Tab* to them at run time. Instead, the focus moves to the next control that can receive it, such as a textbox or button.**

Visual Basic .NET has a very handy feature for displaying the tab order of your controls. Select View | Tab Order and your form will look something like this:



To remove the numbers, just select View | Tab Order once more.

> **It is possible to assign shortcut keys to labels in a similar way to that in which shortcut keys can be assigned to menu items (see Chapter 9). With keyboard shortcuts on labels, pressing the shortcut key doesn't move the focus to the label, but it moves focus to the edit field (or other user input field) with a `TabIndex` one greater than the label's `TabIndex`.**

Multiple Forms

All Windows applications have two types of windows – normal windows and dialog boxes. A normal window provides the main UI for an application. For example, if we use Word we use a normal window for editing our documents.

On occasion, the application will display a dialog box when we want to access a special feature. This type of window "hijacks" the application and forces us to use just that window. For example, when we select the Print option in Word, a dialog box appears, and from that point on we can't actually change the document – the only thing we can use is the Print dialog box itself. We call forms that do this **modal**.

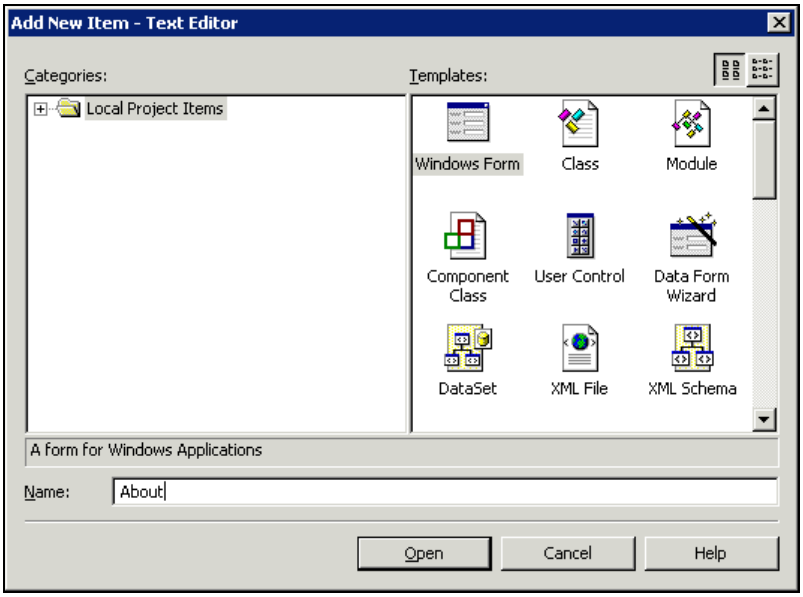
We'll talk about dialog boxes in more detail in Chapter 8. For now, we'll look at adding additional forms to our application. The form that we will add will be a simple modal form.

Help About

Most applications have an About box that describes the application's name and copyright information. As we already have a toolbar button for this feature, let's wire it in now.

Try It Out – Adding an About Box

1. To add a new form to the project, we need to use the Solution Explorer. Right-click on the Text Editor project and select Add | Add Windows Form. Enter the name of the form as About and click Open to create the new form:



2. When the form's Designer appears, set its Text property to About Wrox Text Editor.

3. Dialog boxes and About boxes traditionally appear centered over the application window. Look for the StartPosition property and change this to CenterParent. This will make the About box start in the correct place.

4. Another tradition about modal windows is that they usually cannot be sized – although thanks to the fact that .NET makes it far easier to create resizable forms this tradition is likely to end. However, we want to make sure that our About box cannot be resized, so find the FormBorderStyle property and change it to FixedDialog.

5. Strangely, although we've told the form that it's a FixedDialog, it still has minimize and maximize buttons. Find the MinimizeBox and MaximizeBox properties and change them both to False.

6. Now we need to add two labels, a LinkLabel control (found in the Toolbox), and a button to the form. Here's what we'll end up with:

The properties of these controls are listed in the following table:

Control	Name	Properties
Label	Label1	Text = Wrox Text Editor; Font (Bold, Size = 10)
Label	lblVersion	Text = Version
LinkLabel	lnkWrox	Text = http://www.wrox.com
Button	btnOK	Text = OK; DialogResult = OK

7. To make the OK button the default button, select the form and change its AcceptButton property to btnOK. This means that when the user is looking at the dialog, if they press the *Enter* key, the dialog will be dismissed. (In effect, the `Click` event on the button is simulated.)

8. Double-click on the form background to create a handler for the form's `Load` event. Add this code:

```
Private Sub About_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

    ' set the version number...
    lblVersion.Text &= " " & Environment.Version.ToString()

End Sub
```

9. Now, return to the form's Designer and double-click on the `LinkLabel` control. Add this code:

```
Private Sub lnkWrox_LinkClicked(ByVal sender As System.Object, _
        ByVal e As System.Windows.Forms.LinkLabelLinkClickedEventArgs) _
        Handles lnkWrox.LinkClicked

    ' run a Web browser and point it at wrox.com...
    System.Diagnostics.Process.Start("http://www.wrox.com/")

End Sub
```

10. We need to write a function that will display the About box, so add this to the `TextEditor` form:

```
' ShowAboutBox - display the about box...
Public Sub ShowAboutBox()
    Dim aboutBox As New About()
    aboutBox.ShowDialog(Me)
End Sub
```

11. Finally, we need to call `ShowAboutBox` from `toolbar_ButtonClick`:

```
    ' do we want to make our text lowercase?
    If e.Button Is toolbarLowercase Then
        LowercaseText()
    End If
```

```
        ' do we want to display the about box?
        If e.Button Is toolbarHelpAbout Then
            ShowAboutBox()
        End If

    End Sub
```

12. Run the project and click on the About button. You should see this dialog and if you click the link Internet Explorer will start up and you'll be taken to http://www.wrox.com/:

How It Works

Each form in our application has to have its own class, so to create a new form we need to get Visual Basic .NET to create a new class. We created a new class called `About`.

When the form starts, it will fire the `Load` event. We take this opportunity to write the version number to the `lblVersion` control:

```
Private Sub About_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

    ' set the version number...
    lblVersion.Text &= " " & Environment.Version.ToString()

End Sub
```

The shared property `Environment.Version` provides access to the version number of the current application. **Version numbers** in .NET have four components. The major and minor version numbers in this case are 1 and 0, and can be set by the developer. The next two numbers (2914 and 11 in the screenshot above) are the build and revision numbers, and are generated by Visual Basic .NET automatically as we compile. The `ToString` method can be used to combine all four components of the version number into a string with the format: `Major.Minor.Revision.Build`.

The `LinkLabel` control is a general-purpose control that can be used in our applications for a variety of different reasons. In this case, we're using it to visit a URL. This is done through the shared `System.Diagnostics.Process.Start` method. What this does is ask Windows to run the supplied "program". Thanks to the way that web browsers integrate with Windows, when we try to run a URL as we are doing here, the default Web browser will open the URL for us:

```
Private Sub lnkWrox_LinkClicked(ByVal sender As System.Object, _
        ByVal e As System.Windows.Forms.LinkLabelLinkClickedEventArgs) _
        Handles lnkWrox.LinkClicked
```

```
        ' run a Web browser and point it at wrox.com...
        System.Diagnostics.Process.Start("http://www.wrox.com/")

    End Sub
```

To display another window, you have to create an instance of it. That's exactly what we've done in the `ShowAboutBox` method. Once we have an instance, we have to use the `ShowDialog` method to show it modally. We need to pass in a reference to the owner form (in this case, the `TextEditor` form):

```
' ShowAboutBox - display the about box...
Public Sub ShowAboutBox()
    Dim aboutBox As New About()
    aboutBox.ShowDialog(Me)
End Sub
```

To dismiss the dialog box, we rely on functionality built into the button control. When we drew the button onto the About form, we set its `DialogResult` property to `OK`. If the button is on a modal dialog this tells it that when it's pressed that modal dialog should be closed and the value in `DialogResult` (in this case, `OK`) should be passed back to the caller.

That brings us to the end of our discussion of the Wrox Text Editor application. Hopefully you should now see how easy it is to put together a Visual Basic .NET application that has all of the usual features you'd expect from a Windows application. Now lets look at how to deploy it.

Deploying the Wrox Text Editor

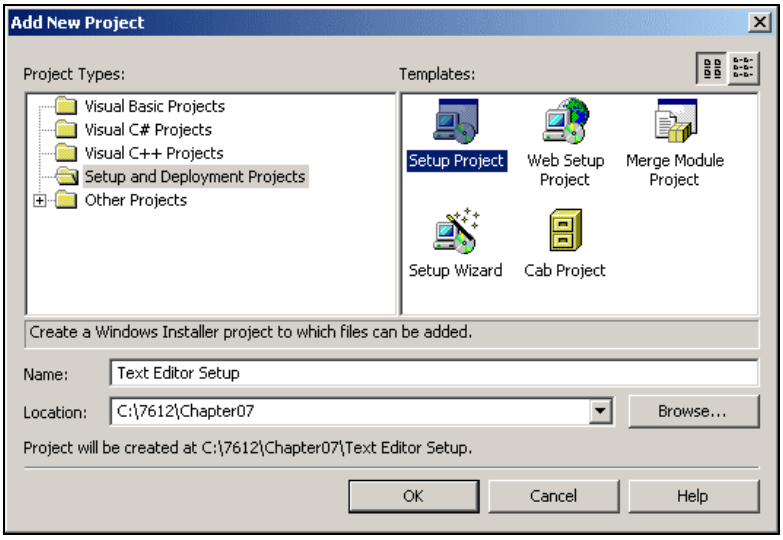
Our Wrox Text Editor works great on our development machine, but how do we get it to the end user? This is where deployment comes into play. For the deployment of Windows applications, Visual Studio .NET provides us with access to the Windows Installer, which provides the following functionality:

- Allows the user to select features they wish to install at the time of installation
- Allows applications to be completely uninstalled
- Allows the repair of files, should a file become corrupted
- Allows files to be copied to the destination machine, adding registry entries and creating desktop shortcuts
- If during an install a component fails to install, a rollback will occur and the system will be left in the state it started in

In the next *Try It Out* we will create a basic Setup Project that can be used to install our Wrox Text Editor to other machines in a professional manner.

Try It Out – Creating an Installer

1. We need to start by adding a new project to our solution. Right-click on Solution 'Text Editor' in the Solution Explorer and select Add | Add New Project. Then click on Setup and Deployment Projects to show the list of project templates.

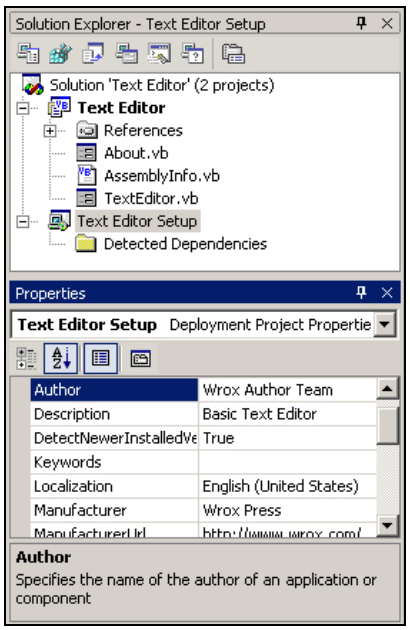


Select Setup Project, rename the project Text Editor Setup, and click the OK button.

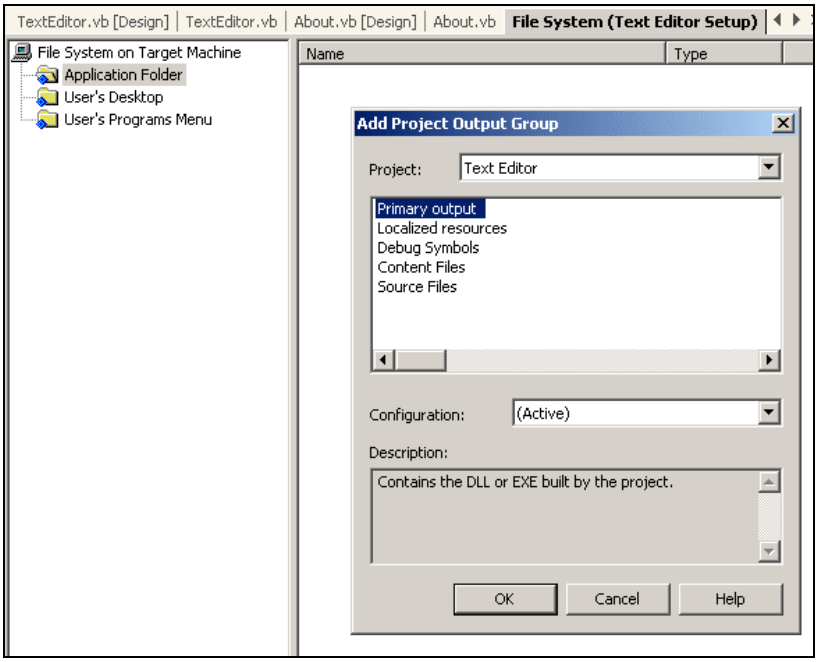
Only Setup Project is available in Visual Basic Standard Edition, and this is the method we will use to create an Installer.

2. Click on the Text Editor Setup project in the Solution Explorer, then change the following information in the Properties window:

Property	Value
Author	Wrox Author Team
Description	Basic Text Editor
Manufacturer	Wrox Press
ManufacturerURL	http://www.wrox.com/
ProductName	Wrox Text Editor
Title	Wrox Text Editor



3. Click on the Text Editor Setup project in the Solution Explorer again. In the File System Editor select the Application Folder. Now go to the main menu bar and select the Action | Add | Project Output.



4. Ensure that Text Editor is displayed in the Project dropdown box. Next, select Primary output from the listbox and finally select (Active) for the configuration. Click the OK button to confirm these actions.

5. In the Solution Explorer right-click on Text Editor Setup and choose Build.

How It Works

Visual Studio makes it very easy for us to build an installation program that will help bundle our application for deployment. The first thing we did was add a new setup project to our solution. Several different types of setup templates are available:

- Setup Project – This was the template chosen, and is designed to create Installer Packages for Windows applications, hence is best suited for the task at hand.
- Web Setup Project – This template provides a convenient way to deploy a web site in a simple install package. It will not only encapsulate the web pages themselves, but also handles any issues with registration and configuration of the web site automatically. This is often used to allow an entire web site to be downloaded or for deployment to multiple web servers. We will not use this project in this chapter because we are deploying a Windows application.
- Merge Module Project – A Merge Module project creates an installer file for components that are to be used in multiple applications. This Merge Module can then be included in the Install Packages of each of the applications.
- Cab Project – used to create cabinet files. Cab files are used to compress single or multiple files together into a small distributable package. They are often used to distribute ActiveX controls from a web server to a web browser.
- Setup Wizard – This wizard provides a step-by-step walkthrough of the templates described above.

After choosing a Setup Project we changed various properties of the project relevant to our application, such as the name and author. These properties will be displayed when the text editor is being installed.

Next we told the installer that we wanted to add an application to the installer package. Here we chose several options:

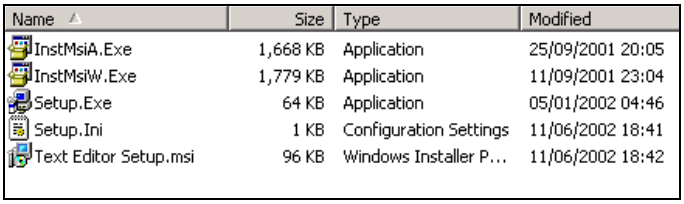
- From the drop-down menu we chose which application we actually wanted to install – in this case it was the Text Editor.
- Next we chose what we want to install; in our case we only want the Primary output files. Other options include choosing debug symbols used for debugging, resources, the actual source code, or a help file for the user who is installing the application on the target machine.
- The default configuration (Active) will usually be sufficient. The other choices are Debug or Release. With Debug selected, the application is compiled along with additional information for the debugger to operate correctly. We use the Release configuration to install our applications on client's machines, as this doesn't include the debugger information, hence the application file is smaller and executes more quickly.
- Finally we built the Installation package and are ready to deploy it to our users.

Deploying the Installer

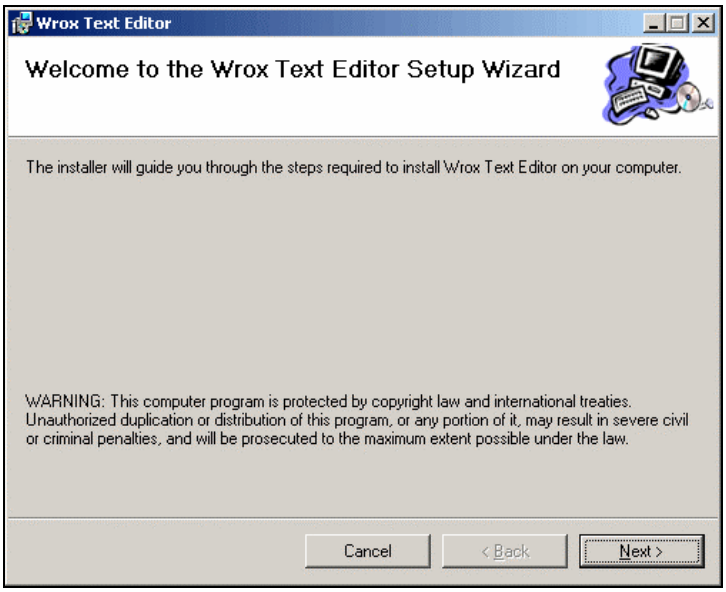
The Installation Package is treated like any other project in Visual Studio .NET. On the first screen when we chose the template we also selected where we wanted the project to exist. If we look in the `Debug` directory (this is my current configuration) it will contain all of the files needed to deploy the Wrox Text Editor. These files could be deployed in many ways:

- Burning them on a CD-ROM
- Placing them on a shared network
- Compressing them and emailing them
- Through a distribution product like SMS

The main point here is that the client machine needs access to all of the files in the directory (`Debug` or `Release`) in order to actually do the install.



All that we need to deploy the solution are these files and a target computer with the .NET Framework installed. Double-clicking on `Setup.exe` will start the install process, and you will be greeted with the following screen.



The user simply follows the on-screen instructions to install the Wrox Text Editor.

We have covered the basic functionality here, but we have really only scratched the surface of how to deploy a Windows application. There are many other capabilities and options available within Visual Studio .NET that you can use to customize the way in which your application is deployed, to suit your needs and preferences.

Summary

In this chapter, we have discussed some of the more advanced features of Windows forms and the commonly used controls. We discussed the event-driven nature of Windows and looked at three events that can happen to a button (namely `Click`, `MouseEnter`, and `MouseLeave`).

We created a simple application that allowed us to enter some text and then choose between counting the number of characters or the number of words by using radio buttons.

We then turned our attention to building a more complex application that allowed us to edit text by changing its color or its case. This application showed how easy it was to build toolbars and status bars. We even added an About box to display the version number and a link to the Wrox web site.

Finally we built an Installation Package that makes our application look professional, and easy to deploy.

Questions

1. What event is fired when the mouse pointer crosses over a button's boundary to hover over it? What event is fired as it moves the other direction, away from the button?
2. How can we prevent our controls from being accidentally deleted or resized?
3. What should we consider when choosing names for controls?
4. What's special about the toolbar and status bar controls?
5. How can you add a separator to your toolbar?