Chapter 13 – Advanced Form/Control Issues

# Objectives:

* Handling events with the mouse
* Changing the shape of the cursor
* Handling events with the keyboard
* Miscellaneous form and control odds and ends
* Creating context menus
* Cloning controls (control arrays)
* Drag and drop
* Making the MessageBox look pretty
* Containers

This chapter will concentrate on some of the more advanced interactions that you can have with forms. So essentially, we’re picking up where the last chapter left off. We will begin by taking a closer look at form input, specifically at the various events that the mouse and keyboard have responses for. From this discussion, you will learn how to fine tune on such things as what keys on the mouse were held down when the user clicked the mouse.

We will also learn how to create pop-up menus (also known as context menus), which are activated when the user right clicks the mouse on a control/form that has a context menu bound to it. You've undoubtedly encountered these types of menus in many Windows applications.

The other major topic of importance for us in this chapter is learning how to create drag and drop applications in which the user can move items within an application or between applications using the mouse. Let's begin by looking at VB's mouse events.

## 

# Handling Events with the Mouse

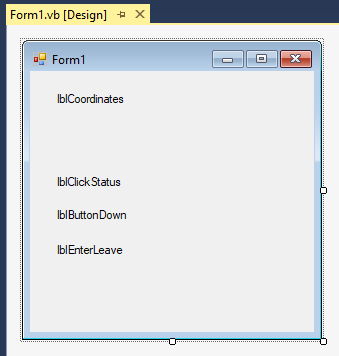
VB provides event handlers for eight different mouse events that you can programmatically respond to:

|  |  |
| --- | --- |
| *Event* | *Triggered When* |
| Click | When a mouse button is clicked once |
| DoubleClick | When a mouse button is double clicked |
| MouseCaptureChanged | When mouse input on the current application is taken away or brought back – think about holding down a mouse button in application one and then task switching to application two |
| MouseDown | When a mouse button is held down |
| MouseHover | When a mouse sits over a control/form |
| MouseUp | When a mouse button is released |
| MouseMove | When the mouse is moved |
| MouseEnter | When the mouse enters the area of a control/form |
| MouseLeave | When the mouse leaves the area of a control/form |
| MouseWheel | When the mouse wheel is rotated up or down a number of detents |

It's obvious from the above table what the purposes of the various events are for. We almost implicitly know and expect that these events should exist since they are fundamental in the operation of a mouse.

Whenever one of the mouse events is generated, a variable named e of type MouseEventArgs is passed through to the event handler. The MouseEventArgs is a class that contains the X & Y coordinates of the mouse's position on the screen along with the status of what mouse buttons were pressed when the event fired, or how many detents the mouse wheel was rotated.

Let's write a VB application that notifies you of various mouse events as they occur. Create a form with four labels on it. The upper label will show the coordinates of the mouse as it is moved around. The next label with notify you of any clicking; the third label will notify you of the mouse buttons that were held down when any click occurred; and the last label will notify you when the mouse enters/leaves the form. Here is a screenshot of the form:



Here's our code for the mouse event testing application:

Public Class Form1

'Chapter 13 - Program 1

Private Sub Form1\_MouseEnter(sender As Object, e As EventArgs)

Handles Me.MouseEnter

'When the mouse enters the form, post a notice. This happens

'both when the mouse moves from outside the app form back

'into the form and also when the mouse moves off of a control

'on the form back to the form canvas.

If sender.Text = "Form1" Then

lblEnterLeave.Text = "Mouse entered form"

End If

End Sub

Private Sub Form1\_MouseHover(sender As Object, e As EventArgs)

Handles Me.MouseHover

'These event fires whenever the mouse sits on top of a control

'for a specified amount of time

lblClickStatus.Text = "Mouse is hovering over some control"

End Sub

Private Sub Form1\_MouseLeave(sender As Object, e As EventArgs)

Handles Me.MouseLeave

'Fires whenever the mouse leaves this form to another form or app,

'or when a control is entered

If sender.Text = "Form1" Then

lblEnterLeave.Text = "Mouse left form"

End If

End Sub

Private Sub Form1\_MouseWheel(sender As Object, e As MouseEventArgs)

Handles Me.MouseWheel

'This will fire every time the mouse wheel is moved up or down

lblButtonDown.Text = "Mouse wheel rolled " & e.Delta & " detents"

End Sub

Private Sub Form1\_MouseClick(sender As Object, e As MouseEventArgs)

Handles Me.MouseClick

'Any time that the form gets clicked on, update the

'Clickstatus label

lblClickStatus.Text = "Clicked on Form"

End Sub

Private Sub Form1\_MouseDoubleClick(sender As Object, e As MouseEventArgs)

Handles Me.MouseDoubleClick

'Any time that the form gets double clicked on, update the

'Clickstatus label

lblClickStatus.Text = "Double clicked on Form"

End Sub

Private Sub Form1\_MouseDown(sender As Object, e As MouseEventArgs)

Handles Me.MouseDown

'Call this routine any time the mouse button is held down.

'We can figure out what button(s) were pressed when this event

'fired by examining the variable e's Button property

'Clear out the ButtonDown label

lblButtonDown.Text = ""

'Was the Left button pressed?

If e.Button = MouseButtons.Left Then

lblButtonDown.Text &= " Left"

Else

'Was the right button pressed?

If e.Button = MouseButtons.Right Then

lblButtonDown.Text &= " Right"

Else

'Was the middle button pressed?

If e.Button = MouseButtons.Middle Then

lblButtonDown.Text &= " Middle"

End If

End If

End If

End Sub

Private Sub Form1\_MouseUp(sender As Object, e As MouseEventArgs)

Handles Me.MouseUp

'As soon as the user lets go of the mouse button(s),

'clear out the Button label

lblButtonDown.Text = ""

End Sub

Private Sub Form1\_MouseMove(sender As Object, e As MouseEventArgs)

Handles Me.MouseMove

'Any time the mouse is moved, this event will be called.

'We can get the mouse's location by looking at variable e's

'x and y properties -- in this case we will display them in

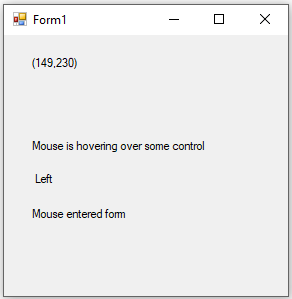
'the Coordinates label

lblCoordinates.Text = "(" & CStr(e.X) & "," & CStr(e.Y) & ")"

End Sub

End Class

Here’s the application when it’s running:



In a similar manner to checking the MouseEventArgs variable for which mouse buttons were pressed down at the time a click event was generated, you can easily determine if any special keyboard keys were held down. We can check for this whenever the mouse was moved, the mouse’s buttons were pressed down or let up through the global shared property named Control.ModifierKeys. Here's a code snippet showing how to do this by adding a bit of logic to the MouseDown event:

Private Sub Form1\_MouseDown(sender As Object, e As MouseEventArgs)

Handles Me.MouseDown

'Call this routine any time the mouse button is held down.

'We can figure out what button(s) were pressed when this event

'fired by examining the variable e's Button property

'Clear out the ButtonDown label

lblButtonDown.Text = ""

'Was the Left button pressed?

If e.Button = MouseButtons.Left Then

lblButtonDown.Text &= " Left"

Else

'Was the right button pressed?

If e.Button = MouseButtons.Right Then

lblButtonDown.Text &= " Right"

Else

'Was the middle button pressed?

If e.Button = MouseButtons.Middle Then

lblButtonDown.Text &= " Middle"

End If

End If

End If

'New code to check for special keyboard keys

If Control.ModifierKeys = Keys.Shift Then

MessageBox.Show("Shift held")

End If

If Control.ModifierKeys = Keys.Control Then

MessageBox.Show("Ctrl held")

End If

If Control.ModifierKeys = Keys.Alt Then

MessageBox.Show("Alt held")

End If

End Sub

Just to make sure that you know, when you click the mouse on a control, four events are raised in this order: MouseDown, Click, MouseClick, MouseUp. In an even larger event context, we would see these events trigger: MouseEnter, MouseMove, MouseDown, Click, MouseClick, MouseUp, MouseLeave. What this means to you is that you have to be careful what and where you chain into events or you could create a very non-responsive mess if you try to react to everything regarding mouse events!

# Changing the Shape of the Mouse

While we are on the subject of the mouse, let's talk about changing the mouse cursor's shape. Most applications will automatically show the busy cursor when an application is performing something that takes some time, the pointer cursor when a user can select items and the I-beam cursor when typing into a TextBox.

A form or control’s MousePointer property determines the cursor shape of the mouse when it is on the form or a particular control. That's it – it really is that simple.

Here are the various cursor shapes that the mouse can take on from the Cursors namespace. Make sure that you preface all cursor names with Cursors. so that VB knows which namespace the cursor shapes are in.

List of typical cursor shapes:

|  |  |
| --- | --- |
| *Shape* | *Typical Purpose* |
| Cursors.Default | The standard arrow pointer |
| AppStarting | The busy cursor with a pointer attached indicating an application is starting up |
| Arrow | The pointer used to indicate selections can be made |
| Cross | For precision |
| Hand | The open hand cursor, usually indicating the user needs to wait |
| Help | The cursor typically shown when a user is allowed to navigate around and click on items for help |
| HSplit | The horizontal split cursor |
| IBeam | The insertion point cursor typically used in textboxes |
| No | The circle with the line through it |
| NoMove2D | Indicates no 2-dimensional move is allowed |
| NoMoveHorz | Indicates no horizontal move is allowed |
| NoMoveVert | Indicates no vertical move is allowed |
| PanEast | Used when panning East |
| PanNorth | Used when panning North |
| PanNE | Used when panning NorthEast |
| PanNW | Used when panning NorthWest |
| PanSouth | Used when panning South |
| PanSE | Used when panning SouthEast |
| PanSW | Used when panning SouthWest |
| PanWest | Used when panning West |
| SizeAll | Used when resizing in all directions |
| SizePointer | Used to indicate resizing |
| SizeNESW | Used to indicate resizing NorthEast and SouthWest |
| SizeNS | Used to indicate resizing North and South |
| SizeNWSE | Used to indicate resizing NorthWest and SouthEast |
| SizeWE | Used to indicate resizing West and East |
| UpArrow | The standard Up arrow pointer |
| VSplit | The vertical split cursor |
| WaitCursor | Used to indicate that the computer is busy |

Let's look at an application that changes the cursor's shape. We will create a form that has a timer and a command button on it. Upon Form.Load(), the cursor will change its shape to the AppStarting cursor. If you move the cursor over the running form, you’ll see this shape. After five seconds, the timer will tick and the cursor will revert back to a normal cursor. When the user presses the command button, the cursor will change to the WaitCursor shape and the timer will start up for five seconds again. Once the timer goes off, the cursor will be set back to the Default cursor. Here’s our code example:

'Chapter 13 - Program 2

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

'Change the cursor to the AppStarting (you have to move the mouse over

'the form to see this cursor)

Me.Cursor = Cursors.AppStarting

'Let this cursor be visible for 5 seconds

Timer1.Interval = 5000

Timer1.Enabled = True

End Sub

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

Handles Button1.Click

'Set the cursor to the Wait cursor and let it be visible for 5 seconds

Me.Cursor = Cursors.WaitCursor

Timer1.Interval = 5000

Timer1.Enabled = True

End Sub

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

'Any time that the timer ticks, shut it off and go back to

'the default cursor shape

Timer1.Enabled = False

Me.Cursor = Cursors.Default

End Sub

End Class

## Handling Events with the Keyboard

Working with the keyboard is about as easy as working with the mouse. There are three basic keyboard events that we would be concerned with interacting with. Let's take a look at these events for dealing with keystrokes at the Form level.

|  |  |
| --- | --- |
| *Event* | *Triggered When* |
| KeyDown | Indicates that a key on the keyboard has been pressed down; also checks for Ctrl, Alt and Shift |
| KeyPress | Returns the key that was pressed on the keyboard |
| KeyUp | Indicates that a key on the keyboard has been released |

When a key is pressed, the event firing order is KeyDown 🡪 KeyPress 🡪 KeyUp.

The KeyPress event receives as a parameter a KeyPressEventArgs object. The KeyPressEventArgs has the following properties available:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| KeyChar | Returns the ASCII character for the key pressed |
| Handled | Indicates whether the KeyPress event was handled or not |

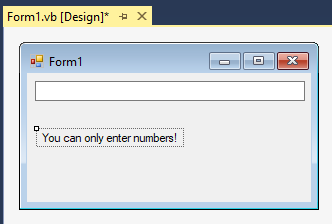
The KeyUp and KeyDown events receive a KeyEventArgs object which has the following properties available:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Alt | Indicates whether the Alt key was pressed |
| Ctrl | Indicates whether the Ctrl key was pressed |
| Shift | Indicates whether the Shift key was pressed |
| Handled | Indicates whether the event was handled or not |
| KeyCode | Returns the key code for the key as a Keys. enumeration; no modifier keys |
| KeyData | Returns the key code for a key combined with modifier keys |
| Modifiers | Returns a Keys. enumeration for any modifier keys |

Normally any control on the form will receive the keystroke before the form will. To override this, set the form’s Me.KeyPreview = True in the Form\_Load event. This now allows the form to intercept any keystrokes before a control receives the keystroke.

You will also most likely want to modify the KeyPress event handler in the control that it was intended for if the form is handling the keystrokes. You can do this by setting KeyChar = 0 in the control’s event handler. Therefore, the form will receive and process the keystroke and essentially nothing will be passed to the control for processing. The form can also set the KeyEventArg’s Handled property to True to indicate that the form took care of the keystroke.

Let’s write an application that only allows numbers into a textbox to illustrate some of the keyboard concepts. Here’s the simple form with a default named TextBox and Label:



This is the code behind it:

'Chapter 13 - Program 3

Public Class Form1

Dim blnNotANumber As Boolean

Private Sub TextBox1\_KeyDown(sender As Object, e As KeyEventArgs)

Handles TextBox1.KeyDown

'Assume the user will be giving us valid numeric data

blnNotANumber = False

' Determine whether the keystroke is a number from the top of the keyboard

'from the KeyCode sent in -- this is a good time to learn about the Keys

'namespace

If e.KeyCode < Keys.D0 Or e.KeyCode > Keys.D9 Then

'Numeric 0 - 9 on main keyboard?

If e.KeyCode < Keys.NumPad0 Or e.KeyCode > Keys.NumPad9 Then

'Not From keypad?

If e.KeyCode <> Keys.Back Then 'Allow backspacing too...

'If it wasn't one of the above, zap it!

blnNotANumber = True

End If

End If

End If

'One more possible set to weed out -- what if the Shift numerics were hit?

If Control.ModifierKeys = Keys.Shift Then

blnNotANumber = True

End If

End Sub

Private Sub TextBox1\_KeyPress(sender As Object, e As KeyPressEventArgs)

Handles TextBox1.KeyPress

'So this gets called whenever a key is about to go into the TextBox...

'If our flag got set, throw the key away...

If blnNotANumber = True Then

'Pretend we handled the keystroke, so it essentially gets zapped

e.Handled = True

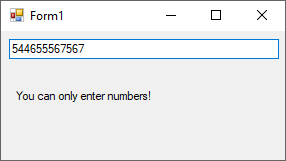
End If

End Sub

End Class

This was a whole lot more work that simply using the MaskTextBox control, but you also learned a bit more about how keystroke processing works behind the scenes.

Here’s our application doing what it does best -- allowing only numbers in:



# Miscellaneous Form & Control Odds and Ends

We have looked at quite a few of the various events and methods that forms, controls, the mouse and the keyboard respond to. There are a few miscellaneous events that we need to talk about before we consider this discussion complete. That is the purpose of this section.

## 

## Changed Event

There are numerous events that end with *changed*. These events are triggered by various controls when something on the control changes. One of the most notable examples is the TextChanged event on the TextBox control, which fires anytime someone types something into the TextBox. Most controls and forms also have a VisibleChanged event that fires whenever the visibility property of the control or form changes.

## GotFocus Event

The GotFocus event is called anytime that a control has the focus; i.e. the control can receive user input from the keyboard or mouse. GotFocus can be used to initialize a control for editing, such that as soon as a TextBox receives focus, you could programmatically clear its contents. In .NET, the Enter event is called prior to GotFocus indicating that the control has been entered. We can also programmatically set focus on a control to force the user there, for example if invalid data was entered and we don’t want them to leave the control until they put something valid in.

## LostFocus Event

The LostFocus event is called anytime that the focus has moved off of a particular control. Lost focus could be used to validate the contents of a control or possibly to prepare it for a future use if focus comes back.

## Focus Method

The Focus method is a method that most controls have, allowing you to programmatically set the focus to a control from within your code. We mentioned above the purpose of the related GotFocus and LostFocus events.

## Validating Event

The Validating event is fired when a control's contents are being validated. This is another suitable place to write code to see if what a user has entered into a TextBox is suitable or not.

The Leave event is called prior to the Validating event. If you choose to use the Validating event, you must set a Cancel flag before the event ends in the event’s CancelEventArgs object that indicates whether the validation was successful or not. If the validation was successful, the Validated event will fire. If the validation was not successful, the user will remain stuck on the control until he or she enters data that passes the validation test.

## Validated Event

The Validation event is called only when a Validating event has successfully completed validation.

Here is a code example that checks to see whether or not the user entered a valid date into a textbox. We will check to make sure the string is a valid date in the Validating event handler. If it is not, we will cancel the validation process, marking the text as not being able to be converted into a date. We also have put a Validating event handler in that will print out a message indicating that the text entered was able to be successfully parsed into a date, if that is the case. Here’s our source code:

Imports System.ComponentModel

'Chapter 13 - Program 4

Public Class Form1

Private Sub TextBox1\_Validating(sender As Object, e As CancelEventArgs)

Handles TextBox1.Validating

'This event is called when the user tries to take focus off of

'the textbox...

'Is the content's of the textbox a valid date?

If IsDate(TextBox1.Text) Then

'Yes, it was valid so do not cancel the validating process

'since it was valid

e.Cancel = False

Else

'No, it was invalid, so tell the user and cancel the

'validating process since the data was not valid

MessageBox.Show("You must enter a valid date")

e.Cancel = True

End If

End Sub

Private Sub TextBox1\_Validated(sender As Object, e As EventArgs)

Handles TextBox1.Validated

'This event will only be fired when the validation event passes

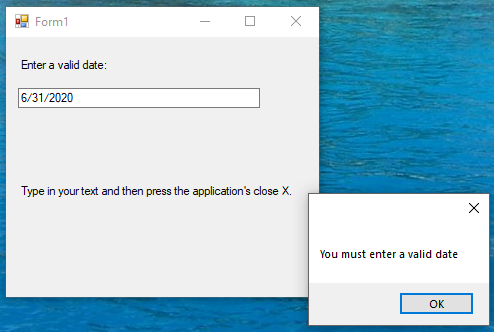
'(in other words the Validating event was not cancelled)

MessageBox.Show("you entered a good date")

End Sub

End Class

Here’s the application running when an invalid date is entered and has been attempted to be validated:

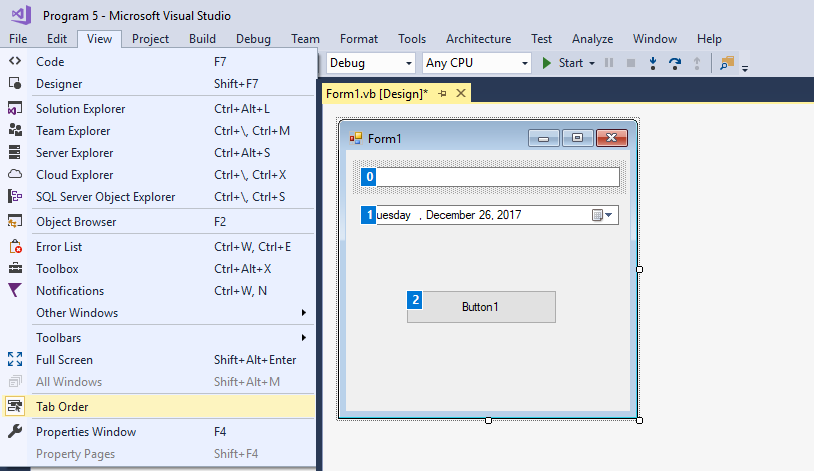


## Tab Order

One more item of note is setting the control’s Tab Order on a form. Users often expect to be able to press the Tab key on the keyboard to move from control to control. In some of your earlier applications you might have tried using the Tab key only to find that it didn't jump around in the order that you expected – that was because you didn’t add the controls that you wanted to use in a strictly sequential manner to your form.

You will notice that most controls have a TabIndex property – this allows you to specify the order in which controls should be visited if the user presses the Tab key. The first control will have a TabIndex value of 0. The next control that you want to be able to tab to should have a TabIndex value set to 1 and so forth. If you don't want a particular control to be tabbed to, then you need to set that control's TabStop property to False.

A much simpler way to set the Tab Order is to turn on the Tab Order View, by clicking on the View🡪Tab Order menu option. You’ll see that each control has a number attached to it on the control’s left hand side – that will be the tab ordering. If things aren’t in the order you want, simply click on the number until you get the value that you want. Pretty slick compared to having to manually set each TabIndex…yuck:



Once you have all the controls set up correctly, you should shut the Tab Order View off by clicking the View🡪Tab Order menu option once again.

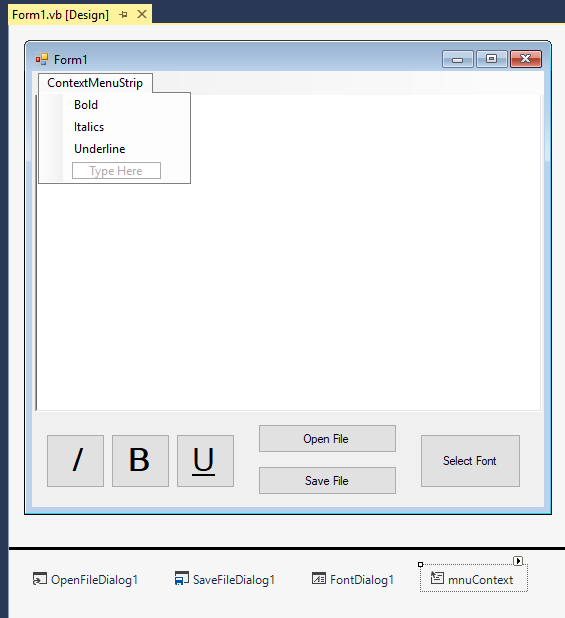
# Creating Context Menus

Context menus (also known as popup menus) typically show up when the user clicks the right mouse button on an object that supports a context menu. Microsoft Word allows you to right click almost anywhere within a document and you get a menu that pops up by the mouse allowing you to perform various operations.

You build the context menu in exactly the same fashion that you build main menus, except that you drag a ContextMenuStrip control to the form and make it look the way you want. Remember to use good names for the menu options that you add to the context menu and you will want to change the default name from ContextMenuStrip1 to something more meaningful since you can have several context menus in one application.

Let’s revisit the RichTextBox example that we saw in the last chapter and create a context menu that will allow for the user to highlight a section of text, then right click the mouse to get a context menu that will apply a font attribute to the selection. We will limit the attributes to bold, italics and underlining.

Here’s what the context menu editor screen looks like. Don't worry about the fact that the context menu appears to be in the same location as a main menu item. When your program runs, the context menu will pop up next to the mouse like it is supposed to.



While we’re talking about renaming things, we should also consider renaming each of the menu options to something more intelligible in our code as well… Now all that we have to do to get the context menu to work is to attach it to the control that should force the display of the menu. In our case we want the RichTextBox control to show the context menu when it is right clicked on. Click on the RichTextBox control in the form designer and change its ContextMenuStrip Property so that the name is set to the new context menu strip control that you added. That's it.

Here’s the code to do all of the work:

'Chapter 13 - Program 5

Public Class Form1

'We already had the heavy lifting done with the push buttons,

'so there are two ways we can handle this:

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

Handles mnuContextBold.Click

'We could write an event handler for the menu that simply calls the

'other event

cmdBold\_Click(sender, e)

End Sub

'Let's do the same thing for the italics

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

Handles mnuContextItalics.Click

cmdItalics\_Click(sender, e)

End Sub

'The second way, really reduces the workload, but you won't find it here --

'Check the existing underline event handler for a change...

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

Handles cmdBold.Click

'This is called whenever someone clicks on the bold button

If Not RichTextBox1.SelectionFont Is Nothing Then

'If we have a font, we will need to create a pointer to it

'so that we can modify the bold property

Dim currentFont As Font = RichTextBox1.SelectionFont

Dim newFontStyle As FontStyle

'Now that we have the font pointer, check and see

'if it is bold.

If RichTextBox1.SelectionFont.Bold = True Then

'if it was bold, make it non-bold

newFontStyle = FontStyle.Regular

Else

'if it wasn't bold, make it bold

newFontStyle = FontStyle.Bold

End If

'Now update the current font

RichTextBox1.SelectionFont = New Font(

currentFont.FontFamily,

currentFont.Size,

newFontStyle)

End If

End Sub

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

Handles cmdItalics.Click

'This gets called if someone clicks on the italics button

If Not RichTextBox1.SelectionFont Is Nothing Then

'Create a pointer to the font

Dim currentFont As Font = RichTextBox1.SelectionFont

Dim newFontStyle As FontStyle

'Check to see if the font is already italics or not and

'see it to the opposite setting

If RichTextBox1.SelectionFont.Italic = True Then

newFontStyle = FontStyle.Regular

Else

newFontStyle = FontStyle.Italic

End If

'Update the font

RichTextBox1.SelectionFont = New Font(

currentFont.FontFamily,

currentFont.Size,

newFontStyle)

End If

End Sub

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

Handles cmdUnderline.Click, mnuContextUnderline.Click

'This is called if someone clicks the underline button

If Not RichTextBox1.SelectionFont Is Nothing Then

Dim currentFont As Font = RichTextBox1.SelectionFont

Dim newFontStyle As FontStyle

'Figure out if we are currently underlined or not and

'set the font up in the opposite manner

If RichTextBox1.SelectionFont.Underline = True Then

newFontStyle = FontStyle.Regular

Else

newFontStyle = FontStyle.Underline

End If

'Update the font

RichTextBox1.SelectionFont = New Font(

currentFont.FontFamily,

currentFont.Size,

newFontStyle)

End If

End Sub

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

Handles cmdOpenFile.Click

'This is called if the user wants to open a file

'Show the OpenFileDialog box

OpenFileDialog1.ShowDialog()

'Take the filename from the dialog box and use the

'richtextbox's LoadFile method to actually load the file in

RichTextBox1.LoadFile(OpenFileDialog1.FileName)

'NOTE: There was no error trapping performed in the load. We

'assume the user supplied a good filename for a valid RTF file.

End Sub

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

Handles cmdSaveFile.Click

'This is called if the user wants to save a file

'Show the SaveFileDialog box

SaveFileDialog1.ShowDialog()

'Take the filename from the dialog box and use the

'richtextbox's SaveFile method to actually save the file out

RichTextBox1.SaveFile(SaveFileDialog1.FileName)

'NOTE: There was no error trapping performed in the save. We

'assume that we can write to the location specified by the

'user.

End Sub

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

Handles cmdSelectFont.Click

'This is called if the user wants to pick a new font

'Show the font selection box

FontDialog1.ShowDialog()

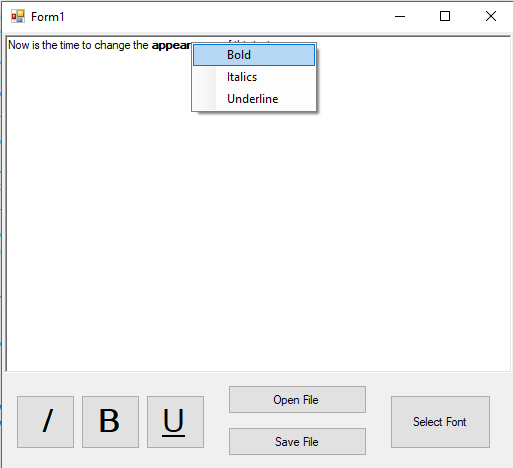
'Take the font that was picked and plug it into our richtextbox

RichTextBox1.SelectionFont = FontDialog1.Font

End Sub

End Class

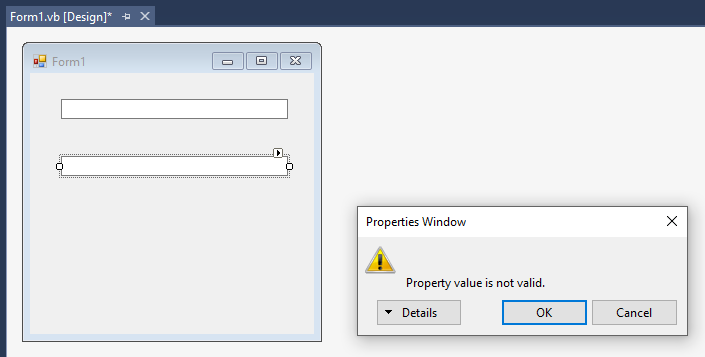
That’s the whole thing. It sure wasn't much work to add the context menu which really helps to make the application look more professional. Now when the user selects some text in the RichTextBox control, he or she can right click and change the text appearance by using the context menu that appears:



# Control Arrays – Cloning Controls

Have you ever wanted to put multiple copies of the same control on a form and yet you didn't want to have to remember all their individual names? That was the purpose of a control array in pre-.NET. VB.NET eliminated control arrays which really is too bad, because now you have to do some extra work to create something similar.

Here's the error message that you will receive if you try to name two controls the same thing in VB.NET. In this case I added TextBox1 without issue. I added a second TextBox and tried to rename it TextBox1 and I get this error message:



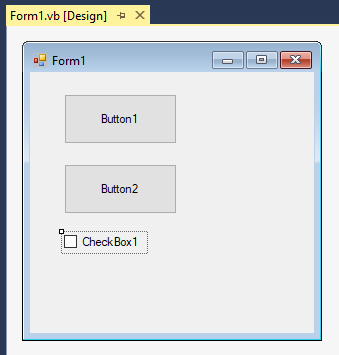
What VB instead allows you to do is write a generic event handler for multiple controls. We saw this in play on the last program, where we had one underline event handler take care of both the command button click event and the context menu click event. The bonus to this technique over the pre-.NET controls array version is that the controls that are to be handled today don't have to be of the same type!

The biggest problem that you face though, is how do you tell one control from another? In other words, if you have five buttons all being handled by the same event handler, how do you tell which one actually fired the event? This is where the old control array concept shined. Each control had a unique index value in the array, so you could tell them apart by the array’s element number.

The simplest way to solve this problem is to use either the TabIndex property (but this could be a pain since numbers are automatically assigned) or the Tag property. The Tag property is a remnant from VB of long ago when Microsoft probably meant to do something with it and didn't. All controls have a Tag property which can essentially hold a string value, and you can use the Tag value for whatever purpose you want. By uniquely assigning a number to each control's Tag property, you can simulate a control array.

To show you the power of this technique, let's consider the following problem: you want to handle two separate command buttons and a single checkbox all with one click handler. Normally Visual Basic would not permit this to happen as it would create a separate click event handler for each control. We can override what VB.NET does by writing our own special handler that names all of the controls (events, actually) that are to be handled by the handler.

You can name this multi-purpose event handler whatever you want. I called mine MixedControl\_Click in the example below. Notice that we will process all three controls’ click events in this single routine. Let's look at how this works by creating a form with two command buttons and a checkbox on it (using their default names). I will then assign numeric values to the controls from 1 to 3, top to bottom respectively via their Tag properties. Here’s a screenshot of the form:



Here's the code that allows us to handle any click the user makes on either command button or the checkbox.

'Chapter 13 - Program 6

Public Class Form1

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

Handles Button1.Click, Button2.Click, CheckBox1.Click

'Note that we have to write this handler -- VB will not do this for

'you since the default is to write one event handler for each

'individual control – notice each control and each control's event

'that is to be handled by this event handler are specifically

'named after the "Handles" statement in the event handler header

Select Case sender.Tag.ToString

Case "1"

MessageBox.Show("Button 1 was clicked")

Case "2"

MessageBox.Show("Button 2 was clicked")

Case "3"

MessageBox.Show("Checkbox 1 was clicked")

End Select

End Sub

End Class

Are you surprised at how clean and easy that example was? The amount of programmatic control that VB now gives you over events is quite impressive – of course, a lot of this is from object-orientation, which we will be getting into in a couple of chapters from here. While we are on the topic of doing some fairly complex things with controls and events, let’s go one step further. We already know that we can add controls to forms at design time by dragging and dropping them onto our form. Did you guess that you can also add controls at runtime programmatically through your application's code?

Let's revisit the example that we just created and modify the checkbox's click so that it places a new third command button on the form. We will have to do two things to get this to work. First, we will have to modify the MixedControl\_Clicked routine to handle the new command button even though it doesn't exist yet (how's that for an interesting trick). Secondly, we will need to write some code to make the checkbox create the new button when the checkbox is clicked on. Let's give the checkbox its own Click event to keep the code clean:

'Chapter 13 - Program 7

Public Class Form1

'We have to create a placeholder for the new button that will

'eventually be created by clicking on the checkbox. Notice the

'use of the WithEvents statement -- we must put this in since

'the new button will indeed be able to raise events, such as click.

Dim WithEvents Button4 As Button

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

Handles Button1.Click, Button2.Click, Button4.Click

'The MixedControls\_Clicked handles only the command buttons.

'Notice that both of the buttons that we added at design time

'and the button that will be created at runtime are all named as

'being handled by this handler. The checkbox has been removed

'and given its own handler to keep the code clean.

Select Case sender.tag.ToString

Case "1"

MessageBox.Show("Button 1 was clicked")

Case "2"

MessageBox.Show("Button 2 was clicked")

Case "4"

MessageBox.Show("Button 4 was clicked")

End Select

End Sub

Private Sub CheckBox1\_Click(sender As Object, e As EventArgs)

Handles CheckBox1.Click

'This event is called when the user clicks on the checkbox. It

'is responsible for creating and displaying the new button.

'The code that you see should looks a lot like what VB.NET

'generates behind the scene in the form's resource file.

'Instantiate a new button, set its text and tag

Button4 = New Button()

Button4.Text = "Button4"

Button4.Tag = 4

'Specify the location of the new button, its name, size

'and tabindex

Button4.Location = New System.Drawing.Point(34, 192)

Button4.Name = "Button4"

Button4.Size = New System.Drawing.Size(113, 50)

Button4.TabIndex = 3

'Add the new button to the form's Controls collection, which

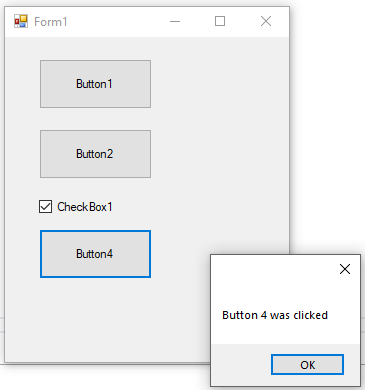
'is a collection of every control that is on the form

Me.Controls.Add(Button4)

End Sub

End Class

The next screenshot shows our finished project at work after clicking the CheckBox. The third dynamically created command button is now showing up and it is able to be clicked on:



Before we continue, there is one more issue to resolve with the application. VB, for whatever reason, will allow you to keep adding controls of the same name and type into the form's Controls collection through code (remember it stops it through the design time GUI builder). This means that if an user clicks on the checkbox twice, there will be two "Button4"s created.

By itself, this isn't a really big issue. The problem is that the new Button4 will be placed behind the current Button4, and, uniquely only one command Button4 can generate the click event. Unfortunately, the one that can create the click is the one behind the one that we can see. Therefore, when we try to click on the Button4 button, it doesn't seem to do anything.

Run the program again and get the Button4 button to appear. Notice that when you click on it, its event handler does what it is supposed to. Then click on CheckBox1 again, and you'll get a new Button4 (even though you don't see anything happen). Now try to click on Button4 again and its event handler doesn't seem to fire.

Fortunately, there is any easy way to handle this. We can modify the CheckBox's click event to resolve the problem. There are two different ways to do this. One way would be to iterate through the Controls collection and see if any control has the name "Button4." If we find an existing Button4, we simply wouldn't add another one.

The second, and programmatically more interesting way would be to iterate through the Controls collection, and if we find a Button4, remove it from the collection so that when we add the Button4 instance, we know that it is unique. Here is the revised CheckBox event handler to do this:

'Chapter 13 – Program 8

Private Sub CheckBox1\_Click(sender As Object, e As EventArgs)

Handles CheckBox1.Click

Dim aControl As Control

'Iterate through every control in the form's Controls

'collection

For Each aControl In Me.Controls

'If we find one named, Button4 then delete it

'and inform the user that we found one

If aControl.Name = "Button4" Then

Me.Controls.Remove(aControl)

MessageBox.Show("Removed Existing Button4 from " & \_

"Controls collection")

End If

Next

'We now can add a new Button4 since we know there isn't one at

'this point

'Instantiate a new button, set its text and tag

Button4 = New Button()

Button4.Text = "Button4"

Button4.Tag = 4

'Specify the location of the new button, its name, size

'and tabindex

Button4.Location = New System.Drawing.Point(30, 152)

Button4.Name = "Button4"

Button4.Size = New System.Drawing.Size(120, 32)

Button4.TabIndex = 3

'Add the new button to the form's Controls collection, which

'is a collection of every control that is on the form

Me.Controls.Add(Button4)

End Sub

If you make this modification and run the program, you will see that the user can now click as many times as he or she wants on the CheckBox and only one instance of Button4 will ever be allowed at any point. This removes the problem of having “disconnected” click events.

Of course, we could have also used another trick in the CheckBox’s click handler to make it impossible to add another Button 4 object:

'Chapter 13 – also in Program 8

Private Sub CheckBox1\_Click(sender As Object, e As EventArgs)

Handles CheckBox1.Click

'Instantiate a new button, set its text and tag

Button4 = New Button()

Button4.Text = "Button4"

Button4.Tag = 4

'Specify the location of the new button, its name, size

'and tabindex

Button4.Location = New System.Drawing.Point(30, 152)

Button4.Name = "Button4"

Button4.Size = New System.Drawing.Size(120, 32)

Button4.TabIndex = 3

'Add the new button to the form's Controls collection, which

'is a collection of every control that is on the form

Me.Controls.Add(Button4)

'Don't let the user do this again, or bad things will happen

CheckBox1.Visible = False

End Sub

Now that we’ve seen this little bit of logic, we could actually build a better control array simulator with a bit of ingenuity. Let’s write a program that dynamically builds ten textboxes, adds them to the screen and programmatically sets an event handler so that all ten reference the same event handler. Oh, and just for fun, let’s make is so that the first five can only accept character input, while the last five can only accept numeric input…

We can programmatically perform this event wiring at runtime through the use of the AddHandler keyword. AddHandler is used to specify, at runtime, what method should be run when an event occurs. So, think about that – we can add a control and then wire up subprograms in our code while we’re executing. AddHandler expects two arguments, a valid control event and the address of the handler code. What this means is that you have to ensure that whatever event you want to wire up on a control exists. Everything we’ve looked at so far is fair game. Eventually we will even learn how to wire up our own events, but not today.

Just as there is an AddHandler, there is also a RemoveHandler keyword which allows us to unwire, or disassociate an event from its code handler at execution time. This all really sounds more complicated that what it is!

We’ll start by building a form with a single command button on it (named cmdMakeTextBoxes). Everything transpires from there after we click the button.

Here’s the source code:

Public Class Form1

'Chapter 13 - Program 9

'Create an array of textbox instances

Dim arrMyTextBoxes() As TextBox

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

Handles cmdMakeTextBoxes.Click

Dim intLoop As Integer

'We're going to build 10 textboxes that will be stored in an array

ReDim arrMyTextBoxes(9) 'size the array for 10

For intLoop = 0 To 9

'Instantiate and set the necessary properties

'You're playing code generator now...

arrMyTextBoxes(intLoop) = New TextBox

arrMyTextBoxes(intLoop).Name = "txtDynTextBox" & intLoop

arrMyTextBoxes(intLoop).Parent = Me

arrMyTextBoxes(intLoop).Top = intLoop \*

arrMyTextBoxes(intLoop).PreferredHeight + 2

arrMyTextBoxes(intLoop).Tag = intLoop

arrMyTextBoxes(intLoop).Visible = True

'This line is huge! We want to make sure that all of the KeyPress

'events are handled by the same handler!!!! So, we will use the

'AddHandler keyword to say that this particular textbox instance's

'Keypress event is handled by our routine TestKeyPress. The

'AddHandler keyword requires two arguments: the event being handled

'and a pointer to the TestKeyPress routine. We can get this pointer

'by using the AddressOf keyword which determines where something lives

'in memory (in this case, a routine, but it really could be anything!)

AddHandler arrMyTextBoxes(intLoop).KeyPress, AddressOf TestKeyPress

Next

End Sub

'Here's the handler, but notice no Handles statement at the end -- this is

'all set up at runtime dynamically using the AddHandler call!

Public Sub TestKeyPress(source As Object, e As KeyPressEventArgs)

Dim intIndex As Integer

'Figure out which textbox we are working with.

intIndex = CInt(source.tag)

'0 through 4 can accept only character input

If intIndex < 5 Then

If e.KeyChar <= "9" And e.KeyChar >= "0" Then

e.Handled = True

End If

Else

'while 5 though 9 will only accept numeric input

If e.KeyChar > "9" Or e.KeyChar < "0" Then

e.Handled = True

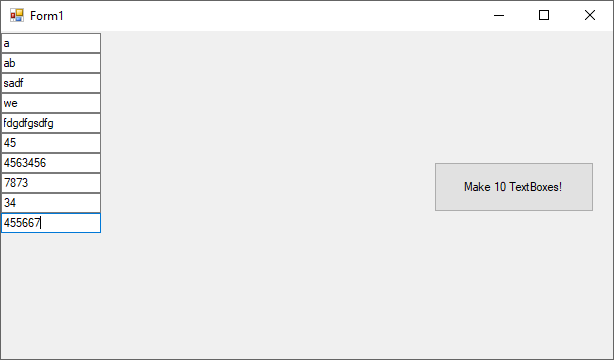
End If

End If

End Sub

End Class

Here’s what the screen looks like after it runs – note that the KeyPress event is doing its job by checking the Tag (think control array index). That’s what determines what that particular TextBox is allowed to accept into its Text property:

Drag and Drop

One of the neat tricks in Windows that makes your application look sharp is if it supports drag and drop. This is the ability to click on an object on the screen, drag it using the mouse and drop it off somewhere else. This used to be a real pain to write, but your friend .NET has really simplified the entire process.

Let's jump right into an example of an application that supports drag and drop. We will create a form that has a PictureBox and a Command button on it, using the default names for these controls. The PictureBox will be set up with its BackColor property initially set to green. What we want to allow is for the Command button to be able to be dragged and dropped on the form. It won't be able to be dropped just anywhere, rather it can be dragged and dropped onto the PictureBox.

As soon as the user moves the Command button over the PictureBox, the PictureBox will turns yellow, meaning that it’s okay for the user to drop the Command button there. If the user drops the button on the PictureBox, the box will turn blue (“drop okay” - that's the result we want when a drop has been successfully completed). If the user decides not to drop the Command button, then when the button leaves the PictureBox, the box is painted red (“no drop”). Remember, I started the application off by painting the PictureBox’s BackColor property green.

Now the question you should ask is how does it work? The first step that you have to set up is enabling the target control (where you are going to drag something to) so that it will listen for and receive drop events. This step is very critical because if you don't set up a target control, no drop event will be triggered, therefore no dragging and dropping will occur in your application. I took care of this in my application's Form\_Load by setting the PictureBox's AllowDrop property to True.

The next major piece of coding work is to allow the user to initiate the Drag process. Somehow, we have to let the application know that the user isn't just pointing and clicking the mouse – he or she actually wants to move something with the mouse. You usually will handle this task in the source control's (what you are going to drag with the mouse) MouseMove event. Usually a mouse button is held down during the drag operation. Therefore, you need to see if any mouse buttons are pressed in MouseMove. If there aren't any, simply exit the event handler. If there was a mouse button pressed, then you've got more work to do and you will start the DoDragDrop method, which notifies our application that there is a drag and drop operation underway. We'll look at the details of how to implement these things in a second.

Some additional work that you must complete is to write DragEnter and DragDrop event handlers for the target control. These handlers respectively will define what to do when we enter the target control and what to do when the user lets go of the mouse button while over the target control. The last handler that you have to concern yourself with is DragLeave, which is called when a drag and drop operation is underway, but the user for some reason has moved the mouse outside of the target control – think of it as drop cancelled. This is all actually much simpler to implement than it sounds. Let's take a quick summary of the pieces of the drag and drop puzzle first.

Here are the drag and drop events, methods and properties:

|  |  |
| --- | --- |
| *Item* | *Purpose* |
| AllowDrop | A property that must be set to true on whatever control(s) are to serve as the "destination" for a drag and drop operation |
| DoDragDrop | This method is invoked when we realize that the user is starting a drag and drop operation. This method informs our application that a drag and drop is underway |
| DragDrop | This method is called when the user completes the drop operation by letting go of the mouse button(s) while the "source" object is over the "destination" control |
| DragEnter | This method is called when the "source" object moves over the "destination" control – usually you will alter the appearance of the destination in some way to clue the user that they are over a droppable location |
| DragLeave | This method is called when the user leaves the "destination" control before letting go of the mouse button(s) – this effectively signals that the drag and drop operation is to be cancelled |
| MouseMove | The event that we usually watch to see if the user was holding down any mouse buttons on a "source" draggable object – if the user was, then initiate the DoDragDrop method |

Here’s the application's source code:

Public Class Form1

'Chapter 13 - Program 10

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles Me.Load

'When we start up, tell the PictureBox it can accept dropped items

PictureBox1.AllowDrop = True

End Sub

Private Sub Button1\_MouseMove(sender As Object, e As MouseEventArgs)

Handles Button1.MouseMove

'This event is triggered whenever the mouse is moved over the

'command button -- we need to determine whether or not we

'should start a drag and drop operation using the command

'button (this is our source control)

'If no mouse button was pressed, get out of here as

'we are not trying to drag and drop

If e.Button = 0 Then

Exit Sub

End If

'Since the mouse is moving and there is a button down,

'we need to start the drag operation

'Set up a copy of the source control by casting the sender

'into, in this case, a button

Dim bbase As ButtonBase = DirectCast(sender, ButtonBase)

'We need to copy any data associated with the control, in

'this case there isn't any, but we still need the DataObject

'to make the DoDragDrop method happy! We'll worry about

'data in another example...

Dim data As New DataObject()

'What type of DragDropEffect do we want...

Dim effect As DragDropEffects = DragDropEffects.Copy

'There is no data to copy, so zap it

data.SetData("")

'Initiate the dragdrop on the copy of the target control we made

effect = bbase.DoDragDrop(data, effect)

'If we didn't want to mess with the casting of the Button Base, we

'could do this and replace everything above:

'Dim DragDropResult As DragDropEffects

'Dim data As New DataObject()

'data.SetData("")

'DragDropResult = PictureBox1.DoDragDrop(data, DragDropEffects.Copy)

End Sub

Private Sub PictureBox1\_DragEnter(sender As Object, e As DragEventArgs)

Handles PictureBox1.DragEnter

'Since the application has been informed that there is a

'drag and drop operation underway, we want to let the user

'know that this control is okay to have things dropped on it

'when the user moves the mouse over a droppable location.

'When we move over the picturebox, e.g. we entered the

'picturebox in the middle of a drag. What do we want to

'do? -- Show the effect

e.Effect = DragDropEffects.Copy

End Sub

Private Sub PictureBox1\_DragOver(sender As Object, e As DragEventArgs)

Handles PictureBox1.DragOver

'As long as the user holds the mouse button down over the

'control we will execute this handler -- they will then either drop

'or leave -- yellow indicates that this is okay to "drop" on

PictureBox1.BackColor = Color.Yellow

End Sub

Private Sub PictureBox1\_DragDrop(sender As Object, e As DragEventArgs)

Handles PictureBox1.DragDrop

'At this point, we know there is a drag and drop operation

'underway and the user has already moved over the destination

'control (DragEnter told us that). If the user lets go of the

'mouse button(s) here, this event (DragDrop) will be called. Here's

'where you write the code that you want carried out when a

'drag and drop is successfully completed.

'Once the user let's go of the mouse, e.g. a "drop" was

'done, what do we need to do? Change color to Blue

PictureBox1.BackColor = Color.Blue

End Sub

Private Sub PictureBox1\_DragLeave(sender As Object, e As EventArgs)

Handles PictureBox1.DragLeave

'At this point, we know there is a drag and drop operation

'underway and the user has already moved over the destination

'control (DragEnter told us that). If the user for some reason

'moves the source out of the target without letting go of the

'mouse button(s), this event (DragLeave) will be called. Here's

'where you write the code that you want carried out when a

'drag and drop is essentially cancelled.

'If they entered the PictureBox but then the left while still holding

'the mouse button(s) down, the drag "left", e.g. "drop" was cancelled

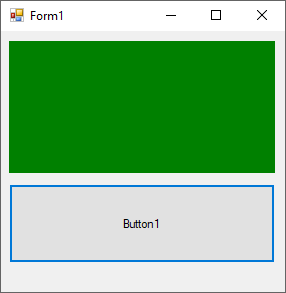
'on this control -- so turn red

PictureBox1.BackColor = Color.Red

End Sub

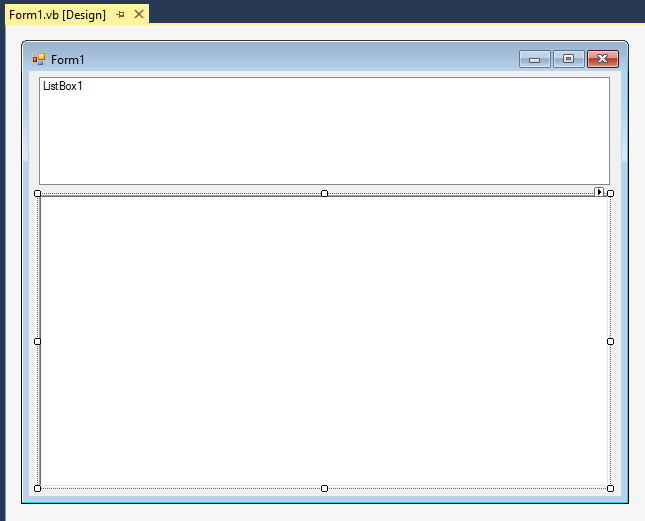
End Class

The next screenshot shows the application in its startup state. Go ahead and run the application. Try dragging and dropping the command button both on and off the PictureBox so that you develop an understanding of how the drag and drop operation really works.



Now let's take a look at another example where we can drag and drop filenames from a ListBox onto a RichTextBox. Once the RichTextBox receives the filename, it will open the file up and display the text contained within the file. This application does things a bit differently than the last one since there is data that must be contended with between the drag source object and the drop destination control.

The application's main form consists of a ListBox and a RichTextBox using the default names:



Here’s the application code:

Imports System.IO

'Chapter 13 – Program 11

Public Class Form1

Private Sub Form2\_Load(sender As Object, e As EventArgs) Handles Me.Load

Dim myDirInfo As String

Dim myFileInfo = New FileInfo(Application.ExecutablePath)

'Sets the AllowDrop property so that data can be dragged onto the control.

RichTextBox1.AllowDrop = True

'Add code here to populate the ListBox1 with paths to text files.

myDirInfo = Convert.ToString(myFileInfo.Directory())

ListBox1.Items.Add(myDirInfo & "\eula.txt")

ListBox1.Items.Add(myDirInfo & "\h323log.txt")

ListBox1.Items.Add(myDirInfo & "\SerialSync.txt")

ListBox1.Items.Add(myDirInfo & "\SQLSRDME.txt")

End Sub

Private Sub ListBox1\_MouseMove(sender As Object, e As MouseEventArgs)

Handles ListBox1.MouseMove

'Everything starts with a mouse move in the original listbox.

Dim dropEffect As DragDropEffects

If e.Button = Windows.Forms.MouseButtons.Left Then

'If the left mouse button is down, proceed with the

'drag-and-drop, passing in the list item as the data.

dropEffect = ListBox1.DoDragDrop(

ListBox1.Items(ListBox1.SelectedIndex),

DragDropEffects.All Or

DragDropEffects.Link)

End If

End Sub

Private Sub RichTextBox1\_DragEnter(sender As Object, e As DragEventArgs)

Handles RichTextBox1.DragEnter

'Once we start the drag in the RTB, just keep the copy effect showing

e.Effect = DragDropEffects.Copy

End Sub

Private Sub RichTextBox1\_DragDrop(sender As Object, e As DragEventArgs)

Handles RichTextBox1.DragDrop

'Loads the file into the control.

'This drop will have data associated with it. In our case the ListBox had

'an item that was selected when the drag drop started containing the name

'of a file.

'We need to pull that data out when the RTB successfully receives a drop

'and pass it to the LoadFile method to actually load the text file into

'the RTB for display.

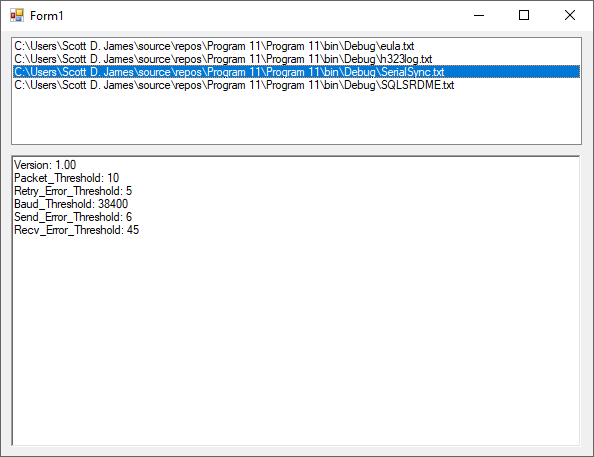
RichTextBox1.LoadFile(e.Data.GetData("Text"),

RichTextBoxStreamType.PlainText)

End Sub

End Class

Here's the application at work:



## 

# Making the MessageBox look pretty

We’ve been using the MessageBox to display things right along. Many Windows applications throw various icons and buttons onto the default MessageBox depending on the intent. We can easily extend our MessageBoxes to behave the same way.

We already know that the first parameter to the MessageBox.Show method provides the string that is to be displayed in the center canvas area of the MessageBox itself. We also have learned that the second parameter is an optional title string for the MessageBox window. There are also several other parameters that we can work with.

The first item that we can modify is which command buttons are displayed on the MessageBox. This is the third parameter in the MessageBox.Show method. Here’s a list of the different button choices (it’s pretty obvious from the names which buttons will be displayed). These are all in the class MessageBoxButtons. namespace:

|  |  |
| --- | --- |
| *Value* | *Buttons Displayed* |
| OK | Displays the OK button only |
| OKCancel | Displays the OK and Cancel buttons |
| AbortRetryIgnore | Displays the Abort, Retry and Ignore buttons |
| YesNoCancel | Displays the Yes, No and Cancel buttons |
| YesNo | Displays the Yes and No buttons |
| RetryCancel | Displays the Retry and Cancel buttons |

The next parameter that you can provide to the MessageBox.Show method tells the method which icon to display in the canvas area. The various icons that can be displayed are provided in the following list (there are all from the MessageBoxIcon. namespace). Notice that the purpose of many of these items are the same:

|  |  |
| --- | --- |
| *Value* | *Purpose* |
| Asterisk | (the i in a thought bubble) – used to give the user information about an action |
| Error | (the x in the red circle) – used to indicate that something critical occurred in the application |
| Exclamation | (the ! in a yield sign) – draws the user’s attention to the message being displayed |
| Hand | (the x in the red circle) – used to indicate that something critical occurred in the application |
| Information | (the i in a thought bubble) – used to give the user information about an action |
| None | Just like it says – no icon is displayed |
| Question | (the ? in a thought bubble) – used when the user must make a decision |
| Stop | (the x in the red circle) – used to indicate that something critical occurred in the application |
| Warning | (the ! in a yield sign) – draws the user’s attention to the message being displayed |

Another parameter allows you to pick which button is set as the default in the MessageBox, e.g. which one gets the focus. This is set by using the following constants which are defined in the MessageBoxDefaultButton namespace:

|  |  |
| --- | --- |
| *Value* | *Purpose* |
| Button1 | Select the first button displayed as the default |
| Button2 | Select the second button displayed as the default |
| Button3 | Select the third button displayed as the default |

Finally, an application can determine if a particular button was selected by a user. The following buttons will be returned by the MessageBox (all return values are stored in the DialogResult. namespace):

|  |  |
| --- | --- |
| *Value* | *Purpose* |
| Abort | Indicates that the Abort button was pressed by the user |
| Cancel | Indicates that the Cancel button was pressed by the user |
| Ignore | Indicates that the Ignore button was pressed by the user |
| No | Indicates that the No button was pressed by the user |
| None | Indicates that the None button was pressed by the user |
| OK | Indicates that the OK button was pressed by the user |
| Retry | Indicates that the Retry button was pressed by the user |
| Yes | Indicates that the Yes button was pressed by the user |

Here's a snippet of code that shows an example of a MessageBox showing the string "Quit?" in a window titled "End Application…". The buttons Yes and No are shown to the user along with the Question icon. Button1 has been designated as the default button (Yes). Finally, the code checks to see if the user pressed the Yes button in the if statement.

If MessageBox.Show("Quit?", "End Application...", \_

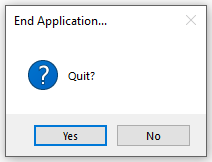
MessageBoxButtons.YesNo, MessageBoxIcon.Question, \_

MessageBoxDefaultButton.Button1) = DialogResult.Yes Then

'If body goes here...

End If

Here’s the MessageBox at runtime:

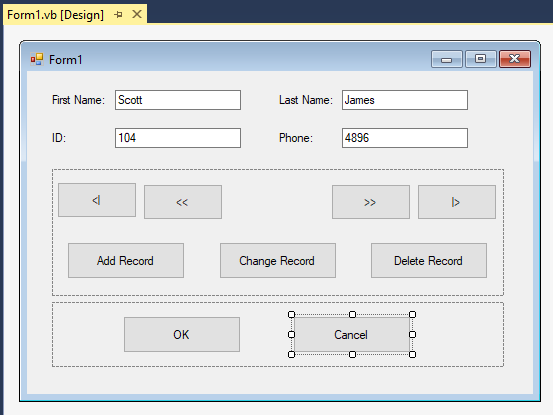


## Containers

The last topic that we will cover in this chapter is the idea of a container. A container is just a control that can host other controls inside of it. The simplest idea of a container is a form…you can put multiple controls on the form and they can all be referenced through *formname.controlname* type notation.

Certain controls such as the Panel, GroupBox, PictureBox, TabControl and Splitter controls will allow you to place multiple controls in them. Let’s take a quick look at how this works. Assume that you need to write the user interface to a database application. Obviously, we will need to have navigation controls that allow us to move through the records. We would also need to be able to save changes that the user makes to records. We probably would not want those controls all bundled together since we will only want to allow certain functionality some of the time. You wouldn’t want a user to move to a new record in the middle of editing an existing record before the changes are saved, would you? By using panels we can simplify our work.

To start the project out, I am going to create two separate panels with various controls placed inside of them. Panel 1 will contain controls for navigating from record to record as well as launching add/change/delete functionality. Panel 2 will contain the controls that lets the user signify that he/she wants to either save the add/change/delete work carried out or cancel it. The finished form would look something like this with the main set of controls showing:



Now, when I run my program, I can start out by having Form Load set Panel2's visible property set to False and only let the user see those controls when he or she actually needs to. Later, when the user clicks on a button (like Add) that requires Panel2, all we have to do is set the visibility of Panel2 to True and that panel’s contents appear.

By simply changing the visibility property of the panel, which is acting as a container, we affect the visibility of all of the controls that are stored within the container as well. This makes one little line of code a very powerful mechanism! Now in practice, as we mentioned, if a user indicates he/she wants to change a record, we would allow the user to modify what’s in the textboxes. The user should not be able to navigate to other records nor click any other function button like Add or Delete while “change” is in use. Therefore, we would want to hide Panel1. Somehow the user needs to indicate that he/she is finished making the change and either wants to keep the changes (OK) or get rid of them (Cancel), so we would want to show Panel2.

We should set up Panel1 and Panel2 in Form Load to be mutually exclusive through the Visible property on each panel. After that, most operations would just result in flipping the state of the two panels to prevent the user from doing things that we don’t want them to do.

Here’s the code:

Public Class Form1

'Chapter 13 - Program 12

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

'Start out by making the two panels mutually exclusive -- in

'other words, you can only ever see one panel at a time

Panel1.Visible = True

Panel2.Visible = False

End Sub

Sub FlipState()

'Since the two panels were started off mutually exclusive

'with regard to visibility, this routine will flip the

'state of the two panels

Panel1.Visible = Not Panel1.Visible

Panel2.Visible = Not Panel2.Visible

End Sub

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

Handles cmdAddRecord.Click

MessageBox.Show("Work to Add Record carried out!")

FlipState()

End Sub

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

Handles cmdChangeRecord.Click

MessageBox.Show("Work to Change Record carried out!")

FlipState()

End Sub

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

Handles cmdDeleteRecord.Click

MessageBox.Show("Work to Delete Record carried out!")

FlipState()

End Sub

Private Sub cmdOK\_Click(sender As Object, e As EventArgs) Handles cmdOK.Click

MessageBox.Show("Whatever work OK does carried out!")

FlipState()

End Sub

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

Handles cmdCancel.Click

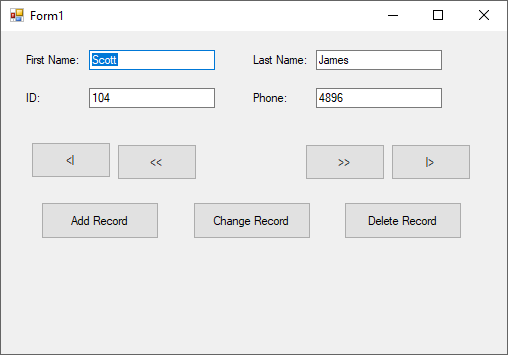
MessageBox.Show("Whatever work Cancel does carried out!")

FlipState()

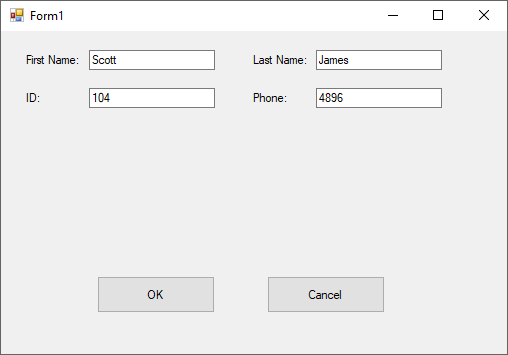
End Sub

End Class

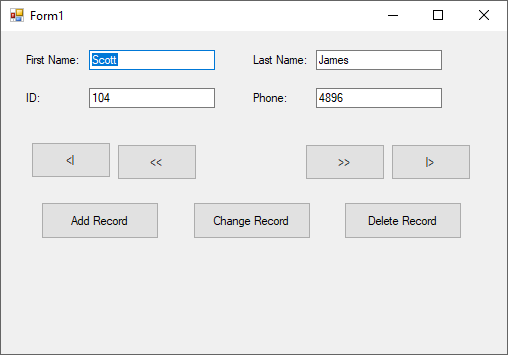
Here’s our application upon startup:



The user clicks Change Record, so the textboxes should be editable. No navigation or function controls should be available. The user should only be able to modify the text in the textboxes and indicate whether the modifications are to be kept or thrown away. We will hide Panel1 and show Panel2 to make this happen:



The user has decided to save the changes and clicks the OK button. Whatever work that needs to take place to save things gets executed. We would then need to swap the panels back around so that navigation/commands can be made again:



Nice, clean design all courtesy of some mutual exclusivity and a routine that flips the states of the two panels!