Chapter 5 – Common Controls, Properties and Events

# Objectives

* Understanding the common properties available on most controls
* Understanding common events that are generated by controls
* Control Sample Programs
* Introductory Error Handling

# Understanding the Common Properties Available on Most Controls

If you haven't realized it yet, most of the power of Visual Basic comes from the fact that you are quickly able to create applications by dragging and dropping controls onto a form. Indeed, if you didn't have these controls available to you, you would spend a great deal of time duplicating the functionality built into the controls. VB blew away the Windows development world of the early 1990s. By VB4 and VB5, entire companies were making their fortunes on turning out smart controls. Microsoft seems to have forgot about that with their lack of ported controls to the Windows Presentation Foundation (WPF) platform.

In order to become a proficient programmer in the VB environment, you must quickly master what controls are available and the functionality that each provides. Fortunately, most controls (and forms) exhibit the same types of functions and properties. Let's take a look at the common properties that most controls have. You'll want to spend some time examining this list so that you know what is available for your use.

Common control properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Anchor | Where the control is bound, relative to the edges of the form |
| BackColor | Sets the background color of the control |
| BorderStyle | Indicates whether or not a border (and, if so, which type) should appear around the form (most controls do not have this property – but forms do) |
| CausesValidation | Determines whether or not the control raises validation events that you can tap into in order to ensure that data entered into the control is correct |
| ContextMenuStrip | Names the context menu to display when the user right clicks on the control |
| Enabled | If set to true (default), the control is accessible to the user |
| FlatStyle | Changes the general appearance of a control. There are four general appearances: Standard, Flat, Popup and System |
| Font | Sets the face, attribute and size of the font used for the text on the control, if the control has any text |
| ForeColor | Sets the foreground color of the control |
| Location | Contains the X and Y coordinates of where the control is placed on the form (actually whatever container the control lives in) |
| Locked | Determines if the control can be moved or resized |
| Margin | Space between this control and any other |
| MaxLength | If the control allows data entry, maximum number of characters permitted |
| Multiline | Again, if the control allows text entry, can it span more than one line? |
| Name | Allows you to name a control; for example txtAge would be a good name for a textbox that allows a user to enter his/her Age |
| ReadOnly | Determines whether the text in the control can be changed |
| Size | Contains the Width and Height coordinates of the control |
| TabIndex | Contains the Tab Stop order in which the control should be moved to if a user tries to move from control to control by pressing the tab button |
| TabStop | Indicates whether or not the user can move to this control by pressing the tab button |
| Tag | An empty placeholder each control has that you can use for your own purposes |
| Text | Sets the text that is displayed on a control if the control displays text; Text will also contain text input from the user if the control permits text to be input (such as the textbox control) |
| TextAlign | Specifies how displayed text should appear in the control (left, center and right vertically; and top, middle and bottom horizontally) |
| ToolTip | The text that shows up in the little yellow sticky-note-like boxes that appear as users move their mouse pointer over a control – this property is only available if a ToolTip control is placed on the form |
| Visible | A true or false setting indicating whether or not a control should be displayed. |
| WordWrap | Indicates that multiline text fields should automatically word wrap or not |

Now that we've spent some time examining the common properties, let's take a look at the controls that you are mostly likely to need when you are creating Visual Basic.Net applications. Probably the best way to get familiar with these controls is to have Visual Basic .NET running and actually add an instance to a form. As you go through the list of controls, try locating each one in the Toolbox and adding it to your form. Playing around with the controls is the best way to learn them. The end of this section will have some sample programs that use the controls.

## Label

This control is used to display text on a form. The contents of the label (the Text attribute) may be static or dynamic.

## Textbox

This control is used to allow users to type in information so that your program can access it. Included below are some additional properties beyond the common properties that are unique to the Textbox control and which will greatly increase the functionality of it.

Additional textbox properties

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| PasswordChar | Indicates that the textbox will allow users to type in sensitive information (such as a password) and that the *PasswordChar* character should be displayed instead of the actual characters entered |
| ScrollBars | Indicates whether or not horizontal and/or vertical scrollbars should be displayed on a multi-line textbox |

## Command Button

This control, which simply shows up as Button in the Toolbox, is used to allow users to push a command button to indicate an action is to take place. This is one of the controls that just about every Windows user is familiar with. OK and Cancel are two commonly used command buttons.

## CheckBox

This control is used to allow users to toggle between two different setting states, such as on/off or enable/disabled. You often see checkboxes for things such as "Print Cover Page?," where if the user clicks on the checkbox and it is filled in with a checkmark, then a cover page will be printed. If the checkbox is empty, then no cover page will be printed.

Additional checkbox properties

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Checked | Indicates that the checkbox is checked |
| CheckState | Indicates the state of the checkbox (Checked, Unchecked, Indeterminate "grayed") |

## Radio Button

This control, which is sometimes referred to as an option button, is used to limit a user to only selecting one option out of a group of options. Typically a GroupBox control is placed on the screen and several related radio buttons are placed in the GroupBox such that only one radio button may be picked at a time. One of the most commonly encountered radio button control sets is that of printing. Usually there is a radio button for printing all, printing select pages, printing the current page or printing the selection. Only one of these items can be chosen at any given time.

Additional radio button properties

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Checked | Contains the state of the option button: True or False |

## 

## ComboBox

This control is used to provide the user with a list of possible choices. However, it also allows the user to enter in a customized choice if that choice is not already presented in the list. The ComboBox is almost programmatically identical to the ListBox, which will be discussed below.

Additional ComboBox properties

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| DropDownStyle | Changes the appearance of a ComboBox |
| Items | Contains the list of items to be shown to the user |
| SelectedIndex | Gets or sets the index specifying the currently selected item (only available when the control is running within a program) |
| SelectedItem | Gets or sets the currently selected item in the ComboBox (only available when the control is running within a program) |
| SelectedText | Gets or sets the text that is selected in the editable portion of the ComboBox (only available when the control is running within a program) |
| Sorted | Indicates whether or not the list of items should be displayed in sorted order |

## ListBox

This control provides the user with a list of possible items and allows the user to select one/some. No new items can be added by the user; if you need that functionality use the ComboBox discussed above.

Additional ListBox properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Items | Contains the list of items to be shown to the user |
| SelectedIndex | Gets or sets the index specifying the currently selected item |
| SelectedItem | Gets or sets the currently selected item in the ListBox |
| SelectedText | Gets or sets the text that is selected in the editable portion of the ListBox |
| SelectionMode | Indicates whether or not the ListBox will allow the selection of multiple items |
| Sorted | Indicates whether or not the list of items should be displayed in sorted order |

## Scrollbars (Horizontal and Vertical)

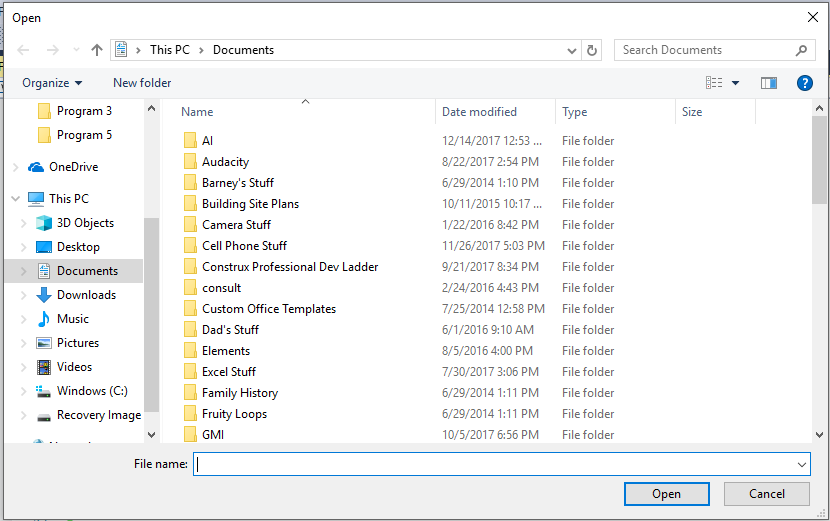
These controls allow the user to change values through scrollbar motion. This is another control that every Windows user has interacted with.

Additional Scrollbar properties:

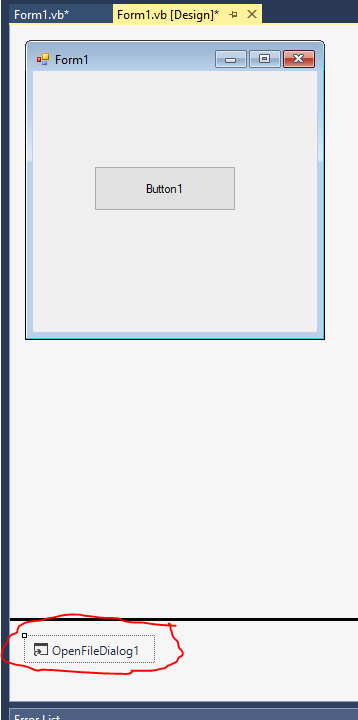
|  |  |
| --- | --- |
| *Property* | *Purpose* |
| LargeChange | Indicates the amount of change to the value property when the user clicks on the scrollbar area itself |
| Maximum | Indicates the largest value on the scrollbar |
| Minimum | Indicates the smallest value on the scrollbar |
| SmallChange | Indicates the amount of change to the value property when the user clicks on the scrollbar arrows |
| Value | The current position of the scrollbar |

## OpenFileDialog

This control automatically brings up the common Open File dialog box that is presented in most Windows applications. The control allows a user to either type in the name of a file to open, or to browse to a file by pointing and clicking the mouse. The next picture shows what the control looks like when it's running.



In Visual Basic.NET, this control is an invisible control that only appears when you tell it to through code. Notice that when you drag an OpenFileDialog control over to your form and drop it, the control is not added directly to the form, but rather in a container area for invisible controls that is underneath the form itself.



An instance of the OpenFileDialog control added to a form

Additional OpenFileDialog methods and properties:

|  |  |
| --- | --- |
| *Property/Method* | *Purpose* |
| FileName | Contains the complete path and name of the file that the user selected |
| ShowDialog | Call this method to actually display the OpenFileDialog box to the user |

## PictureBox

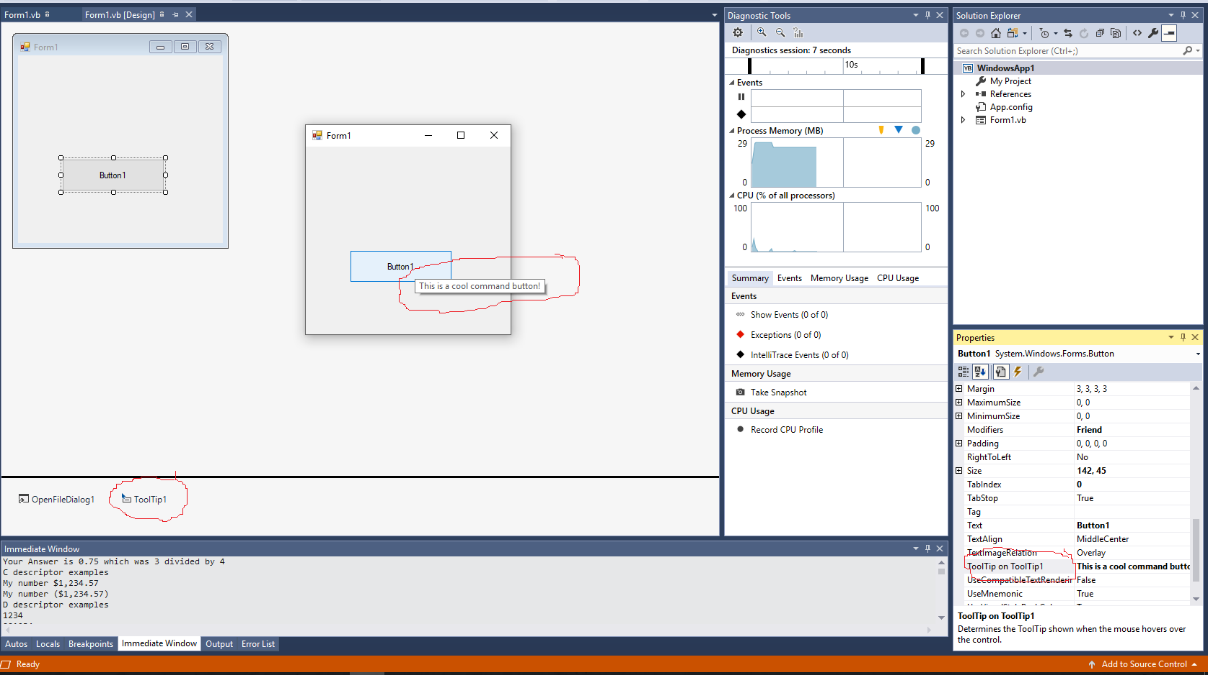
This control allows a user to display almost any kind of image on a form.

Additional PictureBox properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Image | Allows the user to enter the name of the image file (.BMP, .GIF, .ICO, .WMF, .JPG, .JPEG, .PNG) to be displayed directly, or by browsing |
| SizeMode | Indicates how to handle displaying the image. There are five styles available whose functions are pretty obvious from the style names: Normal, StretchImage, AutoSize, CenterImage, Zoom |

## ToolTip

This is how you make those neat little yellow sticky notes appear by a control whenever a user pauses his or her mouse on the control. Just like the OpenFileDialog control, the ToopTip is also an invisible control. Drag one to your form and a new icon will appear in the invisible control section. After you have added the ToolTip control to your form, every other control on the form will then have its ToolTip property appear when hovered over with the mouse. You can edit the ToolTip property on the controls that you want to have popup text on. Simply type in whatever text you want to appear in ToolTip and you are done. Here’s an example of the ToolTip at work:



Please note that this discussion of controls is just to get you proficient with enough controls to start building GUIs with a bit more complexity. We will continue to add to the list of the controls that you know over the course of the semester. Likewise, the properties that we talked about here (as well as the events that we will look at in the next section), are skimming the surface. If you want to consider yourself a proficient Windows Forms application developer, then you really need to sit down with the MSDN help and try out each control, studying the myriad of properties and events that are available!

# Understanding Common Events Raised by Controls

Now that we have an understanding of the common properties that can be modified within controls, we need to examine the common events that most controls respond to.

Let's take a second and revisit the idea of an event. Remember that an event is generated (technically, the term is raised) by something within our application. For example, when a user activates a command button, our VB program needs to be notified that the button was pressed so that it can run the corresponding functionality that is supposed to be carried out whenever that particular command button is activated.

Notice that I used the word activated in the above discussion. How many ways are there to activate a particular button? One way would be to move the mouse over the command button and click on it. A second way would be to use the tab key to move on the form to the button and then press the enter key on the keyboard. A third possible way would be to use a hotkey combination if there is one present, for example Alt-F is a hotkey combination that opens the file menu in most applications. The beauty of VB's event architecture is that it doesn't matter how the user got the button to activate, the button will always raise a Click event to say it was activated.

Whenever the Click event is raised, the code that you write within the Click event handler will be executed. In other words, if you click a Save button, then you would write code in the Save event handler's subroutine to actually do the saving.

The concepts of events are critical for building VB applications. You need to write event handlers to respond to all of the events that you want to handle. Remember, you have to write these handlers carefully since you never know when a user will decide to do something that will raise an event. This is much different than in non-graphical languages where you always have programmatic control over what a user can do and when the user can do so.

Let's take a look at the common events that most controls and forms raise.

List of commonly raised events:

|  |  |
| --- | --- |
| *Event* | *Purpose* |
| Change | Indicates that the control's contents have changed |
| Click | Indicates that the user has clicked on the control |
| MouseDown | Indicates that a mouse button was pressed down over the control |
| MouseEnter | Indicates that the mouse has moved over the control |
| MouseHover | Indicates that the mouse has entered the control and is now hovering over the control |
| MouseLeave | Indicates that the mouse was on the control and just left |
| MouseMove | Indicates that the mouse changed position over the control |
| MouseUp | Indicates that a mouse button was released over the control |

By using these events together with the various controls that we just discussed, we now have the ability to build some decent interfaces for many applications. The next few pages will provide some example applications that show how to do this.

# Control Sample Programs

## Control Example 1

In this example, we are going to build a Fahrenheit to Celsius converter that works via scrollbars. If you move either one of the scrollbars, you will automatically convert the current temperature to the other temperature base. Here are the controls that you will need to add to a form along with their name and miscellaneous properties:

Horizontal Scroll Bar

Name: hscCelsius

Minimum: 0

Maximum: 109 (to account for scrollbar’s scroll width)

Horizontal Scroll Bar

Name: hscFahrenheit

Min: 32

Max: 221 (to account for scrollbar’s scroll width)

Label

Name: lblInCelsius

Text: 0

Label2

Name: lblInFahrenheit

Text: 0

The next image shows a picture of how your form should look when it is completed.

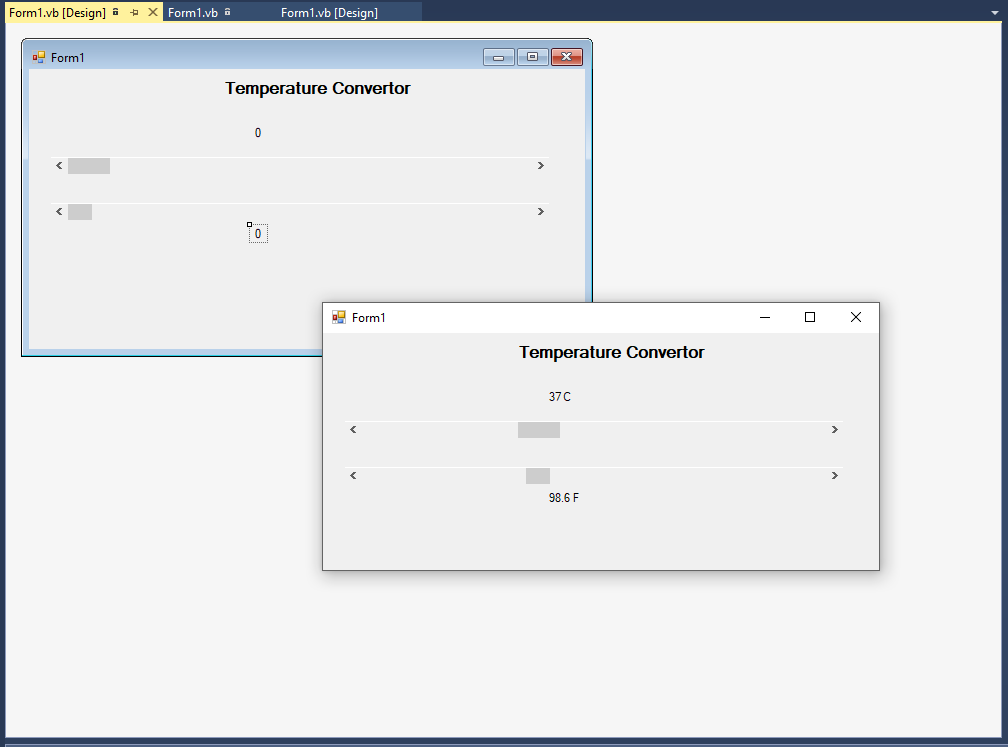


Figure - The temperature converter form

Now that you have the physical form completed, you need to add the following code behind the form:

'Chapter 5 - Program 1

Public Class Form1

Private Sub hscCelsius\_Scroll(sender As Object, e As ScrollEventArgs)

Handles hscCelsius.Scroll

Dim sngFar As Double

sngFar = (CSng(hscCelsius.Value) \* 9.0 / 5.0) + 32.0

hscFahrenheit.Value = CInt(sngFar)

lblInCelsius.Text = hscCelsius.Value & " C"

lblInFahrenheit.Text = sngFar & " F"

End Sub

Private Sub hscFahrenheit\_Scroll(sender As Object, e As ScrollEventArgs)

Handles hscFahrenheit.Scroll

Dim sngCel As Double

sngCel = (CSng(hscFahrenheit.Value) - 32.0) \* (5.0 / 9.0)

hscCelsius.Value = CInt(sngCel)

lblInFahrenheit.Text = hscFahrenheit.Value & " F"

lblInCelsius.Text = sngCel & " C"

End Sub

End Class

Try it out and see what happens when you move either scrollbar!

## 

## Controls Example 2

This example will try out some of the other commonly used controls. We will create an ice cream builder application in which a user can enter his or her name into TextBox txtName, select a flavor (Chocolate, Vanilla or Strawberry) from a ComboBox cboFlavor and select a serving type from a ListBox lstServeAs (Cone, Bowl or Shake).

Once the user has made all of the necessary selections, he or she will press the "Build It!" button cmdBuildIt and a line detailing what the user ordered will be placed in txtOrder. This example teaches you how to interact with listbox and combobox in order to read the values selected by a user. Begin this application by creating the form as shown below:

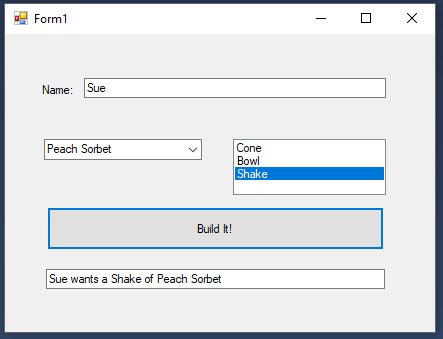


Figure 2 - The ice cream parlor application form detailing a customer’s order

Again, after the user enters his or her name, flavor selection, serving type and presses Build It!, your program should print out a line detailing the user’s order. Figure 2 shows the application in action. Here’s some VB code to help out:

'Chapter 5 - Program 2

Public Class Form1

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

'Handle populating the default combo and listbox choices through Form Load

cboFlavor.Items.Add("Chocolate")

cboFlavor.Items.Add("Vanilla")

cboFlavor.Items.Add("Strawberry")

lstServeAs.Items.Add("Cone")

lstServeAs.Items.Add("Bowl")

lstServeAs.Items.Add("Shake")

End Sub

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

Handles cmdBuildIt.Click

txtOrder.Text = txtName.Text & " wants a " &

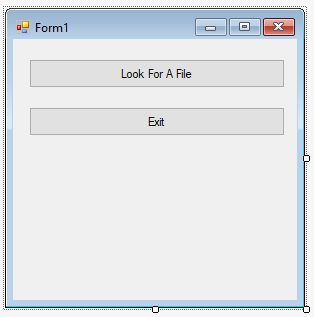
lstServeAs.SelectedItem.ToString & " of " & cboFlavor.Text

End Sub

End Class

## Controls Example 3

Let’s build a file/directory browser by using the OpenFileDialog control. I am going to just use the default name for the OpenFileDialog control in this example. The way that this application will work is that the user will press a button (cmdLookForAFile) to show the FileDialog. Once the user has selected a file, its name will be displayed using a MessageBox. Initially there will also be an Exit button (cmdExit), but you won't be able to see it until the user has looked for a file the first time. We will hide this and disable the form’s X button in the load subroutine. The next screenshot has a sample of what the form should look like.



Here’s the code that you will need to enter into the application:

'Chapter 5 - Program 3

Public Class Form1

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

'Upon form load, shut off the form's control box to hide the minimize,

'maximize and exit button in the form's upper right hand corner

Me.ControlBox = False

'Also hide the Exit command button, so that the user can't end the

'application until a file has been selected.

cmdExit.Visible = False

End Sub

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

Handles cmdExit.Click

'Not really a good idea on ending the application. This is a throwback to

'old school basic -- it does end the program, but doesn't guarantee a nice

'neat orderly shutdown of things!

End

End Sub

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

Handles cmdLookForAFile.Click

'Show the open file dialog box

OpenFileDialog1.ShowDialog()

'Make the exit button visible

cmdExit.Visible = True

'Show the file that the user selected

MessageBox.Show("You picked " & OpenFileDialog1.FileName)

End Sub

End Class

## Controls Example 4

This example is going to look at using one of the controls that we really haven’t talked about yet – the timer. The timer is a very easy way to handle actions that you want performed at specific times or intervals.

Let's create a form that has a textbox that allows the user to enter how many seconds until the timer goes off (txtSeconds). We also need to drop a timer control on the form too (leave the default name of Timer1 alone). Once the timer goes off, we will pop up a quick message with the MessageBox command. We’ll start the timer by pressing the command button (cmdStartTimer). Here’s the application's form:

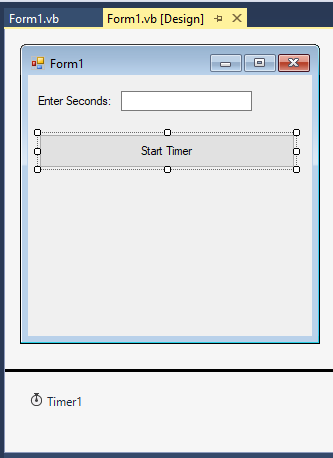


Figure 3 - The form for the timer application

Note: The timer is another one of those “invisible controls” that will not show up on the form when the application is run! It just shows up during design time in the invisible controls tray section of the project…

Additional timer properties:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Enabled | Indicates whether the timer is active or not |
| Interval | Specifies the timer's interval in milliseconds |

To use the timer there are only a couple of things that you need to be aware of. First of all, you have to specify the timer's interval property (how often it is to go off) in milliseconds. That means there are 1000 milliseconds for each second you want it use. Secondly, the timer interval cannot exceed about 65,000 milliseconds (there are ways around this using loop counters). Finally, you have to set the timer’s Enabled property to true for the timer to start working. If you don't shut the timer's Enabled property off, once the timer triggers, it will start counting down again. The bottom line is once the timer's enabled, it will keep triggering infinitely until you disable it or your program ends.

Here’s the code behind the scenes of this application:

'Chapter 5 - Program 4

Public Class Form1

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

Handles cmdStartTimer.Click

'The timer's interval is set in milliseconds, so there are 1000ms in 1s

Timer1.Interval = CSng(txtSeconds.Text) \* 1000

'Turn the timer on

Timer1.Enabled = True

End Sub

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

'This event is called when the timer's interval has elapsed.

'We put in whatever code we want executed here

'Shut off the timer, otherwise as each time interval passes,

'the timer will go off again

Timer1.Enabled = False

'It does matter if the disable is before or after

'the MessageBox. A slow user may not immediately

'press the OK on the MessageBox and so the timer

'interval would elapse and trigger again...we

'shut the timer off and then the user can take

'as long as he/she wants to press the button

MessageBox.Show("Boom! The timer went off...")

End Sub

End Class

# Introductory Error Handling

One of the best features that VB has possessed for a long time is how easy it is to detect and handle errors. Let’s say that we have created a form with a text box on it and I want to convert the contents of the textbox to an integer and then display that result into the second textbox.

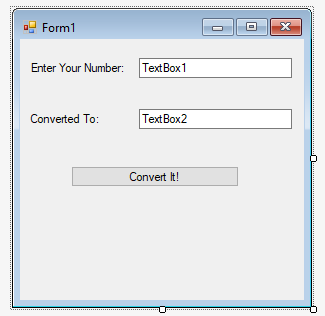


Figure 4 - Sample application that attempts to convert a string into an integer

What will happen if the user enters gibberish into the first textbox? The answer is quite simple: our program will stop executing due to an error since there was no way to convert a value like "KAI#4n" into any kind of a valid number. Since we really don’t like programs that blow up, we need to learn to catch errors ourselves rather than letting the program do it (especially since it didn't do a very good job after all).

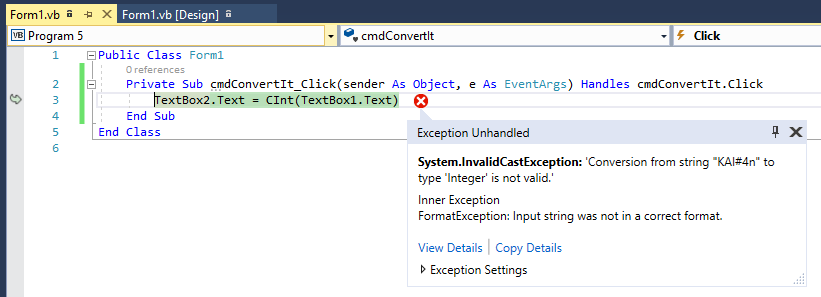


Figure - The result of when an error arises in the application

Fortunately, anytime that an error occurs in VB, we can utilize an exception handler which receives information about an exception that occurs. By using exception handling correctly, we can gracefully handle just about any unexpected error in our programs. Let’s look at the corrected code where the conversion is carried out. Pay special attention since we are performing the error handling ourselves now so the program won’t crash as it did before.

'Chapter 5 - Program 5

Public Class Form1

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

Handles cmdConvertIt.Click

'Prepare for an error to possibly happen...

Try

'If the next line blows up, we catch it and handle it with an

'exception handler

TextBox2.Text = CInt(TextBox1.Text)

'If there was no exception, TextBox2 now contains the converted number

'and we skip to the End Try line

Catch Ex As Exception

'If we caught an exception, here's the code to execute in response to

'it

MessageBox.Show("You Must Enter a Valid Number!")

TextBox1.Text = "0"

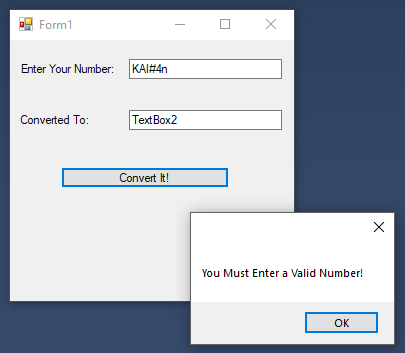
TextBox2.Text = ""

End Try

End Sub

End Class

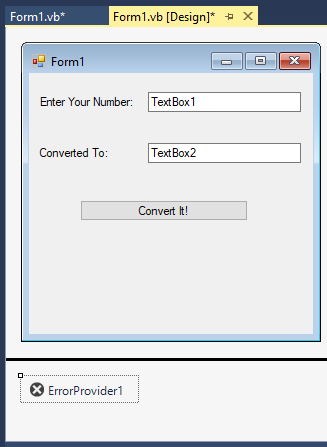
Here’s what happens now when we run the program and the user types in junk:



Notice that we properly detected the error and handled it before things crashed. This is one of the areas that makes a good program "good!" Users have come to expect programs to behave themselves and any program that you write today needs to take responsibility for cleaning up its own mess. We'll look at structured exception handling in a couple chapters from now. I just wanted to show you a quick way of tightening up your programs and taking away some of the issues of program crashes.

## Using the ErrorProvider Control

While we certainly did trap the error, a much better way to handle this would to be to flag the user on the control that has bad data itself, e.g. don’t let the user leave the control with junk data in place. To handle this, we are going to incorporate two new ideas: 1) we will drag and drop an ErrorProvider control to the form (which shows up at the bottom of the design area) and 2) we will tap into the TextBox1’s Validating event:



Now we simply need to add a bit of code:

'Chapter 5 - Program 6

Imports System.ComponentModel

Public Class Form1

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

Handles TextBox1.Validating

'Unless we disable it, all textboxes are

'validated -- so when the user leaves the

'textbox this fires

'Let's see if the junk in the textbox is

'something than can be converted to a number

If Not (IsNumeric(TextBox1.Text)) Then

'If not, set the ErrorProvider to point

'to the control in trouble and provide a

'message if the user pauses over the icon

ErrorProvider1.SetError(Me.TextBox1, "Must be numeric")

'Since we couldn't validate the data to a

'number, get out of here by cancelling the

'event

e.Cancel = True

Else

'If we made it here, we have a valid number

'so just make sure any old providers are

'cleared

ErrorProvider1.SetError(Me.TextBox1, Nothing)

End If

End Sub

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

Handles cmdConvertIt.Click

'No error can occur because we don't leave

'the validation until we get a valid number

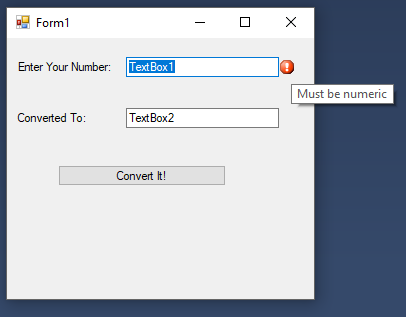
'in the textbox!

TextBox2.Text = CSng(TextBox1.Text)

End Sub

End Class

Here’s the application running:



So by placing in some error providers and validating controls as we leave them, we are able to cut down on exception handling, which ultimately speeds things up!