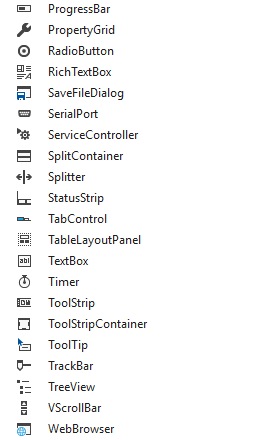
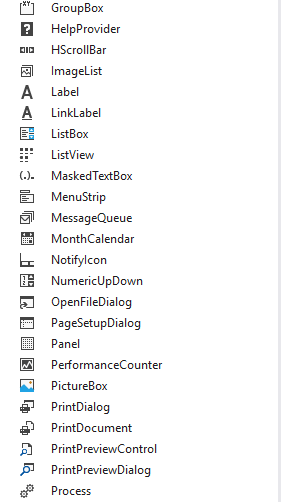
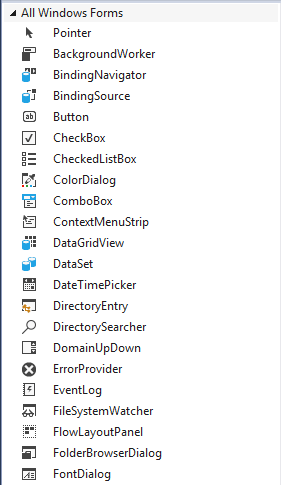
Chapter 3 – More on Forms Based Applications

# Objectives

* Introducing Form Controls
* Application Startup Control
* Application Termination

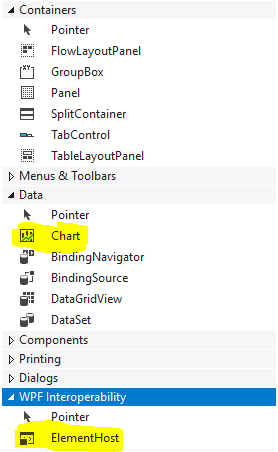
# Controls

So the last chapter took a look at the minimal controls that you should be aware of, but there is a ton more functionality provided right out of the box by the control sets that we have access to. We’ll take a look at each of the standard controls and what they are for:

By the pictures and descriptions, you probably get a pretty good gist of what each control is used for. The next table lists each control and its respective function. There’s a lot here, but don’t freak out, we’ll cover them as we need them over the semester – I just want you to have some idea about what is there.

|  |  |
| --- | --- |
| **Control** | **Purpose** |
| Pointer | For selecting controls placed on a form |
| Background Worker | Executes an operation on another thread |
| BindingNavigator | User interface for moving through data bound controls |
| BindingSource | Hooks a data source to a form |
| Button | Command button usually used for things like OK or CANCEL |
| CheckBox | Two state item which can be checked or unchecked |
| CheckedListBox | Similar to a listbox but to where each item can be selected by being checked/unchecked |
| ColorDialog | Used to allow the user to select a color |
| ComboBox | Similar to a listbox but allows the user to type in items that do not appear in the list |
| ContextMenuStrip | Used to create a menu that will appear when the user right clicks |
| DataGridView | Used to show the contents of a database’s table |
| DataSet | Holds an in-memory cache of data |
| DateTimePicker | A control that allows a user to select a date and time |
| DirectoryEntry | Holds an item from an Active Directory |
| DirectorySearcher | Performs queries against Active Directory |
| DomainUpDown | Allows a user to scroll through a group of related items |
| ErrorProvider | Used to add error text to controls on a form |
| EventLog | Allows application interaction with Windows logs |
| FileSystemWatcher | Watches for changes to files or directories |
| FlowLayoutPanel | Automatically arranges components in a flow layout |
| FolderBrowserDialog | Allows the user to select a folder |
| FontDialog | Used to allow the user to select a font |
| GroupBox | Used to combine a set of radio buttons together in which only one item in the group box can be selected |
| HelpProvider | Used to add help text to controls on a form |
| HscrollBar | A horizontal scrollbar |
| ImageList | A way to store and manage multiple images |
| Label | Typically used for displaying static text |
| LinkLabel | Used to link from one item to another, typically in the context of web pages |
| ListBox | Provides a way to allow a user to select an item from a list of items |
| ListView | Allows items to be displayed in a list – think of the Windows Explorer in which items are displayed as an iconic list |
| MaskedTextBox | User data must conform to the mask presented |
| MenuStrip | Allows users to add a menu to a form |
| MessageQueue | Provides access to a Message Queue server |
| MonthCalendar | A control that allows a user to pick a date off of a calendar |
| NotifyIcon | Used to indicate when the status of some item has changed |
| NumericUpDown | Allows a user to scroll through a range of numbers |
| OpenFileDialog | Used to allow the user to navigate and select a file to open |
| PageSetupDialog | Used to set up how a document will be printed |
| Panel | Used to create work areas of other controls within a form |
| PerformanceCounter | A windows performance counter entity |
| PictureBox | Used to display images of various formats such as .GIF, .JPG, .PNG and .WMF |
| PrintDialog | Used to allow the user to set up a print job |
| PrintDocument | Adds a control that allows a document to be printed |
| PrintPreviewControl | Provides the functionality to preview how a document will look when it is printed |
| PrintPreviewDialog | Used to allow the user to interact with the PrintPreviewControl |
| Process | Access to all processes; ability to start and stop local processes |
| ProgressBar | Used as a visual depiction of the amount of progress completed in some operation |
| PropertyGrid | Provides a user interface to work with the properties of some object |
| RadioButton | Usually used as a series in which only one of a radio button set can be selected at a time |
| RichTextBox | A powerful text editing control |
| SaveFileDialog | Used to allow the user to navigate and name a file to save to |
| SerialPort | Access to a serial port |
| ServiceController | Ability to connect, query and manipulate services |
| SplitContainer | Divides a work area into two resizable partitions |
| Splitter | A control that allows a form to be split into multiple sections where the amount apportioned to each part of the splitter can be dynamically resized |
| StatusStrip | Adds a status line at the bottom of an application |
| TabControl | Allows a user to create a form that has tabbed pages on it – typically used for option pages |
| TableLayoutPanel | Arranges components automatically in a table form |
| TextBox | Allows users to type in alphanumeric text |
| Timer | A control that will count down a specified amount of time |
| ToolStrip | Used to create a graphical toolbar within an application |
| ToolStripContainer | A panel container for one or more ToolStrips, MenuStrips, StatusStrips |
| ToolTip | Used to add popup information when the user places the mouse over a control for a few seconds |
| TrackBar | Used to allow a user to select preset values from a range of values with specific set detents – think of the volume control |
| TreeView | Allows a hierarchy of items to be displayed – think of the Windows Explorer where drives and directories are shown as trees |
| VScrollBar | A vertical scrollbar |
| WebBrowser | Access to a web browser component |

So we’ve seen pretty much the full gamut of what Visual Studio offers – and the list is overwhelming. VS also breaks things down into categories and so if we look at some of the more esoteric categories, you start to see that a bunch of these items may not be controls you need in every application you write. I highlighted a few controls that don’t show up immediately in the list above.

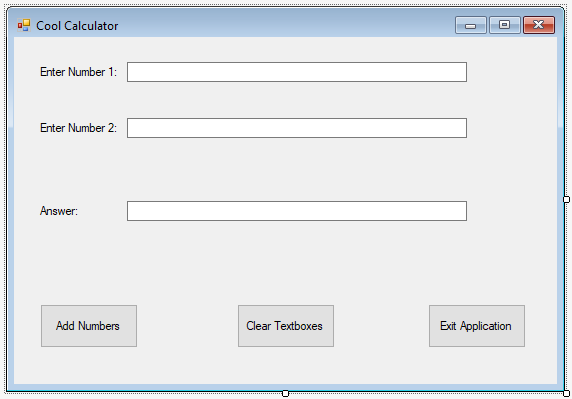


Now just so you know – the data controls allow you to work with could be just about any kind that you can think of: Access, SQL Server, Informix, Oracle and so on. Every major database system today provides add-in support so that the database can be accessed externally by a development environment such as .NET.

Some people wonder why I don’t teach WPF development instead of Windows Forms. There are two reasons: (1) The huge learning curve in getting WPF down and (2) the lack of having so many built in controls. VB was the first RAD for a reason and Microsoft seems to have forgotten that when it rolled out its WPF control line up!

# Application Startup Control

Let’s review the basic application process by writing a simple calculator that takes two numbers and adds them together when the user directs the application to. Here’s the GUI:



The code behind it ought to be pretty familiar to you:

Public Class frmCoolCalculator

'Chapter 3 - Program 1

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

cmdAddNumbers.Click

txtAnswer.Text = CStr(CDbl(txtNumber1.Text) + CDbl(txtNumber2.Text))

End Sub

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

cmdClear.Click

txtAnswer.Text = "0"

txtNumber1.Text = "0"

txtNumber2.Text = "0"

End Sub

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

cmdExit.Click

Me.Close()

End Sub

End Class

So if we run our application at this point, it fires up allowing the user to type a number into each of the upper two textboxes. The user then presses the Add Numbers command button to get a sum. Clear Boxes will, well, clear the three textboxes and Exit Application shuts everything down.

This only thing that might seem a bit bizarre is the Exit command button event handler. It is pretty straightforward after you understand the Me keyword. If you’ve worked with Java, the “this” keyword allows an object to refer to itself. The Me keyword does the same thing in VB.

In the case of the Exit handler, the Me refers to the container in which the event handler lives. That container is the form – the Close method tells the form to close itself, which will end the application since the only form that makes up the application is no longer available.

In terms of starting the form up, we might like to see it automatically initialize the three textboxes all to 0s. We can handle this through the Load event which is called once whenever the form is first brought into memory:

Private Sub frmCoolCalculator\_Load(sender As Object, e As EventArgs) Handles

Me.Load

txtAnswer.Text = "0"

txtNumber1.Text = "0"

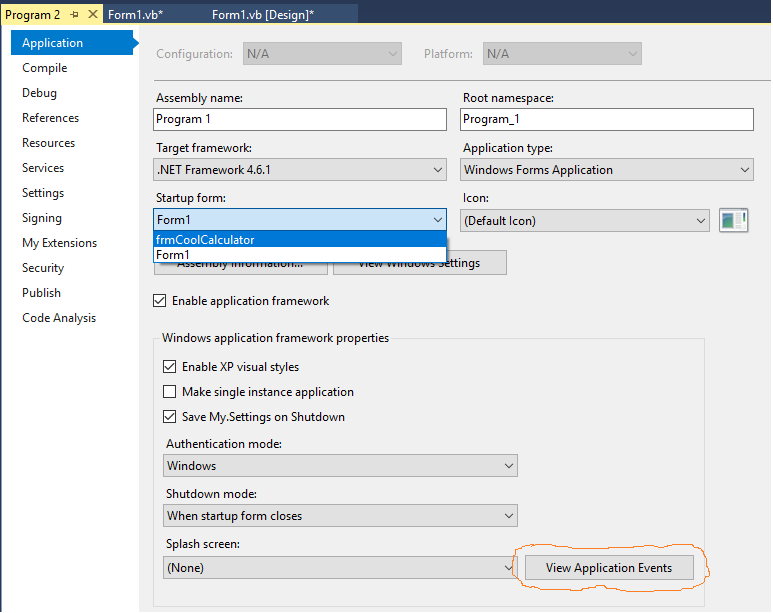
txtNumber2.Text = "0"

End Sub

The Load event is a really convenient way to set up a form the way that we want it to look just before it is rendered to the user. If we run the application again, we see that the textboxes are all nicely cleaned out to our initial 0 values.

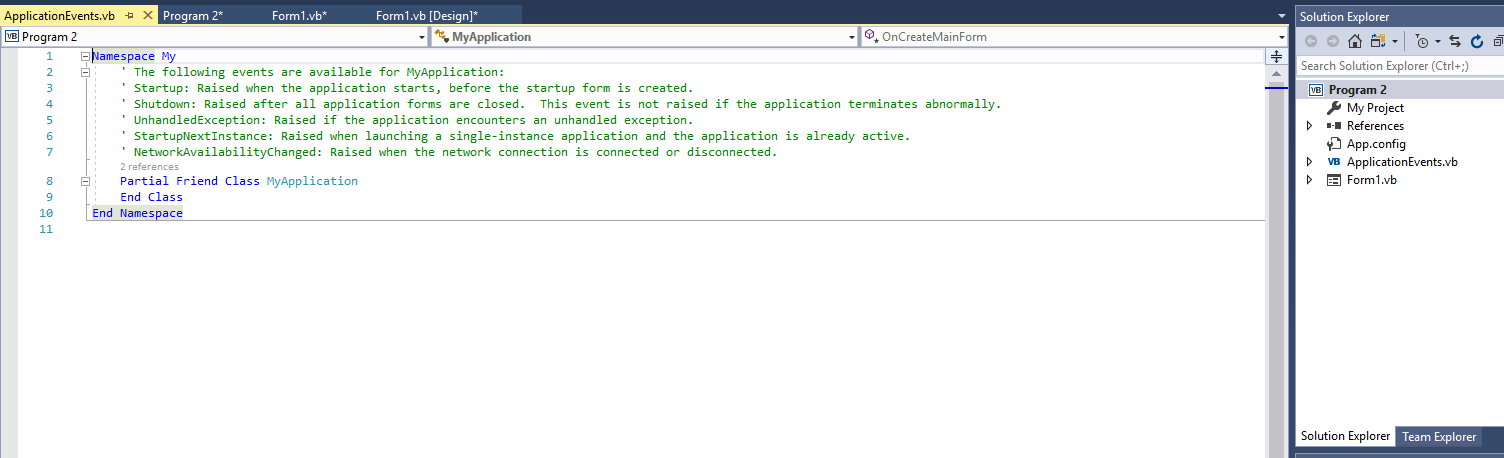
So life is pretty peachy at this point, unless we wanted to do some set up work before the application starts or some clean up work after the application ends. We don’t have any control to do that! Think about it – VB automatically throws this form in your face and said “There you go.” You didn’t get to ask for it and you certainly didn’t get to do anything before it showed up. For those of you thinking about the Form\_Load event handler – it still really doesn’t matter – at that point the form has been instantiated and is in control. Yikes! Think back to other languages: what did you do there? You always had control from the time your program started until it ended with the use of the main() function…

What can we do? Let’s take a look at the Application tab in the Project’s properties window (from last chapter: right click on the Project name in the Solution Explorer window and select Properties):

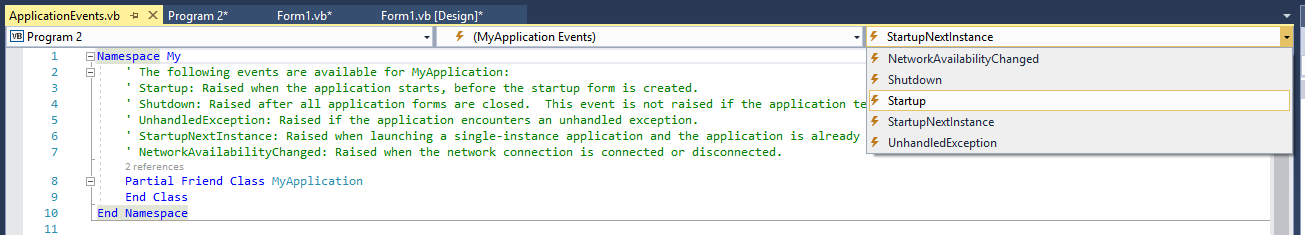


There is a section where we can select the Startup form which might be useful if we had more than one form. In this application, it ain’t going to matter because there’s only one… But we have come to the right place. There is a button toward the bottom of the screen that says View Application Events. Let’s click on that critter!

We see the following screen:



It may not look like much happened: we were taken to new text editor screen and an ApplicationEvents.vb file was added to the solution, but tons have changed. We now have access to starting and stopping the application ourselves. In the editor’s events pulldowns, change the middle dropdown to (MyApplicationEvents) and then on the right dropdown select Startup:



Add the following MessageBox line to the event handler that was written for you:

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

ApplicationServices.StartupEventArgs) Handles Me.Startup

MessageBox.Show("Start up")

End Sub

Now run your project and see what happens…the MessageBox shows up and then my form shows up. Let’s go back and add a Shutdown handler in the same way:

Private Sub MyApplication\_Shutdown(sender As Object, e As EventArgs)

Handles Me.Shutdown

MessageBox.Show("Shut down")

End Sub

Run it again! Full control – we can do things before the main form shows (like initialize database connections over a network) and then do things before the application ends (like clean up any files we have open, write logs, et cetera). With a little ingenuity, we can also get the application to pick which form we want it show, by modifying the MyApplication\_Startup event:

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

ApplicationServices.StartupEventArgs) Handles Me.Startup

MessageBox.Show("Start up")

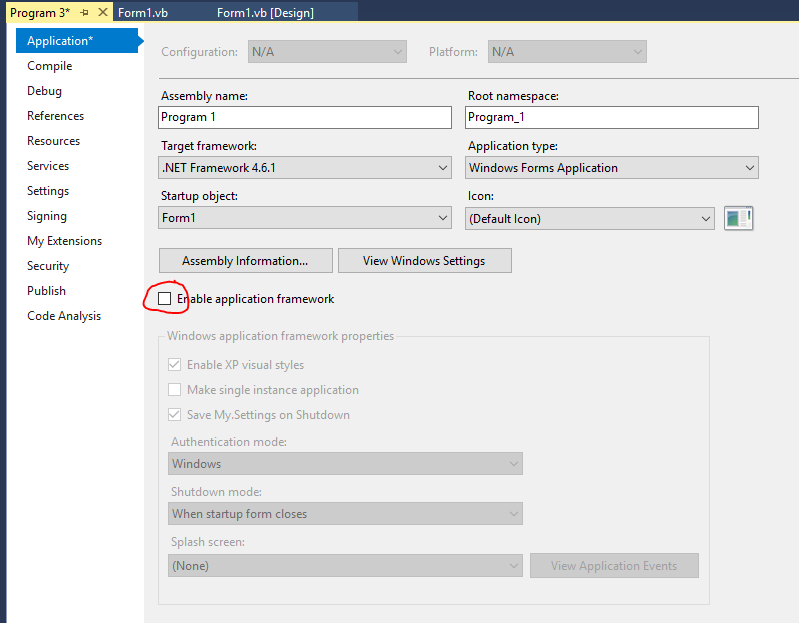
My.Application.MainForm = frmCoolCalculator

End Sub

So if we had multiple forms that we created, we could pick which one to run as the “main” form. Slick and easy.

## Startup Control Version 2

While the example I showed above is the more procedurally correct way to do things, there is a second option and that is where we write our own Sub Main routine that handles everything. To do this, we need to do two things: 1) delete the ApplicationEvents.vb file and 2) Uncheck the Enable Application Framework checkbox on the Project Properties window:



You can see that all of the goodies in the bottom half of the screen, like View Applications Events are gone now. So why do this? Converting some old-school VB projects may have been very dependent on Sub Main to the point that porting it to the Application Events style is traumatic.

Now what we need to do is go under Project🡪Add Module and call it modMain.vb. Once we have the code editor back, add the following code:

Module modMain

Sub Main()

'Chapter 3 - Program 3

MessageBox.Show("Main is in control")

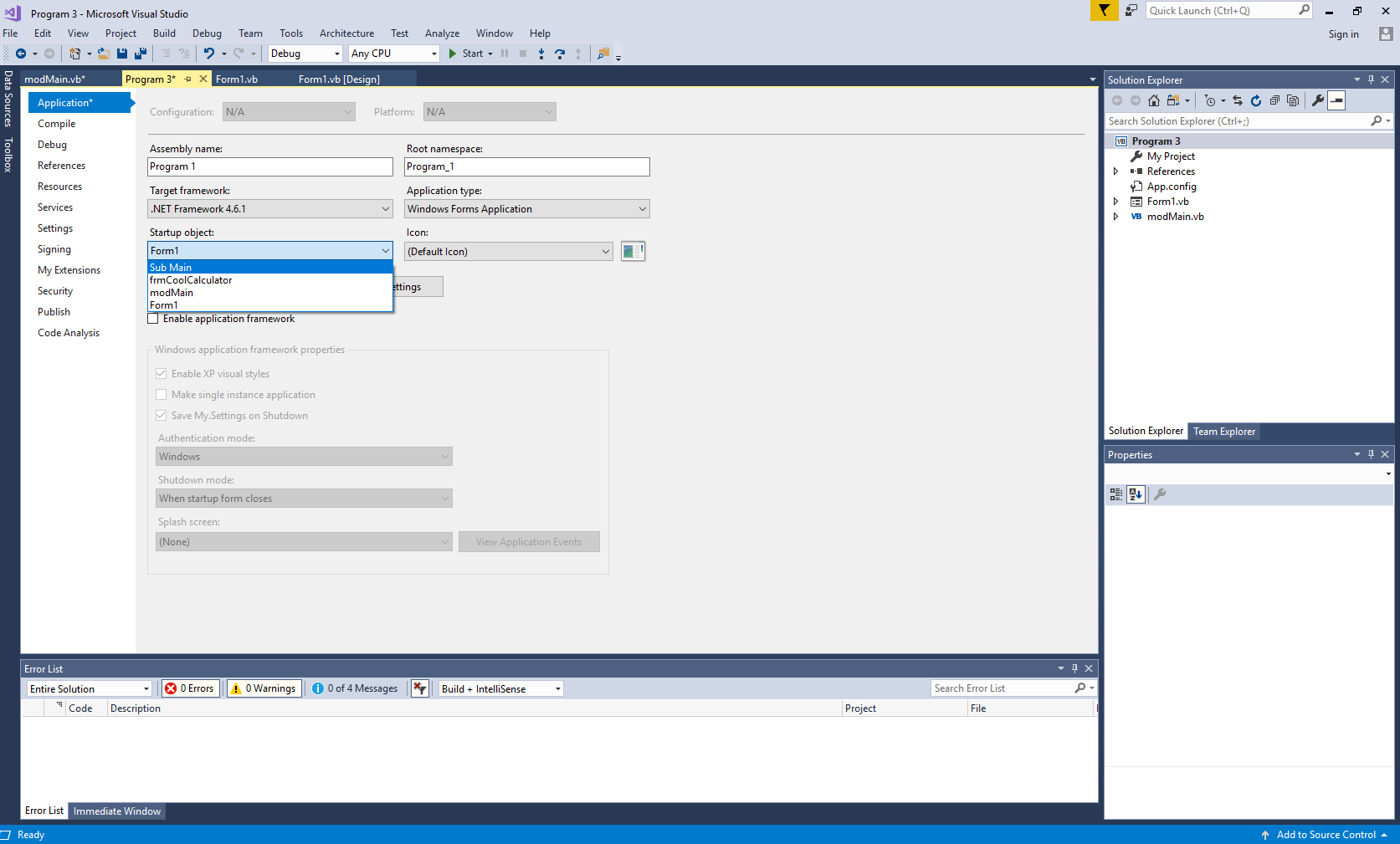
frmCoolCalculator.ShowDialog()

MessageBox.Show("Main is about to exit")

End Sub

End Module

We just have to tell our application to run Sub Main instead of the main form and we’ll have control again…



Now run the application…any way you slice it, you have the ability to get things running and set up the way you want. You’re in full control of the application’s start up process now, which is the way it should be!

# Application Termination

If there’s some magic to starting an application is there something to shutting it down too? Going back to our first version of the program (one form – no Sub Main – no ApplicationEvents), I am going to modify the Exit button’s click event handler to throw up a MessageBox before the application ends:

'Chapter 3 - Program 4

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

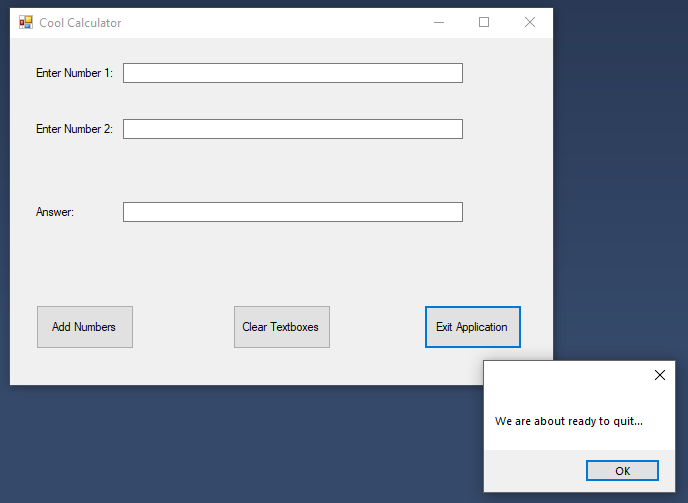
cmdExit.Click

MessageBox.Show("We are about ready to quit...")

Me.Close()

End Sub

Sure enough if I click on the Exit Application button, I see the MessageBox before the application closes down…



But, if I run the application again and press the X in the upper right hand corner of the application, the application just stops! What happened to my MessageBox? The problem lies in the fact that the form can be closed in multiple different ways and we need to chain into the common event handler so that no matter how the user chooses to close the application, it is consistently executing the same code body.

There is an event handler attached to the form called FormClosing. I am going to make this slight code modification to cmdExit\_Click first through:

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

cmdExit.Click

'Remove the MessageBox.Show from here

Me.Close()

End Sub

Next, I’ll add this handler (again using the chevron’s at the top of the code editor window to build the skeleton code for me):

Private Sub frmCoolCalculator\_FormClosing(sender As Object, e As

FormClosingEventArgs) Handles Me.FormClosing

MessageBox.Show("We are about ready to quit...")

End Sub

Now, no matter how I run the application, whenever the form receives a Close event (which both the X and the Exit button generate), the same common code base is run. We can even get a bit fancier asking the user if he/she really wants to close the application down:

Private Sub frmCoolCalculator\_FormClosing(sender As Object, e As

FormClosingEventArgs) Handles Me.FormClosing

'Chapter 3 - Program 5

If MessageBox.Show("Are you sure you want to close the application down?",

"Cool Calculator",

MessageBoxButtons.YesNo) = DialogResult.No Then

e.Cancel = True

End If

End Sub

This makes our application look a whole lot more polished…we are prompted whether or not we want to quit. If the user selects No, then we tell the FormClosing event to cancel the closing process. There is a parameter sent to this handler of type FormClosingEventArgs and inside of that parameter, there is a property called Cancel. If Cancel is set to true inside of the event handler, then the closing process is aborted.

