Chapter 11 – Form Development Details

# Objectives

* Forms
* Working with Menus
* Building More Forms
* MDI Forms
* Understanding the OpenForms, MDIChildren and Controls collections

This chapter is going to provide a more detailed coverage of forms' properties, methods and events. In addition, the discussion will examine how to create MDI (multiple document interface) applications in which a single application can have multiple child windows attached to it.

We will also look at how to build various menu types and you will learn how to respond to the menu choices that have been selected by a user. Menus are one of the concepts that almost all programs must implement in order to provide a user with a common interface.

Finally, the chapter will end with an examination of the OpenForms and MDIChildren collections, which allow us to access and iterate across the forms that make up an application.

# Forms

The form is the most basic device that you can use to provide a graphical user interface to an application's user. We have already worked through examples of dragging and dropping controls onto a form. We have also looked at the common events that most controls will raise and we have learned how to respond to them through code. This section will examine the Form object itself and what can be done with it.

As with controls, there are many properties, methods and events available to a form. Let's begin by spending some time examining what's available.

Common Form Properties:

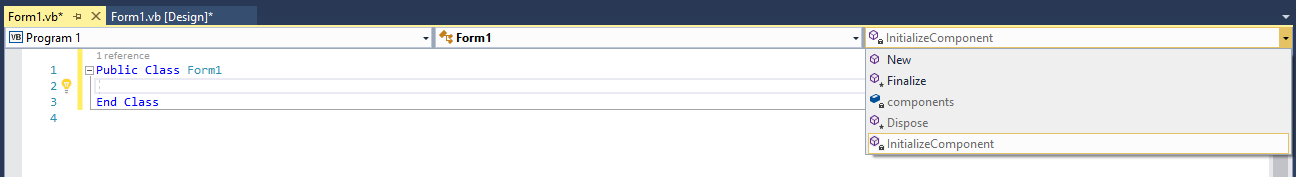
|  |  |
| --- | --- |
| *Property* | *Purpose* |
| AcceptButton | Gets or sets the button on the form that is clicked when the user presses the ENTER key |
| ActiveControl | Gets or sets the active control of the form |
| ActiveForm | Gets or sets this form as the currently active form for the application |
| AllowTransparency | Gets or sets a value indicating whether the opacity of the form can be adjusted |
| AutoScroll | Gets or sets a value indicating whether the form enables autoscrolling |
| AutoSize | Resize the form according to the setting of the AutoSizeMode property |
| AutoValidate | Gets or sets a value that indicates whether controls in this form will be automatically validated when the focus changes |
| BackColor | Determines the background color of the form |
| BackgroundImage | Gets or sets the background image of the form |
| Bottom | Distance in pixels between bottom edge of the form and the top edge of its container’s client area |
| Bounds | Gets or sets the size and location of the form |
| CancelButton | Gets or sets the button control that is clicked when the user presses the ESC key |
| CausesValidation | Gets or sets a value indicating whether the form causes validation to be performed on any control that requires validation when it gets focus |
| ContextMenu | Gets or sets the shortcut menu (context menu) associated with the form |
| ContextMenuStrip | Gets or sets the ContextMenuStrip associated with this control |
| ControlBox | If set to true (default), the control menu in the upper left corner of the title bar is enabled |
| Controls | Collection of all of the controls attached to this form |
| Cursor | Gets or sets the cursor that is displayed when the mouse pointer is moved over the form |
| DefaultCursor | Gets or sets the default cursor for the form |
| DialogResult | Gets or sets the dialog result for the form (think MessageBox) |
| Enabled | Indicates whether a user can interact with the form |
| Events | Gets the list of event handlers attached to this form |
| Focused | Gets a value indicating whether the form has input focus |
| Font | Gets or sets the font of the text displayed by the form |
| ForeColor | Gets or sets the foreground color of the form |
| FormBorderStyle | Allows the user to select how the border of the form behaves |
| HasChildren | Gets a value indicating whether form contains child forms |
| Height | Gets or sets the height of the form |
| HScroll | Gets or sets a value indicating whether the horizontal scroll bar is visible |
| Icon | Gets or sets the icon for the form |
| IsMdiChild | Gets or sets a value indicating whether the form is an MDI child form |
| IsMdiContainer | Gets or sets a value indicating whether the form is a container for MDI children |
| Left | Gets or sets the distance in pixels between the left edge of the form and the left edge of the container’s client area |
| Location | Gets or sets the Point that represents the upper-left corner of the form on the screen |
| MainMenuStrip | Gets or sets the primary menu container for the form |
| Margin | Gets or sets the space between the controls on the form |
| MaximizeBox, MinimizeBox | If set to true (default), the Maximize and Minimize buttons are shown in the upper right hand corner of the title bar |
| MdiChildren | Gets the forms the are MDI children to this parent form |
| MdiParent | Gets or sets the current MDI parent form of this form |
| Menu | Gets or sets the MainMenu that is displayed on the form |
| MergedMenu | Gets the merged menu for the form |
| Modal | Gets a value indicating whether this form is displayed modally (Show vs. ShowDialog) |
| Name | Gets or sets the name of the form |
| Opacity | Gets or sets the opacity level for the form |
| OwnedForms | Gets an array of Form objects that represent all forms that are owned by this form |
| Owner | Gets or sets the form that owns this form |
| Padding | Gets or sets padding within the form |
| Parent | Gets or sets the parent container of the control |
| ParentForm | Gets the form that the container control is assigned to |
| ResizeRedraw | Gets or sets a value indicating whether the control redraws itself when resized |
| Right | Gets the distance in pixels between the right edge of the form and the left edge of its container’s client area. |
| ScaleChildren | Gets a value that determines the scaling of child controls |
| ShowIcon | Gets or sets a value indicating whether an icon is displayed in the caption bar of the form |
| ShowInTaskbar | Gets or sets a value indicating whether the form is displayed in the Windows taskbar |
| Size | Gets or sets the size of the form |
| SizeGripStyle | Gets or sets the style of the size grip to display in the lower-right corner of the form |
| StartPosition | Where the form should be located when it is first displayed |
| Text | Caption for the form |
| Top | Gets or sets the distance between the top edge of the form and the top edge of its container’s client area |
| TopLevel | Gets or sets a value indicating whether to display the form as a top-level window |
| TopLevelControl | Gets the parent control that is not parented by another Windows forms control |
| TopMost | Gets or sets a value indicating whether the form should be displayed as a topmost form |
| TransparencyKey | Gets or sets the color that will represent transparent areas of the form |
| UseWaitCursor | Gets or sets a value indicating whether to use the wait cursor |
| Visible | Indicates whether the form is visible or not (this property is only available at runtime) |
| VScroll | Gets or sets a value indicating whether the vertical scroll bar is visible |
| Width | Gets or sets the width of the form |
| WindowState | Indicates the initial state of the form (normal, minimized or maximized) |

Common Form Methods:

|  |  |
| --- | --- |
| Method | Purpose |
| Activate | Activates the form and gives it focus |
| ActivateMdiChild | Activates the specified MDI child |
| CenterToParent | Centers the form within its container |
| CenterToScreen | Centers the form within the current screen resolution |
| Close | Closes the form |
| CreateGraphics | Returns a graphics context that can be used to draw on the form |
| CreateHandle | Returns an O/S handle used to refer to the form |
| FindForm | Retrieves the named form |
| Focus | Sets input focus to the form |
| Hide | Hides a form which was shown, but does not unload it from memory |
| LayoutMdi | Specifies how to arrange the child forms within the MDI parent form |
| Refresh | Repaints a form |
| Select | Activates the form |
| Show | Forces a form to be displayed – if the form is not in memory, it will be loaded from disk first. The user may jump between forms |
| ShowDialog | Forces a form to be displayed (loaded first, if necessary). ShowDialog forces the form to be displayed modally, in other words, the user has to interact with the form displayed and cannot work with any other forms while this remains shown |

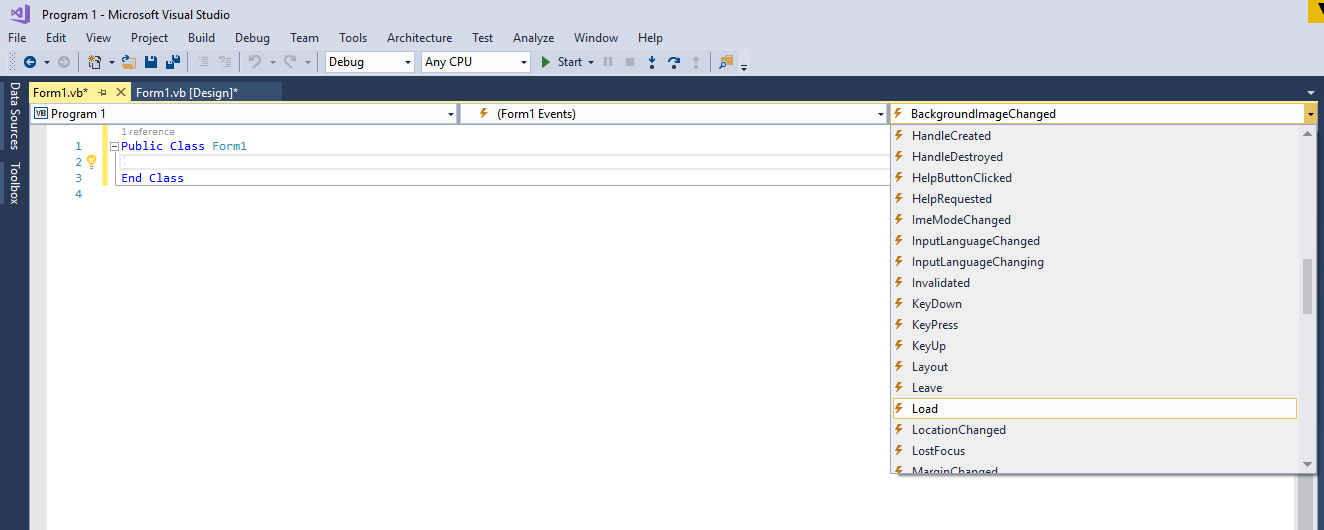
There are also many events that you can have a form respond to. One that is very useful is the Load event that gets called whenever a form is initially loaded. As we’ve previously mentioned, the Load event is often a very convenient place to put in initialization work that you need done any time that a form is instantiated.

As you may have already discovered, just about every event is available from within VB's code editor window. There is a list of our forms and controls in the middle dropdown box immediately above the code window. To the right of the object list, is the event list for that particular object. If we click on one of the events, the skeleton event handle is added to the code window for us.



Whenever I start a new form, however, and I jump over to the code window, I can’t see many of the events. In fact, most of what shows up looks like events pertaining to the construction and destruction of the form. One things for sure: our Load event ain’t there! Take another look at the list of events in the right dropdown in the previous screenshot.

To access this form’s events, I simply selected the proper form in the middle dropdown, and under Form1, I selected (Form1 Events). Having set that choice in the middle dropdown, I can now go to the right-hand dropdown box to pick the event I am interested in. This is a list of every event that the base class has available to it. You might be wondering what the base class is to the form that you are working on. Quite simply it's "Form", the prototype to every form instance we derive. You can see that the Load event is now available over in the right-hand dropdown box:



Whenever you build a Form in Visual Basic, there is a bunch of code being written on your behalf. While we likey the draggy-droppy, think about this – everything must eventually be boiled down to code that can be compiled. So, if we look into the bowels of the form generator code written for you (remember you just drag and drop controls from the toolbox to create your form), we see something like this at the top:

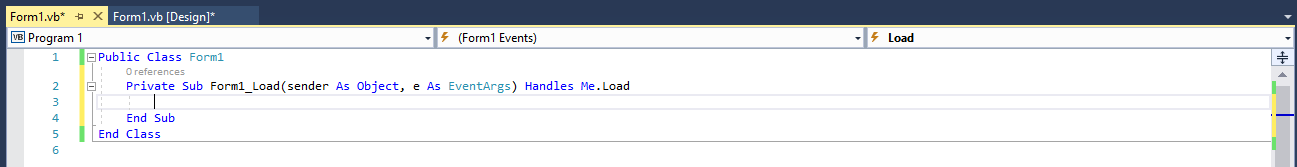
Partial Class Form1

Inherits System.Windows.Forms.Form

This shows you that "Form1" in this case is based on the granddaddy of all forms, System.Windows.Forms.Form. Therefore, all that we've done by accessing the base class events is told VB.NET that there are some events that the granddaddy Form class has that we want to use in our particular form instance.

Now don’t worry about the fact that you aren’t seeing the lines of code that I showed above. You won’t find them in your normal code editor window. I will show you later on how I accessed this information (and then you’ll probably wish that I hadn’t…). But trust me, they are there.

For now, go ahead and click on the Load event handler in the right-hand dropdown and you will see that VB writes the skeleton subroutine event handler for you:



Now that the load event handler is available to us, we could write any code that we want to run each time an instance of this form type gets loaded. You can basically equate a Load event firing whenever a form instance gets created. Another item to note is that the Load event in the right-hand box is now displayed in bold text. VB does this to any event which has an event handler in the code editor – this is to make it easy for you to see which event handlers you may have already either implemented, or at least have the skeleton handler generated.

We have also seen that there are many ways to indicate that the application is supposed to end. We really don't want our program to have “ending” code that's located in several spots. Good programming design dictates that there ought to be a single entrance point for an application and a single exit point. This isn't to say that you can't have several different ways for the user to quit an application, it just means that you need to funnel each way through one common set of code.

Remember that we have already solved this problem in a past chapter – I’m just throwing it back in here to refresh your memory. We want every exit point to simply tell the form Me.Close() and then throw our exit handling code into the FormClosing event:

Public Class Form1

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

Handles cmdExit.Click

'Don't check about closing code here -- just call the common Closing event

'sequence and write your closing code there instead!

Me.Close()

End Sub

Private Sub Form1\_FormClosing(sender As Object, e As FormClosingEventArgs)

Handles Me.FormClosing

'This will be called anytime the form is told to close. Ask the user if

'he/she is sure about closing. If the answer is no, cancel the closing

'event...

If MessageBox.Show("Are you sure you want to quit?", "",

MessageBoxButtons.YesNo) = DialogResult.No Then

e.Cancel = True

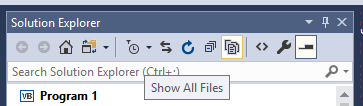
End If

End Sub

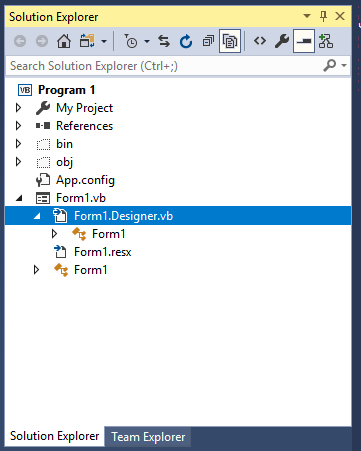
End Class

Now the user can click on the "X" or the “End” command button and he or she will be consistently provided with the same dialog box. We could then write whatever kind of clean up code we want since both ways to exit the application are not funneled through a common event handler, FormClosing.

By the way, in case you are interested in the code that the form designer wrote for you, here’s how you can take a peek – do not alter anything or you’ll break the ability for the designer to be able to edit that code! First of all, show all of the files that make up your solution, by clicking on the Show All Files icon in the icon row about the Solution Explorer:



Next expand the Form1.vb set and you’ll see Form1.Designer.vb. This is the file that was written as you dragged and dropped by the form designer. Go ahead and double click on that file to see the source code…



Here’s the code that’s there (including the earlier mentioned Partial Class line):

<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> \_

Partial Class Form1

Inherits System.Windows.Forms.Form

'Form overrides dispose to clean up the component list.

<System.Diagnostics.DebuggerNonUserCode()> \_

Protected Overrides Sub Dispose(ByVal disposing As Boolean)

Try

If disposing AndAlso components IsNot Nothing Then

components.Dispose()

End If

Finally

MyBase.Dispose(disposing)

End Try

End Sub

'Required by the Windows Form Designer

Private components As System.ComponentModel.IContainer

'NOTE: The following procedure is required by the Windows Form Designer

'It can be modified using the Windows Form Designer.

'Do not modify it using the code editor.

<System.Diagnostics.DebuggerStepThrough()> \_

Private Sub InitializeComponent()

Me.cmdExit = New System.Windows.Forms.Button()

Me.SuspendLayout()

'

'cmdExit

'

Me.cmdExit.Location = New System.Drawing.Point(75, 58)

Me.cmdExit.Name = "cmdExit"

Me.cmdExit.Size = New System.Drawing.Size(123, 39)

Me.cmdExit.TabIndex = 0

Me.cmdExit.Text = "Exit"

Me.cmdExit.UseVisualStyleBackColor = True

'

'Form1

'

Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)

Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font

Me.ClientSize = New System.Drawing.Size(284, 261)

Me.Controls.Add(Me.cmdExit)

Me.Name = "Form1"

Me.Text = "Form1"

Me.ResumeLayout(False)

End Sub

Friend WithEvents cmdExit As Button

End Class

You can see that a lot of things were written on your behalf! Notice that the command button is created entirely through code, as is the form. When I first started writing Microsoft Windows code back with Windows 2.0, this was all done by you and typing code – there was no drag and drop, nothing graphical. VB was the language that was responsible for introducing this new graphical based development philosophy. Did VB change the way programmers approached Windows coding? You better believe it!

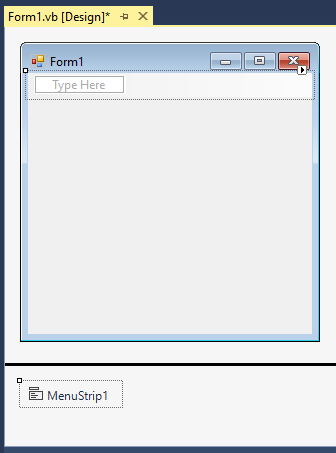
You can close the Form1.Designer.vb window and then click on the Show All Files button again to put things back to normal. I warn you one more time – be careful what you’re doing with that Designer.vb file or you can really mess things up! If you try to edit the contents using the code editor, the designer will not be able to render your form again. Your quickest solution to that problem would be to delete the form and then recreate it from scratch. Not a good thing if the form was pretty complex with lots of controls and code wired to it!

# Working with Menus

Menus are the common way that we allow users to navigate through lots of functionality options. Almost every Windows application has a menu system, so users tend to expect a menu. If I ask you some questions about menus, you'll see how commonplace they have become.

What's the first option on a main menu bar? File. What's the last option? Help. What's just to the left of help? Windows. What's the first option usually under the File menu choice? New. What's the last option under File? Exit. You get the picture.

Visual Basic provides a great facility for building menus. It's very easy to create a menu and pretty simple to respond to a menu option clicked by a user. You can put a menu on any form by dragging a MenuStrip control from the toolbox to the form. The next screenshot has an example of this. A MenuStrip control is placed in the invisible controls tray even though the menu immediately appears on your form:

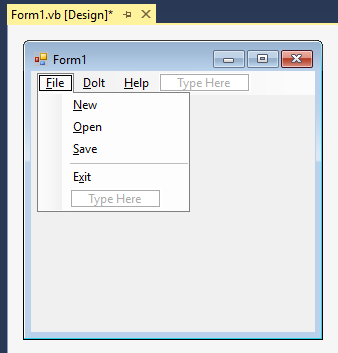


To edit the menu, all you need to do is click on the MenuStrip control you have added and then point and click where you want to add a new menu option and then type in your menu option. You can build up as complex of a menu as you want by doing this… Wherever you see the message "Type Here", this indicates a location in which you can add a menu item. You can typically add items to the right and to the bottom of any other item. As you add menu items, you will see that the number of places where you can new items will automatically increase.

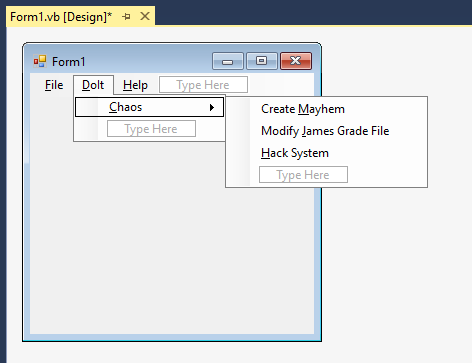
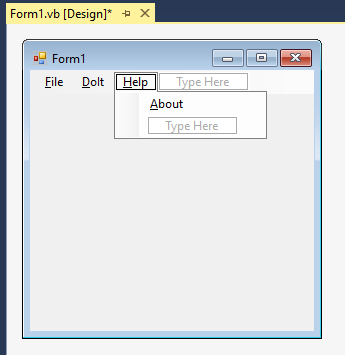
Again, clicking on a "Type Here" location indicates that you are starting to create a menu item. You will enter the menu caption (what’s actually displayed on the menu bar) by simply typing the value in from the keyboard. If you want to make a menu item have an associated hotkey, you place an & in front of the letter that you want to use as that item's hotkey. The most common example is setting the "F" in the File menu option as a hotkey, where if you press Alt-F, the same action is taken as if you pointed and clicked on the menu item using the mouse.

You can also put a separator bar into a menu by typing a single hyphen in as the caption. To create submenus for a particular menu item, simply click to the right or below the menu option that you are currently working on and the submenu will automatically be handled for you.

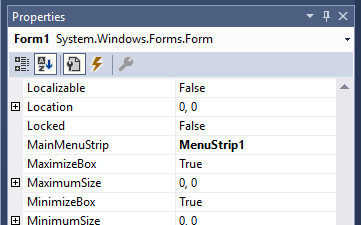
Here’s a simple menu system that I built which contains some submenus and hotkeys.



Here’s our menu after adding some options under the DoIt and Help Menu Structures:

Forms can use different menus, so it is possible to create multiple menus. The MainMenuStrip property on the form will identify which MenuStrip should be used as the main menu. I only have one menu strip, so the only current choice is the default MenuStrip1 that I created:



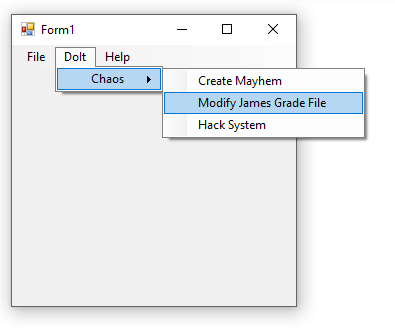
VB automatically assumed that menu should be used when I finished up editing it, but if you run into problems getting menus to work right, check that property out.

Once you have the menu working the way that you want it to, you should go back through and click on each menu item. You'll notice that the Properties window will show the details on that each menu option. By default, the name of each menu item is something like XXXToolStripMenuItem, which while being somewhat descriptive is: A) long and B) not hierarchically nested. Sometimes we like to make the menu options show where they live with regard to the physical MenuStrip structure.

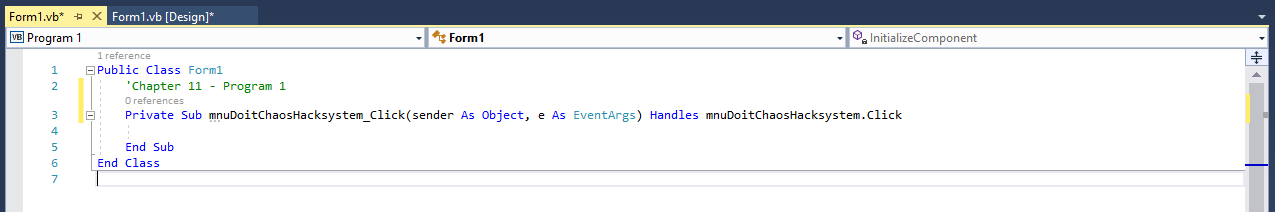
Since we will eventually need to interact with the user clicking on menu items, and we will have to deal with XXXToolStripMenuItem\_Click events, I would strongly recommend renaming the menu option from its default name to something a little shorter and clearer. Following the Hungarian notation scheme we examined in Chapter 3, I would change the name of my File menu option mnuFile, for example. When I have to deal with the event handler for that menu option it will now show up as mnuFile\_Click, which is much clearer than FileToolStripMenuItem\_Click. I would then continue on modifying the names of all of the other menu options. For example, the File🡪New menu option would be renamed to mnuFileNew.

You should get into the habit of modifying these menu names as soon as you’re done building the menu system, but before writing any code. This will make it easy to keep the naming clean and consistent, and, as your skeleton event handlers are generated, the resulting names there will be much nicer in the long run as well.

I would like to take a second and show what the application looks like once I run it. Notice that the menus are "live" and the user can interact with them, even without writing any menu event handlers:



To set up code to be executed for any menu item, all I need to do is double click on any menu option when I am in the code editor. I will then be taken to the skeleton event handler code window for that option. In my example, I double-clicked on Doit🡪Chaos🡪Hack System:

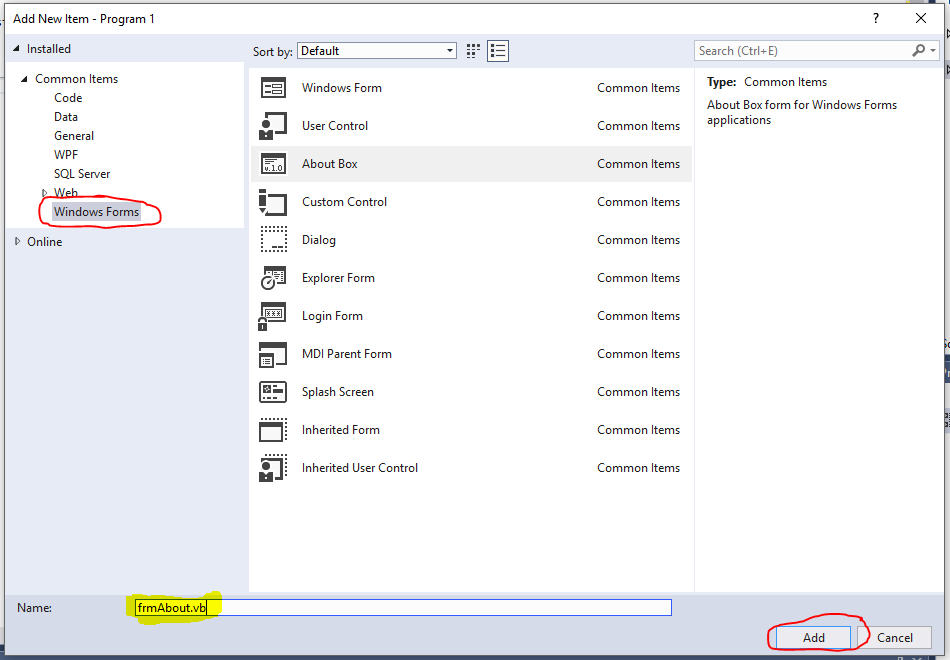


Now that we've completed a basic discussion on working with a main menu, let's pause for a minute and look at a related topic – having multiple forms in an application. We will come back to the menu discussion in a bit.

# Building More Forms

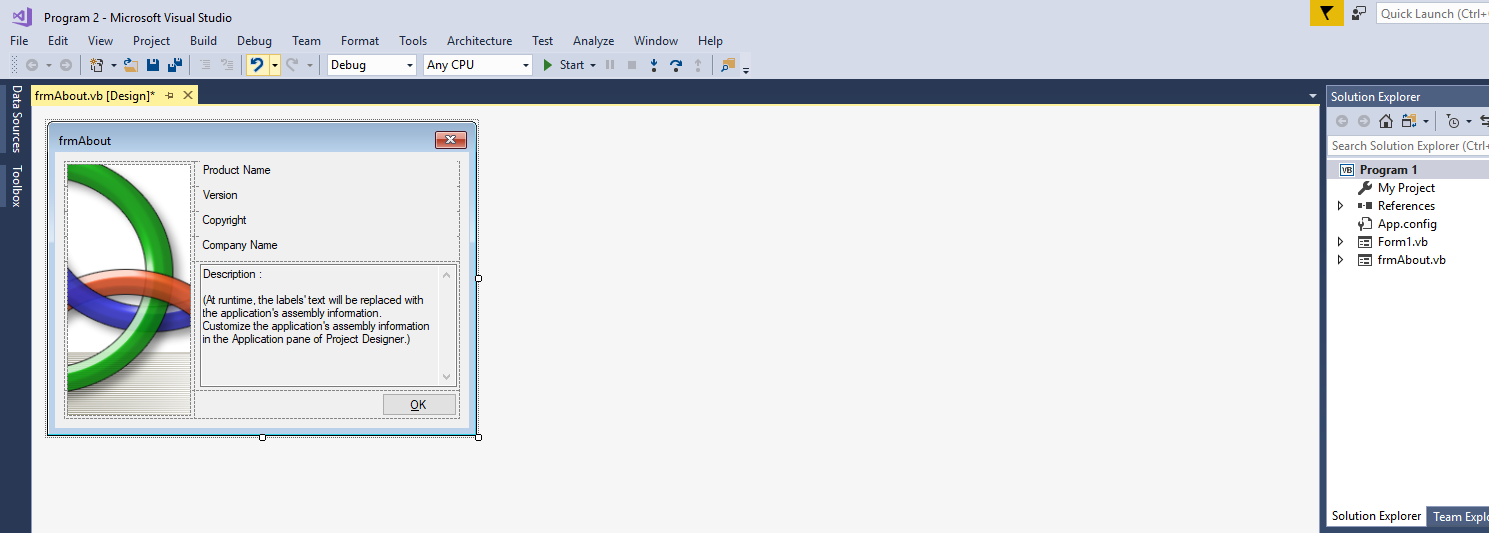
Visual Basic has a bunch of form wizards available that will build About Forms, Splash Screens, Login Screens and so forth. This is great because the wizard based forms take care of a lot of the tedious issues with building these kind of forms.

As you know most applications have more than one form in them. Almost every Windows application has the main application form and at least a Help🡪About form. Let’s say that we decide we want to add an About box to our application and then have a menu option that is responsible for showing it. Good news: VB can auto build an About form for you! The first step is to go into the VB.NET menu and select Project🡪Add Windows Form. After you have selected the Add Windows Form menu item, click on the Windows Forms option in the left hand pane of the dialog that appears and you will be shown the following options:



Now if we just want a plain, new, empty Windows form, just click Add and you'll see that a new form will appear in your project, named awesomely Form2.vb. But before we get too click happy, let’s look around in the list of what can be built. Sure enough, down the list a couple of entries, you’ll see the ability to create an About Box. Let's rename this form frmAbout.vb in the Name textbox and then click the Add button (so the setup should look like the screenshot above).

In a blink, you’ll see your About form created! Notice that the frmAbout.vb file has also been added to the files listed in the Solution Explorer:



Now, how’s that for cool? The About form has got a picture on it and everything. The other interesting point is that the form is complete and ready to go – the OK button even has the necessary code added for you to close the form. Thank you very much Visual Studio!

The next task that we must complete is wiring the About form up to the main application form's MenuStrip system. We want to display the About form whenever a user clicks on the About menu option under help. Let's go back to our main application form, which should be called Form1. Navigate through the form's menu structure and double-click on the Help🡪About menu option. The code window will reappear and then you need to enter the single line of code shown below in the mnuHelpAbout\_Click handler:

Public Class Form1

'Chapter 11 - Program 2

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

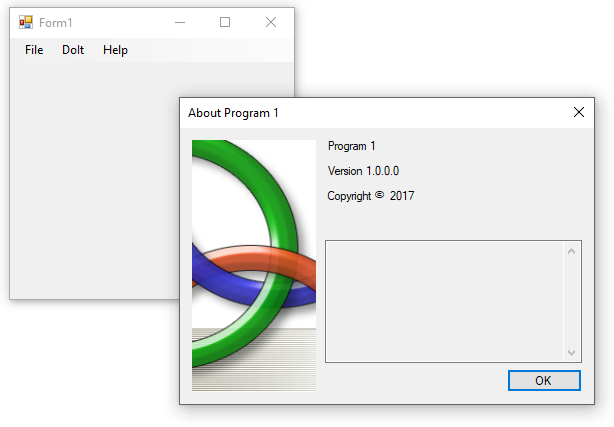
Handles mnuHelpAbout.Click

frmAbout.ShowDialog()

End Sub

End Class

If we run the application now and then click on the Help🡪About menu option, our About form will be displayed. And, because we displayed the About form with ShowDialog, the user must close the About form before he or she can continue with the application. The next figure shows a picture of the About form displayed:



Notice that there aren't any close, minimize or maximize buttons on the About form. When the user clicks the OK button, the form will close itself and the user will be able to interact again with the main application. As you can see from this discussion, it is fairly easy to add some common forms to an application. Using the menu system is a very convenient way to allow navigation from form to form.

Now one thing you might be wondering about is where the About box pulled the information that it displayed. Well let’s take a look at the code behind the scenes in the About form:

Public NotInheritable Class frmAbout

Private Sub frmAbout\_Load(ByVal sender As System.Object,

ByVal e As System.EventArgs) Handles MyBase.Load

' Set the title of the form.

Dim ApplicationTitle As String

If My.Application.Info.Title <> "" Then

ApplicationTitle = My.Application.Info.Title

Else

ApplicationTitle = System.IO.Path.GetFileNameWithoutExtension

(My.Application.Info.AssemblyName)

End If

Me.Text = String.Format("About {0}", ApplicationTitle)

' Initialize all of the text displayed on the About Box.

' TODO: Customize the application's assembly information in the

' "Application" pane of the project properties dialog (under the "Project"

' menu).

Me.LabelProductName.Text = My.Application.Info.ProductName

Me.LabelVersion.Text = String.Format("Version {0}",

My.Application.Info.Version.ToString)

Me.LabelCopyright.Text = My.Application.Info.Copyright

Me.LabelCompanyName.Text = My.Application.Info.CompanyName

Me.TextBoxDescription.Text = My.Application.Info.Description

End Sub

Private Sub OKButton\_Click(ByVal sender As System.Object,

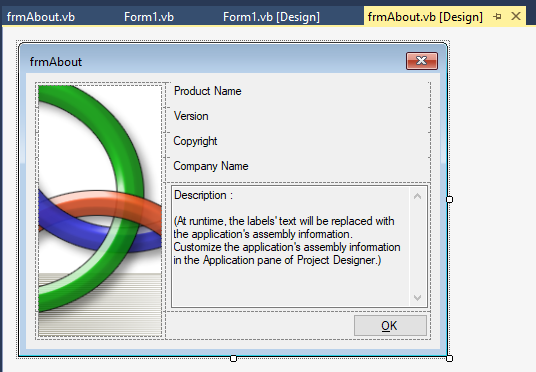
ByVal e As System.EventArgs) Handles OKButton.Click

Me.Close()

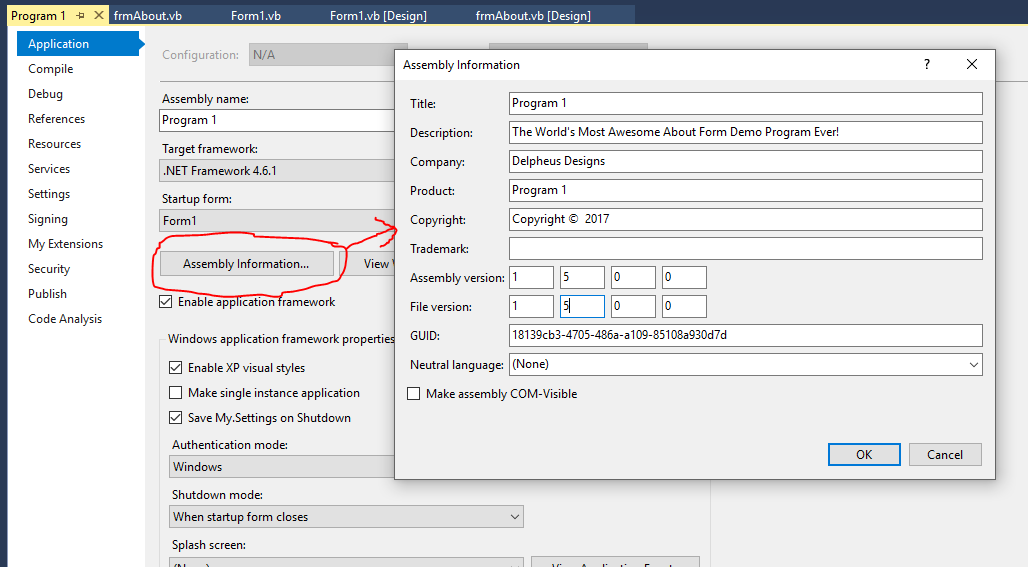
End Sub

End Class

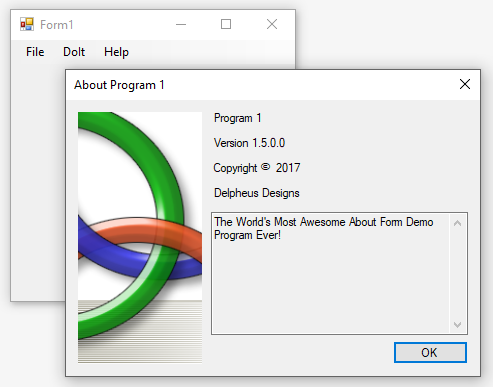
If we look at the About form in the designer, we see that 5 pieces of information will be drawn from somewhere and displayed on the form:



Both the About form design screen and the code tell us that we can edit this information by going into the Application pane of the Project’s Properties. Remember, we right click on the Project’s Properties, select the Application Pane and then click on the Assembly Information button. A new popup screen shows up where we can fill out whatever information we want:

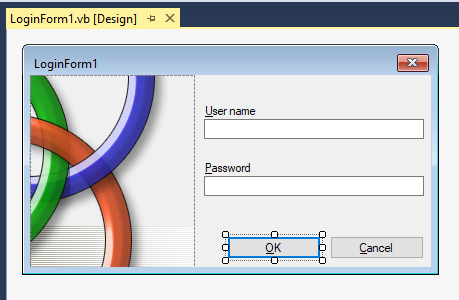


After making some changes and clicking OK, we see those reflected in the About form when we run the application again:



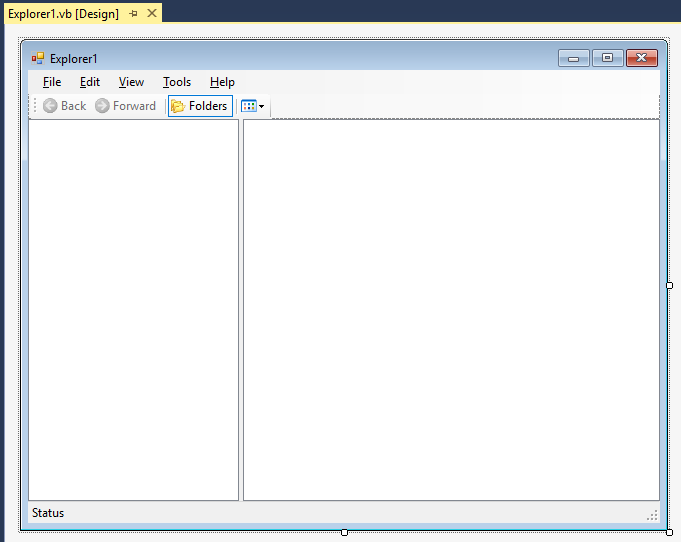
Sweet! All we had to do was edit some textboxes and our changes showed up with no code changes necessary! Since we are on the subject of forms that we can get from free, let’s finish this up, by taking a look at a couple more that are built in.

Here’s the Login Form:

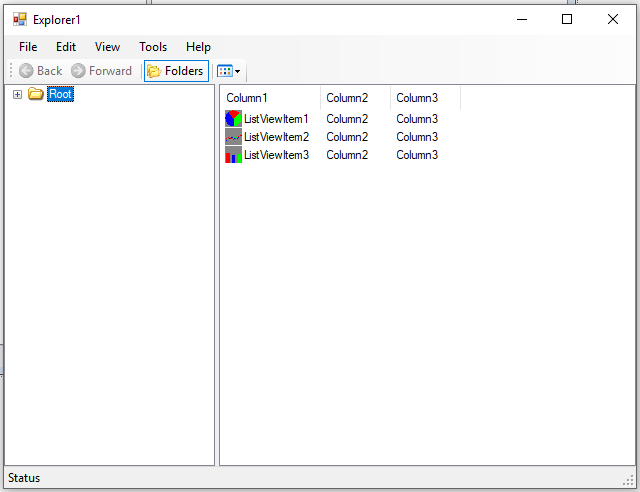


Now before you get all excited about this form, realize that while the OK and Cancel buttons are functional, they don’t do anything. You would have to wire up the actual authentication of the user’s credentials. At least you didn’t have to create the form though!

We also can get an Explorer form built for us:



If we run the program and show this form, it looks pretty good at first. It won’t take long and you’ll soon realize that it’s nothing more than a skeleton that you have to wire up! Not so bueno after all…



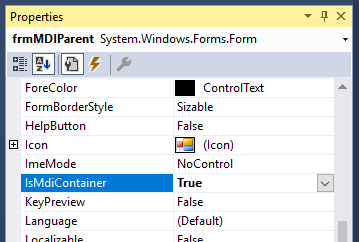
# MDI Forms

The last major topic to discuss regarding forms is creating MDI applications. MDI stands for Multiple Document Interface and this refers to the ability for an application to have several child document windows open at the same time. Think of old school Microsoft Word as an example of MDI – you can open multiple documents simultaneously under one copy of Word. Each child document lives within its own window.

To create an MDI application, we will need to create two different types of forms. The first form type is the MDI parent form, which serves as the main application form. The second form type is the child document, of which, multiple instances can be created and contained within the MDI parent.

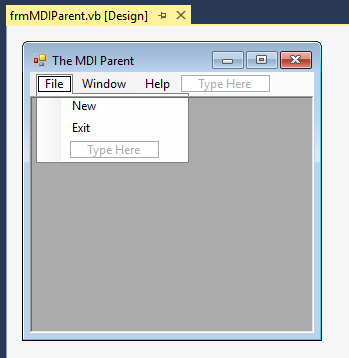
To create the parent form, all you need to do is set the IsMDIContainer property to True in the form's property window on whatever form that you want to serve as the MDI parent. There can only be one MDI parent in any application! The child form is just a normally created form that has the IsMDIContainer property set to its default value of False. You will notice that the MDI parent visually will have a much darker background than the child form.

Let's try this out. Create a new project that has two forms. Name one frmMDIParent and one frmChild. Set the frmMDIParent up as the MDI container by setting its IsMDIContainer property to True.

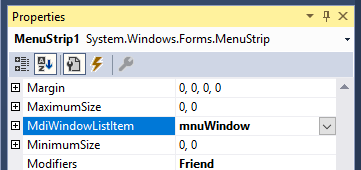


Next, add a Main Menu control to the MDI parent form. Create a menu consisting of the following main options: File, Window and About. You can also add New and Exit submenu options under File. Be sure you give the menu options good names since we will need to talk to them through code.

Here’s a quick look at the MDI parent form:

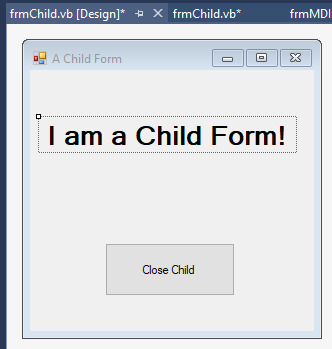


Pay careful attention to this little tip: once you have your menu built, click on the MenuStrip object so that it’s what’s selected (make sure it’s not a menu item that you’ve selected instead). Next, change the MenuStrip’s MdiWindowListItem property to whatever you called your Window menu item. If you’re following the good naming convention that I showed you, it should be called mnuWindow. This little change will allow your program to switch between child windows. Any time that the user clicks on the Window menu option, a list of all child windows that are currently open will appear. The user can then change between child windows simply by clicking on one of the windows in the list. The next screenshot shows the MdiList property in the MenuStrip1 properties window:



Now, go to the child form and put a label on it that says that it's a child window. Also add one command button that will close the child window – the command button simply should perform a Me.Close().

You have just successfully created the child form. Here's a screen shot of what the child form should look like:



Now let’s take a look at the code required to get the New menu option to open a new child form. Before looking at the code, let's talk about the logic needed to permit this to happen. We would need to create a new instance of the child form and remember there can be many instances of the child. We then need to tell the child window who its parent is. This will be handled by setting the child form's MdiParent property. Finally, we tell the child form to show itself to the user.

Programmatically, we can do all of this work inside of the MDI parent's New menu handler. Since the menu belongs to the parent and we are creating the child inside of the parent, we simply need to set the child's MdiParent property = Me, where Me is the parent form, which really is the parent of the child. Here's the code:

'Chapter 11 - Program 3

Public Class frmMDIParent

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

Handles mnuFileNew.Click

'Create another child

Dim aNewChildForm As frmChild = New frmChild()

'Hook the child to the parent

aNewChildForm.MdiParent = Me

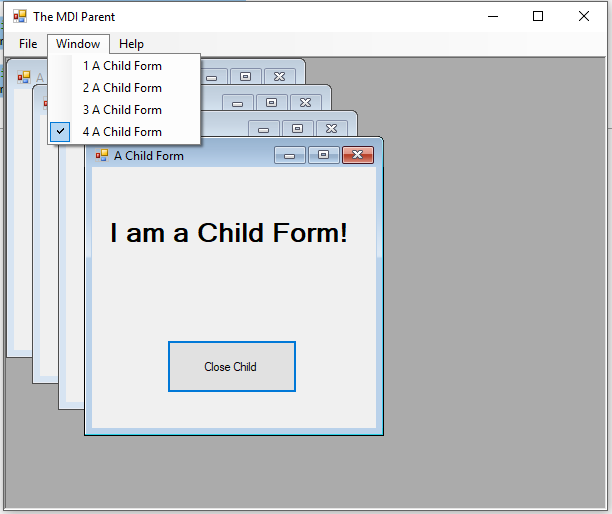
'Show the child

aNewChildForm.Show()

End Sub

End Class

That’s it! We can now create as many child windows as we want, whenever we want just by clicking on File🡪New. This works exactly like other “real” Windows applications that are MDI based. Our window menu option even works just like we’d expect – every child window shows up in the list:



Now for a good question: How do I know which child window is being worked with? This is again where the Me keyword becomes important. Me refers to whatever form is currently being interacted with. So prefacing a control with Me., such as in Me.Command1, allows you to have programmatic control on the current child form being used – this is talking about the Command1 control in whatever form currently is running. Pretty simple, isn’t it?

One more common feature that most Windows applications have besides switching between child windows is the ability to quickly arrange how the child windows look within the MDI parent container. I added menu options under the Windows option to Cascade, Tile🡪Vertical and Tile🡪Horizontal.

Here's the code to handle the various MDI window arrangements:

'Chapter 11 - Program 4

Public Class frmMDIParent

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

Handles mnuFileNew.Click

'Create another child

Dim aNewChildForm As frmChild = New frmChild()

'Hook the child to the parent

aNewChildForm.MdiParent = Me

'Show the child

aNewChildForm.Show()

End Sub

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

Handles mnuWindowCascade.Click

Me.LayoutMdi(MdiLayout.Cascade)

End Sub

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

Handles mnuWindowTileHorizontal.Click

Me.LayoutMdi(MdiLayout.TileHorizontal)

End Sub

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

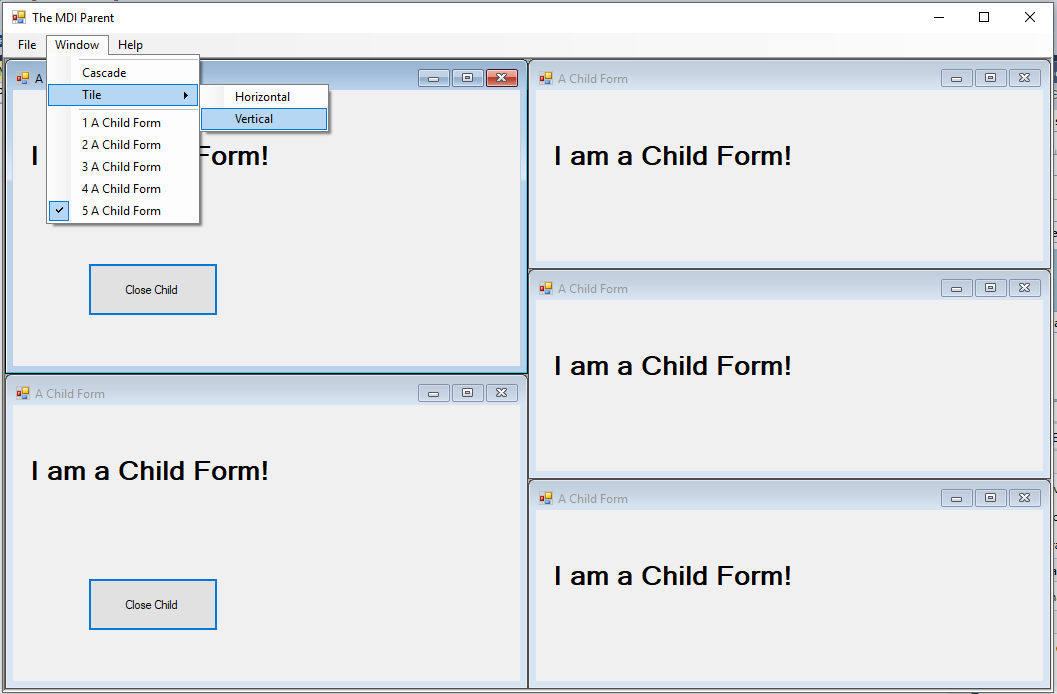
Handles mnuWindowTileVertical.Click

Me.LayoutMdi(MdiLayout.TileVertical)

End Sub

End Class

Here's a screenshot of the application in action. Notice that the child windows still show up in the WindowList in addition to presenting the arrangement menu options:

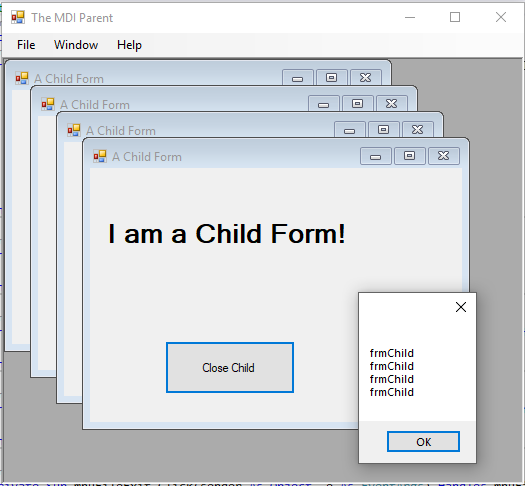


One final note to end the MDI section on: you already have seen that an MDI parent can have multiple MDI children. What you may not have thought about yet is the fact that there can be different types of children! In our examples, each child form looked like every other child form. This does not have to be the case though – we could create numerous child form types and we could create multiple instances of each child type that are all owned by the same parent. Visual Basic gives you the ability to build any kind of an application configuration that you can think of!

# Understanding the OpenForms, MDIChildren and Controls Collections

The MDIChildren collection is a container of every child form that belongs to an MDI parent form within an application. Each form is accessible by iterating through the MDIChildren collection using a For Each loop.

Here’s an example of some VB code that loops through all of the MDI children in an MDI application and then prints a messagebox containing the names of all of the child forms out to the user.



Here’s the related code to carry out the work:

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

Handles mnuFileExit.Click

Dim strNames As String

Dim aForm As Form

For Each aForm In Me.MdiChildren

strNames &= aForm.Name & vbCrLf

Next

MessageBox.Show(strNames)

End

End Sub

While we printed out the names of each child, the names were all the same. How useful is that? Now what could we do if we really want to give each child form a unique name? Well, we could create a global variable and then increment it…

Add this to the Application by clicking on the View Application Events button in the Project Properties:

Namespace My

'Chapter 11 - Program 5

Partial Friend Class MyApplication

Public intFormCount As Integer = 0

End Class

End Namespace

Next, modify the New menu routine in the frmParent’s code:

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

Handles mnuFileNew.Click

'Create another child

Dim aNewChildForm As New frmChild()

'Increment the number for this child form

My.Application.intFormCount += 1

'Change the form's name and text

aNewChildForm.Name &= "-" & Trim(CStr(My.Application.intFormCount))

aNewChildForm.Text = aNewChildForm.Name

'Hook the child to the parent

aNewChildForm.MdiParent = Me

'Show the child

aNewChildForm.Show()

End Sub

Run the application, add a few children and check the WindowList:



Likewise, when we exit the application, you can see that the forms have been renamed and can be referred to through their names:



Now that we’ve seen how to rename the MDI children and that there is an MDI container, let’s take a look at something a bit more interesting. We can actually walk over every form and every control, because VB has other containers behind the scenes allowing us to do this. The My.Application.OpenForms collection contains, well, every open form that the application has. We already know that if we have a particular form, we can get to all of its MDI children via the MdiChildren collection. About the only other new tidbit of news here, would be that every form has a Controls collection which contains every control that has been added to that particular form.

By carefully stringing this logic together, we can programmatically iterate over all of our forms and all of their controls. I added this code to the mnuFileExit event handler:

'Chapter 11 - Program 6

Public Class frmMDIParent

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

Handles mnuFileExit.Click

Dim strNames As String

Dim aForm, aForm2 As Form

Dim aParentControl, aChildControl As Control

For Each aForm In My.Application.OpenForms

strNames &= "Application Form: " & aForm.Name & vbCrLf

For Each aParentControl In aForm.Controls

strNames &= " Control: " & aParentControl.Name & vbCrLf

Next

For Each aForm2 In aForm.MdiChildren

strNames &= " Child Form: " & aForm2.Name & vbCrLf

For Each aChildControl In aForm2.Controls

strNames &= " Control: " & aChildControl.Name & vbCrLf

Next

Next

Next

MessageBox.Show(strNames)

End

End Sub

This is the result after running the application and adding three child forms:

