# Introduction to Windows Forms

This document contains notes pertaining to the Pluralsight course “Introduction to Windows Forms”. Some of the clips of the Pluralsight course are documented. The names of those clips are copied to the section headers in this document and to the table below. Using Microsoft Word, you can use the names in the table as hyperlinks to navigate to any particular clip. But using Apache Open Office, these hyperlinks do not work; instead, they merely serve as a table of contents. You can navigate to the start of any clip via bookmarks; type F5 to bring up the Navigator; then double-click Bookmark1 for 1st clip header, Bookmark 2, for 2nd clip header, etc.

|  |  |
| --- | --- |
| [Forms](#_Forms) | [Demo: Forms](#_Demo:_Forms) |
| [Demo: Controls](#_Demo:_Controls) | [Demo: Event Handlers](#_Demo:_Event_Handlers) |
| [Demo: Using The Designer](#_Demo:_Using_The) | [Generated Code & Demo](#_Generated_Code_&) |
| [Form Lifecycle](#_Form_Lifecycle) | [Demo: Form Lifecycle (plus Anchor property)](#_Demo:_Form_Lifecycle) |
| [Demo: Dialogs (MessageBox)](#_Demo:_Dialogs_(MessageBox)) | [Custom Dialogs](#_Custom_Dialogs) |
| [Demo: Custom Dialogs](#_Demo:_Custom_Dialogs) | [Menu and Toolbars](#_Menu_and_Toolbars) |
| [Demo: Menu and Toolbars](#_Demo:_Menu_and) | [Demo: User Controls](#_Demo:_User_Controls) |
| [Demo: Creating the (MDI) Forms](#_Demo:_Creating_the) | [Demo: Window Menu](#_Demo:_Window_Menu) |
| [Demo: Edit Menu](#_Demo:_Edit_Menu) | [Demo: File Menu](#_Demo:_File_Menu) |
| [Demo: Toolbar](#_Demo:_Toolbar) | [SDI Applications](#_SDI_Applications) |
| [Demo: Creating the (SDI) Project](#_Demo:_Creating_the_1) | [Demo: Configuring The Application Object (VB)](#_Demo:_Configuring_The) |
| [Demo: Configuring The Application Object (CS)](#_Demo:_Configuring_The_1) | [Demo: File and Edit Menu](#_Demo:_File_and) |
| [Demo: Window Menu](#_Demo:_Window_Menu_1) | [Demo: Creating The Project (explorer style)](#_Demo:_Creating_The_2) |
| [Demo: Populating The Tree View](#_Demo:_Populating_The) | [Demo: Populating The List View 1 – Items and Images](#_Demo:_Populating_The_1) |
| [Demo: Populating The List View 2 – Views](#_Demo:_Populating_The_2) | [Data Binding](#_Data_Binding) |
| [Demo: Preparing The Project (Data Binding)](#_Demo:_Preparing_The) | [Demo: Complex Data Binding](#_Demo:_Complex_Data) |
| [Demo: Simple Data Binding](#_Demo:_Complex_Data) | [Demo: Add and Delete](#_Demo:_Add_and) |
| [BindingSource Control](#_BindingSource_Control) | [Demo: Using The BindingSource](#_Demo:__Using) |

## Forms

The form displayed when the application starts initially is normally invoked automatically. With regard to secondary forms that are displayed later in the process of application execution, the statements to display such forms are

Show()

or

ShowDialog()

Show() displays the form modelessly, whereas ShowDialog() displays the form modally.

Application start-up also involves the use of the **Application** object. The statement

Application.Run(form)

performs initialization in addition to displaying the start-up form. To end the application one could use

Application.Exit, but this is not really necessary, because closing the start-up form implicitly invokes Application.Exit.

## Demo: Forms

We start by building a code-only project; we will not use template code, and we will not use the **Toolbox**.

* Open Visual Studio. In the right panel click **Create a new project**. Visual Studio displays the **Create a new project** dialog.
* Open the languages drop-down list near the top of the right panel. Choose **Visual Basic**. Type “empty” in the search box at the top of the right panel. Visual Studio reduces the number of candidates in the right-panel list. Select **Empty Project (.NET Framework)** with **Visual Basic** as the language. Click **Next**. Visual Studio displays the **Configure your new project** dialog.

***It is not necessary to have allocated a folder for the solution before creating the solution and 1st project. Visual Studio automatically allocates a new subfolder under the folder selected in the following step, and it copies the solution name as the name of this new subfolder.***

* Type “Simple” into the **Project name** text box. Use the **Location** browser to select a folder into which the solution will reside. Type “CodeOnly” into the **Solution name** text box. Uncheck the check box below the **Solution name** text box. Click **Create**. Visual Studio displays the solution with empty left and center panels. If the **Solution Explorer** is not visible in the right panel, navigate to make it visible.
* Now create the class that contains the windows form. Right click “Simple”; then select **Add -> Class...** from the context menu. Visual Studio displays the dialog **Add New Item**.
* Change the name in the **Name:** text box (bottom left) to “MyForm”. Click **Add**. Visual Studio displays the class MyForm in the center panel.

***By default, classes created in C# are not “public”. It is necessary to type “public” as a prefix to the class name, unless you deliberately want it to be private. The same applies to properties and methods in the class.***

* We want class MyForm to inherit from Systems.Windows.Forms.Form. In order to do that we will have to reference the Systems.Windows.Forms assembly. Right-click “Simple”; then select **Add -> Reference...** from the context menu. Visual Studio displays the dialog **Reference Manager**.
* Expand **Assemblies** in the left panel. Click **Framework** to make it highlighted. Scroll down the list in the center panel to **System.Windows.Forms**, and hover the mouse over this item. Click - to check - the checkbox to the left of this item. Click **OK**. Visual Studio displays **System.Windows.Forms** in the **Solution Explorer** under **References**.
* Navigate back to the body of Class MyForm. Type the line

Inherits Form

Hover the mouse over “Form”, and type <ctrl>.; then accept the invitation “Imports System.Windows.Forms”.

* Add the following code to Class MyForm – a skeleton of the constructor.

Public Sub New() (the VB constructor evidently uses “New” instead of the class’s name)

End Sub

* The constructor can be used to set values for various desired properties. At this point the only property will be the “Text” property, which shows up as the window’s title. In the body of the constructor type

Me.Text = “My Form”

Now we need to supply the “Main” method, the entry point for this application.

* The “Main” method goes inside a module. Right click “Simple”; then select **Add -> Module...** from the context menu. Visual Studio displays the dialog **Add New Item** with **Module** selected (by default) in the center pane. (***In C# we add a new class instead of “module”.***)
* Change the name in the **Name:** text box (bottom left) to “Startup”. Click **Add**. Visual Studio displays the module “Startup” in the center panel.
* Type the following code in the body of Startup.

Public Sub Main()

Dim form as New MyForm

Application

End Sub

***In C# we write “public static void Main()” instead of “Public Sub Main()”.***

* Hover the mouse over “Application”, and type <ctrl>.; then accept the invitation “Imports System.Windows.Forms”.
* Append “.Run(form)” after “Application”. The statement “Application.Run(form)” does 3 things. (1) It starts listening in on the Windows message loop. (2) It sets “form” as the main form for the application. (3) It shows the form.

This completes the construction of the “Simple” application. Build and run the application. The application shows a form whose title is “My Form”. When the C# version of this application runs, the form is displayed, but the console-window is also displayed. To prevent the console window from displaying

* Right-click the project name. Then select **Properties** from the context menu. Visual Studio displays the properties with tab selectors in the left panel.
* With the **Application** tab selected, refer to the **Output type:** dropdown list near the top of the right panel. Use the down-arrow to the right of the dropdown list to change the type to **Windows Application**.
* Close the **Properties** dialog. Rerun the application, and observe that console window is no longer displaying.

## Demo: Controls

We will be adding 3 controls to the Main form: a text box, a button, and a label.

* Navigate to Visual Studio, MyForm.vb.
* To declare the 3 controls, add the following code after “Inherits Form”.

Private MessageTextBox As TextBox

Private MessageLabel As Label

Private ShowMessageButton As Button

* To place these controls into the form, add the following code at the end of the constructor (Sub New()). This code instantiates each of the controls, sets properties of that instance, and then adds that instance to the controls collection of the form.

MessageTextBox = New TextBox()

MessageTextBox.Left = 25

MessageTextBox.Top = 25

MessageTextBox.Width = 200

Me.Controls.Add ( MessageTextBox )

ShowMessageButton = New Button()

ShowMessageButton.Left = 25

ShowMessageButton.Top = 75

ShowMessageButton.Width = 200

ShowMessageButton.Text = “Show Message”

Me.Controls.Add ( ShowMessageButton )

MessageLabel = New Label()

MessageLabel.Left = 25

MessageLabel.Top = 125

MessageLabel.Width = 200

MessageLabel.Text = "[Label]”

Me.Controls.Add ( MessageLabel )

* Build and run the application. The form and its controls display as expected.

## Demo: Event Handlers

The 3 controls – text box, button, and label – are intended to work together. The user types some text into the text box; then he clicks the command button; the application is supposed to respond by copying the text into the label. We start by adding an event handler for the click event of the button. We’ll begin with the static method.

***Do not attempt the following steps in C#. The “static” method is not available in C#.***

* Insert the “WithEvents” keyword into the button’s declaration statement – near the top of the code in MyForm.vb. The result is

Private WithEvents ShowMessageButton As Button

* Add the handler for the button’s click event. All event handlers have a common signature: the 1st parameter is of type **object**, where we normally name the sender of the event; the 2nd parameter – normally named “e” - is of type **EventArgs** (or a class derived from **EventArgs**); e provides information that is specific to the event. In Visual Basic the signature ends with a Handles clause. To get help in coding the event handler signature . . .
  + Refer to the dropdown list at the top of the center panel that contains (by default) the name of the form; click the down arrow, and select the name of the control that we want to work with (ShowMessageButton in this case).
  + Click the rightmost dropdown list at the top of the center panel, and select the name of the event (**Click** in this case). Visual Studio displays a skeleton of the event handler for the selected control.

By default, Visual Studio uses – for the name of the event handler *NameOfControl\_NameOfEvent*. But there is nothing magic about this name; the **Handles** clause at the end of the event signature specifies the name of the control and the name of the event. Just to drive this point home, change “ShowMessageButton\_Click” to “ClickHandler”.

* Type the following statements into the body of ClickHandler().

MessageLabel.Text = MessageTextBox.Text

MsgBox("Button clicked")

Now we’ll repeat the exercise using the dynamic method of adding an event handler.

* Remove the “WithEvents” keyword from the button’s declaration statement.
* Remove the **Handles** clause from the signature of ClickHandler().
* Insert the following statement after “Me.Controls.Add ( ShowMessageButton )”.

AddHandler ShowMessageButton.Click, AddressOf ClickHandler

* Build and run the application. Note that the text is copied successfully after clicking the button.

***Use the following steps to add an event handler in C# (dynamically).***

* Insert the following statements after “this.Controls.Add ( ShowMessageButton )”.

ShowMessageButton.Click += ShowMessageButton\_Click;

Visual Studio responds by creating a skeleton for the mouse-click event handler, and it displays an invitation to rename the event handler. Accept the invitation and rename it to “ClickHandler”.

* Type the following statements into the body of ClickHandler().

MessageLabel.Text = MessageTextBox.Text;

MessageBox.Show ( "Button clicked" );

* Build and run the application. Note that the text is copied successfully after clicking the button.

## Demo: Using The Designer

This project will be similar to the code-only project; but this time we will use template code and the **Toolbox**.

* Open Visual Studio. In the right panel click **Create a new project**. Visual Studio displays the **Create a new project** dialog.
* Open the languages drop-down list near the top of the right panel. Choose **Visual Basic**. Type “windows” in the search box at the top of the right panel. Visual Studio reduces the number of candidates in the right-panel list. Select **Windows Forms App (.NET Framework)** with **Visual Basic** as the language. Click **Next**. Visual Studio displays the **Configure your new project** dialog.

***It is not necessary to have allocated a folder for the solution before creating the solution and 1st project. Visual Studio automatically allocates a new subfolder under the folder selected in the following step, and it copies the solution name as the name of this new subfolder.***

* Type “Designer” into the **Project name** text box. Use the **Location** browser to select a folder into which the solution will reside. Keep “Designer” as the **Solution name** text box. Check the check box titled **Place solution and project in the same directory**. Click **Create**. Visual Studio displays the solution with an empty form titled “Form1”. If the **Solution Explorer** is not visible in the right panel, navigate to make it visible.

The form has two views – the design view and the code view. One can switch between the 2 views by right-clicking *FormName*.vb (in the **Solution Explorer**) and selecting either **View Code** or **View Designer** in the context menu.

* Rename the form in the **Solution Explorer** to “MyForm”. Visual Studio displays a pop-up dialog asking whether to rename all references to the code element ‘Form1’. Click **Yes**. Notice that Visual Studio has renamed the class (in code view) to “MyForm”.
* Look at the **Properties** window for MyForm.
* To change MyForm’s title, change the **Text** property to “My Form”.
* Change the **Size** property to “500, 430”.

Now we have enough room to add controls to the form.

* Open the **Toolbox** (in the left-hand panel). Expand the **Common Controls** node.
* Drag/drop a **TextBox** from the **Toolbox** to MyForm. Similarly drop a **Button** and a **Label** to MyForm.
* Play with moving the controls around on the form. Notice vertical snap lines are displayed when the controls are aligned vertically. If the controls have similar sizes, Visual Studio shows a snap line between their right edges when their sizes are the same. Also, when a control is close to an edge of the form, Visual Studio shows a snap line from the center of the control’s edge to the edge of the form - indicating that the control is at the correct distance from the edge of the form in accordance with the standard Windows Forms guidelines. Similarly, when two controls are positioned close to each other the same small snap line is displayed to indicate the correct distance between the controls.
* In this example, we won’t follow the guidelines necessarily, because we are trying to make the controls look the same as in the code-only project.
* Set the TextBox’s **Name** property to “MessageTextBox”.
* Expand the TextBox’s **Location** property; set both **X** and **Y** to 25.
* Expand the TextBox’s **Size** property; set **Width** to 200.
* Set the Button’s **Name** property to “ShowMessageButton”
* Set the Button’s **Location** property to “25, 75”.
* Expand the Button’s **Size** property; set **Width** to 200.
* Set the Button’s **Text** property to “Show Message”.
* Set the Label’s **Name** property to “MessageLabel”.
* Set the Label’s **Location** property to “25, 125”.
* Setting the Label’s **Width** doesn’t work, because the **AutoSize** property is **True** by default. Change **AutoSize** to **False**, and then change **Width** to 200.
* Set the Label’s **Text** property to “[Label]”.
* Click the lightning icon at the top of the Button’s **Properties** window to switch the “events” view.
* Click the **Click** event in the Button’s events view; verify that it is highlighted. Then double-click ShowMessageButton in the form’s designer view. Visual Studio creates a skeleton of the event handler, and displays it in the form’s code view.
* Type the following statements into the body of the event handler.

MessageLabel.Text = MessageTextBox.Text;

Build and run the application. Note that the text is copied successfully after clicking the button.

***I created a similar C# project named “DesignerCS”. I did it quickly, and I deliberately made it look different from the VB version.***

## Generated Code & Demo

The important take-away information from the two Pluralsight clips “Generated Code” and “Demo: Generated Code” are (1) the identity of the files that contain the generated code, and (2) how to navigate to these files. (Since Current Pixel will not be concerned with Visual Basic, I have omitted the answers to these questions for VB from this document.)

In the Solution Explorer expand the node for MyForm.cs, and examine MyForm.Designer.cs. We wrote button event handler in MyForm.cs. Visual Studio automatically generated MyForm.Designer.cs, which contains the method InitializeComponent(). InitializeComponent() contains code that (1) declares and instantiates the classes for the controls, (2) sets the properties of the instances, and (3) adds the instances to the form’s controls collection. Much of this code is hidden by default, but you can see it by expanding the **+** (region) named “Windows Form Designer generated code”.

With regard to the start-up code, execution starts at Main() in Program.cs (which was automatically generated by Visual Studio). In particular “Application.Run()” is the last statement in Main(), and Application.Run() creates an instance of class MyForm. Note that Visual Studio has automatically added MyForm’s constructor (in MyForm.cs). This constructor contains the statement “InitializeComponent();”, which we discussed above.

## Form Lifecycle

The following are the names of event handlers pertaining to a form’s lifecycle events.

**Load** – fires after the form’s constructor is executed, just before a form is shown

**Shown** – fires just after the form shows on the screen

**Activated** – fires when the form gets focus

**Deactivate** – fires when the form loses focus

**FormClosing** – fires just before the form is closed, and closing can be cancelled

**FormClosed** – fires just after the form is closed

A fast way to start coding the **Load** event is to double-click anywhere on the form that does not contain a control; for example, double click the title bar of the form.

## Demo: Form Lifecycle (plus Anchor property)

The 1st part of the clip illustrates using the form’s **Load** event to copy data into a listbox.

The 2nd part of the clip introduces the **Anchor** property for controls on Windows Forms. There are four parts of the anchor: top, right, bottom, and left. Each part of the anchor can be present or missing. For horizontal anchor parts…

If the left part is present and the right part is missing, then

when the form is widened,

the distance between the control and the form’s left edge remains fixed.

If the left part is missing and the right part is present, then

when the form is widened,

the distance between the control and the form’s right edge remains fixed.

If both the left and right parts are present, then

when the form is widened,

the width of the control increases by the same amount.

If both the left and right parts are missing, then

when the form is widened,

the control is moved to the left or right (relative to the form) to maintain the same proportional distance between the left and right edges of the form.

The behavior for vertical anchor parts is the same as for horizontal anchor parts – replacing “left” with “top”, and “right” with “bottom”.

## Demo: Dialogs (MessageBox)

This section shadows the 2nd MessageBox example in the Pluralsight clip “Demo: Dialogs”. This 2nd example deals with the situation where the user has clicked the close-form button (“x” in the upper-right corner); the example uses MessageBox to ask the user whether he really wants to exit. The project that I am using as a vehicle for this example is the same C# project that I wrote for [Demo: Using The Designer.](#_Demo:_Using_The)

* Navigate to the C# Visual Studio project, to the “designer” view of the form. Open the Object Explorer. Click the lightning bolt to get the list of events. Double-click **FormClosing** in the list of events. Visual Studio creates a skeleton of the event handler for **FormClosing**.
* Type the following code into the body of the event handler.

var msg = "Are you sure you want to close?";

if ( MessageBox.Show ( msg, this.Text, MessageBoxButtons.YesNo, MessageBoxIcon.Question,

MessageBoxDefaultButton.Button2 )

{

e.Cancel = true;

}

For this (overloaded) variant of MessageBox.Show()

The 1st argument (msg) is the text to display in the message-box window.

The 2nd argument (this.Text) specifies the text in the window’s title bar (the same text as in the form’s title bar).

The 3rd argument (MessageBoxButtons.YesNo) specifies the style: a box containing 2 buttons – **Yes** and **No**.

The 4th argument (MessageBoxIcon.Question) specifies an icon to display in the dialog: a picture of a question mark.

The 5th argument (MessageBoxDefaultButton.Button2) specifies which of the buttons is highlighted by default: the 2nd button (**No**). If the 1st button were highlighted instead, and if user pressed the **Enter** key carelessly, the form would have closed. But using the default argument in this way forces the user to select the **Yes** button deliberately to close the form.

The return value from MessageBox.Show is an element of an enumerator, **DialogResult**. When the user responds by clicking **No**, DialogResult.No is returned, and the e.Cancel statement is executed.

Build and run the application. Click the “x” in the upper-right corner. The message box will display. Click **No**, or press the **Enter** key. The form does not close. Click “x” again. This time respond to the message box by clicking **Yes**. The form closes, and the application exits.

## Custom Dialogs

The following is – verbatim – what the Pluralsight instructor displayed at the beginning of this clip. It would seem to be a great checklist to use when building a custom dialog.

* Create the form as you would any other
* Set form properties to add dialog look, feel and behavior
  + Set FormBorderStyle to FixedDialog
    - Changes visual style
    - Disables resizing of dialog
  + Set ControlBox property to false
    - Removes minimize, maximize and close buttons from title bar
  + Set AcceptButton and CancelButton properties
    - AcceptButton: pressing Enter is the same as clicking the button
    - CancelButton: pressing Escape is the same as clicking the button
* Set dialog return value on button clicks
  + In event handler, or
  + Using the DialogResult property of the Button

*Important note: Setting the DialogResult property closes the dialog*

## Demo: Custom Dialogs

In this demo we create a custom dialog to perform the same action that we provided in [Demo: Dialogs (MessageBox)](#_Demo:_Dialogs_(MessageBox)) using MessageBox.Show(). But I want to avoid creating a new project, and I don’t want to lose the code used in [Demo: Dialogs (MessageBox.](#_Demo:_Dialogs_(MessageBox)) Therefore, I am going to add - to that project - a new button **Exit** that will have the same functionality as the “x” in the upper-right corner.

* Navigate to the C# Visual Studio project that was written for [Demo: Using The Designer.](#_Demo:_Using_The) Navigate to the “designer” view of the form.
* Add a button named btnExit with “Exit” for its **text** property.
* Double-click btnExit to induce Visual Studio to create the skeleton for btnExit\_Click().
* In the **Solution Explorer** right-click the project name. Click **Add -> Form (Windows Form)…** in the context menu. Visual Studio displays a dialog titled **Add New Item** with **Form (Windows Form)** selected in the center panel.
* Rename the form “ConfirmDialog” via the **Name:** text box at the bottom of the dialog. Click **Add**. Visual Studio responds by adding ConfirmDialog.cs to the solution, and it displays ConfirmDialog.cs in designer mode.
* Resize ConfirmDialog.cs to make it smaller than the main form. Change its **Text** property to “Confirm”.
* Drop a **Panel** (from the **Containers** section of the **Toolbox**) onto ConfirmDialog, Change the **Dock** property of the **Panel** to **Top**. Change the **BackColor** property of the **Panel** to **HighlightText**.
* Add a **Label** to the **Panel**. Change the **Text** property of the **Label** to “Are you sure you want to close?”
* Drop two buttons onto ConfirmDialog, The left-hand button is named “YesButton”, and its **Text** property is “Yes”; the right-hand button is named “NoButton”, and its **Text** property is “No”.
* Following the checklist from [Custom Dialogs](#_Custom_Dialogs) we start by setting the **FormBorderStyle** property of ConfirmDialog; change it to **FixedDialog**; this prevents the user from resizing the dialog.
* Remove the 3 buttons in the upper-right corner; change the **ControlBox** to **False**. (Double-click it to toggle from **True** to **False**.)
* Set YesButton to be the **AcceptButton**; change the **AcceptButton** property to **YesButton**.
* Set NoButton to be the **CancelButton**; change the **CancelButton** property to **NoButton**.
* The Pluralsight instructor’s suggestion was to change the **DialogResult** property of YesButton to be **Yes**. This may work for Visual Basic, but it did seem to work for C#; instead I set the dialog property to **OK.**
* The Pluralsight instructor’s suggestion was to change the **DialogResult** property of NoButton to be **No.** This may work for Visual Basic, but it did seem to work for C#; instead I set the dialog property to **Cancel**.
* Navigate to the handler btnExit\_Click().
* Add the following code to the body of btnExit\_Click().

var confirmDialog = new ConfirmDialog();

var dr = confirmDialog.ShowDialog();

if ( dr == DialogResult.OK )

{

this.Close();

}

You might notice a difference between the **if** statement above and the **if** statement suggested by the Pluralsight instructor. The **if** statement above tests the dialog-result value for the value **DialogResult.OK**, not **DialogResult.Yes**; the reason for this that **ShowDialog**’s return value is either **DialogResult.OK** or **DialogResult.Cancel**. This might be because of the difference between Visual Basic and C#; I did not try this example in Visual Basic.

I tried running the example, and it was successful, except for the use of the **Enter** key. (The **Esc** key worked.) The **Enter** key failed, because the control in ConfirmDialog that had focus (immediately after the dialog was shown) was NoButton, and typing the **Enter** key with this focus is the same as clicking NoButton. I resolved this problem by changing the **TabIndex** properties of YesButton and NoButton to 1 and 2 respectively.

***We do not have to rely on the return value from* ShowDialog() as *the example below illustrates.***

Add a button (btnNote) and a label (lblNote) to the form in [Demo: Dialogs (MessageBox).](#_Demo:_Dialogs_(MessageBox)) Double-click btnNote to create a skeleton event handler (btnNote\_Click()). Add a dialog (frmNote) to the solution - that contains a text box (txtNote) and a button (btnNote). Add the following code to btnNote\_Click().

var dlgNote = new frmNote();

dlgNote.ShowDialog();

var textBoxCtls = dlgNote.Controls.Find ( "txtNote", true );

lblNote.Text = textBoxCtls[0].Text;

The code above instantiates dlgNote from the frmNote class, opens it modally, and copies the text from its text box, when control returns to the main form. btnNote\_Click() in class frmNote contains the single statement

this.Close();

Run the application, and click the “Note” button. When the “Note” dialog displays, type some text into its text box, and then click its “Exit” button. Observe that the application copies its text to lblNote.

## Menu and Toolbars

Menus employ the **MenuStrip** and **ToolStripMenuItem** controls. Menu items are described by text, icons, and shortcut keys. When a character in menu text is preceded by “&”, the user can use <alt> + <that character> to invoke that text (instead of clicking it). The user invokes a shortcut of a menu item by typing <ctrl> + <some letter>. You can display images with menu entries. Images are available from the Visual Studio image library. ***I have downloaded and installed this library on the Windows-10 computer at***

***C:\Program Files (X86)\Microsoft Visual Studio\2019\VS2019 Image Library.zip***

The toolbar employs the **ToolStrip** control. An example of a control on a tool strip is the **ToolStripComboBox**, which is a wrapper around an ordinary combo box. Some, but not all, of the properties and events of wrapped control are accessible directly from the wrapper control; for others, it would be necessary to gain access via the wrapped control.

Windows Forms under .NET Framework, does not provide the ribbon control.

## Demo: Menu and Toolbars

We are going to need some thumbnail images for the following demo. The Pluralsight instructor advised downloading “Visual Studio Image Library”; this is a compressed (.zip) file containing a very large number of subfolders, each of which contains several image files with different extensions. In Windows Forms the “.png” extension is used very often. I selected the 2019 and 2012 versions of “Visual Studio Image Library”, and I downloaded these compressed files onto my Windows-10 computer - into subfolders of

C:\Program Files (X86)\Microsoft Visual Studio

But I found it very difficult to find the appropriate .png files from either of these Image Libraries. Ultimately, I copied a folder from the Pluralsight “Exercise Files” titled “Images” that contains the 9 .png files that we need for this course. I copied this folder to a location that makes it the sister folder to the folders that contain the various Visual Studio solution (.sln) files that shadow this course.

* Create a new solution and application. Name the solution and project “MenuTool”; we’ll use “MenuToolCS” for the C# version.
* Drag a **MenuStrip** from the toolbox to the main form. Visual Studio automatically docks the control at the top of the form.
* Click the **MenuStrip**. Visual Studio opens a place for the designer to type to the name of the leftmost menu. Type “&File”; then **Enter**. Visual Studio displays File; this means that “F” is the “quick” key, and that – instead of clicking File, the user can achieve the same effect typing <alt>F (case insensitive).
* Click under File, and then type “&New”; then **Enter**.
* Click under New, and then type “&Open”; then **Enter**.
* Before adding Save, we want a horizontal separator. Type “-” under Open; then **Enter**.
* What you should see now is File, *indent* New, *indent* Open, *indent* ----------- stacked vertically.
* Click under -----------, and then type “&Save”; then **Enter**.
* Click under Save, and then type “Save &As”; then **Enter**.
* For an example of a second top-level menu item, click in the box to the right of File, type “&Edit”; then **Enter**.
* Then one could add the children of Edit in a manner similar to File’s children.
* Let us return to the File menu; click File, then click New. Visual Studio displays a right-arrow followed by a box inviting a sub-menu under File -> New.
* Open the **Properties** window. Notice that the name of this menu-item control (at the top of **Properties**) is newToolStripMenuItem.
* Similarly examine the names of Open, Save, and Save As in **Properties**.
* Click New again. Click the property **ShortcutKeys**. Click the box to right of **ShortcutKeys**; then click the down-arrow that appears. Visual Studio opens the **Modifiers:** dialog for this menu item.
* Click the **Ctrl** checkbox. Open the **Key:** drop-down list; select “N”. Then click anywhere else. Visual Studio displays Ctrl+N to the right of New.
* Let us associate an image with this “File -> New” menu item. One of the **Properties** of this menu item is **Image**. Click **Image**. Visual Studio responds by displaying a small box containing “...” to the right of **Image**.
* Click “...”. Visual Studio responds by displaying the **Select Resource** dialog.
* Click the **Project Resource** radio button. Then click **Import...** Use the file navigator to go to **Images**; then click DocumentHS.png**.** Click **Open**. Visual Studio displays the name of the image in the box under **Properties\Resources.resx**.
* Click **OK** in the **Select Resource** dialog. Visual Studio displays the image to the left of New.
* Repeat the last 3 steps for File -> Open with openHS.png, and for File -> Save with saveHS.png.

***The first time I tried adding images to the menu items, I may have failed to follow the steps exactly, and the box the box under Properties\Resources.resx showed some strange file names. I saw files named*** DocumentHS1.png, openHS1.png, ***and*** saveHS1.png. ***There is a remedy. Open Resources.resx - via the Solution Explorer; in Visual Basic by double-clicking My Project , and then selecting the Resources tab (in the left panel). You will see these same strange files displayed in Visual Studio’s center pane. To remove one of these files, click the file, and then click RemoveResource in the toolbar at the top of the center pane. But, as a result of removing a resource, when you return to the designer view of the form, Visual Studio – at least in the C# version - might display an error message complaining that the removed resource is needed (e.g. in a menu item). The error message refers to a line of code in Form1.Designer.cs - in the region titled*** “Windows Form Designer Generated Code” - ***where the programmer is warned not to make modifications. The example that I encountered was in a statement that ended with***

= global::MenuToolCS.Properties.Resources.openHS1;

“openHS” ***was the correct name of the image;*** “openHS1” ***was erroneous. In spite of the warning, I deleted “*1*” from the statement, and that resolved the problem.***

* Drag a **ToolStrip** from the toolbox to the main form. Visual Studio automatically docks the control at the top of the form.
* Click the dropdown button at the right edge of the **ToolStrip**. Visual Studio displays a dropdown list showing the various toolbar options. **Button** is very commonly used.
* Select **Button**, and Visual Studio automatically adds it to the horizontal list of tools. This button will be used for “new”.
* Add two more buttons for “open” and “save”. Add a vertical separator. Add three more buttons for “cut”, “copy”, and “paste”.
* Click the leftmost button, and open the **Properties**  window. Rename the toolbar button to “NewToolStripButton”.
* Next to change its image, click **Image** property. Visual Studio responds by displaying a small box containing “...” to the right of **Image**.
* Click “...”. Visual Studio responds by displaying the **Select Resource** dialog.
* Click the **Project Resource** radio button. Visual Studio enables the list of resources below the **Project Resource** radio button.
* Select DocumentHS from the list of resources. Then click **OK**. Visual Studio changes NewToolStripButton’s image to the DocumentHS image.
* Click the **ToolTip Text** property; then type “New” to set its property. This text is shown when the mouse hovers over a control.
* Rename tool-strip buttons, add images, and supply tool-tip text for the remaining tool-strip buttons. You will need to import images for “cut”, “copy”, and “paste”.
* Select the File -> New menu item. Click the lightning icon at the top of the item’s **Properties** window to switch the “events” view.
* Select **Click** in the list of events. Double-click the menu item. Visual Studio creates a skeleton of the event handler. Repeat this for File -> Open and File -> Save.
* In the body of the File -> New’s **Click** event handler, type

MsgBox("You have clicked New")

* Put similar code into the event handlers for File -> Open and File -> Save.
* Select the “new” toolstrip button. Click the lightning icon at the top of the item’s **Properties** window to switch the “events” view.
* Select **Click** in the list of events. Click the down-arrow at the right edge of the “value” for this event. Visual studio displays a candidate list of events. Select the event handler for the File -> New menu item. (We are letting the menu item share an event handler with the toolstrip item.)
* Repeat the last two steps for the “open” and “save” toolstrip buttons.
* The “cut”, “copy”, and “paste” toolstrip buttons could also have event handlers, but for now they would be new event handlers; we don’t yet have “cut”, “copy”, and “paste” menu items.

You can repeat all of the steps in this section for MenuToolCS, the C# version of this project. To make it possible to use this application without the mouse, I have made the following enhancements.

* Change the **TabStop** property to **True** for the Menu Strip and for the Tool Strip.
* Set the **TabIndex** property to 1 for the Menu Strip and to 2 for the Tab Strip.

The **TabStop** **True** value makes it possible for the user to navigate to the Menu Strip and to the Tool Strip by typing the **Tab** key on the keyboard. When the cursor is at the Menu Strip, the user employ use all 4 arrow keys on the keyboard to navigate through the menu. When the cursor is at the Tool Strip, the user can employ the right and left arrow keys to navigate.

## Demo: User Controls

***I have added the Visual Basic code for this demo to the Visual Studio solution*** MenuTool.sln***. Similarly, I have added the C# .NET code for this demo to the Visual Studio solution*** MenuToolCS.sln***.***

* Add a class called “Person”.
* Person has 3 **Public** properties: FirstName (string), LastName (string), and Age (integer).
* Allocate a **Load** event for the form. In the body of this event type

Dim p As New Person With {

.FirstName = "Rob",

.LastName = "Windsor",

.Age = 49

}

This code instantiates and initializes p. Notice that the “\_”, which (in Visual Basic) indicates continuation to the next line, seems to be unnecessary – at least in this case.

* Open a context menu for the project and click **Add -> New Item**, then click **User Control (Windows Forms)** in the center pane of the popup dialog. Change the name to “PersonControl”; then click **Add**. Visual Studio displays a designer square to be used by the programmer to add controls.
* Enlarge the square to accommodate 3 text boxes and 3 labels. Add the 3 text boxes from the toolbox stacked vertically. Add the 3 labels from the toolbox stacked vertically – to the left of the 3 text boxes. Change the text in Label1 to “First Name”. Change the name TextBox1 to FirstNameTextBox. Change the properties of the other controls similarly.
* Lasso the 6 controls and move them close to the upper-left corner. (Once lassoed, you can move them with the keyboard arrow keys.) Shrink the control to remove the wasted space.
* Navigate to the body of PersonControl. Type “Property”, and then the **Tab** keyboard key twice. Visual Studio responds by creating skeleton code for (1) a **Private** property, and (2) a **Public** property to **Get** and **Set** values.
* In the **Private** statement, change NewPropertyValue to \_person, and change the type String to Person.
* In the **Public** property, change NewProperty to Person, and make certain that the end of the statement is “As Person”.
* The names and types are settled, but we need to pack and unpack values from PersonControl’s text boxes in the **Public** property. Add the following code immediately before “Return \_person”.

\_person.FirstName = FirstNameTextBox.Text

\_person.LastName = LastNameTextBox.Text

\_person.Age = CInt(AgeTextBox.Text)

Add the following code immediately before “\_person = value”.

FirstNameTextBox.Text = value.FirstName

LastNameTextBox.Text = value.LastName

AgeTextBox.Text = value.Age

* Next we want to add an instance of PersonControl to the main form. Navigate to the main form in design mode. Notice that PersonControl does not show up in the **Toolbox**. We can make that happen by building the project. I rebuilt the entire solution, and PersonControl appeared in the **Toolbox** at the top. ***In C#, however, rebuilding the solution did not seem to work immediately. I was looking for a resolution via Google, and trying different remedies – none of which seemed to work. Then (magically) the control appeared in the toolbox.***
* Drag/drop PersonControl onto the main form.
* Navigate back to the **Load** event for the main form. Add a final statement “PersonControl1.Person = p”. PersonControl1 is the name of the PersonControl instance on the Main form. .Person is the **Public** property that we built above. “p” is the initial value defined at the start of the **Load** event.
* We can build and run the application now. The initial value is displayed. But we would also like to test our ability to change the value, and prove that the main form can obtain the changed value by copying a value from a property of the user control.
* Add a button to the main form. In the event handler for that button display – for example – the last name from the user control. Use “MsgBox(PersonControl1.Person.LastName)” in the button’s event handler.

***In the C# code, I experimented with an alternative that avoids the “Person” class, at least for the last step.***

***private void button1\_Click\_1(object sender, EventArgs e)***

***{***

***var textBoxCtls = personControl1.Controls.Find ( "txtLastName", true );***

***MessageBox.Show ( "last name: " + textBoxCtls[0].Text );***

***}***

***I am confident that I could dispense entirely with the “Person” class, unless there are future situations where it would be convenient to have first name, last name, and age wrapped into a class.***

## Demo: Creating the (MDI) Forms

This Pluralsight clip illustrates how to create the project and forms for an MDI application.

* Open Visual Studio. In the right panel click **Create a new project**. Visual Studio displays the **Create a new project** dialog.
* Open the languages drop-down list near the top of the right panel. Choose **Visual Basic**. Type “windows” in the search box at the top of the right panel. Visual Studio reduces the number of candidates in the right-panel list. Select **Windows Forms App (.NET Framework)** with **Visual Basic** as the language. Click **Next**. Visual Studio displays the **Configure your new project** dialog.
* Type “MDI” into the **Project name** text box. Use the **Location** browser to select a folder into which the solution will reside. Keep “MDI” as the **Solution name** text box. Check the check box titled **Place solution and project in the same directory**. Click **Create**. Visual Studio displays the solution with an empty form titled “Form1”. If the **Solution Explorer** is not visible in the right panel, navigate to make it visible.
* Rename the form in the **Solution Explorer** to “MainForm”. Visual Studio displays a pop-up dialog asking whether to rename all references to the code element ‘Form1’. Click **Yes**. Notice that Visual Studio has renamed the class (in code view) to “MainForm”.
* Resize the form, so that occupies the majority of the center pane in Visual Studio.
* Change the title of the form (**Text** property) to “MDI Text Editor”. In response you should see “MDI Text Editor” in the title bar (top left) of the form.
* Change the **IsMdiContainer** property to **True**. This is the key to making this an MDI application. In response Visual Studio darkens the form’s background color.
* Open a context menu for the project and click **Add -> New Item**, then click **Form (Windows Forms)** in the center pane of the popup dialog. Change the name to “ChildForm”; then click **Add**. Visual Studio displays a new form in Visual Studio’s center panel.
* Change the **Text** property of ChildForm to “New Document”.
* Drag/drop an instance of the **RichTextBox** control onto ChildForm.
* We want the **RichTextBox** to occupy the entire form. There are 2 convenient methods to do this. (1) Click the “smart tag” of the **RichTextBox** (an arrow at the upper-right corner), and select **Dock in Parent Container** from the context menu. (2) Set the **Dock** property of the **RichTextBox** to **Fill**.
* Rename the **RichTextBox** to DocumentTextBox.
* Drag/drop a **MenuStrip** onto MainForm.
* Click the “smart tag” of the **MenuStrip** (an arrow at the upper-right corner), and select **Insert Standard Items**. In response Visual Studio adds 4 frequently used menu items – File, Edit, Tools, and Help.
* There are two ways to implement the **Click** event for File -> New. (1) In the **Properties** window for File -> New, choose the **Events** (lightning-bolt) view. Double-click the **Click** event. (2) Use the default event for menu items: double-click File -> New. As a result of either technique Visual Studio creates the skeleton of the event handler.
* Now we use the event handler to create a new child form, and make it visible. Add the following code to the skeleton event handler created in the previous step.

Dim childForm As New ChildForm()

childForm.TopLevel = False

childForm.MdiParent = Me

childForm.Show()

I had to add the 2nd statement to get around an exception; by default ChildForm forms have the **TopLevel** property set to True.

* Before running the application, resize ChildForm to make it considerably smaller than MainForm.
* Run the application, and try clicking File -> New a few times. After each time the application adds a new ChildForm to the MainForm’s window. But refer to the notes a few lines farther down this document.
* The C# version is almost identical. I deliberately set the **Text** property of MainForm to “MDI Text Editor (C#)”.

***There were a few problems with the Visual Basic version – as follows; the C# version did not encounter these problems.***

* If the statement

childForm.TopLevel = False

is not present, an exception is raised in response to the *childForm.MDIParent = Me* statement:

**Top-level control cannot be added to a control.**

The only work-around that I could find is to include the *childForm.TopLevel = False*

* When File -> New is used to add a new child form, this child form is placed at the top of the parent form with half of its title bar hidden.
* When a child form is minimized, it temporarily disappears completely. It reappears (minimized) when File -> New is invoked to add a subsequent child form.
* I used the debugger in (File -> New)’s event handler. I set a breakpoint at the last statement. When the breakpoint caused the execution to pause . . .
  + I examined childForm.IsMdiChild, and I found that the value was False. The value should have been True.
  + I examined me.MdiChildren, and I found that its Length was 0. The value remained at 0 even after I added a few more child forms.
* While shadowing the Pluralsight instructor in the next clip (Demo: Window Menu) . . .
  + When I tried the Window submenus – Cascade, Tile Horizontally, and Tile Vertically, nothing changed on the screen.
  + The list of child windows was missing from the bottom of the Window menu.

These anomalies are not surprising, since the child forms are missing from the MdiChildren list.

***I have not been able to find a work-around for the problems listed above. Therefore, for the remainder of the MDI clips, I will develop only with C# code, and abandon Visual Basic.***

## Demo: Window Menu

* Delete the **Help** menu from the MainForm by clicking the **Help** menu, and by pressing the **Delete** key on the keyboard. Delete the **Tools** menu by clicking the **Tools** menu, and pressing the **Delete** key.
* Add the Window menu (type &Window).
* Add submenus under the Window menu: &Cascade, Tile &Horizontally, and Tile &Vertically. Notice – in the **Properties** window - that Visual Studio has already provided sensible names for these submenus.
* Change a property of the menu strip itself. Click the menu strip, and examine the **Properties** window. Click the property **MdiWindowListItem**; open the dropdown list to the right, and select WindowToolStripMenuItem. As we will see, this selection automatically creates a list of all of the child forms, and it places this list at the bottom of the Window pull-down menu. It will also indicate which of these is the “active” form, and can be used to facilitate switching “active” to a different child form.
* Double-click “Window->Cascade” to add its event handler. Add “This.LayoutMdi(MdiLayout.Cascade);” to the body of this event handler.
* Double-click “Window->Tile Horizontally” to add its event handler. Add “This.LayoutMdi(MdiLayout.TileHorizontal);” to the body of this event handler.
* Double-click “Window->Tile Vertically” to add its event handler. Add “This.LayoutMdi(MdiLayout.TileVertical);” to the body of this event handler.
* Build and test the application. Observe that the Window menu buttons provide the appropriate functionality. Observe, also, that **MdiWindowListItem** provides the functionality described above.

## Demo: Edit Menu

In the clip [Demo: Creating the (MDI) Forms](#_Demo:_Creating_the) I obtained **Standard Items** for the menu. Examine the Edit menu and its submenus – Undo, Redo, Cut, Copy, Paste, and Select All. We will be using all of these options for the editor; therefore, we can leave this menu unchanged, and simply implement their event handlers. These actions apply to the text in the currently active window. The work we need to do is (1) get a reference to the form in the currently active window, and this will be used – in turn – to get access to its **RichTextBox** control. We note that the actions (Undo, Redo, Cut, Copy, Paste, and Select All) are all methods of the **RichTextBox** class; therefore, implementation is straightforward.

* Implement Undo. Double-click to create its event handler. Then type the following text into the body of this event handler.

If ( this.ActiveMdiChild != null )

{

var childForm = (ChildForm) this.ActiveMdiChild;

childForm.documentTextBox.Undo();

}

The name we gave to our **RichTextBox** instance is “DocumentTextBox”, but the statement above refers to this instance as “documentTextBox”. (See ChildFormDesigner.cs.) Observe that “DocumentTextBox” is a **private** property; therefore, I wrote a **get** accessor immediately below DocumentTextBox’s declaration and I named it “documentTextBox”.

* I implemented the remaining 5 Edit submenus in similar fashion – with changes only to different methods of the **RichTextBox** control.
* Compile and test the application. Try the various Edit menu buttons.

The Pluralsight instructor suggested a different way of getting around the fact that “DocumentTextBox” is a **private** property. He changed DocumentTextBox’s **Modifiers** property from **private** to **internal**. (Incidentally, he also changed the name of this instance to “documentTextBox”.) I’m commenting out my accessor code and trying the instructor’s suggestion.

**internal** means “The type or member can be accessed by any code in the same assembly, but not from another assembly.”

## Demo: File Menu

In the clip [Demo: Creating the (MDI) Forms](#_Demo:_Creating_the) I obtained **Standard Items** for the menu. Examine the File menu and its submenus. I find that will not need some of these, and I will have to add one new submenu. (Use the **delete** key to remove menu items and horizontal separators. To insert above a menu item or separator, right-click and select **Insert** from the context menu.)

* Delete Print, Print Preview, and the horizontal separator that preceded Print.
* Delete Save As.
* Insert Close before the topmost horizontal separator.
* Use the **Properties** window to rename the Close menu item to “CloseToolStripMenuItem”.
* Change the implementation of File->New to rename the child forms to make the titles unique. Navigate to the top of MainForm.cs. Insert the statement

private int \_counter = 0;

In newToolStripMenuItem\_Click(), immediately before “childForm.show()”, insert the statement

ChildForm.text = “New Document “ + (++\_counter).ToString();

* Implement the Close submenu. Double-click to produce a skeleton event handler. Insert the following into the body of the event handler.

if (this.ActiveMdiChild != null)

{

this.ActiveMdiChild.Close();

}

* Implement the Exit submenu. Double-click to produce a skeleton event handler. Insert the following into the body of the event handler.

this.Close();

* Implement the Save submenu. Double-click to produce a skeleton event handler. Insert the following into the body of the event handler.

if (this.ActiveMdiChild != null)

{

var childForm = (ChildForm) this.ActiveMdiChild;

var dialog = new SaveFileDialog();

dialog.Filter = "Rich text files|\*.rtf";

dialog.AddExtension = true;

var result = dialog.ShowDialog();

if (result == DialogResult.OK)

{

childForm.documentTextBox.SaveFile(dialog.FileName);

childForm.Text = dialog.FileName;

}

}

The dialog.AddExtension statement causes the file name to be saved with the default extension, which – in this case – will be “.rtf” (derived from the dialog.Filter` setting).

* Implement the Open submenu. Double-click to produce a skeleton event handler. Insert the following into the body of the event handler.

var dialog = new OpenFileDialog();

dialog.Filter = "Rich text files|\*.rtf";

var result = dialog.ShowDialog();

if (result == DialogResult.OK)

{

var childForm = new ChildForm();

childForm.documentTextBox.LoadFile(dialog.FileName);

childForm.Text = dialog.FileName;

childForm.MdiParent = this;

childForm.Show();

}

* Build the application, and test the various file-menu facilities.

## Demo: Toolbar

In this clip we will add a toolbar.

* Drag/drop the **ToolStrip** control from the **Toolbox**.
* Click the “smart tag” of the **ToolStrip** (an arrow at the upper-right corner), and select **Insert Standard Items**. In response Visual Studio adds several frequently used toolbar icons.
* Delete “print”, “help” (**?**), and the final separator. (To delete, use right-click; then click **Delete** in the context menu.)
* We need an event handler for the “New” toolbar button. The simple way to do this is to invoke the File->New event handler to this button.
  + Click the “New” toolbar button.
  + Navigate to the **events** view in the **Properties** window.
  + Expand the drop-down list to the right of the **Click** event.
  + Click newToolStripMenuItem\_Click in the context menu.
* Repeat the above for the remaining tool-strip items.
* Build and test the MDI application – specifically the toolbar buttons.

## SDI Applications

SDI Applications are like MDI Applications, but SDI Applications do not have a top-level form. SDI Applications allow the programmer the option of **single** instance, which means . . .

If an SDI Application is already running when the user tells Windows to start the application again, Windows presents the user an additional form, but this form runs under the same instance of the application as the forms that were already presented to the user. The Application continues to run until all of the forms are closed.

When programming in Visual Basic, this functionality is provided by configuring a Visual Studio using a dialog called **Application Framework**. This produces a class “WindowsFormsApplicationBase”.

When programming in C#, the **Application Framework** dialog is not available, but some of the functionality can still be obtained by creating the application in a class derived from “WindowsFormsApplicationBase”. What seems to be unavailable to C# is the same integration (as provided to Visual Basic SDI), and the same kind of code generation. (***I don’t understand what the last sentence means, but I am hoping it will make more sense when I follow the demos in subsequent clips.***)

## Demo: Creating the (SDI) Project

* Open Visual Studio. In the right panel click **Create a new project**. Visual Studio displays the **Create a new project** dialog.
* Open the languages drop-down list near the top of the right panel. Choose **Visual Basic**. Type “windows” in the search box at the top of the right panel. Visual Studio reduces the number of candidates in the right-panel list. Select **Windows Forms App (.NET Framework)** with **Visual Basic** as the language. Click **Next**. Visual Studio displays the **Configure your new project** dialog.
* Type “SDI” into the **Project name** text box. Use the **Location** browser to select a folder into which the solution will reside. Keep “SDI” as the **Solution name** text box. Check the check box titled **Place solution and project in the same directory**. Click **Create**. Visual Studio displays the solution with an empty form titled “Form1”. If the **Solution Explorer** is not visible in the right panel, navigate to make it visible.
* Rename the form in the **Solution Explorer** to “DocForm”. Visual Studio displays a pop-up dialog asking whether to rename all references to the code element ‘Form1’. Click **Yes**. Notice that Visual Studio has renamed the class (in code view) to “DocForm”.
* Make the form a little bit larger.
* Change the title of the form (**Text** property) to “New Document”.
* Copy the menu and toolbar from the **MDI** application to DocForm. (There is no “main” form in SDI; so, each of the document forms has its own menu and toolbar.)
  + Open the **MDI** Visual Studio project.
  + Navigate to the Main form (design view).
  + Copy the menu into the clipboard.
  + Navigate to the **SDI** Visual Studio project.
  + Paste from the clipboard to DocForm (design view).
  + Navigate to the Main form of the **MDI** Visual Studio project
  + Copy the toolstrip into the clipboard.
  + Navigate to the **SDI** Visual Studio project.
  + Paste from the clipboard to DocForm (design view).
* Drag/drop an instance of the **RichTextBox** control onto DocForm.
* We want the **RichTextBox** to occupy the entire form. There are 2 convenient methods to do this. (1) Click the “smart tag” of the **RichTextBox** (an arrow at the upper-right corner), and select **Dock in Parent Container** from the context menu. (2) Set the **Dock** property of the **RichTextBox** to **Fill**.
* Rename the **RichTextBox** to DocumentTextBox.

Build and run the application. You can type into rich text box, but the menu/toolbar buttons are not yet wired, and it’s also not an SDI application. For that, we need to configure the properties of the application object. Refer to the next clip.

Repeating these instructions for C# is straightforward.

## Demo: Configuring The Application Object (VB)

This clip describes how to configure the Application object (for SDI in Visual Basic). This is relatively easy, because we can use the **Application Framework** dialog.

* Navigate to the **Solution Explorer**. Right-click the project name (“SDI”), and click **Properties** in the context menu. Visual Studio displays a dialog with no title, but it contains a vertical list of tabs on the right edge – starting with **Application**. (If **Application** is not selected by default, click this tab.)
* Examine the drop-down list titled **Startup form:**. If “DocForm” (the result of our renaming) is not selected by default, use the drop-down arrow to select “DocForm”.
* Note the subtitle **Windows application framework properties** about half-way down the page.
* Ensure that **Make single instance application** is checked. (Refer to “single instance” in [SDI Applications](#_SDI_Applications).)
* Ensure that **Shutdown mode** contains **When last form closes**.
* Click **Save All** - either under Visual Studio’s **File** menu, or in Visual Studio’s toolbar.
* Navigate to the **Solution Explorer**. Click **Show All Files** (in **Solution Explorer**’s toolbar) to make hidden files visible.
* Expand the tree SDI / My Project / Application.myapp, and then click (to view) Application.Designer.vb. In this automatically generated file, we can see the selections that we made in the previous steps.

It may appear that we have achieved our goal – having expressed out intent for “single instance” and “shutdown when last form closes”. But Visual Studio has not provided all of this functionality automatically; there is additional configuration that we need to express via code.

The method **OnCreateMainForm()** (in Application.Designer.vb) contains the statement that specifies the chosen form (DocForm) to display when the application starts up. But the user might decide to invoke another instance of the application, and we need to tell the program what to do in that case.

* Again, navigate to the **Solution Explorer**. Right-click the project name (“SDI”), and click **Properties** in the context menu. Visual Studio displays a dialog with no title, but it contains a vertical list of tabs on the right edge – starting with **Application**. (If **Application** is not selected by default, click this tab.)
* Click the command button **View Application Events** (in the bottom line of the screen). Visual Studio displays the file **ApplicationEvents.vb**. Notice that one of the comments in this file contains “StartupNextInstance:”.
* Expand the **MyApplication** dropdown list at the top-center of the screen; then click **(my application events)**. Visual Studio shows a dropdown list of events in the top-right corner of the screen.
* Expand the dropdown list of events, and click **StartupNextInstance**. Visual Studio adds a skeleton of the **StartupNextInstance** event handler into the body of class **MyApplication**.
* It is our responsibility to express (in code) what to do in this event. What we want to do is create a new instance of DocForm, and show it. The Pluralsight instructor has the foresight to know that this same functionality is needed in more than one place; therefore, we will write a method that can be shared, and we will place this method in DocForm’s class.
* Navigate to the code view of DocForm. Add the following code into Class DocForm.

Private Shared \_counter As Integer = 0

Public Shared Function CreateForm() As DocForm

Dim form as New DocForm()

\_counter += 1

form.Text = “New Document “ & \_counter

My.Application.ApplicationContext.MainForm = form

form.Show()

Return form

End Function

* Return to the **StartupNextInstance** event handler in the body of class **MyApplication**. Insert the statement

DocForm.CreateForm()

Into the body of the event handler.

* Build the application.

To test the application, we want to watch what happens each time the application is invoked: that a new copy of the form is displayed without introducing a new instance of the application. Moreover, the application should continue to run until we close the last instance of the form.

* Navigate in a Windows file explorer to the folder (SDI) that contains the solution.
* Drill down to SDI/bin/debug to expose SDI.exe.
* Create a shortcut of SDI.exe onto the desktop.
* Display only the desktop (e.g. type WindowsKey+d).
* Display the Task Manager.
* Double-click the SDI shortcut. SDI should appear in the Task Manager, and a form titled “New Document” should appear on the desktop.
* Double-click the SDI shortcut again a few times. You should see only one instance of SDI in the Task Manager. You should also see additional instances of the form on the desktop, labeled “New Document 1”, “New Document 2”, etc.
* Delete the forms, but arrange to delete “New Document 2” last. SDI should remain visible in the Task Manager until you delete “New Document 2”.

## Demo: Configuring The Application Object (CS)

* Open Visual Studio. In the right panel click **Create a new project**. Visual Studio displays the **Create a new project** dialog.
* Open the languages drop-down list near the top of the right panel. Choose **C#**. Type “windows” in the search box at the top of the right panel. Visual Studio reduces the number of candidates in the right-panel list. Select **Windows Forms App (.NET Framework)** with **C#** as the language. Click **Next**. Visual Studio displays the **Configure your new project** dialog.
* Type “SDIcs” into the **Project name** text box. Use the **Location** browser to select a folder into which the solution will reside. Keep “SDIcs” as the **Solution name** text box. Check the check box titled **Place solution and project in the same directory**. Click **Create**. Visual Studio displays the solution with an empty form titled “Form1”. If the **Solution Explorer** is not visible in the right panel, navigate to make it visible.
* Rename the form in the **Solution Explorer** to “DocForm”. Visual Studio displays a pop-up dialog asking whether to rename all references to the code element ‘Form1’. Click **Yes**. Notice that Visual Studio has renamed the class (in code view) to “DocForm”.
* Change the title of the form (**Text** property) to “New Document”.
* Copy the menu and toolbar from the **MDIcs** application to DocForm. (There is no “main” form in SDI; so, each of the document forms has its own menu and toolbar.)
  + Open the **MDIcs** Visual Studio project.
  + Navigate to the Main form (design view).
  + Copy the menu into the clipboard.
  + Navigate to the **SDIcs** Visual Studio project.
  + Paste from the clipboard to DocForm (design view).
  + Navigate to the Main form of the **MDIcs** Visual Studio project
  + Copy the toolstrip into the clipboard.
  + Navigate to the **SDIcs** Visual Studio project.
  + Paste from the clipboard to DocForm (design view).
* Drag/drop an instance of the **RichTextBox** control onto DocForm.
* We want the **RichTextBox** to occupy the entire form. There are 2 convenient methods to do this. (1) Click the “smart tag” of the **RichTextBox** (an arrow at the upper-right corner), and select **Dock in Parent Container** from the context menu. (2) Set the **Dock** property of the **RichTextBox** to **Fill**.
* Rename the **RichTextBox** to DocumentTextBox.

It would be convenient if we could proceed to configure the application in the same manner as in the previous clip – using a dialog in the **Properties** of the project. But Visual Studio has not provided this within the C# framework. ***Confirm this by (1) right-clicking the project name (SDIcs), (2) selecting*** Properties ***from the context menu, and (3) examining what is provided under the*** Application ***tab – by comparison with the same dialog under the Visual Basic framework.*** Instead, it will be necessary to provide the application configuration via C# code.

* Right-click the project (SDIcs), and select **Add->Class...** In the context menu. Visual Studio pops up a dialog **Add New Item** with **Class** selected (by default) in the center pane.
* Type SDIApplication into the **Name** box at the bottom of the dialog. Click **Add**. Visual Studio adds the class to the solution.
* Add the attribute **public** in front of “class SDIApplication” (in SDIApplication.cs).
* To make this a single-instance class use the following code in the body of class SDIApplication:

private static SDIApplication \_instance = null;

public static SDIApplication Instance

{

get

{

if (\_instance == null)

\_instance = new SDIApplication();

return \_instance;

}

}

* To make this an “application”, we will let it inherit from WindowsFormsApplicationBase. Since WindowsFormsApplicationBase is built into the VisualBasic assembly, we will need an assembly reference.
  + Right-click **References** in the **Solution Explorer**. Visual Studio pops up a dialog titled **Reference Manager**.
  + Make certain that **Assemblies** is selected in the left panel. Then click a check box next to **Microsoft.VisualBasic**.
  + Click **OK**.
  + Navigate back to SDIApplication.cs.
  + Type “ : WindowsFormsApplicationBase” after “public class SDIApplication”.
  + Hover the mouse on WindowsFormsApplicationBase; Visual Studio will display a few suggestions about to remedy the error; click the suggestion to add the **using** statement.

Next we’ll add a constructor to the SDIApplication class and set some properties.

* After the line that declares \_instance, type “ctor” followed by the **Tab** key. Visual Studio creates the skeleton constructor.

What properties do we need to set to get the same functionality as in SDI (for Visual Basic)? The answer is found by reviewing what we did when we used the dialog to set the application’s properties. Refer to [Demo: Configuring The Application Object (VB)](#_Demo:_Configuring_The) in the paragraph pertaining to the portion of the dialog with the subtitle **Windows application framework properties**. These properties were (1) **Make single instance application** (turn this “on”), and (2) ensure that **Shutdown mode** is set to **When last form closes**. The code for these settings is in Application.Designer.vb.

* Open SDI (Visual Basic) to examine Application.Designer.vb.
* The two pertinent statements are “Me.IsSingleInstance = true” and “Me.ShutdownStyle = Global.Microsoft.VisualBasic.ApplicationServices.ShutdownMode.AfterAllFormsClose”.
* Type similar statements into the SDIApplication() constructor.

this. IsSingleInstance = true;

this. ShutdownStyle = ShutdownMode.AfterAllFormsClose;

In addition to the above there were two events requiring event handlers in SDI (for Visual Basic): (1) when the Main form is created, and (2) when the user attempts to launch a new instance of an application that is already active. The code for (1) is in Application.Designer.vb - method OnCreateMainForm(); this was part of the code generated by Visual Studio. The code for (2) is in ApplicationEvents.vb - method MyApplication\_StartupNextInstance(); this is code that we added. In both cases the appropriate code is code that is similar to DocForm.CreateForm().

* At the end of the body of class SDIApplication type code for the two event handlers.

protected override void OnCreateMainForm()

{

DocForm.CreateForm();

}

protected override void OnStartupNextInstance ( StartupNextInstanceEventArgs eventArgs )

{

DocForm.CreateForm();

}

* As we did in SDI (for Visual Basic), we code the implementation of CreateForm() in DocForm.cs.

private static int \_counter = 0;

public static DocForm CreateForm()

{

DocForm form = new DocForm();

form.Text = "New Document " + (++\_counter);

SDIApplication.Instance.ApplicationContext.MainForm = form;

form.Show();

return form;

}

If you build and run the application now, you might notice that the form’s title (**Text** property) is “New Document”, not “New Document 1”. This is because the application’s start-up code is not invoking OnCreateMainForm() (in class SDIApplication). We need to modify the application’s start-up code.

* Navigate to Program.cs.
* Replace the last statement in Main() with

SdiApplication.Instance.Run ( args );

* And “args”, which is not yet defined, specifies the command-line arguments when invoking a program. Change “static void Main()” to “static void Main ( string[] args )”.
* Build the application; there should be no compile errors.
* Run the application, and observe that the form’s title is now “New Document 1”.

To test the application, we want to watch what happens each time the application is invoked: that a new copy of the form is displayed without introducing a new instance of the application. Moreover, the application should continue to run until we close the last instance of the form.

* Navigate in a Windows file explorer to the folder (SDIcs) that contains the solution.
* Drill down to SDIcs/bin/debug to expose SDIcs.exe.
* Create a shortcut of SDIcs.exe onto the desktop.
* Display only the desktop (e.g. type WindowsKey+d).
* Display the Task Manager.
* Double-click the SDI shortcut. SDI should appear in the Task Manager, and a form titled “New Document 1” should appear on the desktop.
* Double-click the SDI shortcut again a few times. You should see only one instance of SDI in the Task Manager. You should also see additional instances of the form on the desktop, labeled “New Document 2”, “New Document 3”, etc.
* Delete the forms, but arrange to delete “New Document 2” last. SDI should remain visible in the Task Manager until you delete “New Document 2”.

## Demo: File and Edit Menu

This describes how to code for the Menu and Tool Strip – with regard to File and Edit. To begin with the following are not needed in an SDI Application: Window -> Cascade, Window -> Tile Horizontally, Window -> Tile Vertically, and File -> Exit. These need to deleted in DocForm.vb - design mode.

The Pluralsight instructor copied (and then modified) the event handlers that were built in the MDI application for Visual Basic. Since I did not complete that application, I created skeleton event handlers for the various menu and tool-strip buttons (see the end of [Demo: Menu and Toolbars](#_Demo:_Menu_and) for a description of how to do this); then I filled in the code using the Pluralsight instructor’s instructions as a guide. The results are as follows.

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

newToolStripMenuItem.Click, newToolStripButton.Click

CreateForm()

End Sub

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

openToolStripMenuItem.Click, openToolStripButton.Click

Dim dialog As New OpenFileDialog()

dialog.Filter = "Rich text files|\*.rtf"

Dim result = dialog.ShowDialog()

If result = DialogResult.OK Then

Me.DocumentTextBox.LoadFile ( dialog.FileName )

Me.Text = dialog.FileName

End If

End Sub

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

CloseToolStripMenuItem.Click

Me.Close()

End Sub

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

saveToolStripMenuItem.Click, saveToolStripButton.Click

Dim dialog As New SaveFileDialog()

dialog.Filter = "Rich text files|\*.rtf"

dialog.AddExtension = True

Dim result = dialog.ShowDialog()

If result = DialogResult.OK Then

Me.DocumentTextBox.SaveFile ( dialog.FileName )

Me.Text = dialog.FileName

End If

End Sub

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

undoToolStripMenuItem.Click

Me.DocumentTextBox.Undo()

End Sub

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

redoToolStripMenuItem.Click

Me.DocumentTextBox.Redo()

End Sub

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

cutToolStripMenuItem.Click, cutToolStripButton.Click

DocumentTextBox.Cut()

End Sub

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

copyToolStripMenuItem.Click, copyToolStripButton.Click

DocumentTextBox.Copy()

End Sub

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

pasteToolStripMenuItem.Click, pasteToolStripButton.Click

DocumentTextBox.Paste()

End Sub

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

selectAllToolStripMenuItem.Click

DocumentTextBox.SelectAll()

End Sub

The instructions (above) in this section can be applied to the C# version of the SDI application (SDIcs). It is straightforward to translate the event handlers from Visual Basic to C#. The results are as follows.

private void newToolStripMenuItem\_Click ( object sender, EventArgs e )

{

CreateForm();

}

private void openToolStripMenuItem\_Click ( object sender, EventArgs e )

{

var dialog = new OpenFileDialog();

dialog.Filter = "Rich text files|\*.rtf";

var result = dialog.ShowDialog();

if ( result == DialogResult.OK )

{

DocumentTextBox.LoadFile ( dialog.FileName );

this.Text = dialog.FileName;

}

}

private void CloseToolStripMenuItem\_Click ( object sender, EventArgs e )

{

this.Close();

}

private void saveToolStripMenuItem\_Click ( object sender, EventArgs e )

{

var dialog = new SaveFileDialog();

dialog.Filter = "Rich text files|\*.rtf";

dialog.AddExtension = true;

var result = dialog.ShowDialog();

if (result == DialogResult.OK)

{

DocumentTextBox.SaveFile ( dialog.FileName );

this.Text = dialog.FileName;

}

}

private void undoToolStripMenuItem\_Click ( object sender, EventArgs e )

{

DocumentTextBox.Undo();

}

private void redoToolStripMenuItem\_Click ( object sender, EventArgs e )

{

DocumentTextBox.Redo();

}

private void cutToolStripMenuItem\_Click ( object sender, EventArgs e )

{

DocumentTextBox.Cut();

}

private void copyToolStripMenuItem\_Click ( object sender, EventArgs e )

{

DocumentTextBox.Copy();

}

private void pasteToolStripMenuItem\_Click ( object sender, EventArgs e )

{

DocumentTextBox.Paste();

}

private void selectAllToolStripMenuItem\_Click ( object sender, EventArgs e )

{

DocumentTextBox.SelectAll();

}

## Demo: Window Menu

With regard to the Window Menu, we have already deleted Window -> Cascade, Window -> Tile Horizontally, Window -> Tile Vertically. But we do want the dropdown list that shows all of the open windows, and that provides an easy way to select the “active” window (the window that has focus). In an MDI application this was easy to accomplish; however, in an SDI application, we’ll have to write a small amount of code in an event handler.

* Navigate to DocForm.vb in design view, and open the **Properties** window for the Wndow-menu button. Switch to the **Events** (lightning bolt) section of the **Properties** window, and double-click the **DropDownOpening** event. Visual Studio creates a skeleton of the event handler in DocForm.vb (code view).
* What we need to do in the event handler is to find out whether the Window menu already has a drop-down list. (This would be the case if the user has clicked “Window” previously.) If so, we want to remove that dropdown list. The code for doing this is

If WindowToolStripMenuItem.DropDownItems.Count > 0 Then

WindowToolStripMenuItem.DropDown.Dispose()

End If

* Next we want to obtain the list of open forms; this is available from the application object. Then rebuild the Window dropdown list. The code for this part of the event handler is

windowToolStripMenuItem.DropDown = New ToolStripDropDown

For Each openForm As Form In My.Application.OpenForms

Dim childItem As New ToolStripMenuItem

childItem.Text = openForm.Text

childItem.Tag = openForm

WindowToolStripMenuItem.DropDownItems.Add ( childItem )

AddHandler childItem.Click, AddressOf WindowMenuItemClick

Next

The only statements above that might not be self-explanatory are “childItem.Tag . . .” and “AddHandler . . .”.

* + .NET makes the **Tag** property available to the programmer to store an object containing additional information about the item. In this case we use the **Tag** property of a menu item to record a reference to the form that the menu item refers to. This form reference will be used after the user clicks the menu item.
  + The AddHandler statement specifies the event handler that is triggered when the user clicks a menu item. We have specified the name of this handler (delegate) to be “WindowMenuItemClick”
* Finally, we implement “WindowMenuItemClick”, the menu-item-click event handler. The code is

Private Sub WindowMenuItemClick ( sender As Object, e As EventArgs )

Dim menuItem = CType ( sender, ToolStripMenuItem )

Dim form = CType ( menuItem.Tag, Form )

form.Activate()

End Sub

The argument “sender” contains a reference to the menu item that the user clicked. The **Tag** property of that menu item contains a reference to the form that the user selected.

Build and test the SDI application.

The instructions (above) in this section can be applied to the C# version of the SDI application (SDIcs). It is straightforward to translate the event handlers from Visual Basic to C#. The results are as follows.

private void windowToolStripMenuItem\_DropDownOpening ( object sender, EventArgs e )

{

if ( windowToolStripMenuItem.DropDownItems.Count > 0 )

{

windowToolStripMenuItem.DropDown.Dispose();

}

windowToolStripMenuItem.DropDown = new ToolStripDropDown();

foreach ( Form openForm in Application.OpenForms )

{

var childItem = new ToolStripMenuItem();

childItem.Text = openForm.Text;

childItem.Tag = openForm;

WindowToolStripMenuItem.DropDownItems.Add ( childItem );

childItem.Click += WindowMenuItemClick;

}

}

private void WindowMenuItemClick ( Object sender, EventArgs e )

{

var menuItem = (ToolStripMenuItem) sender;

var form = (Form) menuItem.Tag;

form.Activate();

}

Build and test the SDI application.

## Demo: Creating The Project (explorer style)

* Open Visual Studio. In the right panel click **Create a new project**. Visual Studio displays the **Create a new project** dialog.
* Open the languages drop-down list near the top of the right panel. Choose **Visual Basic**. Type “windows” in the search box at the top of the right panel. Visual Studio reduces the number of candidates in the right-panel list. Select **Windows Forms App (.NET Framework)** with **Visual Basic** as the language. Click **Next**. Visual Studio displays the **Configure your new project** dialog.
* Type “Explorer” into the **Project name** text box. Use the **Location** browser to select a folder into which the solution will reside. Keep “Explorer” as the **Solution name** text box. Check the check box titled **Place solution and project in the same directory**. Click **Create**. Visual Studio displays the solution with an empty form titled “Form1”. If the **Solution Explorer** is not visible in the right panel, navigate to make it visible.
* Rename the form in the **Solution Explorer** to “ExplorerForm”. Visual Studio displays a pop-up dialog asking whether to rename all references to the code element ‘Form1’. Click **Yes**. Notice that Visual Studio has renamed the class (in code view) to “ExplorerForm”.
* Change the title (**Text** property) to Explorer-Style Application.
* Resize the form to take up most of the design area.
* Add a **ToolStrip** control from the **Toolbox**.
* Drop a **Panel** control from the **Toolbox** onto the form.
* Change the **BackColor** of the **Panel** to “Cornsilk”; you will find “Cornsilk” among the **Web** collection of colors.
* Make the **Panel** about 100% larger.
* Drop a **TextBox**, **Label**, and a **Button** from the **Toolbox** onto the **Panel**.
* To move the **Panel**, click it once with the mouse. The response is cross with 4 arrows – 2 horizontal and 2 vertical. Drag the 4-arrow cross to move the panel.
* Observe that when you move the **Panel**, the 3 controls that you added - in a previous step - move with the **Panel**. The **Panel** is a “container” control. It has its own collection of controls. The controls on the container are children of the container. The **Panel** is part of the form’s control collection, but the **TextBox**, **Label**, and **Button** are part of the **Panel**’s control collection.
* Delete the **Panel** (select the Panel, and type the **delete** key). Observe that the **TextBox**, **Label**, and **Button** are also deleted.
* Drag a **SplitContainer** from the **Toolbox** onto the form. Notice that this provides a panel on the left side, a splitter (2 closely spaced vertical lines) in the middle, and a panel on the right side. Hover the mouse on the splitter, and observe the 2-direction horizontal arrow. Drag the splitter to the left and then to right; notice that it moves the position of the splitter.
* Drop a **TreeView** control onto the left panel. Click the “smart tag” of the **TreeView** (an arrow at the upper-right corner), and select **Dock in Parent Container** from the context menu. Similarly drop and dock a **ListView** onto the right panel.
* Build and test the application. Drag the splitter to reapportion the two panels.

The C# version of this application is almost identical.

## Demo: Populating The Tree View

This describes how to populate the **TreeView** control in the Explorer-style application. The exercise is to show all of the folders under the “My Documents” folder. (In Windows 10, “My Documents” is now named C:\Users\*user\_*name\Documents.) In a subsequent section, we will write code to respond when the clicks the name of a folder; the application will show the names of all of the files in that folder (via the **ListView** control).

We will 1st add “My Documents” as a root node into the **TreeView**. Then we will add the names of the folders under “My Document” by recursively traversing its child folders.

* Navigate to ExplorerForm.vb in code view. If the skeleton of the ExplorerForm\_Load() event handler does not yet exist, create it via the **Events** section of the form’s **Properties** window. In the body of this event handler we will create the tree node for “My Documents”.
* Type

Dim docs As New TreeNode ( “My Documents” )

* Keep a record of the path to “My Documents” in the **Tag** property. Type

docs.Tag = My.Computer.FileSystem.SpecialDirectories.MyDocuments

* Add this **TreeNode** instance into the **Nodes** collection of the **TreeView** control. Type

TreeView1.Notes.Add ( docs )

* Now, we add the child nodes. GetFolders ( treeViewNode ) is a function that we will soon define. It processes the nodes that are descendants of treeViewNode. Invoke GetFolders() for “docs”. Type

GetFolders ( docs )

Create “GetFolders ( node As TreeNode )” as a private subroutine in Class ExplorerForm. The argument “node” specifies the node (or subfolder) of interest; node’s **Tag** property contains the full path of the subfolder. The statements in the body of GetFolders() are as follows.

* Get the full path of the subfolder from node.Tag. Type

Dim dir As New DirectoryInfo ( node.Tag )

DirectoryInfo is not yet recognized by this Visual Basic file; hover the mouse over DirectoryInfo, and type <ctrl>. to allow Visual Studio to generate the appropriate **Imports** statement.

* Create a loop to process each subfolder of the folder whose path is “dir”. Type

For Each childDir In dir.GetDirectories()

. . .

Next

* The following statements comprise the body of the loop. Start by allocating a tree node for the subfolder. Type

Dim childNode As New TreeNode ( childDir.Name )

* Keep a record of the path to childNode in the **Tag** property. Type

childNode.Tag = childDir.FullName

* Add the new node as a child of "node". Type

node.Nodes.Add ( childNode )

* Process the descendants of childNode. Type

GetFolders ( childNode )

Build and run the application. There is an error in the code that prevents the application from running to completion. The Pluralsight instructor described an exception that was handled silently; the exception halted execution without displaying a pop-up diagnostic. But on my computer the unhandled exception did result in a pop-up diagnostic.

The exception occurred because access is denied to the entities “My Music”, “My Pictures”, and “My Videos”. Access is denied, because these are really not folders; instead, they are links to folders stored elsewhere in the computer.

(Evidently exceptions in the Form-load event handler were silently handled at the time when this course was written. But with my newer version of Visual Studio, the exception does induce a pop-up diagnostic.)

As a work-around introduce **Try** and **Catch** blocks into GetFolders.

Try

. . .

Catch ex As Exception

. . .

End Try

Move the **For** loop into the **Try** block. Type the following into the **Catch** block.

MsgBox ( ex.Message )

Build and run the application again. This time we see messages telling us that access is denied to “My Music”, “My Pictures”, and “My Videos”, and all of the descendants of “My Documents” are displayed in the left panel.

There is an easy way to prevent “My Music”, “My Pictures”, and “My Videos” from being displayed in the left panel; check the attributes of the folder to determine whether it is “hidden”. Type the following immediately after the **For Each** statement.

If childDir.Attributes.HasFlag ( FileAttributes.Hidden ) Then

Continue For

End If

There remains one minor cosmetic change. When the application starts up, it would be nice to have the top tree node (“My Documents”) expanded. To attain this, at end of the body of ExplorerForm\_Load(), type

docs.Expand()

Translation of the above code to C# is straightforward. The result is as follows.

private void ExplorerForm\_Load ( object sender, EventArgs e )

{

var docs = new TreeNode ( "Documents" );

docs.Tag = Environment.GetFolderPath ( Environment.SpecialFolder.MyDocuments )

TreeView1.Nodes.Add ( docs );

GetFolders ( docs );

docs.Expand();

}

private void GetFolders ( TreeNode node )

{

var dir = new DirectoryInfo ( node.Tag.ToString() );

try

{

foreach ( var childDir in dir.GetDirectories() )

{

if ( childDir.Attributes.HasFlag ( FileAttributes.Hidden ) )

{

continue;

}

var childNode = new TreeNode ( childDir.Name );

childNode.Tag = childDir.FullName;

node.Nodes.Add(childNode);

GetFolders(childNode); }

}

catch ( Exception ex )

{

MessageBox.Show ( ex.Message );

}

}

## Demo: Populating The List View 1 – Items and Images

This describes how to populate the **ListView** control with items and images in the Explorer-style application. When the user clicks a tree-view node, the application will display only files (not subfolders) in the folder represented by the clicked node. Windows File Explorer provides view options specifying how to display the files – small icons, large icons, details, etc. The application will provide some of these options.

* Navigate to ExplorerForm.vb in design view. Double-click the **TreeView** control in the left panel. Visual Studio creates a skeleton for the **AfterSelect** event (by default).
* In the body of the **AfterSelect** event handler, create a **DirectoryInfo** instance to represent the folder selected in **TreeView**. The second event-handler calling-sequence argument, e, contains a reference to the **Node** object; the **Tag** property of the **Node** object contains the path to the folder. Type

Dim dir As New DirectoryInfo ( e.Node.Tag )

* Erase anything that might have been displayed from a previously selected folder. Type

ListView1.Items.Clear()

* Loop through the items in the selected folder. Type

For Each thisFile In dir.GetFiles()

. . .

Next

* 1st statement in body of loop; create a **ListViewItem** instance. One of the constructors for **ListViewItem** takes a string argument – the caption of the item; we’ll supply the item name. Type

Dim item As New ListViewItem ( thisFile.Name )

* 2nd statement in body of loop; add the item to the list view. Type

ListView1.Items.Add ( item )

Build and test the application. Select any folder known to have files. We see part of the captions, but we don’t see images. Navigate to the **Properties** of the **ListView** control; click the right-hand panel opposite the **View** property; click the down-arrow to examine the list of options - **LargeIcon**, **Details**, **SmallIcon**, **List**, and **Tile**. **LargeIcon** is the default; do not change it. The images for the items in the **ListView** control are provided in an image-list control.

* Drag two **ImageList** controls from the **Toolbox** onto the form. Rename the 1st of these “LargeImageList” and the 2nd of these “SmallImageList”.
* Set the **ColorDepth** property of LargeImageList to **Depth32Bit**.
* Set the **ImageSize** property of LargeImageList to 32, 32.
* Set the **ColorDepth** property of SmallImageList to **Depth32Bit**.
* Retain the **ImageSize** property of SmallImageList as 16, 16.
* Set the **LargeImageList** property of the **ListView** control to LargeImageList.
* Set the **SmallImageList** property of the **ListView** control to SmallImageList.
* Navigate back to the **AfterSelect** event handler.
* Insert the following code immediately after the “Dim item . . .” statement.

If Not LargeImageList.Images.ContainsKey ( thisFile.Extension ) Then

Dim thisIcon = Icon.ExtractAssociatedIcon ( thisFile.FullName )

LargeImageList.Images.Add ( thisFile.Extension, thisIcon )

End If

item.ImageKey = thisFile.Extension

* One can usually depend upon the fact that there is 1-1 association between file extensions and icons. But this does not necessarily hold true for executable files (with extension “.exe”). For that reason we do not want to reuse the icon that was given for a previous executable file. Therefore, insert the following statement near the start of event handler (i.e. before the “For Each . . .” statement).

LargeImageList.Images.RemoveByKey ( “.exe” )

* Rebuild and test the application. Now we see more of the file-name captions, and some of the associated icons are displayed, also.

Translation of the above code to C# is straightforward. The result is as follows.

private void treeView1\_AfterSelect ( object sender, TreeViewEventArgs e )

{

var dir = new DirectoryInfo ( e.Node.Tag.ToString() );

listView1.Items.Clear();

Large\_imageList.Images.RemoveByKey(".exe");

foreach ( var thisFile in dir.GetFiles() )

{

var item = new ListViewItem ( thisFile.Name );

if ( ! Large\_imageList.Images.ContainsKey ( thisFile.Extension ) )

{

var thisIcon = Icon.ExtractAssociatedIcon ( thisFile.FullName );

Large\_imageList.Images.Add ( thisFile.Extension, thisIcon );

}

item.ImageKey = thisFile.Extension;

listView1.Items.Add(item);

}

}

## Demo: Populating The List View 2 – Views

This section describes software to allow the user to change the view to a different choice in the Explorer Style application. In particular, support will be added for the “details” view.

The Pluralsight instructor made a couple of small changes to the code developed in the preceding clip. in the **AfterSelect** event handler he inserted 2 extra statements. Before

LargeImageList.Images.RemoveByKey ( “.exe” )

add

SmallImageList.Images.RemoveByKey ( “.exe” )

Also, before

LargeImageList.Images.Add ( thisFile.Extension, thisIcon )

add

SmallImageList.Images.Add ( thisFile.Extension, thisIcon )

Make similar changes for the C# version of the application. You can build and test these changes, by changing the **View** property of the **ListView** control to **SmallIcon**. (Don’t forget to change it back to **LargeIcon** after testing the application.)

The toolbar will be used to allow the user to change the view.

* Navigate to ExplorerForm.vb in design view. Click the down-arrow in the **ToolStrip** control. Select **Label** in the context menu.
* Change the **Text** property of the **ToolStrip Label** to “View”.
* To the right of the **ToolStrip Label**, add a **ComboBox** via the context menu. Set the **Name** of this **ComboBox** to ViewToolStripComboBox.
* Click the **Items** property of ViewToolStripComboBox. Click **…** at the right-hand edge of this property. Visual Studio displays a dialog titled **String Collection Editor**. Type into the dialog

Large Icons

Small Icons

Details

* Click **OK**. Visual Studio dismisses the dialog.
* Add an event handler for ViewToolStripComboBox’s **SelectedIndexChanged** event. Visual Studio creates a skeleton for the event handler.
* Type the following code into the body of the event handler.

Select Case ViewToolStripComboBox.Text

Case "Large Icons"

ListView1.View = View.LargeIcon

Case "Small Icons"

ListView1.View = View.SmallIcon

Case "Details"

ListView1.View = View.Details

End Select

* We need the default setting of ViewToolStripComboBox’s selected index to be the appropriate value corresponding to “Large Icons”. This is not a property that can be set at design time; it will need to be set in the **ExplorerForm\_Load** event. Type the following as the 1st statement in the **ExplorerForm\_Load** event handler.

ViewToolStripComboBox.SelectedIndex = 0

Build and test the application. Notice that the Large Icons view comes up by default. Use the combo box to change to Small Icons. That, too, is successful. Finally, try Details. Nothing is shown, because we have not yet configured this view.

* Start by specifying the columns. The Pluralsight instructor recommends doing this in code. Navigate to the **ExplorerForm\_Load** event handler.
* Type the following after the “ViewToolStripComboBox.SelectedIndex = 0” statement.

ListView1.Columns.Add ( "Name", 250 )

ListView1.Columns.Add ( "Date modified", 150 )

ListView1.Columns.Add ( "Size", 75, HorizontalAlignment.Right )

The header text of the 3 columns are “Name”, "Date modified", and "Size". Their widths (pixels) are 250, 150, and 75 respectively. We specified the horizontal alignment for the 3rd column. Evidently (based on the behavior of the application – left horizontal alignment is the default).

Build and test the application. We can see the icon and file name in the “Name” column. But the other two columns are empty, because we have not yet provided code to add their values to the list view.

The leftmost column contains the “Item” information of the ListView control. Columns 2 and 3 are sub-items. The “Item” was inserted into the ListView in the **AfterSelect** event handler; refer to the statement “ListView1.Items.Add(item)”; “item” and its properties were set earlier in the handler. After allocating “item”, the subitems and their values can be specified.

* After

Dim item As New ListViewItem ( thisFile.Name )

add

Dim lastWrite = thisFile.LastWriteTime

Item.SubItems.Add ( lastWrite.ToShortDateString() & " " & lastWrite.ToShortTimeString() )

Item.SubItems.Add ( System.Math.Ceiling ( thisFile.Length / 1024 ) & " KB" )

Build and test the application. Observe that the “Details” view now provides the pertinent information.

Translation of the above code to C# is straightforward. The results after the changes are as follows.

private void ExplorerForm\_Load(object sender, EventArgs e)

{

viewToolStripComboBox.SelectedIndex = 0;

ListView1.Columns.Add ( "Name", 250 );

ListView1.Columns.Add ( "Date modified", 150 );

ListView1.Columns.Add ( "Size", 75, HorizontalAlignment.Right );

var docs = new TreeNode("Documents");

docs.Tag = Environment.GetFolderPath ( Environment.SpecialFolder.MyDocuments );

TreeView1.Nodes.Add ( docs );

GetFolders ( docs );

docs.Expand();

}

private void treeView1\_AfterSelect(object sender, TreeViewEventArgs e)

{

var dir = new DirectoryInfo ( e.Node.Tag.ToString() );

listView1.Items.Clear();

Small\_imageList.Images.RemoveByKey ( ".exe" );

Large\_imageList.Images.RemoveByKey ( ".exe" );

foreach ( var thisFile in dir.GetFiles() )

{

var item = new ListViewItem ( thisFile.Name );

var lastWrite = thisFile.LastWriteTime;

// Columns 2 and 3 are sub-items

Item.SubItems.Add ( lastWrite.ToShortDateString() + lastWrite.ToShortTimeString() );

Item.SubItems.Add ( Math.Ceiling ( (double)thisFile.Length / 1024.0 ) + " KB");

if ( ! Large\_imageList.Images.ContainsKey(thisFile.Extension) )

{

var thisIcon = Icon.ExtractAssociatedIcon ( thisFile.FullName );

Small\_imageList.Images.Add(thisFile.Extension, thisIcon);

Large\_imageList.Images.Add(thisFile.Extension, thisIcon);

}

item.ImageKey = thisFile.Extension;

ListView1.Items.Add ( item );

}

}

private void viewToolStripComboBox\_SelectedIndexChanged(object sender, EventArgs e)

{

Switch ( viewToolStripComboBox.Text )

{

case "Large Icons":

listView1.View = View.LargeIcon;

break;

case "Small Icons":

listView1.View = View.SmallIcon;

break;

case "Details":

listView1.View = View.Details;

break;

}

}

## Data Binding

When data binding is used to connect simple controls (text box, check box, etc.) to a collection of data, this is called “Simple Data Binding”. The collection of data is typically a table, the control is connected to a column of the table, and the data elements in the column are of a simple type (string, integer, Boolean, etc.). Another way to look at Simple Data Binding is associating simple controls to **List<**SomeClass**>**, where the control is connected to a simple-type property of SomeClass. ***What determines the row of the table (or the index of the list)? I don’t know. But experiments seem to indicate that Data Binding uses the most recently selected row of the table. At start-up the default seems to be 0.***

When data binding is used to connect a list-type object (e.g. **List**<*type*>) to a list-type control (e.g. **ListBox** or **ListView**), this is called “Complex data binding”.

For many of the simple controls, there is the **DataBindings** property, which is inherited from the **Control** class; one uses the **DataBindings.Add()** method to specify the association between the control and the data source. This method has several overloads, but three pieces of information are required: (1) the name of the property of the control that is being bound; (2) a reference to the data source; (3) the name of the property that identifies the column in the data source

For complex data binding the association between the list-type object and the control is expressed via three properties of the control: (1) the value of the **.DataSource** property is set to the name of the list-type object; (2) the **.DisplayMember** property is set to the name of the property of the underlying type that can be displayed; (3) the **.ValueMember** property is set to the name of the property of the underlying type that would be used to identify the particular member of the list (e.g. a key).

## Demo: Preparing The Project (Data Binding)

This demo prepares the project that will be used to deal with Data Binding.

* Open Visual Studio. In the right panel click **Create a new project**. Visual Studio displays the **Create a new project** dialog.
* Open the languages drop-down list near the top of the right panel. Choose **Visual Basic**. Type “windows” in the search box at the top of the right panel. Visual Studio reduces the number of candidates in the right-panel list. Select **Windows Forms App (.NET Framework)** with **Visual Basic** as the language. Click **Next**. Visual Studio displays the **Configure your new project** dialog.
* Type “Data” into the **Project name** text box. Use the **Location** browser to select a folder into which the solution will reside. Keep “Data” as the **Solution name** text box. Check the check box titled **Place solution and project in the same directory**. Click **Create**. Visual Studio displays the solution with an empty form titled “Form1”. If the **Solution Explorer** is not visible in the right panel, navigate to make it visible.
* Rename the form in the **Solution Explorer** to “DataForm”. Visual Studio displays a pop-up dialog asking whether to rename all references to the code element ‘Form1’. Click **Yes**. Notice that Visual Studio has renamed the class (in code view) to “DataForm”.
* Change the title (**Text** property) to Data Binding.
* Resize the form to take up most of the design area.
* Add a class named “Category”. This is one of the objects to which we will be binding. Type the following code in the body of “Category”.

Public Property CategoryID As Integer

Public Property CategoryName As String

Public Sub New ( id As Integer, name As String )

Me.CategoryID = id

Me.CategoryName = name

End Sub

***When the above code is translated to C#, it is important to append*** { get; set; } ***after each of the field declarations; this makes them “properties” in addition to fields. Without*** { get; set; } ***errors occur when classes such as these are bound to controls.***

* Add a class named “Product”. This is another one of the objects to which we will be binding. Type the following code in the body of “Product”.

Public Property ProductID As Integer

Public Property ProductName As String

Public Property CategoryID As Integer

Public Property QuantityPerUnit As String

Public Property UnitPrice As Decimal

Public Property UnitsInStock As Integer

Public Property UnitsOnOrder As Integer

Public Property Discontinued As Boolean

Public Sub New ( id As Integer, name As String, categoryId As Integer, unitQuant As String,

price As Decimal, inStock As Integer, onOrder As Integer,

discontinued As Boolean )

Me.ProductID = id

Me.ProductName = name

Me.CategoryID = categoryId

Me.QuantityPerUnit = unitQuant

Me.UnitPrice = price

Me.UnitsInStock = inStock

Me.UnitsOnOrder = onOrder

Me.Discontinued = discontinued

End Sub

***When the above code is translated to C#, it is important to append*** { get; set; } ***after each of the field declarations; this makes them “properties” in addition to fields. Without*** { get; set; } ***errors occur when classes such as these are bound to controls.***

* Add a class named “ObjectSource”. Type the following code in the body of “ObjectSource”.

Private \_categories As List(Of Category)

Private \_products As List(Of Product)

Public Sub New()

\_categories = New List ( Of Category )

\_categories.Add ( New Category ( 1, "name 1" ) )

\_categories.Add ( New Category ( 2, "name 2" ) )

\_products = New List ( Of Product )

\_products.Add ( New Product ( 1, "name 1", 1, "description 1", 1.0, 1, 0, False ) )

\_products.Add ( New Product ( 2, "name 2", 2, "description 2", 2.0, 2, 1, True ) )

End Sub

Public Function GetCategories() As IList ( Of Category )

Return \_categories

End Function

Public Function GetProducts ( categoryId As Integer ) As IList ( Of Product )

Dim result = From p In \_products

Where p.CategoryID = categoryId

Select p

Return result.ToList()

End Function

If you want realistic data, you can replace the body of the constructor ( New() ) with data from Pluralsight’s “Exercise Files”, or you can copy it from the “Data” project on Stephen’s Windows 10 computer. Alternatively, you can add or change the artificial data that I started above – using imagination to attain variety.

Observe (above) that the “Dim result = . . . “ statement in GetProducts() uses a LINQ query.

* Navigate to DataForm.vb in design view.
* Drag a **ToolStrip** control from the **Toolbox** to DataForm.
* Drag a **TabControl** from the **Toolbox** to DataForm. We are going to have two views of the data – form view and a grid view; one page from the **TabControl** for each of these.
* Set the **TabControl**’s **Dock** property to **Fill**.
* Click the **TabControl**’s **TabPages** property. Click the button labeled “...” to the right of “(Collection)”. Visual Studio displays a pop-up dialog titled **TabPage Collection Editor**. Use this dialog to change the **Text** property of TabPage1 to “Form”, and the **Text** property of TabPage2 to “Grid”.
* Select the 2nd tab page (Grid). Drag a **DataGridView** control from the **Toolbox** to the **TabControl**. Visual Studio automatically displays the **DataGridView Tasks** dialog. Click **Dock in Parent Container.**
* Rename the **DataGridView** control to ProductDataGridView.

This completes preparation for data binding in Visual Basic. Translate the above steps to C#. The constructor for ObjectSource contains several statements, each of which allocates a new Product. One of the arguments in each of these statements takes a decimal value; in C# use the suffix “m” after a numeric literal to express “decimal literal”.

## Demo: Complex Data Binding

This demo illustrates complex data binding. The result will bind the category and products data to a Combo Box and to a List Box.

* Navigate to DataForm.vb in design view. Click the “Form” tab.
* Drag a **ComboBox** from the **Toolbox** to DataForm. Put it near the upper-left corner. Make it a little bit wider (to accommodate long names).
* Set the **Name** property of the **ComboBox** to CategoriesComboBox.
* Set the **DropDownStyle** property to **DropDownList**.
* Double-click DataForm’s title to induce Visual Studio to create a skeleton of the form-load event.
* Immediately before the event handler, type

Private \_source As New ObjectSource()

Recall that the constructor of ObjectSource populates the instance (\_source) with category data and product data.

* In the body of form-load event handler type

CategoriesComboBox.DataSource = \_source.GetCategories()

CategoriesComboBox.DisplayMember = "CategoryName"

The 1st statement assigns the **List** to the combo-box's **DataSource** property. The 2nd statement instructs – more specifically – to display the category object’s CategoryName member.

* Build and run the application. The complete list of categories shows up in the dropdown list.
* Drag a **ListBox** from the **Toolbox** to DataForm. Put it under the combo box, and align it horizontally with the combo box. Make it a little bit longer (to accommodate several product names).
* Set the **Name** property of the **ListBox** to ProductsListBox.
* Add a SelectedIndexChanged event handler of CategoriesComboBox. The intent is to respond when the user changes the selection in CategoriesComboBox: create a list of the products that are associated with that category, and bind this list to ProductsListBox. This association uses CategoryID – a property of both the Category class and the Product class.
* The CategoryID value can be stored in the **ValueMember** of CategoriesComboBox. Type the following as an additional statement in the body of the form-load event handler.

CategoriesComboBox.ValueMember = "CategoryID"

* Retrieve this CategoryID value in the SelectedIndexChanged event handler of CategoriesComboBox. Type the following statement into the event handler.

Dim catID = CategoriesComboBox.SelectedValue

* Build the list by using the CategoryID value (catID) as the argument to the method GetProducts(). Add the statement into the event handler.

ProductsListBox.DataSource = \_source.GetProducts ( catID )

* Instruct ProductsListBox to display the Product object’s ProductName member. Add the statement into the event handler.

ProductsListBox.DisplayMember = "ProductName"

* Build and run the application. It crashes with an Invalid Cast Exception. The problem is in the 1st statement of the Categories SelectedIndexChanged event handler. The expression

CategoriesComboBox.SelectedValue

provides a “category” object. (You can see this by hovering the mouse over “SelectedValue”.) What we wanted, instead, was the “value” property that the combo box holds for the selected category.

* It turns out the way to get the **SelectedValue** property to work correctly is to set the **ValueMember** property before setting the **DataSource** property. This means that the correction is to move the 1st statement in Form-load event handler two lines lower (after the “CategoriesComboBox.ValueMember” statement.
* Build and run the application. The error has been fixed.

In the C# version of this data-binding example, I tried an experiment (see CategoryNamePlus in the C# code) to test whether DisplayMember binding can be used with a property that is calculated (not merely retrieved from one of the class's fields). The answer is "yes". A practical example would be a combo box that displays a combination of FirstName and LastName, each of which is a property of a "person" class.

## [Demo: Simple Data Binding](#_Demo:_Complex_Data)

Recall that with Simple data binding (see [Data Binding) the software expresses an association between simple](#_Data_Binding) controls (text box, check box, etc.) and a column in a tabular data source. This clip uses simple data binding to display a few details about an instance of the “Product” class using data-bound text boxes.

* Navigate to DataForm.vb in design view. Click the “Form” tab.
* Drag three text boxes from the **Toolbox** to DataForm. Put them near the upper-center of the form.
* Set the names of the text boxes to NameTextBox, PriceTextBox, and StockTextBox.
* The method used to do Simple data bindings is

*SimpleControl* . DataBindings . Add ( *PropertyUsedToDisplay, DataSource,*

*PropertySpecifyingColumnInDataSource* )

* Data Bindings statements need to be invoked only once – when *DataSource* is specified. *DataSource* is the list of products for the selected category. Therefore, Data Binding statements are placed in CategoriesComboBox\_SelectedIndexChanged() - after the statements

Dim catID = CategoriesComboBox.SelectedValue

ProductsListBox.DataSource = \_source.GetProducts ( catID )

* The value of *DataSource* in the Data Bindings statements is “\_source.GetProducts ( catID )”, which also appears in the 2nd of the two statements in the previous step. Therefore, it is good practice to economize by using a common local variable “products” Replace the 2nd of the two statements above with

Dim products = \_source.GetProducts ( catID )

ProductsListBox.DataSource = products

* The Data Bindings statements can be placed at the bottom of the event handler.

NameTextBox.DataBindings.Add ( "Text", products, "ProductName" )

PriceTextBox.DataBindings.Add ( "Text", products, "UnitPrice" )

StockTextBox.DataBindings.Add ( "Text", products, "UnitsInStock" )

* Build and test the application. The Beverages display without a problem. And we also notice that selecting different products in the list box induces appropriate changes in the text boxes. But when we select a different category, we get an unhandled exception. The description of the error is that two bindings in the collection to bind to the same property. The simple explanation is that bindings have to be cleared first before new bindings are added.
* Before each of the Simple Data Bindings statements add

*SimpleControl* . DataBindings . Clear()

* Build and test the application. Note that clearing the data bindings has resolved the problem.

Note, also, that the second part of 2-way data binding is also working. Choose some product. Type new values into one or more of the text boxes. Then navigate away from this product. Then return and notice that the data has persisted. (The persistence is, of course, not permanent. This is in-memory data. When the application is restarted, the values revert to the initialized values derived from the source code.)

## Demo: Add and Delete

This clip describes how to implement “Add” and “Delete” functionality.

* Navigate to DataForm.vb in design view. Click the “Form” tab.
* Add buttons to the tool strip as shown in the Pluralsight course. My major concern about doing this was obtaining the image files. The way I resolved this was to download the Pluralsight Exercise files, unzip the appropriate folder that pertained to this demo, and open the Visual Studio solution. At the same time I had the Visual Studio solution open for the shadowed application. Then, for each of the buttons, I clicked **copy** to put the button from the downloaded toolbar into the clipboard, navigated back to the shadowed application, and clicked **paste** to put the button into my toolbar.
* Add – at the left edge of the toolbar - the label “Category:”, a combo box for the categories, the label “Source:”, and its combo box. Set the **Name** of the 1st combo box to CategoryToolStripComboBox; change its **DropDownStyle** to **DropDownList**. Set the **Name** of the 2nd combo box to SourceToolStripComboBox; change its **DropDownStyle** to **DropDownList**.
* Add - to the toolbar – vertical separators after each of the combo boxes, after the right-arrow button, and after the delete button.
* We will be using CategoryToolStripComboBox in place of CategoriesComboBox. Therefore, in DataForm\_Load() event handler, replace the 3 instances of CategoriesComboBox with CategoryToolStripComboBox.ComboBox.
* Drag 2 additional text boxes and a check box from the **Toolbox** to DataForm. Put them below the existing text boxes. Name them QuantityTextBox, OrderTextBox, and DiscontinuedCheckBox.
* Set the **Text** property of DiscontinuedCheckBox to an empty string.
* Change the **ReadOnly** property of **NameTextBox** to **True**; we don’t want the user to be able to modify the name by typing into this text box.
* Drag 6 labels from the **Toolbox** to DataForm. Put them to the left of the 5 text boxes and the checkbox. Set their **Text** properties to Name, Unit Price, Units in Stock, Quantity Per Unit, Units on Order, and Discontinued.
* Create a SelectedIndexChanged event handler for CategoryToolStripComboBox.
* Copy the body of code - from CategoriesComboBox’s SelectedIndexChanged event handler – to CategoryToolStripComboBox’s SelectedIndexChanged event handler.
* In the 1st statement (Dim CatID = . . .), replace CategoriesComboBox with CategoryToolStripComboBox.ComboBox.
* In CategoryToolStripComboBox’s SelectedIndexChanged event handler, bind the two additional text boxes and the check box in the same manner as the 1st three earlier text boxes.
* Add a statement to CategoryToolStripComboBox’s SelectedIndexChanged event handler to bind ProductDataGridView to the “products” data source. (***The code in the Pluralsight Exercise Files does not make the “Name” column read-only, and the user could change the name of product via the grid. But I am confident that there is a way to make a column read-only.***)
* Build and test the application. Note that the text boxes are displaying the correct data. Click the “Grid” tab, and observe that the values are displayed for all of the products in the category. Observe, also, that you are permitted to change the contents of an item in the grid as well as in the discrete controls on the “Form” tab; the changed value persists as long as the application is running, but the value reverts when you restart the application.

Translate the code to C#, and repeat the tests.

The next thing to do is to provide facilities for adding an instance to the list of products, and for deleting a specified instance from the list of products. The Pluralsight instructor illustrates object deletion by using the **Remove()** method in **List(Of** *ObjectType***)**. ***Although* Remove() *does the job, it makes me uncomfortable, because it incurs comparing several instances of*** *ObjectType,* ***each of which has several properties****.* ***I believe the* RemoveAt() *method might be better.***

* **List ( Of** Product **)** is declared in Class ObjectSource, and this is an appropriate place to provide the DeleteProduct() and AddProduct() methods. Type the following code in Class ObjectSource.

Public Sub DeleteProduct ( product As Product )

\_products.Remove ( product )

End Sub

Public Sub AddProduct ( product As Product )

\_products.Add ( product )

End Sub

* Create a new dialog titled “Add Product”. Give this dialog the name AddProductForm. The form contains the same controls – 5 text boxes and a check box – as the content of the right side of the main form. Add a 6th text box – below NameTextBox; give it the name CategoryTextBox. Change the **ReadOnly** property of CategoryTextBox to **True**; we don’t want to invite the user to modify the product’s category by typing into this text box. Add the label “Category” to the left of CategoryTextBox. Provide the exit buttons OK and Cancel. Set their **DialogResult** properties to **OK** and **Cancel** respectively.
* Navigate to View Code of AddProductForm.
* Add \_category of type Category as a Private property of class AddProductForm.

Private \_category As Category

* Add a constructor for class AddProductForm that takes the argument “category”.

Public Sub New(category As Category)

InitializeComponent()

\_category = category

CategoryTextBox.Text = category.CategoryName

End Sub

* Provide a property of class AddProductForm named Product that retrieves the values supplied in the dialog.

Public ReadOnly Property Product

Get

Dim result As New Product (

0, NameTextBox.Text, \_category.CategoryID,

QuantityTextBox.Text, PriceTextBox.Text, StockTextBox.Text,

OrderTextBox.Text, DiscontinuedCheckBox.Checked )

Return result

End Get

End Property

* Navigate to the main form. Introduce a button-click event for the “Add” button (the blue cross button on the toolbar).
* In this add-product event handler, we need to employ the dialog to get a product. Instantiate the dialog; as a prerequisite to showing the dialog, we need the currently selected category. Type

Dim category = CType ( CategoryToolStripComboBox.ComboBox.SelectedItem, Category )

Dim form As New AddProductForm ( category )

* Show the dialog, and test whether the user exited the dialog via the “Cancel” button.

Dim result = form.ShowDialog()

* If the user did not Cancel, employ the values that the user has entered to build a product, and add it to the collection.

If result = DialogResult.OK Then

\_source.AddProduct ( form.Product )

End If

* Navigate to the main form. Introduce a button-click event for the “Delete” button (the large X button on the toolbar).
* Get the instance of the selected product, and relay that to the DeleteProduct() method. Type the following into the body of the “Delete” button event handler.

Dim product = CType ( ProductsListBox.SelectedItem, Product )

\_source.DeleteProduct ( product )

Build and test the application. Try choosing a category, clicking one of its products, and finally clicking the “delete” icon on the toolbar. There is no visual indication in the list of products that the product has been deleted. But if you choose a different category, and then come back to the 1st category, you will notice that the product has, indeed, been deleted. Similarly, after choosing a category, click the “add” icon on the toolbar. After supplying values for a new product’s properties, click the “OK” button. Again, there is no visual indication in the list of products that the product has been added. But if you choose a different category, and then come back to the 1st category, you will notice that the product has, indeed, been added. The lack-of-visibility problem will be addressed in a subsequent Pluralsight clip.

Translate the code to C#, and repeat the tests.

## BindingSource Control

The **BindingSource** Control provides useful functionality – navigating, sorting/filtering, adding/deleting, and event notification to user interface. Instead of binding the data object directly to the display object, one uses an instance of the **BindingSource** Control as an intermediary between the data object and a display control. Bind the data object to the **BindingSource** instance; bind the **BindingSource** instance to the display control.

In our sample application **BindingSource** resolves two problems and opens an opportunity for additional functionality.

* As we saw in the last clip, an addition to the collection of products is not visible to the user until he navigates to a different category and returns. **BindingSource** resolves this problem.
* As we saw in the last clip, a deletion from the collection of products is not visible to the user until he navigates to a different category and returns. **BindingSource** resolves this problem.
* **BindingSource** can be used with the left-arrow and right-arrow toolbar buttons to navigate forward and backwards through the list of products.

## Demo: Using The BindingSource

This demo introduces two instances of **BindingSource**, one for categories and the 2nd for products.

* Navigate to DataForm.vb in code view.
* At the beginning of the body of “Class DataForm” type

Private \_categoryBindingSource As New BindingSource

Private \_productBindingSource As New BindingSource

* In DataForm\_Load() insert the binding source as an intermediary between the data object and the combo box.
  + Add the statement at the beginning of the event handler

\_categoryBindingSource.DataSource = \_source.GetCategories()

* Use the **BindingSource** as the data source for the combo box. Modify the statement “CategoryToolStripComboBox.ComboBox.DataSource =” to . . .

CategoryToolStripComboBox.ComboBox.DataSource = \_categoryBindingSource

* With regard to “products” data, refer to the combo-box's SelectedIndexChanged event handler. Move all of its code – except the two “Dim” statements - to DataForm\_Load(). Then replace the instance “products” with \_productBindingSource. Now that the bindings are done only once (at form load) instead of when the category changes, the .Clear() statements are no longer necessary; delete them. In the combo-box's SelectedIndexChanged event handler, replace “Dim products” with \_productBindingSource.DataSource.
* Changes should also be made for the add-button event handler and for the delete-button event handler.
  + Navigate to AddToolStripButton\_Click(). Comment out \_source.AddProduct(form.Product); replace it with a statement that adds the newly created product to the **BindingSource** for products. The replacement statement should read

\_productBindingSource.Add ( form.Product )

* Navigate to DeleteToolStripButton\_Click(). Comment out \_source.DeleteProduct(product); replace it with a statement that removes the selected product from the **BindingSource** for products. The replacement statement should read

\_productBindingSource.Remove ( product )

* The methods AddProduct() and DeleteProduct() - in class ObjectSource - are obsolete. ***I am not deleting them at this point, in case I want to refer to an earlier version of this application.***

Build and test the application. The tests are

* The appropriate products should display when the category changes.
* The product details should display correctly in the text boxes and check box when a product is selected.
* The grid should display the products and details correctly as a function of category selection.
* Make changes to product details via the text boxes or check box. The changes should persist.
* Make changes to product details via the grid. The changes should persist.
* Add a product via the “Add” toolbar button. The newly added product should display immediately. Look, also, at the grid.
* Remove a product via the “Delete” toolbar button. The deleted product should disappear immediately. Look, also, at the grid.

Do additions or deletions of products persist? Try navigating to a different category - after performing an addition and a deletion - and return to the category where the addition and deletion were performed. The added product has vanished, and the deleted product has come back. This is a bug. I had a discussion with the Pluralsight instructor, and he agreed that is a bug. I made the following code changesto resolve the problem. (The Pluralsight instructor agreed that these were the appropriate changes.)

* The changes that we made (above) to AddToolStripButton\_Click() and to DeleteToolStripButton\_Click() should not have included commenting out \_source.AddProduct(form.Product) nor \_source.DeleteProduct(product).
* Uncomment and resurrect these statements.
* As a result, the newly created product is added both to the BindingSource and to \_products; and a deleted product is removed both from the BindingSource and from \_products. Furthermore, the methods AddProduct() and DeleteProduct() - in class ObjectSource - are not obsolete after all.

What about changes to a product’s property (e.g. a change to the price of a product)? If the user changes a property, navigates to a different category, and returns to the original category, does the changed property persist. Yes. The reason why changes are not subject to the same fate as additions/deletions is that a property change results in a change both to the BindingSource and to the underlying collection (\_products).

The BindingSource control is also useful to provide a convenient navigation tool for the user.

* Create a **Click** event handler for BackToolStripButton. Type the following statement for the body of the event handler.

\_productBindingSource.MovePrevious()

* Create a **Click** event handler for ForwardToolStripButton. Type the following statement for the body of the event handler.

\_productBindingSource.MoveNext()

Build and test the application. Try clicking the “back” and “forward” toolbar buttons. Observe that “forward” moves the selected product down in the ProductsListBox, and the “back” moves the selected product up in the ProductsListBox – even when the ProductsListBox is not the control that has focus.

It is a good idea to use the **TabIndex** and **TabStop** properties when designing a form. If unfamiliar with these, Use Google to see definitions. A useful trick to simplify the **TabIndex** settings is to select the form, and then click **View -> Tab Order** on Visual Studio’s pull-down menu. Not only does this show the **TabIndex** values, but you can also set the controls to have consecutive values by clicking each of the controls in the desired order. When finished, click **View -> Tab Order** a second time to revert to normal view.

Translate the code to C#, and repeat the tests.