# 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) |  |

## 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 (design 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.