Windows Forms
Using Visual Basic

Student Guide

Revision 4.0

Object Innovations Course 4230

Windows Forms Using Visual Basic
Rev. 4.0

Student Guide

Authors: Robert J. Oberg and Dana Wyatt

Object Innovations
877-558-7246
www.objectinnovations.com

Printed in the United States of America.

Table of Contents (Overview)

Directory Structure

- **The course software installs to the root directory *C:\OIC\WinVb*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap02**, and so on.
  - The **Labs** directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.
  - The **Demos** directory is provided for hand-on work during lectures.
  - The **Deploy** directory is provided to test deployment.
- **Data files install to the directory *C:\OIC\Data*.**

Table of Contents (Detailed)

Chapter 2

Visual Studio and the Forms Designer

Visual Studio and the Forms Designer

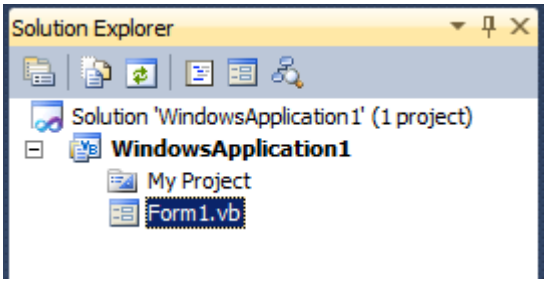
Objectives

After completing this unit you will be able to:

- **Use Visual Studio to build simple Windows Forms applications.**
- **Use the Forms Designer to visually design forms.**
- **Trap events using the Forms Designer.**
- **Create an attractive visual design for your forms.**
- **Create an efficient design for your forms, including setting a tab order and implementing keyboard shortcuts.**

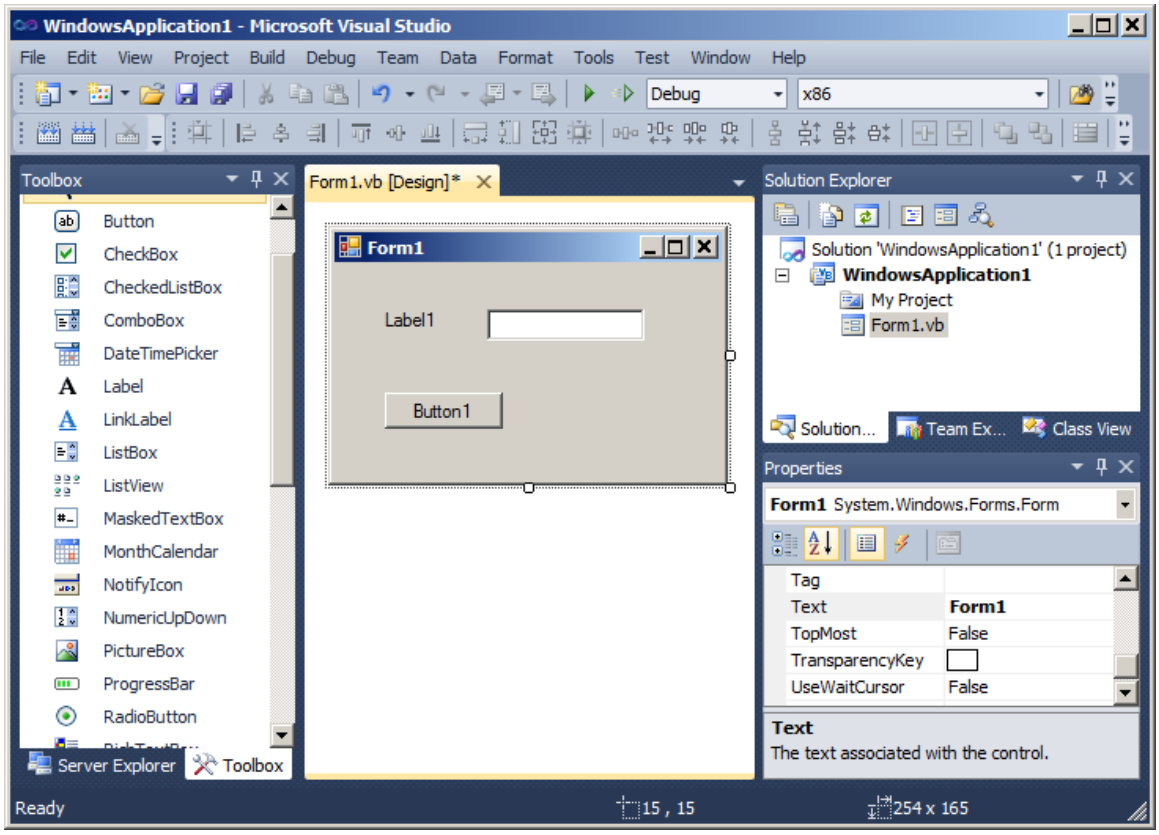
Visual Studio

- **Visual Studio, which we used in the first chapter to create and run simple projects, is extremely useful in developing Windows applications.**
- **Visual Studio allows us to design forms using a drag-drop interface.**
  - If you are familiar with the IDEs in Visual Basic or Visual C++, the Visual Studio IDE will look very familiar!
- **The drag-drop interface is generally referred to as the *Forms Designer*.**
  - The Forms Designer will be available any time a Windows Forms class has been added to a project.
  - It can be opened by selecting the View Designer icon from the Solution Explorer window.



Using the Forms Designer

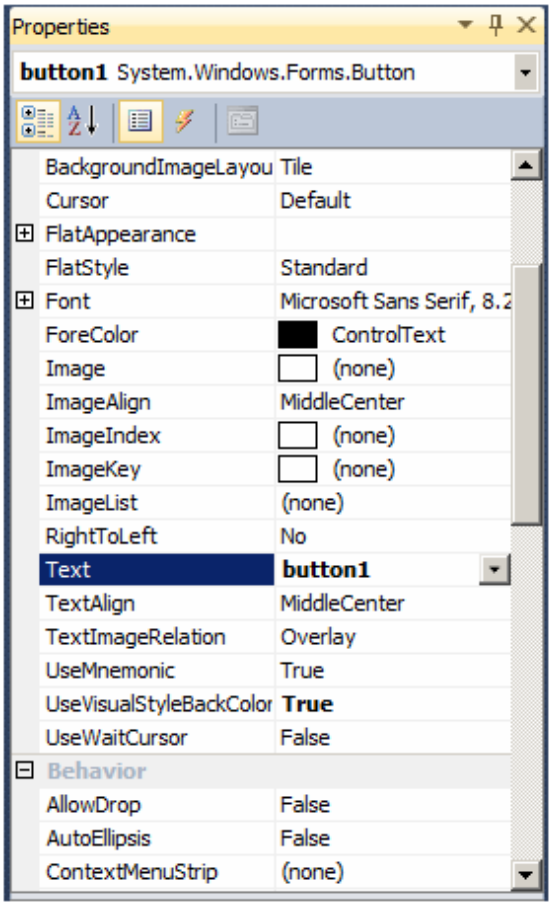
- **The Forms Designer allows a programmer to drag and drop controls from a toolbox onto a form.**
  - If the toolbox isn't visible, you can select it from the View | Toolbox menu.



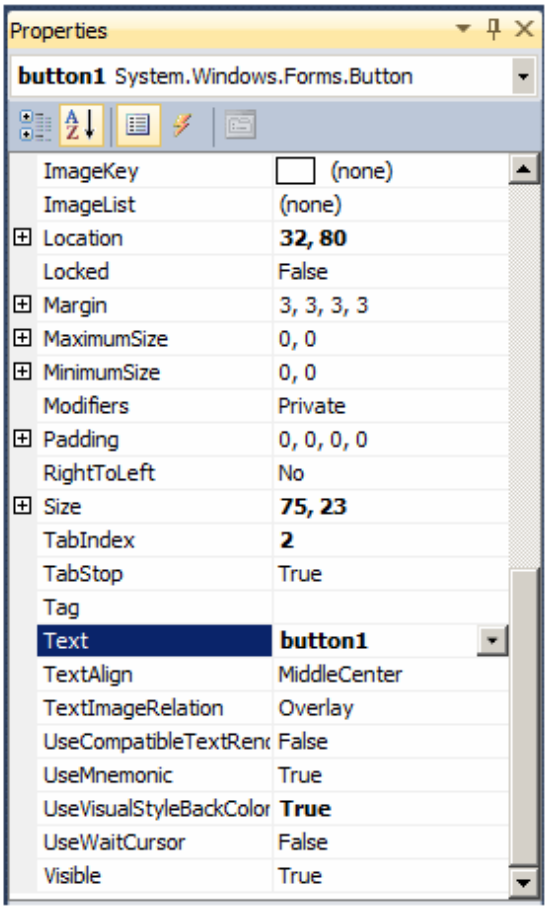
Using the Forms Designer (Cont'd)

- **You can modify the properties of a control using the Properties window (shown in the lower right).**
  - If the Properties Window isn't visible, you can select it from the View | Properties Window menu.
  - The properties can be shown by category or alphabetically be selecting an icon from the Properties Window toolbar.

By category: 

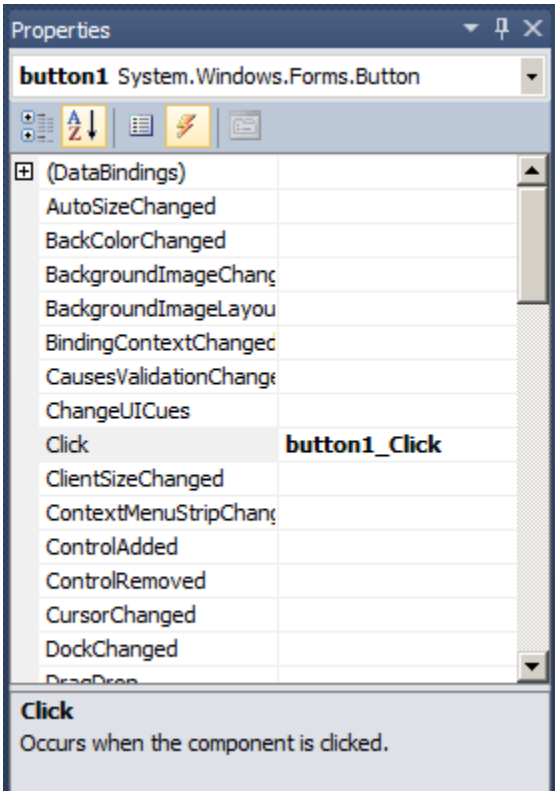


Alphabetically: 



Using the Forms Designer (Cont'd)

- **You can add, modify and view the event handlers for each control using the Properties window.**
  - To add an event handler and associated delegate, double-click on the appropriate event from the left-hand side of the scrolling grid. Select Events by the  icon.

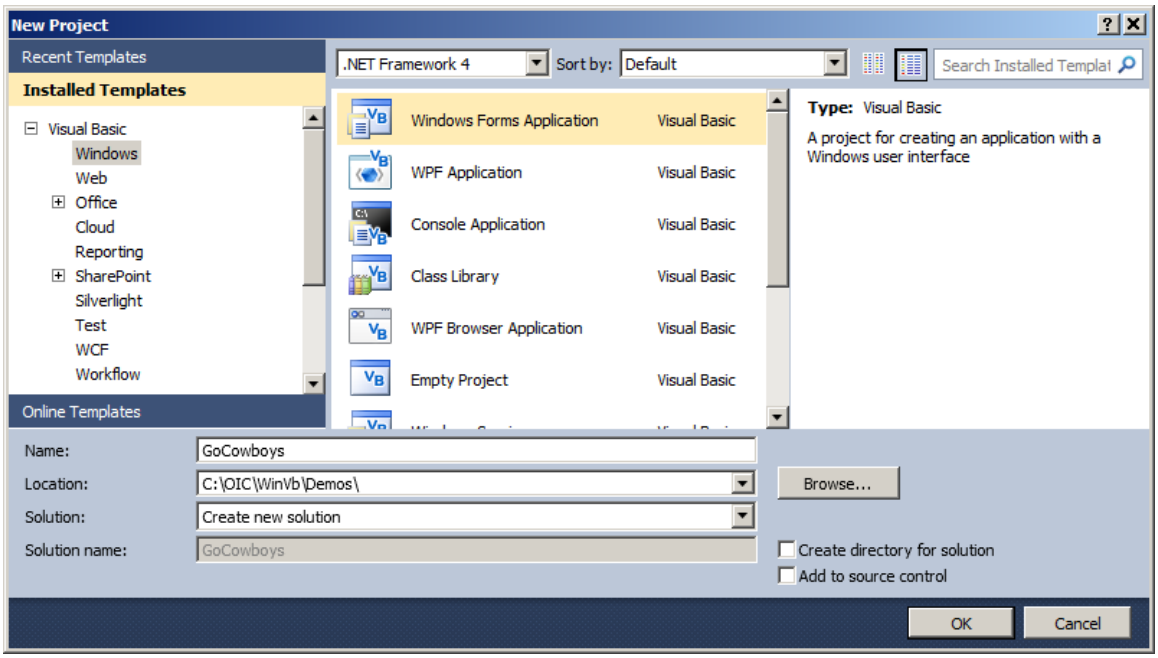


  - You can add the "default" event handler for each control by double-clicking the control in design view.

Example: Creating a Windows Forms Application

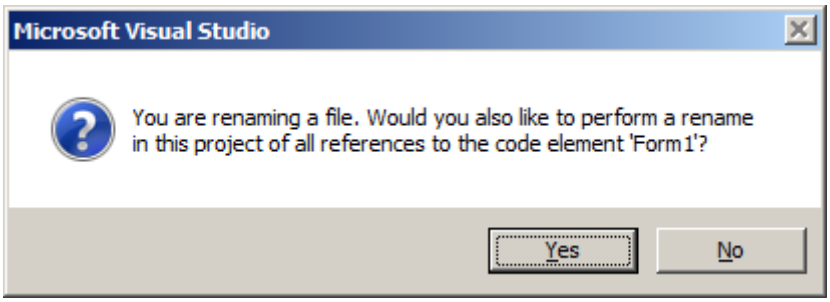
- **It is easy to create a Windows Forms application using Visual Studio.**
  - A copy of the application is saved in **Chap02\GoCowboys** directory.
  - If you want to follow along, you should do your work in the **Demos** directory.

1. We begin by creating a new VB Windows Forms Application project named **GoCowboys** in the **Demos** directory. Leave unchecked "Create directory for solution."

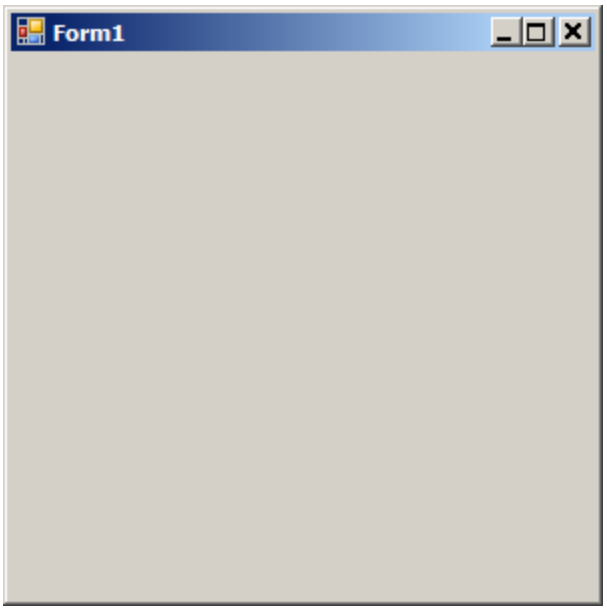


Example: Creating ... (Cont'd)

2. We then rename the source file for the form to **MainForm.cs**. You will be asked if you want to rename the corresponding code elements. Say yes.

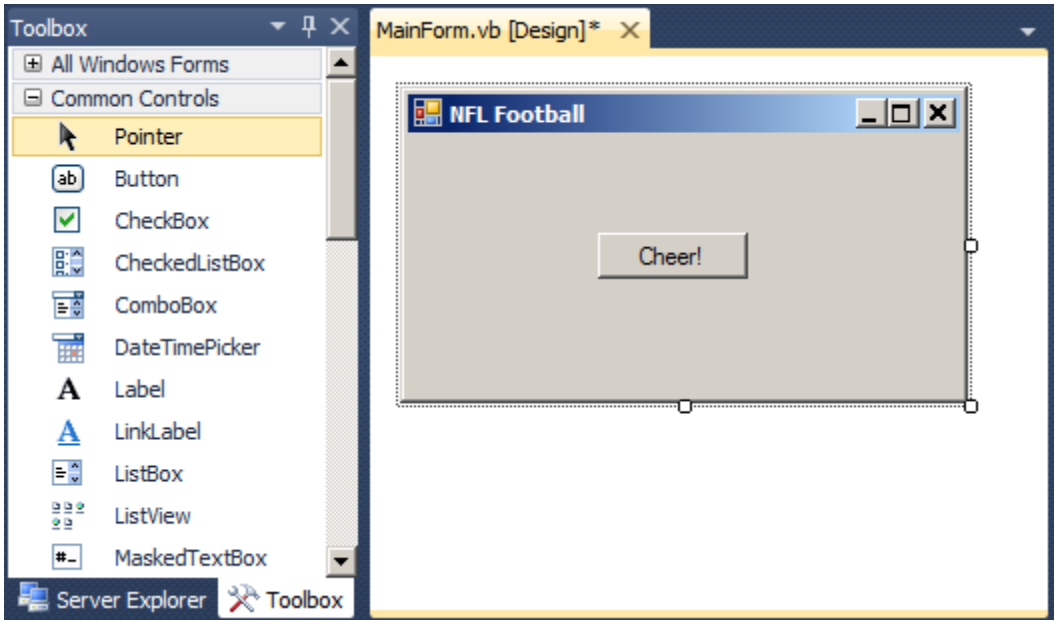


3. To verify that the required code elements have been renamed, build and run the application. You should see a bare form, which can be resized.



Example: Creating ... (Cont'd)

4. We will use the toolbox to drag a button control to the form. To make everything look nifty, we will resize the form.



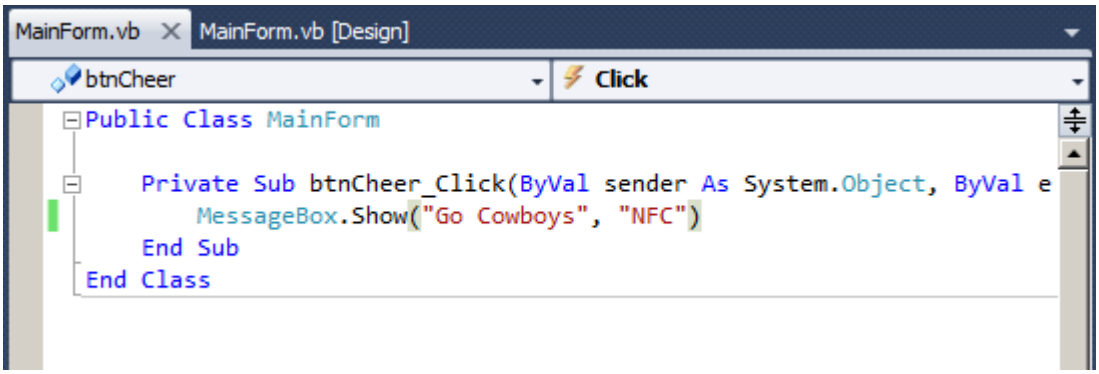
5. We want to make sure the following property values are set for the button and form:

Object	Name Property	Text Property
Button	btnCheer	Cheer!
Form	MainForm	NFL Football

Example: Creating ... (Cont'd)

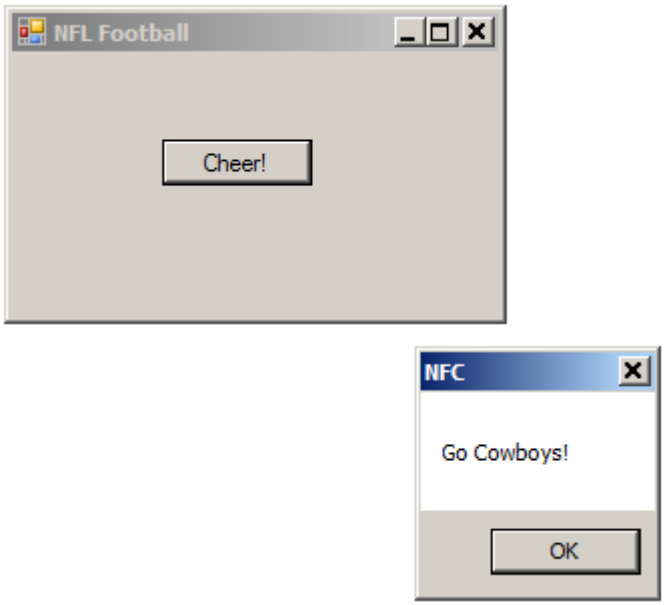
6. We need to trap the **Click** event for the **btnCheer** button. To do this, we can double-click on the **btnCheer** button. It will write the **Click** event handler for us and position us at the handler function in the code window.

   – We will add code to display a message box that shows the message "Go Cowboys!"



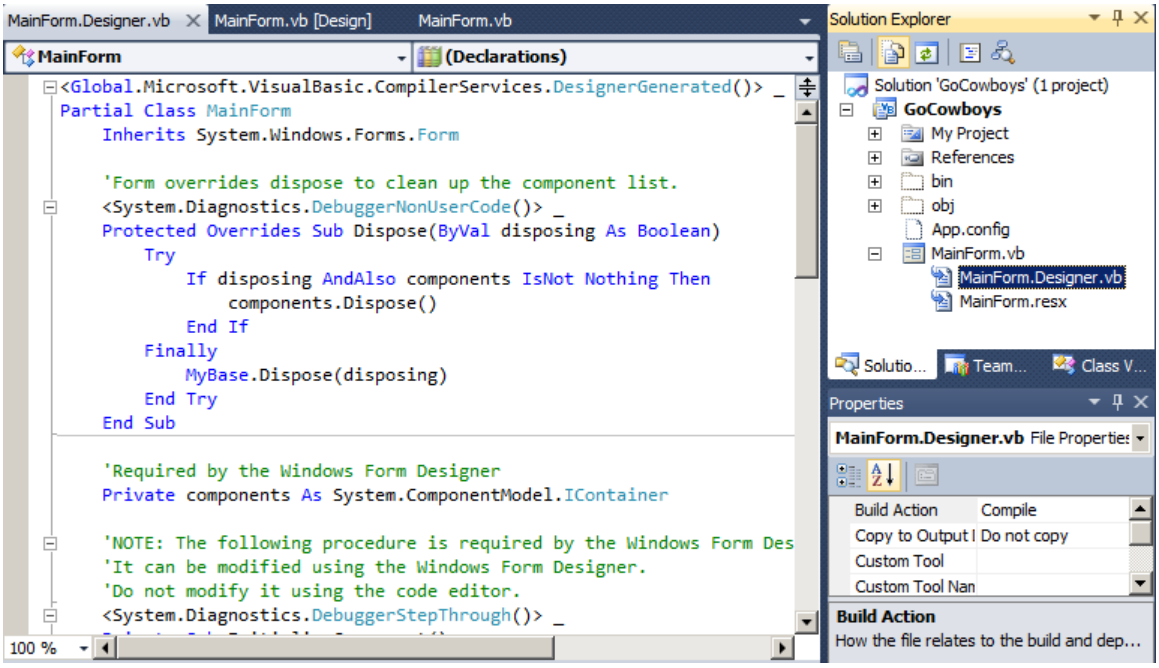
Example: Creating ... (Cont'd)

7. Finally, we can build and run the application.



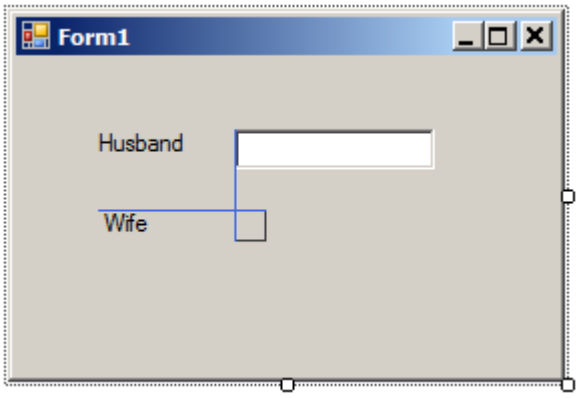
Examining the Forms Designer Generated Code

- **The Forms Designer generated code as we designed the form.**
  - The code is in the separate file **MainForm.Designer.vb**.
  - Click the button  in Solution Explorer to show all files.
  - Note use of the **Partial** modifier on the class, which enables this wizard-generated code to be maintained in a separate file.

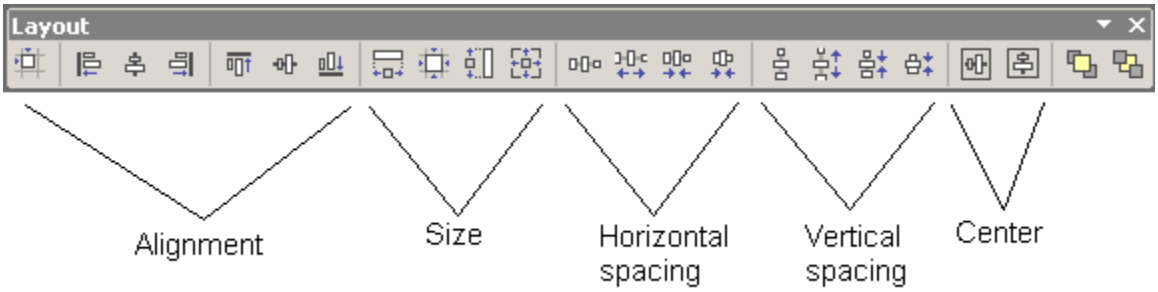


Designing "Pretty" Forms

- **In order to make your forms look nice, you will want to:**
  - Size your controls so that similar are approximately the same size.
  - Align your controls on some meaningful X and Y axis.
  - Visual Studio 2010 provides alignment lines as you drag controls, making your job easier.



- **The Forms Designer allows you to use a special toolbar to perform a variety of layout tasks.**
  - You can use the View | Toolbars | Layout menu option to display the toolbar.



Designing "Easy-to-Use" Forms

- **In order to make your forms easy to use, you will want to:**
  - Lay out the controls in a meaningful way.
  - Provide meaningful labels.
- **However, there are some other steps you can take to make the form easy to use:**
  - Set the tab order of the controls.
  - Define keyboard shortcuts to provide fast access to controls.
  - Define default and cancel buttons.
- **Our example illustrates a form that is both “pretty” and easy to use.**
  - See **PrettyDialog** in the current chapter directory.

Setting the Tab Order

- **To set the tab order of the controls on a form, you must use the View | Tab Order menu.**
  - The form must have focus for this option to be visible.
  - You must select View | Tab Order when you are done to turn off tab ordering.
- **When you click on a tab number, it changes.**
  - The first tab you click on becomes 0, the second tab you click on becomes 1, etc.
  - If you accidentally give a tab an incorrect number, keep clicking on the tab… the number cycles through the available number.

  - You should always include your labels in the tab order directly before the control they label. The reason will be apparent on the next page!

Defining Keyboard Shortcuts

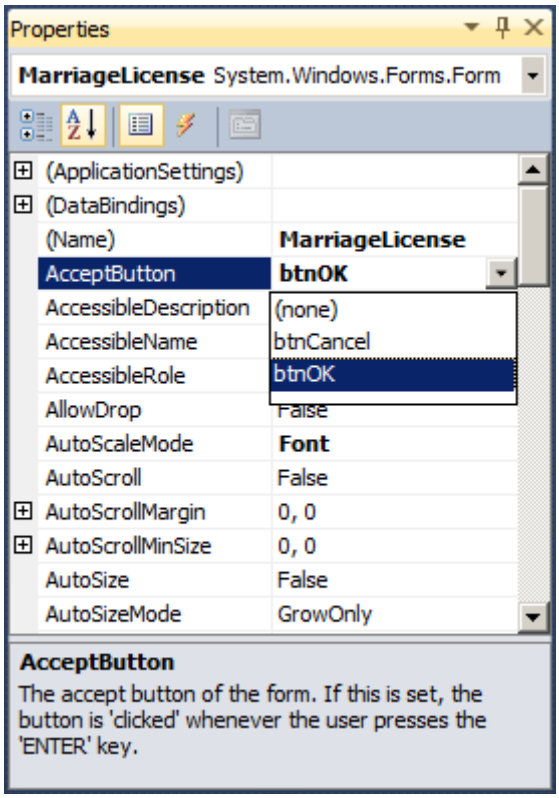
- **You can define keyboard shortcuts for controls on your form.**
  - These shortcuts allow the user to press Alt + *shortcutKey* to move focus to the control that defines the shortcut.
- **To assign a shortcut to controls, you must:**
  - Place an ampersand (&) in front of the shortcut letter in the control's **Text** property.
  - For example, if the **Text** property of an OK button is &OK, then the text will appear as <u>O</u>K on the screen to let the user know O is the shortcut key. When the user presses Alt + O, focus moves to the control.
  - When you use the shortcut key on a button, it invokes the click event handler for the button instead of setting focus to it.
- **When a control does not have a static *Text* property (for example, a textbox), you must place a label that defines the key directly in front of the control—based on tab order.**
  - For example, if the textbox has tab order 7, its label should have tab order 6.
  - The label's **Text** property must have an ampersand (&) in front of the shortcut letter (example: &Husband will define H as the shortcut; Hus&band will define B as the shortcut).

Defining Keyboard Shortcuts (Cont'd)

- **When running the program, to use the keyboard shortcuts the user should press the Alt key.**
  - Then the underlines will appear.
  - When the user enters the keyboard shortcut, focus will go to control the following the label (based on ***tab order***) because a label does not have a ***tab stop***.

Defining Default and Cancel Buttons

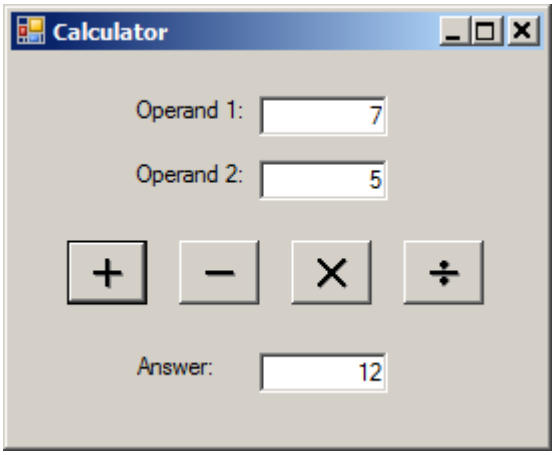
- **It is common among Windows Forms applications to define default and cancel buttons for each form.**
  - The default, or accept, button is one that is invoked if the user hits the Enter key in any control on the form that does not have its own **AcceptReturn** property set to True.
  - The cancel button is the one that is invoked if the user hits the Escape key in any control on the form.
- **Windows Forms makes this assignment easy:**
  - The form has two properties, **AcceptButton** and **CancelButton**, which can be assigned a reference to a button using a drop-down list.



Lab 2

My Calculator

In this lab you will you will use Visual Studio and the Forms Designer to build a simple calculator that performs addition, subtraction, multiplication and division on floating point numbers.



Detailed instructions are contained in the Lab 2 write-up at the end of the chapter.

Suggested time: 30 minutes

Summary

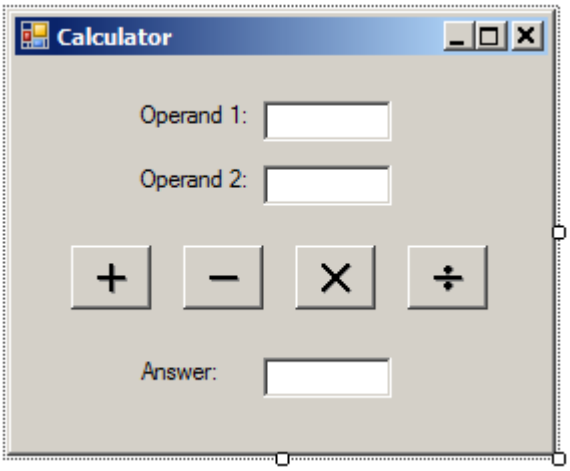
- **Visual Studio makes it easy for programmers to build Windows Forms applications.**
- **The Forms Designer allows programmers to drag controls from a toolbox and visually place them on a form.**
- **The Properties Window can be used to specify values for form and control properties.**
- **The Forms Designer generates code based on the programmer's drag-drop actions and property settings.**
- **Visual Basic and the tools inside Visual Studio make it easy to build Windows Forms applications that look nice and are easy to use.**

Lab 2

My Calculator

Introduction

In this lab you will you will use Visual Studio and the Forms Designer to build a simple calculator that performs addition, subtraction, multiplication and division on floating point numbers. Your form should resemble the following:



You will need to use exception handling to make sure that "garbage" data in the operand 1 and/or operand 2 textbox does not cause your program to "crash".

Suggested Time: 30 minutes

Root Directory: **OIC\WinVb**

Directories:	**Labs\Lab2**	(do your work here)
	OIC\Data\Graphics	(contains icon files)
	Chap02\MyCalculator	(contains lab solution)

Instructions

1. Create a new VB Windows Forms Application named **MyCalculator**. Name the form class and the associated file **Calculator**. Save the solution.

2. Design the form similar to that shown above.

3. Copy the icon files for the four arithmetic operations, MISC18.ICO, MISC19.ICO, MISC20.ICO and MISC21.ICO, from **OIC\Data\Graphics** to the working directory.

Chapter 10

Applications and Settings

Applications and Settings

Objectives

After completing this unit you will be able to:

- **Use the *Application* object to obtain information about the application and its environment.**
- **Build applications that filter messages from the message loop.**
- **Build a configuration file to store application-specific settings.**
- **Use the application settings facilities in .NET to persist both application-wide and user-specific settings.**
- **Use .NET classes to read configuration files and use their settings in applications.**
- **Access the registry from a .NET application.**

The Application Class

- **The *Application* class, found in the *System.Windows.Forms* namespace, represents the class that manages a Windows Forms application.**
- **It has several interesting properties, including:**
  - The **StartupPath** property, which contains the path of the .exe that started the application.
  - The **ExecutablePath** property, which contains the path and filename of the .exe that started the application.
  - The **ProductName** property, which contains the name of the application.
- **It has several interesting methods, including:**
  - The **Run** and **Exit** methods, which start and stop applications.
  - The **ExitThread** method, to close all windows running on the current thread.
  - The **DoEvents** method, which processes all Windows messages in the queue.
- **All members of the class are shared methods.**

Starting and Stopping Applications

- **As you have seen thus far, Windows Forms applications have a *Main* function[1] that identifies the window that will appear when the application is launched.**
  - The **Run** method starts an application message loop on the current thread and, optionally, makes a form visible.

```
Public Shared Sub Main()
   Application.Run(New MainForm())
End Sub
```

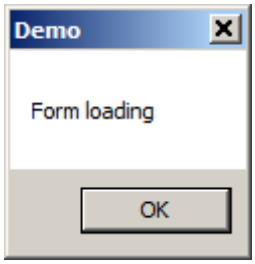
- **Windows Forms applications can call the *Exit* method to stop a message loop, effectively terminating the application.**

```
Private Sub btnExit_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles btnExit.Click
   Log.WriteLine("Exit clicked")
   Application.Exit()
End Sub
```

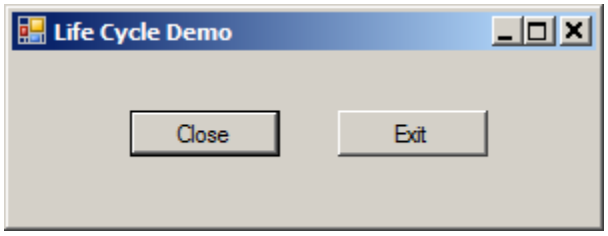
[1] Normally this happens behind the scenes in Visual Basic, but we saw the Main() function in the examples from Chapter 1.

Life Cycle Demonstration

- **A simple demo program illustrates closing an application in various ways, by buttons on the form and by using the standard "X" button.**
  - See **LifeCycle** in the chapter directory.
  - Message boxes and simple logging to a file are used.
- **Here is one scenario.**
  - Program is started. A message box is displayed as the main form is loaded.



  - The main form is then displayed:



  - The Exit button is clicked.

Application Events

- **The Application class fires several events:**
  - The **ApplicationExit** event fires after all forms have closed and the application is about to terminate.
  - The **Idle** event fires when the application is entering the idle state. This occurs after the application has processed all messages in the input queue.
  - The **ThreadExit** event fires when a thread is about to terminate. It fires before the **ApplicationExit** event.
  - The **ThreadException** event fires when an unhandled exception occurs. It can be used to allow the application to continue executing.
- **We can write an event handler for *ApplicationExit*:**

```
Private Shared Sub OnExit(ByVal sender As Object, _
ByVal e As EventArgs)
   Log.WriteLine("OnExit called")
   MessageBox.Show("Exiting application", "Demo")
End Sub
```

- **We can add the event handler in *MainForm_Load*:**

```
Private Sub MainForm_Load(ByVal sender As Object, _
ByVal e As EventArgs) Handles MyBase.Load
   Log.WriteLine("Form loading")
   MessageBox.Show("Form loading", "Demo")
   AddHandler Application.ApplicationExit,  _
      AddressOf OnExit
End Sub
```

Logging to a File

- **In running our little example program, we may be unsure whether the *OnExit* event handler was actually called, because the message box was not displayed.**
  - There are some subtleties in Windows that may prevent a window from being displayed too late in the application shut down process.
- **Besides displaying message boxes, our *LifeCycle* demo program also logs to a file.**

```
Public Class Log
   Public Shared Sub WriteLine(ByVal str As String)
      Dim writer As _
         New StreamWriter("c:\OIC\log.txt", True)
      writer.WriteLine(str)
      writer.Close()
   End Sub
   ...
End Class
```

- **The *OnExit* event handler contains this code:**

```
Log.WriteLine("OnExit called")
```

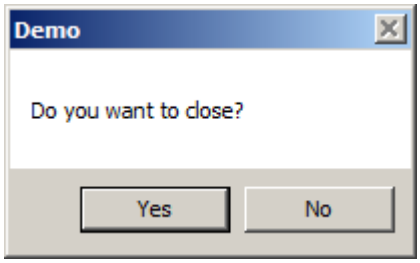
- **The log file *log.txt* for the simple run shows that the event handler was indeed called.**

Closing a Window

- **As mentioned earlier, you should normally shut down an application more gracefully by closing its main window.**
  - This is illustrated by the Close button in our example.

```
Private Sub btnClose_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles btnClose.Click
   Log.WriteLine("Close clicked")
   Close()
End Sub
```

- **There are two key events associating with closing a window.**
  - **FormClosing** is fired first, and the handler for the event can prevent the closing by setting a Cancel flag.
  - **FormClosed** is fired when the window is actually closed.
- **You may query the user in the handler of the Closing event.**



Closing a Window (Cont'd)

- **Here are the handlers for *FormClosing* and *FormClosed*.**

```
Private Sub MainForm_FormClosing(ByVal sender As _
Object, ByVal e As _
System.Windows.Forms.FormClosingEventArgs) _
Handles MyBase.FormClosing
   Log.WriteLine("Form closing")
   Dim status As DialogResult = MessageBox.Show( _
      "Do you want to close?", "Demo", _
       MessageBoxButtons.YesNo)
   If status = DialogResult.No Then
      e.Cancel = True
   End If
End Sub

Private Sub MainForm_FormClosed(ByVal sender As _
System.Object, ByVal e As _
System.Windows.Forms.FormClosedEventArgs) _
Handles MyBase.FormClosed
   Log.WriteLine("Form closed")
End Sub
```

- **Here is the complete log file for running the application, clicking the Close button, and saying Yes to the query to close the window.**

```
Form loading
Close clicked
Form closing
Form closed
OnExit called
```

Processing Windows Messages

- **.NET programs that perform computationally intensive processing on their main message loop are problematic.**
  - We have all used an application that we thought had “hung.” We click everywhere and are just about to kill it, when all of a sudden it "springs to life" and processes all of our intermediate clicks.
- **We can solve this problem in one of two ways:**
  - Use a background thread to perform computationally intensive processing—which frees the UI thread to process Windows messages.
  - Use **DoEvents** to periodically allow Windows to process queued messages when performing computationally intensive processing.

```
Private Sub mnuDoSomethingLong_Click(ByVal sender _
As Object, ByVal e As System.EventArgs)

   Dim i As Integer
   For i = Int32.MinValue To Int32.MaxValue
      ' Do something computationally intensive
      If Math.Abs(i) Mod 10000 = 0 Then
         Application.DoEvents()
      End If
   End For

End Sub
```

Filtering Messages

- **.NET allows Windows Forms programmers to add a message filter to the application message pump to monitor Windows messages.**
  - You should be quite familiar with Windows SDK programming before attempting this.
- **You begin by defining a class that implements the *IMessageFilter* interface.**
  - This class can view the messages before they are processed, potentially stopping an event from being processed.
  - The **IMessageFilter** interface defines the method **PreFilterMessage**. It returns true to block the message from being processed.

```
' From Windows SDK file winuser.h
' #define WM_RBUTTONDOWN                  0x0204
Private Class FilterMouseMessages
   Implements IMessageFilter

   Public Function PreFilterMessage( _
   ByRef m As Message) As Boolean _
      Implements IMessageFilter.PreFilterMessage

      ' Filter right mouse button clicks
      If m.Msg = &H204 Then
          Return True
      Else
          Return False
      End If

   End Function
End Class
```

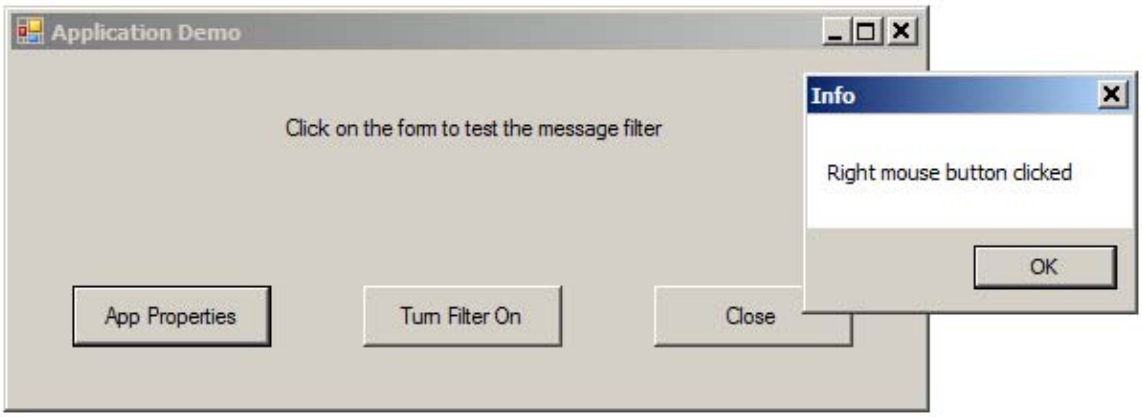
Filtering Messages (Cont'd)

- **You must then call the *AddMessageFilter* method and pass it a reference to the class that implements *IMessageFilter*.**
  - You can remove the filter by calling **RemoveMessageFilter**.

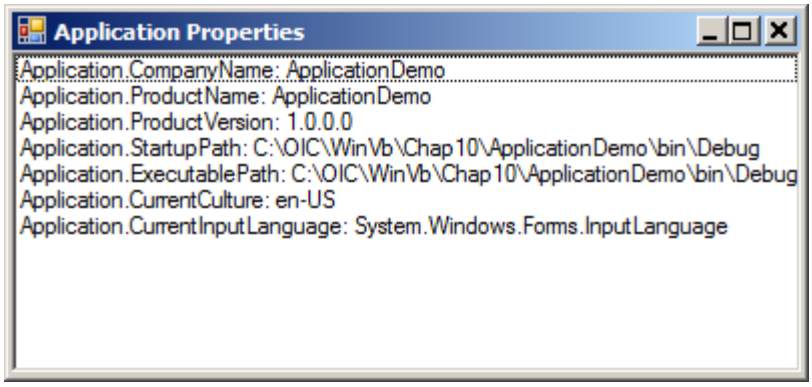
```
Private msgFilter As IMessageFilter = Nothing

Private Sub btnFilter_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles btnFilter.Click

   ' Toggle filter
   If msgFilter Is Nothing Then
      msgFilter = New FilterMouseMessages
      btnFilter.Text = "Turn Filter Off"
      Application.AddMessageFilter(msgFilter)
   Else
      Application.RemoveMessageFilter(msgFilter)
      msgFilter = Nothing
      btnFilter.Text = "Turn Filter On"
   End If

End Sub
```

Example: Using Application Class

- **The example program *ApplicationDemo* demonstrates the use of the application class.**



- **The App Properties button displays some of the Application properties in a separate window.**



- **The Message Filter button toggles filtering out right mouse button clicks on the form.**
  - Before the button is pressed, the form displays a message for left and right mouse clicks. After the button is pressed, it displays a message only when the left button is clicked.
- **The Close button calls *Close()*.**

Configuration Files

- **Configuration files are XML files that provide configuration parameters to applications.**
  - They can be changed without having to recompile the application.
- **Several configuration files exists:**
  - Each application can have a .config file named *applicationName*.config which is located in the application's directory. For example, if the application were called Notepad.exe, the config file would be Notepad.exe.config.
  - The machine has a .config file named machine.config which is located in the directory Windows\Microsoft.NET\Framework\vx.y.zzzz\Config (where x.y.zzzz is the version number of .NET)
- **Microsoft suggests all application-specific configuration settings be stored as key/value pairs in the <appSettings> section of the application's config file.**

Configuration Files (Cont'd)

- **For example, a config file that specifies a default user name and connection string for database access might resemble:**

```
<configuration>
<appSettings>
  <add key="Default User" value="BWW" />
  <add key="Connection String"
   value="Data Source=(local);Initial Catalog=pubs"
   />
</appSettings>
</configuration>
```

- ***NOTE:*** **Important new features pertaining to configuration files were introduced in .NET 2.0.**
  - They will be discussed later in the chapter.

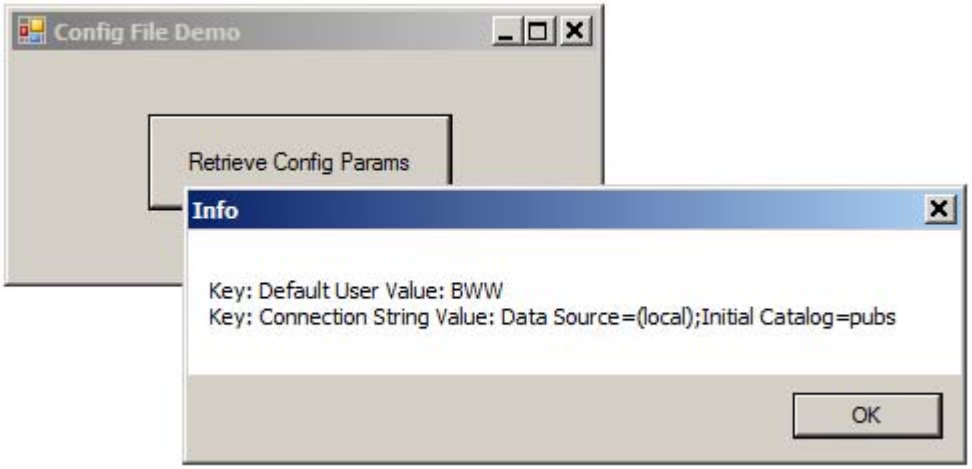
Reading Configuration Files

- **The System.Configuration namespace contains classes which can be used to read the .config file, including:**
  - The **ConfigurationManager**[2] class provides access to the AppSettings or user-defined sections of a .config file.
- **The class provides access to the key/value pairs via a NameValueCollection object. This type includes:**
  - The **Count** property, which identifies the number of key/value pairs.
  - The **AllKeys** property, which returns an array of strings representing each key.
  - The **Get** method, which accepts a key and returns the value associated with that key.

[2] The **ConfigurationManager** class supersedes the **ConfigurationSettings** class, which is now obsolete.

Example: Using Config Files

- **In the example program *ConfigFiles*, we will read initialization parameters from a config file and display them in a message box.**
  - The application's name is **ConfigFiles.exe**, therefore the .config file is named **ConfigFiles.exe.config** and must reside in the same directory as the application.



- **The code that read the config file is shown below:**

```
Private Sub btnRetrieve_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles btnRetrieve.Click
   Dim parms As NameValueCollection
   parms = ConfigurationManager.AppSettings

   Dim msg As String = ""
   Dim key As String
   For Each key In parms.AllKeys
      msg &= String.Format("Key: {0} Value: {1}", _
          key, parms.Get(key)) & vbNewLine
   Next
   MessageBox.Show(msg, "Info")
End Sub
```

Example: Using Config Files (Cont'd)

- **In order for the code shown above to work, we must have two using statements:**

```
Imports System.Collections.Specialized
Imports System.Configuration
```

- **Your project also needs a reference to *System.Configuration*.**
- **To run the example, you should copy the config file down to the *bin\Debug* (or *bin\Release*) directory.**

Configuration File and Visual Studio

- **To conveniently work with a configuration file in Visual Studio, name it *App.config* and add it to your project.**
- **When you build the project, Visual Studio will copy the configuration file to *bin\Debug* (or *bin\Release*) and rename it based on the name of the assembly.**
- **See the example project *ConfigFilesVs* for this chapter.**
  - The configuration file is renamed to **ConfigFiles.exe.config** when the project is built.
  - The configuration file is automatically copied to the folder containing the executable.

Application Settings

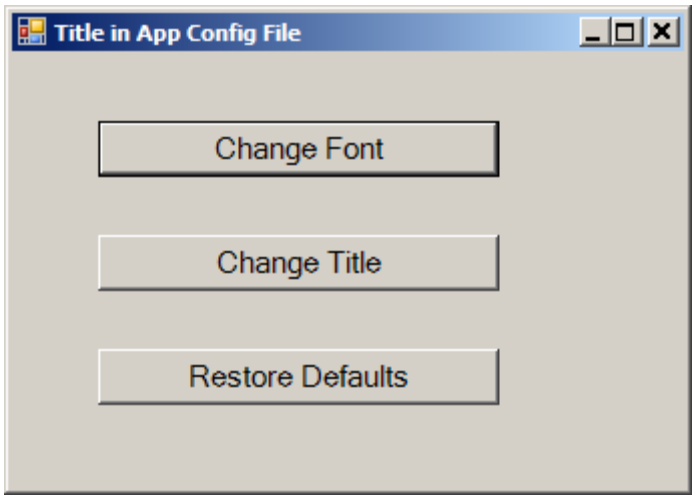
- **The .NET Framework has always provided the capability to store application settings as an XML fragment in a configuration file.**
  - Before .NET 2.0, configuration setting information pertained only to the application as a whole (not individual users) and was read-only.
- **Beginning in .NET 2.0 the support for application settings is more comprehensive.**
  - Setting information for the application as a whole can be stored in ***app*.exe.config**, where ***app*** is the name of your main executable file. This information is read-only.
  - Setting information for individual users can be stored in a new file **user.config**. This information is read-write.
  - The **user.config** file is stored in a Local Settings area specific to the individual user.
  - Default user settings can be stored in **app.exe.config**.
- **Application settings may be retrieved and set programmatically by using a class derived from *ApplicationSettingsBase*.**
  - This class is in the **System.Configuration** namespace.

Application Settings Using Visual Studio

- **Visual Studio 2010 provides strong support for using application settings in your program.**
  - Studying the code generated by Visual Studio can also help you in working with application settings via code written manually. You will also have to manually edit a configuration file if you don't use Visual Studio.
  - You can bind application settings to properties of a form or controls on the form.
- **Perform the following steps to create a new setting.**
  - Select the form or control whose properties are to be bound to the new setting.
  - Use the Property Editor to open the Application Settings dialog box.
  - Use the user interface in this dialog to select the property you want to bind to the new setting.
  - Configure the new setting by giving it a name, a scope (user or application) and a default value (if any).
- **You may then manipulate the new setting by using the *Settings* object in the file *Settings.Designer.vb* that is generated by Visual Studio, in the Properties folder of the project.**

Application Settings Demo

- **Let's illustrate the use of application settings with a simple demonstration.**
  - We will first create a setting using Visual Studio.
  - We will then manually code another setting.
- **The final project is in *DemoSettings* in the code folder for this chapter.**
  - Build and run.



  - Try changing the font and title. Exit the application and run again. You should see the changes preserved.
  - Now restore the defaults. Again, exit the application and run again.

Application Settings Demo (Cont'd)

- **Let's create this application using Visual Studio 2010.**
  - Do your work in the **Demos** directory.

1. Create a new Windows Forms application **DemoSettings**, and save the solution in the **Demos** directory.

2. Drag three buttons onto the main form. Set the Text of the buttons to "Change Font," "Change Title" and "Restore Defaults" as shown in the screen capture on the preceding page. For now, just go with the default font. Set the names of these controls to **btnFont**, **btnTitle** and **btnDefault**.

3. Drag a FontDialog control onto your form. Change the name of the control to **fontDlg**.

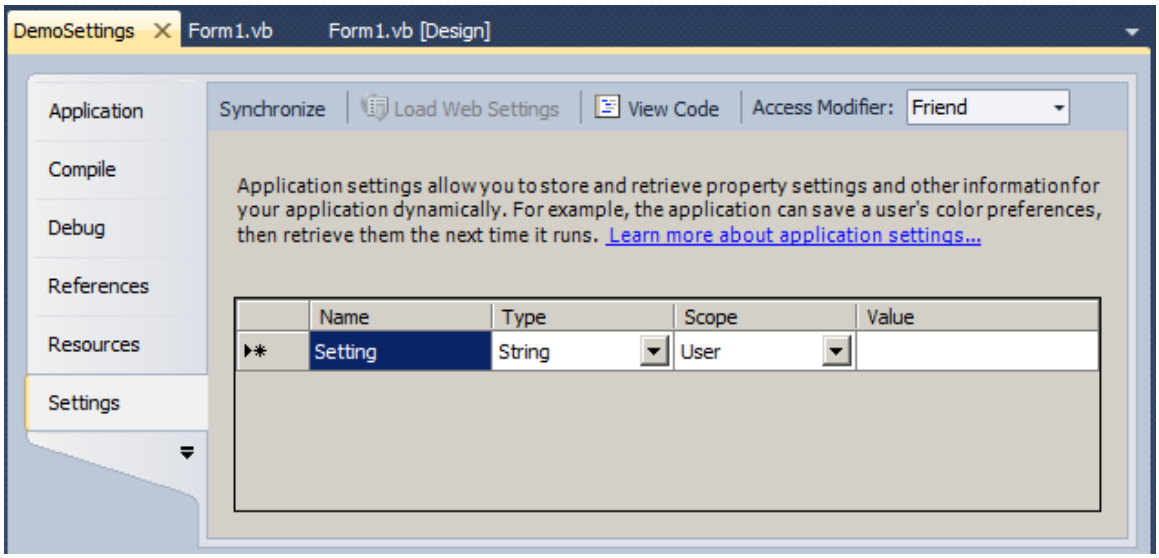
4. Add a handler for the Change Font button. Provide the following code.

```
Private Sub btnFont_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles btnFont.Click
   If fontDlg.ShowDialog() = DialogResult.OK Then
      Dim f As Font = fontDlg.Font
      Me.Font = f
   End If
End Sub
```

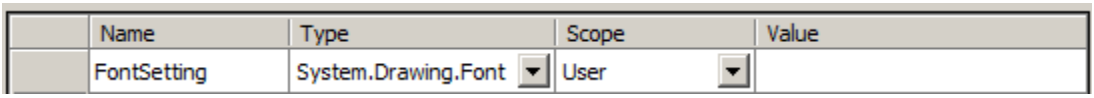
5. Build and run. Try changing the font. Note that the size of the buttons change size to harmonize with the size of the font. Note that if you run the application again, any changes you made to the font will not be preserved.

Application Settings Demo (Cont'd)

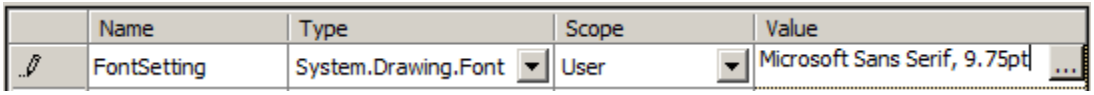
6. Next, let's use the Property Designer to add an application setting that will let us save the font. Double-click My Project in Solution Explorer and select Settings from the side menu.



7. Make the Name **FontSetting**, the Type **System.Drawing.Font**, and the Scope **User**.



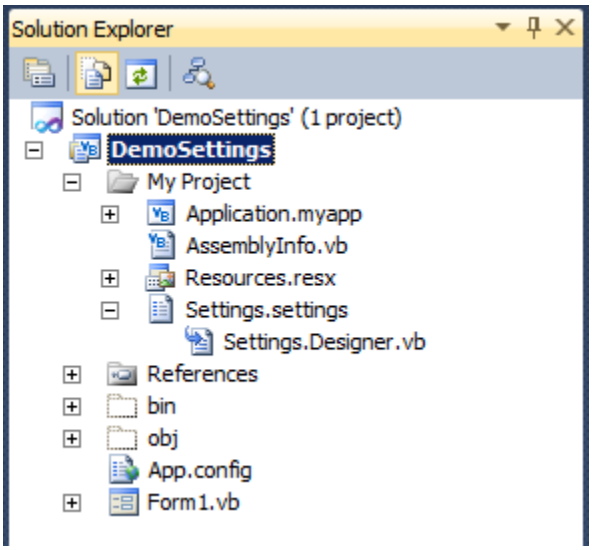
8. Click the Ellipsis in the Value column. The Font common dialog comes up. Leave the font name as "Microsoft Sans Serif" and select 10 as the font size (it will actually become 9.75).



9. Click outside the first row, and the pencil editing icon will go away.

Application Settings Demo (Cont'd)

10. In Solution Explorer click the Show All Files button  . There will now be new files **Settings.Designer.vb** and **app.config**.



11. Examine the **Settings.Designer.vb** file. There is a class **Settings** derived from **ApplicationSettingsBase**.

12. Add handlers for the form's **Closing** and **Load** events.

Application Settings Demo (Cont'd)

13. Add the following code to **Form1.vb**

```
Private Sub Form1_Load(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles MyBase.Load
   Dim stg As Settings = Settings.Default
   stg.Reload()
   Me.Font = stg.FontSetting
End Sub

Private Sub Form1_FormClosing(ByVal sender As _
System.Object, ByVal e As _
System.Windows.Forms.FormClosingEventArgs) _
Handles MyBase.FormClosing
   Dim stg As Settings = Settings.Default
   stg.FontSetting = Me.Font
   stg.Save()
End Sub
```

14. Build and run. The new font will now be persisted.

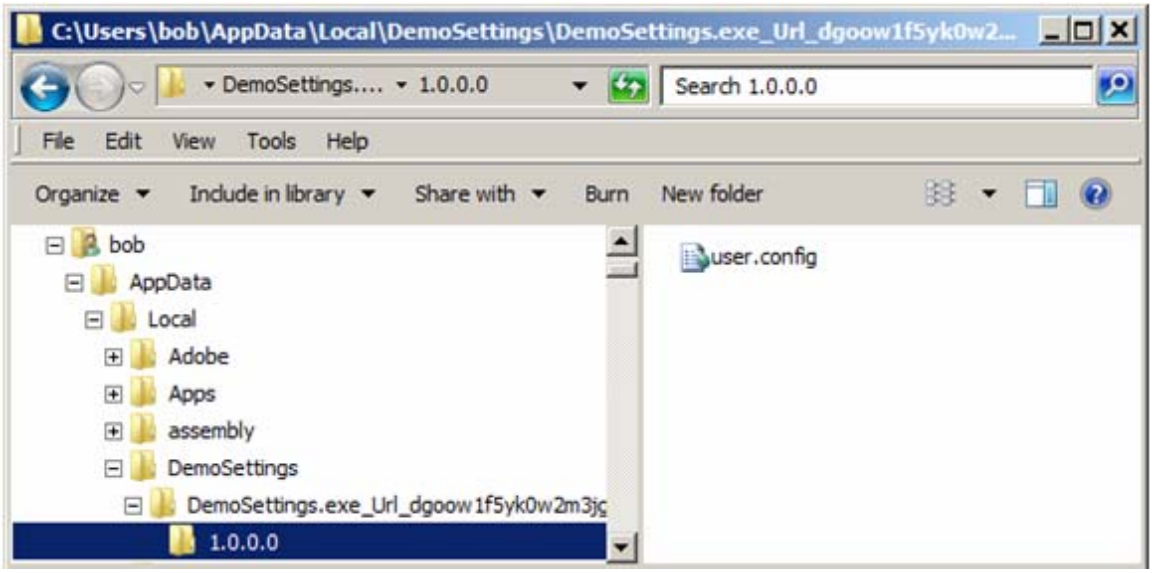
Application Configuration File

- **Visual Studio has created a file *app.config*. When the project is built, the configuration file is copied to *DemoSettings.exe.config* in the same folder as the application's executable file *DemoSettings.exe*.**
  - Two sections are defined: **userSettings** and **applicationSettings**.
  - In the <**userSettings**> group the setting **FontSetting** is defined, with default value "Microsoft Sans Serif, 9.75pt."

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="userSettings" ... >
      <section
         name="DemoSettings.Properties.Settings"
         ... />
    </sectionGroup>
    <sectionGroup name="applicationSettings"
         ... />
       </sectionGroup>
  </configSections>
  <userSettings>
    <DemoSettings.Properties.Settings>
      <setting name="FontSetting"
               serializeAs="String">
        <value>Microsoft Sans Serif, 9.75pt</value>
      </setting>
    </DemoSettings.Properties.Settings>
  </userSettings>
  <applicationSettings>
    <DemoSettings.Properties.Settings />
  </applicationSettings>
</configuration>
```

User Configuration File

- **When the form closes, the *Save()* method of the *ApplicationSettingsBase* class will persist all the user setting information to the file *user.config*.**
  - This file can be found in the **AppData\Local** area for the user who ran the program.



  - The file **user.config** contains the saved values of the settings.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <userSettings>
    <DemoSettings.Properties.Settings>
      <setting name="FontSetting"
               serializeAs="String">
        <value>Microsoft Sans Serif, 12pt,
               style=Bold</value>
      </setting>
    </DemoSettings.Properties.Settings>
  </userSettings>
</configuration>
```

Manual Application Settings

- **We may also manually code an application setting.**
  - The code generated by Visual Studio can serve as a good guide.
- **Let's add a new setting *TitleSetting* that can be used to save the title (Text) of the form.**

1. Add a new file **ManualSetting.vb** to your project, defining a class **ManualSetting**, derived from **ApplicationSettingsBase**.

```
Imports System.Configuration

Public Class ManualSettings : Inherits _
ApplicationSettingsBase
   <UserScopedSetting(), _
   DefaultSettingValue("Default Title")> _
   Public Property TitleSetting() As String
      Get
         Return CStr(Me("TitleSetting"))
      End Get
      Set(ByVal value As String)
         Me("TitleSetting") = value
      End Set
   End Property
End Class
```

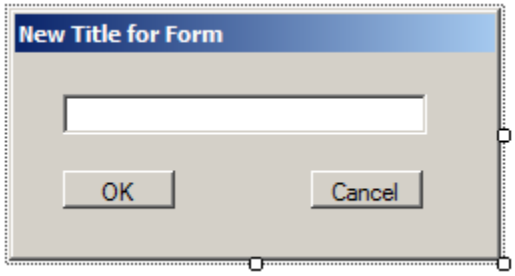
Manual Application Settings (Cont'd)

2. Add information about this new setting to the application configuration file. You may use the configuration information generated by the Designer as a model. New section is **DemoSettings.ManualSettings**. New setting within that group is **TitleSetting**.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="userSettings" ... >
          <section
name="DemoSettings.Properties.Settings" ... />
         <section
name="DemoSettings.ManualSettings" ... />
      </sectionGroup>
...
    </configSections>
    <userSettings>
      <DemoSettings.Properties.Settings>
       <setting name="FontSetting"
                 serializeAs="String">
        <value>Microsoft Sans Serif, 9.75pt</value>
       </setting>
        </DemoSettings.Properties.Settings>
      <DemoSettings.ManualSettings>
         <setting name="TitleSetting"
                  serializeAs="String">
           <value>Title in App Config File</value>
         </setting>
      </DemoSettings.ManualSettings>
   </userSettings>
    <applicationSettings>
        <DemoSettings.Properties.Settings />
    </applicationSettings>
</configuration>
```

Manual Application Settings (Cont'd)

3. Add a new form **TitleDialog** to the project. Drag a text box and two buttons onto the form. Provide the names **txtTitle**, **btnOK** and **btnCancel** for these controls.



4. Set the following properties of the new form and buttons that are normal for model dialogs:

    a. FormBorderStyle is FixedDialog

    b. No control box, minimize box, or maximize box.

    c. The OK button should have DialogResult OK, and the cancel button DialogResult of Cancel.

5. Add a handler for clicking the Change Title button.

Manual Application Settings (Cont'd)

6. Implement the handler by adding the following code to change the title of the form by bringing up the dialog. If the user clicks OK, the title of the form should be changed, and the application setting for the title should also be saved.

```
Dim dlg As TitleDialog = New TitleDialog()
dlg.txtTitle.Text = Me.Text
If dlg.ShowDialog() = DialogResult.OK Then
   Me.Text = dlg.txtTitle.Text
   Dim ms As ManualSettings = New ManualSettings()
   ms.TitleSetting = Me.Text
   ms.Save()
End If
```

7. Add code to the handler of the Load event to load the setting for the title and use it to initialize the title of the form.

```
Dim stg As Settings = Settings.Default
stg.Reload()
Me.Font = stg.FontSetting
Dim ms As ManualSettings = New ManualSettings()
ms.Reload()
Me.Text = CStr(ms.TitleSetting)
```

8. Build and run. You should now be able to change both the font and the title, and have these values persisted. You may examine the **user.config** file.

Default Values of Settings

9. Add a handler for the Restore Defaults button and supply the following code.

```
Dim stg As Settings = Settings.Default
stg.Reset()
Me.Font = stg.FontSetting
Dim ms As ManualSettings = New ManualSettings()
ms.Reset()
Me.Text = CStr(ms.TitleSetting)
ms.Save()
```

10. Specify some defaults in **app.config**.

```
<DemoSettings.Properties.Settings>
   <setting name="FontSetting"
            serializeAs="String">
      <value>Microsoft Sans Serif, 11pt</value>
   </setting>
</DemoSettings.Properties.Settings>
<DemoSettings.ManualSettings>
   <setting name="TitleSetting"
            serializeAs="String">
      <value>Title in App Config File</value>
   </setting>
</DemoSettings.ManualSettings>
```

11. Build and run. Change the values of the font and the title. Now click the Restore Defaults button, and observe that the values stored in the application configuration file are used.

Accessing the Registry

- **.NET allows you to access the Windows system registry via classes in the Microsoft.Win32 namespace.**
  - These are platform-specific classes and are not available under the **System** namespace.
  - Their use limits the portability of the application.
- **However, if you are committed to the Win32 platform, you may need to access settings stored in the Windows registry.**
- **The *Registry* class exposes six read-only static properties that return a *RegistryKey* reference to a registry hive.**
  - The properties are: **ClassesRoot**, **CurentConfig**, **CurrentUser**, **DynData**, **LocalMachine**, **PerformanceData** and **Users**.

```
Dim reg As RegistryKey = Registry.ClassesRoot
```

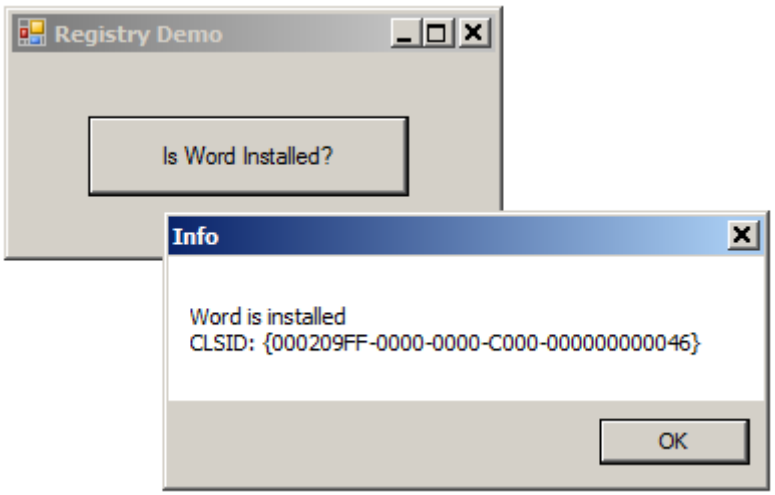
- **The RegistryKey class has three instance properties:**
  - The **Name** property is the name of the key.
  - The **ValueCount** property is the count of the values for the key.
  - The **SubKeyCount**, if greater than 0, is the number of subkeys for this key.

Accessing the Registry (Cont'd)

- **The RegistryKey class also contains several methods, including:**
  - **OpenSubKey**, which returns a reference to a subkey of the instance key.
  - **GetValueNames**, which returns a list of names for the values associated with the key.
  - **GetValue**, which returns a value for a key.
  - **Close**, which closes the key.
- **There are many other methods to manipulate the registry that you should research using MSDN.**

Example: Manipulating the Registry

- **The example *ManipulatingTheRegistry* illustrates the use of the *Registry* and *RegistryKey* classes.**
  - This example checks the registry to determine if Microsoft Word is installed. It also displays the CLSID for Word if it is installed.



- **The code that accomplishes this is shown on the next page.**

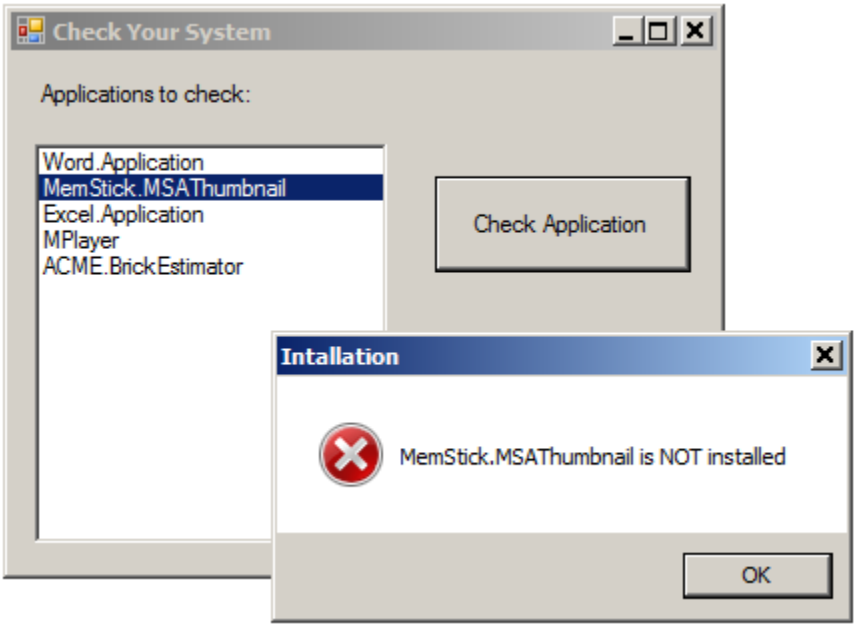
Manipulating the Registry (Cont'd)

```
Private Sub btnWord_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs) _
Handles btnWord.Click
   ' Get a reference to HKEY_CLASSES_ROOT
   Dim hive As RegistryKey = Registry.ClassesRoot
   ' Lookup the ProgID Word.Application
   Dim wordProgID As RegistryKey = _
   hive.OpenSubKey("Word.Application")
   If Not wordProgID Is Nothing Then
      Dim msg As String = "Word is installed"
      ShowWordInformation(msg, wordProgID)
      wordProgID.Close()
   Else
      MessageBox.Show("Word is not installed", _
         "Info")
   End If
End Sub

Private Sub ShowWordInformation(ByVal msg As _
String, ByVal progID As RegistryKey)
   ' Get a reference to HKEY_CLASSES_ROOT
   Dim hive As RegistryKey = Registry.ClassesRoot

   ' Lookup the value of CLSID
   Dim clsid As RegistryKey = _
      progID.OpenSubKey("CLSID")
   Dim strCLSID As String = _
      CStr(clsid.GetValue(""))
   clsid.Close()

   msg &= vbNewLine & "CLSID: " & strCLSID
   MessageBox.Show(msg, "Info")
End Sub
```

Lab 10

Checking Your System

In this lab, you will use a config file to specify a set of application ProgIDs that should be loaded into a listbox. The program will allow the user to select a ProgID from the listbox and check to see if the corresponding application is loaded on the system.



Detailed instructions are contained in the Lab 10 write-up at the end of the chapter.

Suggested time: 45 minutes

Summary

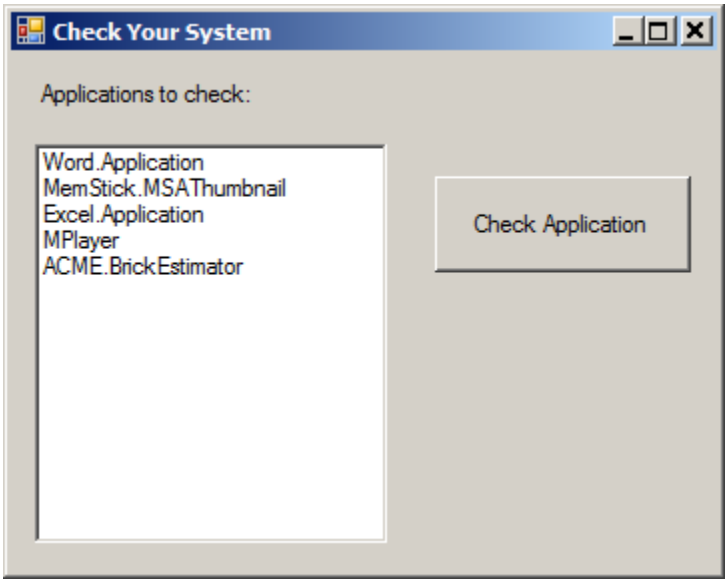
- **The Application class represents the class that manages a Windows Forms application.**
  - It contains methods to start and stop a Windows Forms application.
- **.NET applications use configuration files encoded using XML.**
  - The **ConfigurationManager** class provides access to the AppSettings or user-defined sections of a .config file.
- **The application settings facilities in .NET make it easy to persist both application wide and user-specific settings.**
- **.NET allows you to access the Windows system registry via classes in the Microsoft.Win32 namespace.**
  - These are platform-specific classes and are not available under the **System** namespace.
  - Their use limits the portability of the application.
- **The *Registry* class provides access to the registry hives.**
- **The *RegistryKey* class contains properties and methods that can be used to navigate the registry.**

Lab 10

Checking Your System

Introduction

In this lab, you will use a config file to specify a set of application ProgIDs that should be loaded into a listbox. The program will allow the user to select a ProgID from the listbox and check to see if the corresponding application is loaded on the system.



Suggested Time: 45 minutes

Root Directory: **OIC\WinVb**

Directories: **Labs\Lab10\CheckingYourSystem** (do your work here)
Chap10\CheckingYourSystem (contains lab solution)

Instructions

1. Create a new Windows Forms Application named **CheckingYourSystem**. Name the form class and the associated file **MainForm**.

2. Build a config file that resembles the following. Name the file **App.config**, save it in the source directory, and add it to your Visual Studio project.

```
<configuration>
  <appSettings>
     <add key="Count" value="3" />
     <add key="App1" value="Word.Application" />
     <add key="App2" value="Excel.Application" />
     <add key="App3" value="MemStick.MSAThumbnail" />
```

```
  </appSettings>
</configuration>
```

3. Design your main form to resemble that shown above.

4. In your form's **Load** event handler, write code to read the config file and load the ProgIDs into the listbox.

```
Private Sub MainForm_Load(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles MyBase.Load
  Dim parms As NameValueCollection = ConfigurationManager.AppSettings
  Dim count As Integer = Convert.ToInt32(parms.Get("Count"))
  Dim key As String = ""
  Dim value As String = ""
  Dim i As Integer
  For i = 1 To count
      key = "App" + i.ToString()
      value = parms.Get(key)
      lstApplications.Items.Add(value)
  Next
End Sub
```

5. Write code in the **Click** event of your Check button to use the ProgID of the selected listbox item and test to see if it is listed in the registry as an installed application. If it is, it will be listed under the HKEY_CLASSES_ROOT hive.

```
Private Sub btnCheckSelected_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnCheckSelected.Click
   ' Get a reference to HKEY_CLASSES_ROOT
   Dim hive As RegistryKey = Registry.ClassesRoot

   ' Get the ProgID from the listbox
   Dim progID As String = ""
   Try
      progID = lstApplications.SelectedItem.ToString()
      If progID = "" Then
         Exit Sub
      End If
   Catch
      Exit Sub
   End Try

   ' Lookup the ProgID
   Dim key As RegistryKey = hive.OpenSubKey(progID)
   If Not key Is Nothing Then
      Dim msg As String = progID & " is installed"
      MessageBox.Show(msg, "Intallation", MessageBoxButtons.OK, _
         MessageBoxIcon.Information)
      key.Close()
   Else
      Dim msg As String = progID & " is NOT installed"
      MessageBox.Show(msg, "Intallation", MessageBoxButtons.OK, _
         MessageBoxIcon.Hand)
   End If
End Sub
```

6. Compile and run your program.