Windows Forms
Using C#

Student Guide

Revision 4.0

Object Innovations Course 4130

Windows Forms Using C#
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)

Chapter 1 Introduction to Windows Forms

Chapter 2 Visual Studio and the Forms Designer

Chapter 3 Using Controls

Chapter 4 List Controls

Chapter 5 Working with Dialogs

Chapter 6 Menus, Toolbars and Status Bars

Chapter 7 Advanced Windows Forms Topics

Chapter 8 Using Advanced Controls

Chapter 9 Resources

Chapter 10 Applications and Settings

Chapter 11 Data Access

Chapter 12 Data Binding

Chapter 13 Windows Forms and WPF Interoperation

Appendix A Learning Resources

Directory Structure

- **The course software installs to the root directory *C:\OIC\WinCs*.**
  - 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 1 Introduction to Windows Forms 1
What are Windows Forms?.. 3
Windows Forms Class Hierarchy 4
Control Class.. 5
Form Class 7
Form Class Methods 9
Building a Form 10
Application Class 12
Simple Windows Application 13
Using the .NET Framework SDK 18
Using Controls 19
Common Control Properties 20
Ambient Control Properties 22
Example: Placing a Control in a Form.. 24
Common Control Events.. 26
Trapping Events 28
Using the Button Control 29
HelloWorld – Step 3 30
Using the Label Control 31
Using the TextBox Control 32
Example – Complete Application 34
MSDN Documentation 37
Lab 1 38
Summary 39

Chapter 2 Visual Studio and the Forms Designer 41
Visual Studio.. 43
Using the Forms Designer 44
Example: Creating a Windows Forms Application 47
Examining the Forms Designer Generated Code... 52
Designing "Pretty" Forms 53
Designing "Easy-to-Use" Forms... 54
Setting the Tab Order.. 55
Defining Keyboard Shortcuts 56
Defining Default and Cancel Buttons... 58
Lab 2 59
Summary 60

Chapter 3 Using Controls... 63
Controls... 65
Using the TextBox – Again 66
Using the Clipboard... 68

Lab 3A (Optional) ... 69
Making Selections ... 70
Using the CheckBox ... 71
Example: Using the CheckBox ... 72
Using the RadioButton with a GroupBox ... 74
Example: Using a Radio Button ... 76
Working with Ranges ... 78
Using the NumericUpDown ... 79
Example: Using the NumericUpDown ... 80
Using the TrackBar ... 81
Example: Using the TrackBar ... 82
Using the ProgressBar ... 83
Example: Using the ProgressBar ... 84
Working with Dates ... 86
Using the MonthCalendar ... 87
Using DateTimePicker ... 90
Example: Using the DateTimePicker ... 91
Other Controls ... 94
Tracing ... 95
Debug and Trace Classes ... 96
Tracing Example ... 97
Viewing Trace Output ... 98
Debug Statements ... 99
Debug Output ... 100
WriteLine Syntax ... 101
Lab 3B ... 102
Summary ... 103

Chapter 4   List Controls ... 107
Working with Lists ... 109
Using a ListBox ... 110
Selected Items ... 111
Selected Indices ... 112
Other ListBox Features ... 113
Adding and Removing Items Dynamically ... 115
Example: Using a ListBox ... 116
Using the ComboBox ... 119
ComboBox Example ... 120
Storing Objects in List Controls ... 122
Lab 4A ... 123
Using the DomainUpDown Control ... 124
Example: Using the DomainUpDown ... 125
Using a ListView ... 127
Adding Columns to a ListView ... 129
Adding Items to a ListView ... 130
Example: Using a ListView ... 132

Lab 4B 134
Summary 135

Chapter 5 Working with Dialogs 141
Modal and Modeless Dialogs 143
MessageBox 144
MessageBox Show Method 145
Closing a Form 148
Custom Dialogs 149
Modal Dialogs 150
Example: Modal Data Entry Dialogs 151
Creating a New Form 152
Common Dialog Properties 153
Designing the Form 154
Configuring the DialogResult 155
Displaying the Form 156
Accessing Data on the Form 157
Changing the Behavior of a Button's DialogResult 160
Displaying Errors with the ErrorProvider Control 162
Validation Using the ErrorProvider 165
Modeless Dialogs 167
Example: Modeless Data Entry Dialogs 168
Designing the Modeless Dialog 169
Displaying the Form 171
Managing the Relationship between the Parent and Modeless Dialog 172
Programming the Apply and Close Buttons 173
Programming the Apply Button 174
Managing the Number of Instances of the Modeless Dialog 176
Lab 5A 178
Common Dialogs 179
Using the Common Dialog Controls 180
Example: Using Common Dialogs 181
Lab 5B 185
Summary 186

Chapter 6 Menus, Toolbars and Status Bars 193
Menus 195
MenuStrip Control 196
Example: Integrating Menus into an Application 197
Attaching a Menu to a Form 199
Configuring Items in a Menu 200
Responding to Menu Events 202
DropDownOpening Event 206
ContextMenuStrip Control 207
Example: Integrating a Context Menu into an Application 208
Context Menu Events 209

Handling Multiple Events ... 210
Lab 6A ... 212
Status Bars ... 213
StatusStrip Example ... 214
StatusStrip Demo ... 215
A Quick Status Bar ... 219
Toolbars ... 220
ToolStrip Demo ... 221
Importing Images ... 223
Associating an Event Handler ... 224
Image and Text on Buttons ... 225
Lab 6B ... 226
Summary ... 227

Chapter 7 Advanced Windows Forms Topics ... 233
Forms and Controls ... 235
Parent/Child Relationships ... 236
Example: Using Parent/Child Relationships ... 238
Owner/Owned Relationships ... 240
Example: Using Owner/Owned Relationships ... 241
Top-Most Forms ... 244
Clipboard Object ... 245
Placing Data on the Clipboard ... 246
Retrieving Data from the Clipboard ... 247
Visual Inheritance ... 248
Building the Base Form ... 249
Example: Using Visual Inheritance ... 250
BackgroundWorker Component ... 257
BackgroundWorker Example ... 258
BackgroundWorker Code ... 260
ClickOnce Deployment ... 261
Web Examples Setup ... 262
Home Page for Web Example ... 266
ClickOnce Demonstration ... 267
Publishing a ClickOnce App ... 268
Uninstalling the Application ... 271
Installing the Application ... 272
Summary ... 273

Chapter 8 Using Advanced Controls ... 275
Panel Control ... 277
Panel and GroupBox Example ... 278
TreeView Control ... 279
TreeView Properties ... 280
TreeView Methods ... 281
TreeView Events ... 282

TreeNode Class 283
Adding Nodes 284
Removing Nodes 285
Iterating Through Nodes 286
TreeView Example 287
TreeView Demonstration 288
ImageList 292
ImageList Demonstration 293
Image Collection Editor 294
Lab 8A 296
TabControl 297
Controls on Tab Pages 298
Selected Index 299
TabControl Demonstration 300
Tab Control Event Handling 301
Lab 8B 302
SplitContainer 303
File Browser Demo 304
WebBrowser Control 309
Lab 8C 310
Summary 311

Chapter 9 Resources 319
Resources 321
Image Resources 322
Loading the Bitmaps 323
Embedded Resources 324
Accessing Embedded Resources 325
Creating String Resources 326
Strings in the Program 327
String Resource Demo 328
Accessing Resources from Code 330
Cultures and Internationalization 335
.NET Support for Cultures 337
Example: Using CultureInfo 338
Changing the Current Culture 342
Building Localizable Forms 343
MainForm.resx File 345
Code for Localization 347
Visual Studio Localization Support 348
Demo: Localizing Using VS.NET 350
Lab 9 355
Summary 356

Chapter 10 Applications and Settings 359
The Application Class 361

Starting and Stopping Applications ... 362
Life Cycle Demonstration ... 363
Application Events ... 364
Logging to a File ... 365
Closing a Window ... 366
Processing Windows Messages ... 368
Filtering Messages ... 369
Example: Using Application Class ... 371
Configuration Files ... 372
Reading Configuration Files ... 374
Example: Using Config Files ... 375
Configuration File and Visual Studio ... 377
Application Settings ... 378
Application Settings Using Visual Studio ... 379
Application Settings Demo ... 380
Application Configuration File ... 385
User Configuration File ... 386
Manual Application Settings ... 387
Default Values of Settings ... 391
Accessing the Registry ... 392
Example: Manipulating the Registry ... 394
Lab 10 ... 396
Summary ... 397

Chapter 11   Data Access ... 401
ADO.NET ... 403
.NET Namespaces ... 404
ADO.NET Architecture ... 405
.NET Data Providers ... 407
Connected Data Access ... 408
SmallPub Database ... 409
Connected Programming Example ... 410
DataSet Architecture ... 414
Why DataSet? ... 415
DataSet Components ... 416
DataSet Example Program ... 417
XML File Schema Definition ... 418
XML File Data ... 419
Reading and Writing XML ... 420
Accessing a DataSet ... 421
Adding a New Row ... 422
Searching and Updating a Row ... 423
Deleting a Row ... 424
Data Adapters ... 425
Data Adapter Example Program ... 426
Lab 11A ... 427

Language Integrated Query (LINQ) 428
Bridging Objects and Data 429
Using Server Explorer 430
LINQ Demo 431
Object Relational Designer 432
IntelliSense 434
Basic LINQ Query Operators 435
Obtaining a Data Source 436
LINQ Query Example 437
Filtering 438
Ordering 439
Aggregation 440
Obtaining Lists and Arrays 441
Deferred Execution 442
Modifying a Data Source 443
Performing Inserts via LINQ to SQL 444
Performing Deletes via LINQ to SQL 445
Performing Updates via LINQ to SQL 446
Lab 11B 447
Summary 448

Chapter 12 Data Binding 457
Data Binding Concept 459
Simple Data Binding 460
Binding to a List 461
SimpleList Example 462
Complex Data Binding Example 463
Binding to a DataGrid 464
DataView 466
Filtering and Sorting 467
DataView and Data Binding 468
DataView Example 469
Column Formatting 471
Lab 12A 472
Data Binding with BindingSource 473
BindingSource Architecture 474
Binding to a List 475
Setting up the Bindings 476
Category Class 477
Code in the Form 478
DataGridView Control 480
DataGridView Sample Program 481
DataGridView Demo 482
Performing a Query 488
Lab 12B 491
Summary 492

Chapter 13 Windows Forms and WPF Interoperation.. 497
Windows Presentation Foundation 499
What Is XAML? 500
Default Namespace 501
XAML Language Namespace... 502
Code-Behind File 503
OneButton Example.. 504
Interoperating with Windows Forms 505
Add a Form to a WPF Application 506
Demo: Form in WPF Application... 507
Add a WPF Window to a Windows Forms Application............................. 511
Mixing WPF and Windows Forms in the Same Window........................... 512
Hosting a Windows Forms Control Using Code 513
WindowsFormsHost via Code 514
Windows Forms MonthCalendar.. 515
WindowsFormsHost via XAML.. 516
Summary 517

Appendix A Learning Resources... 519

Chapter 2

Visual Studio and the Forms Designer

Visual Studio and the Forms Designer

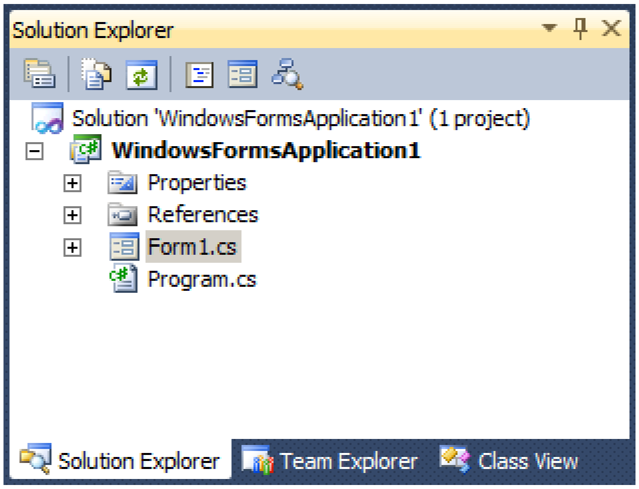
Objectives

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.**

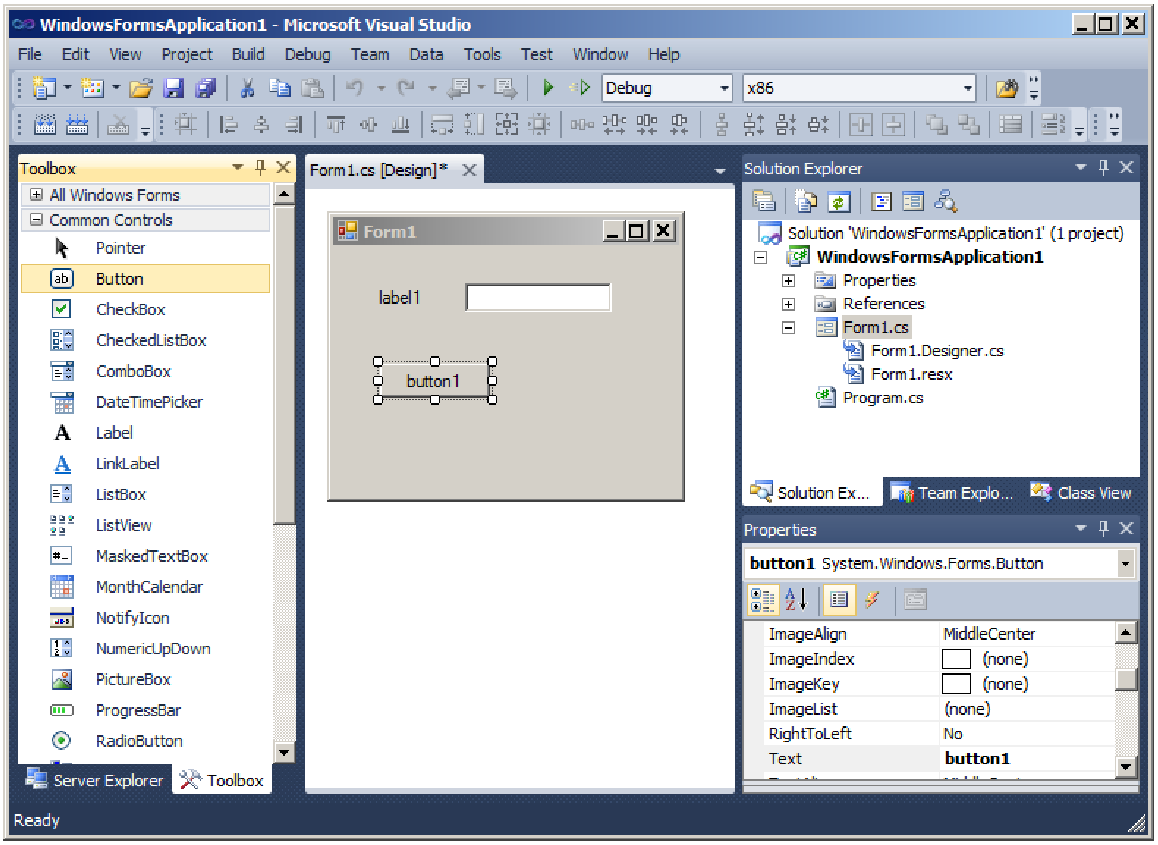
Visual Studio

- **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.

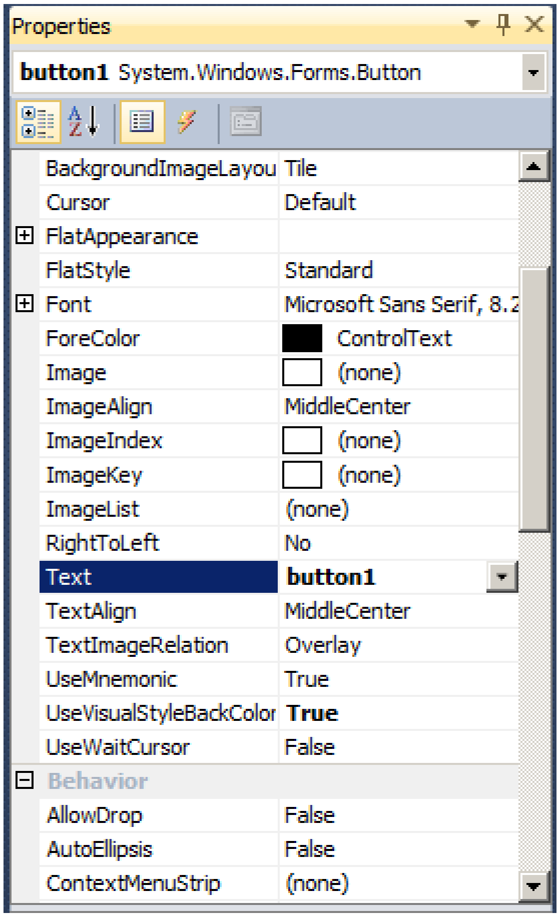


ImageAlign	MiddleCenter
ImageIndex	□ (none)
ImageKey	□ (none)
ImageList	(none)
RightToLeft	No
Text	**button1**

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.

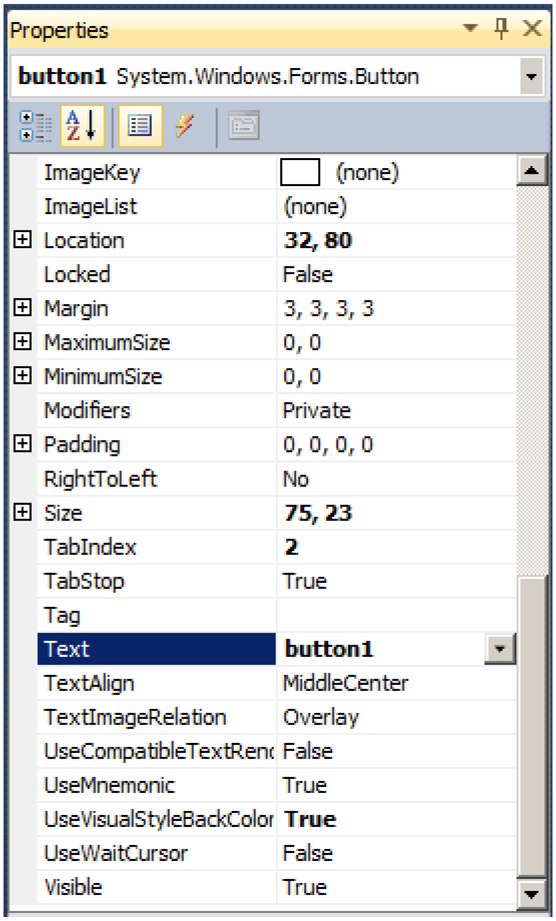
By category: Alphabetically:



Properties

button1 System.Windows.Forms.Button

BackgroundImageLayou	Tile
Cursor	Default
⊞ FlatAppearance	
FlatStyle	Standard
⊞ Font	Microsoft Sans Serif, 8.2
ForeColor	■ ControlText
Image	□ (none)
ImageAlign	MiddleCenter
ImageIndex	□ (none)
ImageKey	□ (none)
ImageList	(none)
RightToLeft	No
Text	**button1**
TextAlign	MiddleCenter
TextImageRelation	Overlay
UseMnemonic	True
UseVisualStyleBackColor	**True**
UseWaitCursor	False
⊟ Behavior	
AllowDrop	False
AutoEllipsis	False
ContextMenuStrip	(none)



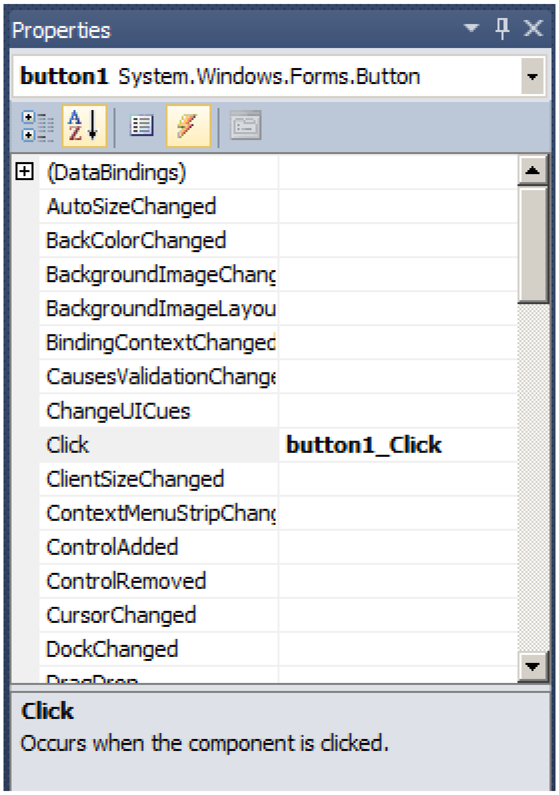
Properties

button1 System.Windows.Forms.Button

ImageKey	□ (none)
ImageList	(none)
⊞ Location	**32, 80**
Locked	False
⊞ Margin	3, 3, 3, 3
⊞ MaximumSize	0, 0
⊞ MinimumSize	0, 0
Modifiers	Private
⊞ Padding	0, 0, 0, 0
RightToLeft	No
⊞ Size	**75, 23**
TabIndex	**2**
TabStop	True
Tag	
Text	**button1**
TextAlign	MiddleCenter
TextImageRelation	Overlay
UseCompatibleTextRen	False
UseMnemonic	True
UseVisualStyleBackColor	**True**
UseWaitCursor	False
Visible	True

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.



⊞ (DataBindings)	
AutoSizeChanged	
BackColorChanged	
BackgroundImageChang	
BackgroundImageLayou	
BindingContextChangec	
CausesValidationChange	
ChangeUICues	
Click	**button1_Click**
ClientSizeChanged	
ContextMenuStripChang	
ControlAdded	
ControlRemoved	
CursorChanged	
DockChanged	
DragDrop	

Click

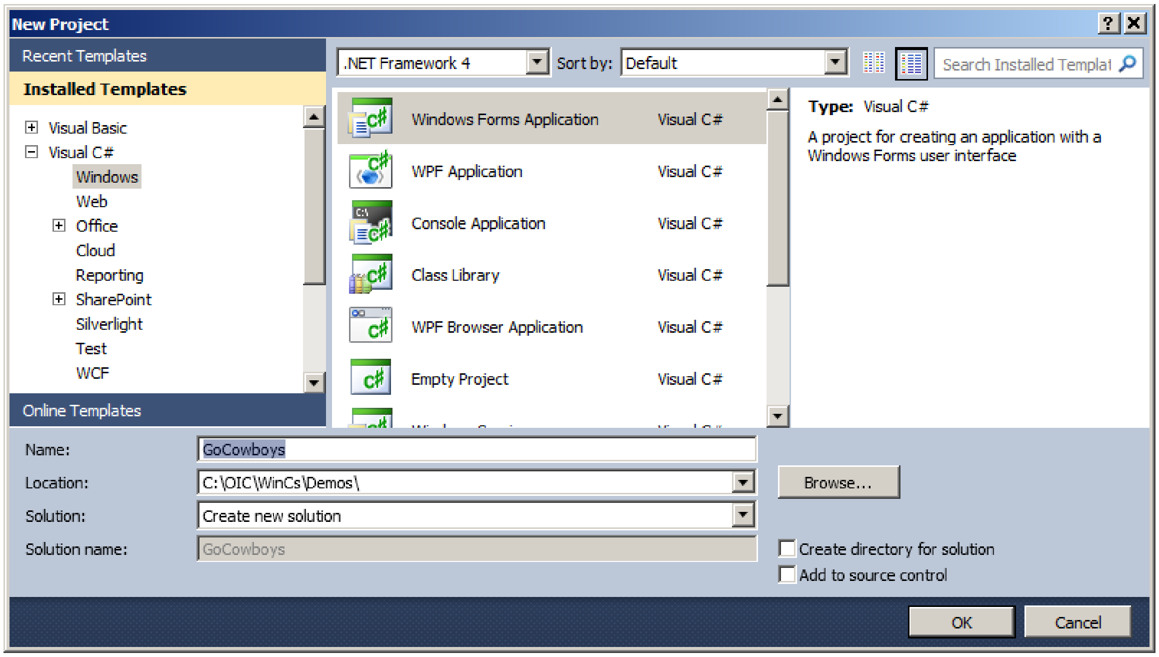
Occurs when the component is clicked.

  - 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 C# Windows Forms Application project named **GoCowboys** in the Demos directory. Leave unchecked "Create directory for solution."



New Project

.NET Framework 4 Sort by: Default

Windows Forms Application Visual C#

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: GoCowboys

Location: C:\OIC\WinCs\Demos\

Solution: Create new solution

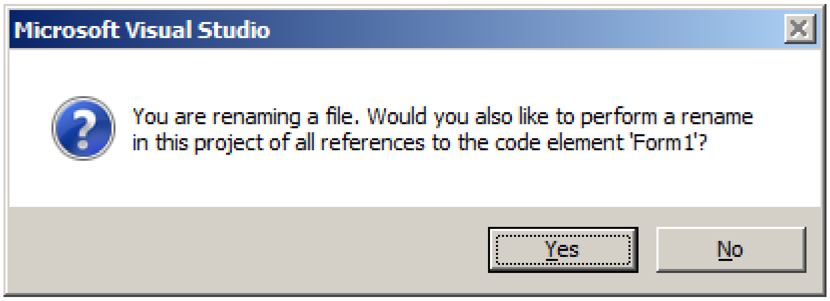
Solution name: GoCowboys

□ Create directory for solution

□ Add to source control

Example: Creating ... (Cont'd)

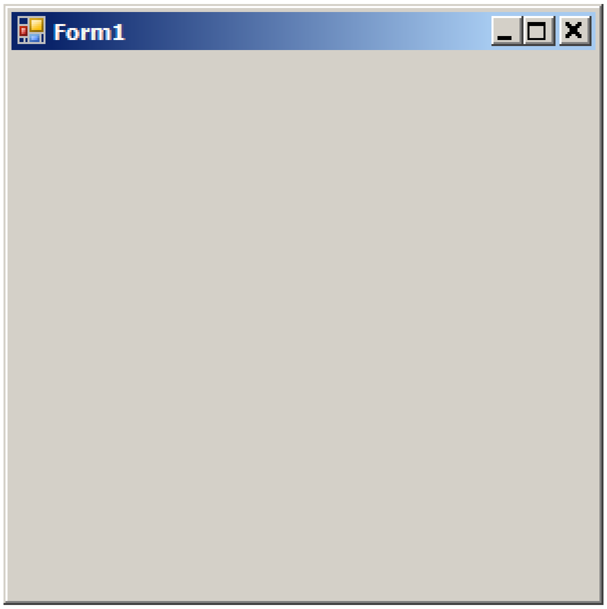
2. We then rename the source file for the form to **MainForm.cs**. You will be asked if you want to rename the corresponding code elements. Say yes.



Microsoft Visual Studio

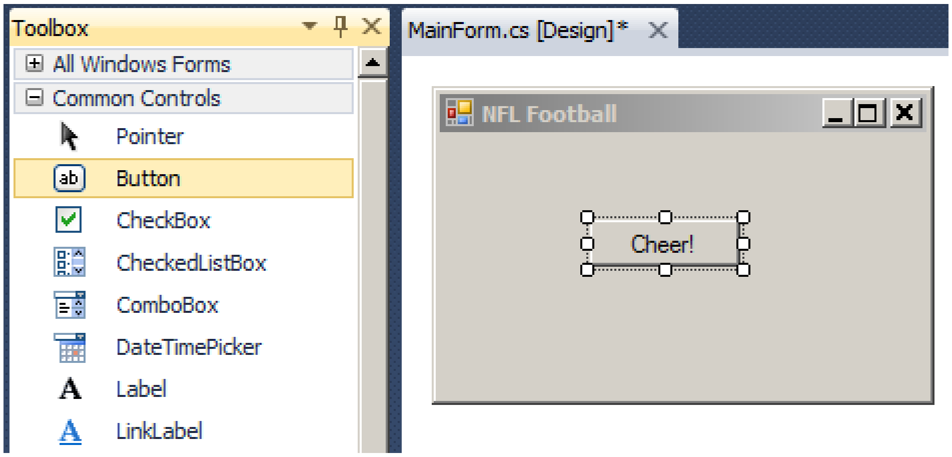
You are renaming a file. Would you also like to perform a rename in this project of all references to the code element 'Form1'?

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.



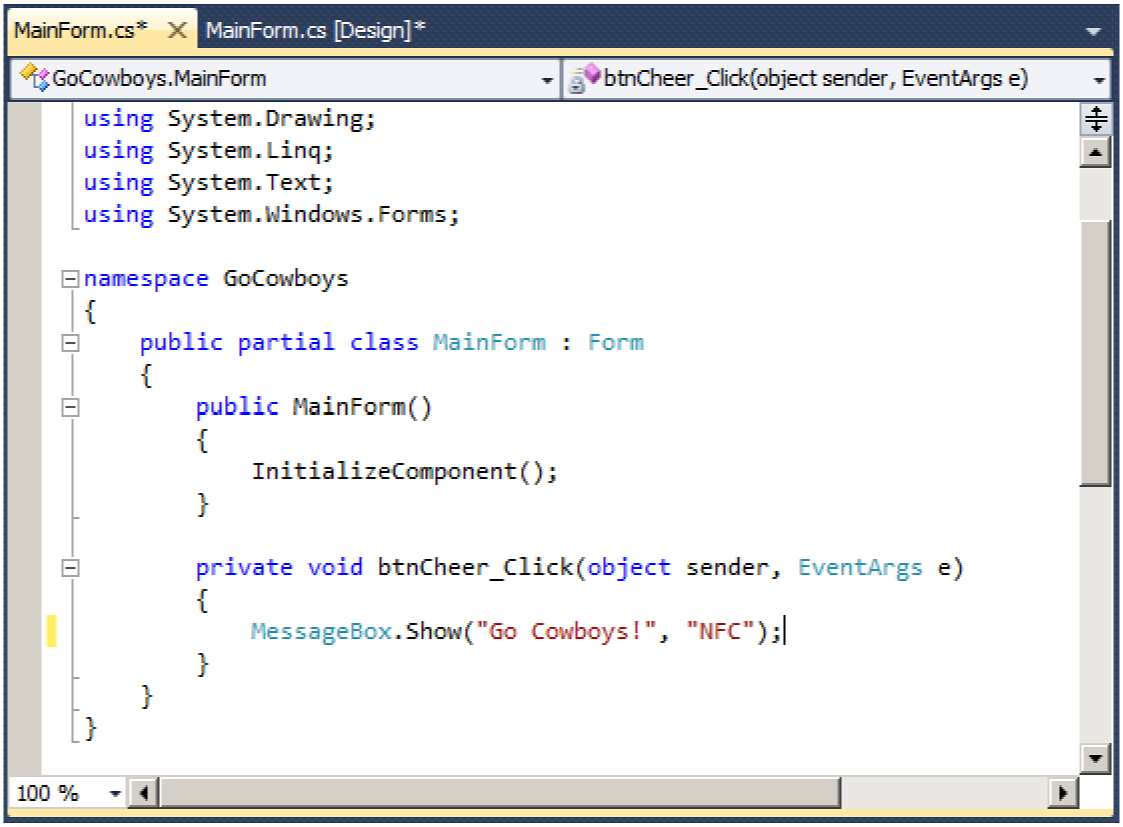
5. We want to make sure the following property values are set for the button and form:

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 and delegate 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!"



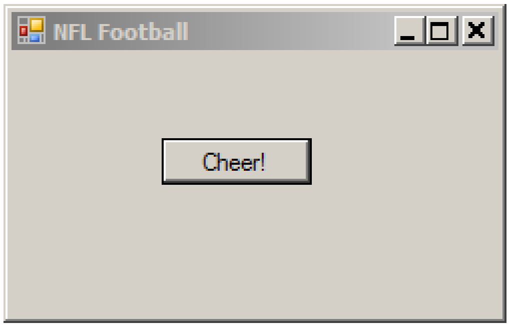
```
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace GoCowboys
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

        private void btnCheer_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Go Cowboys!", "NFC");
        }
    }
}
```

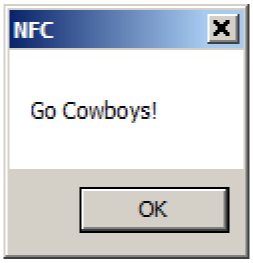
- **Note the *partial* modifier on the class.**

   – This feature, introduced in .NET 2.0, enables wizard-generated code to be maintained in a separate file.

Example: Creating ... (Cont'd)

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

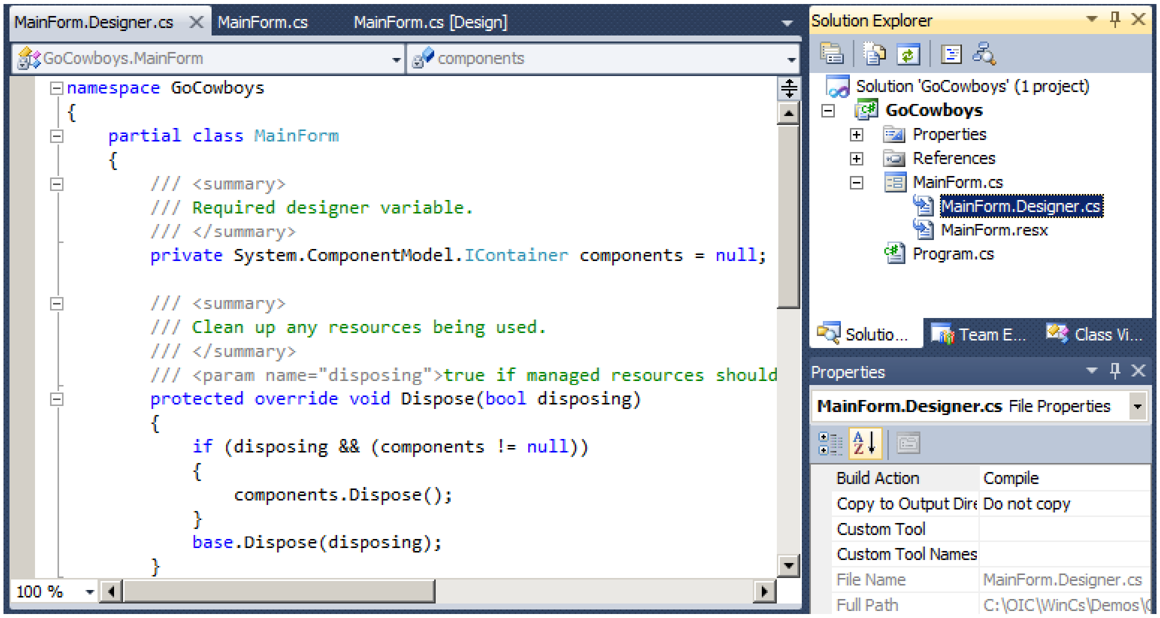




Go Cowboys!

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.cs**.

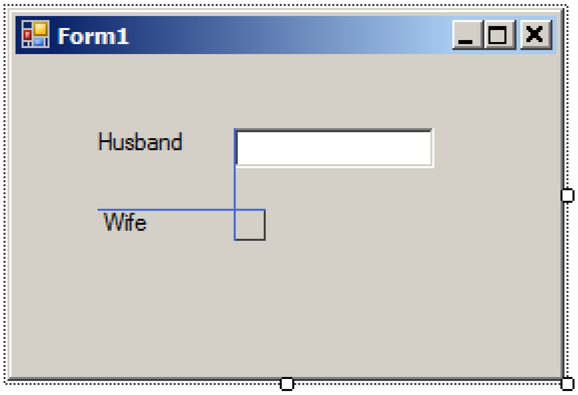


```
namespace GoCowboys
{
    partial class MainForm
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
```

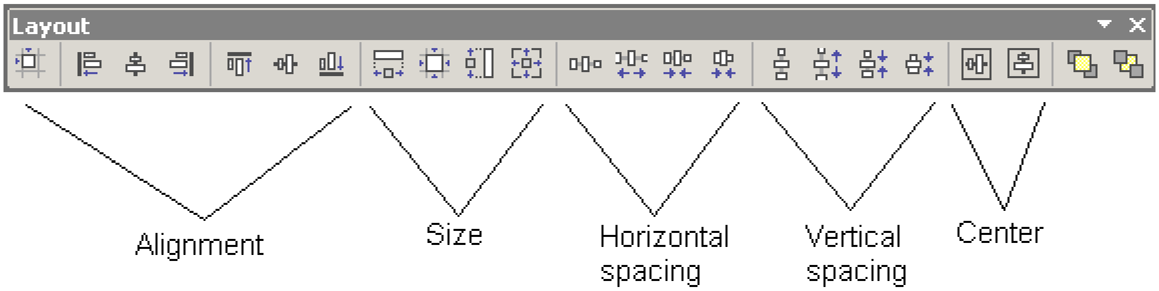
Build Action	Compile
Copy to Output Dire	Do not copy
Custom Tool	
Custom Tool Names	
File Name	MainForm.Designer.cs
Full Path	C:\OIC\WinCs\Demos\

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.

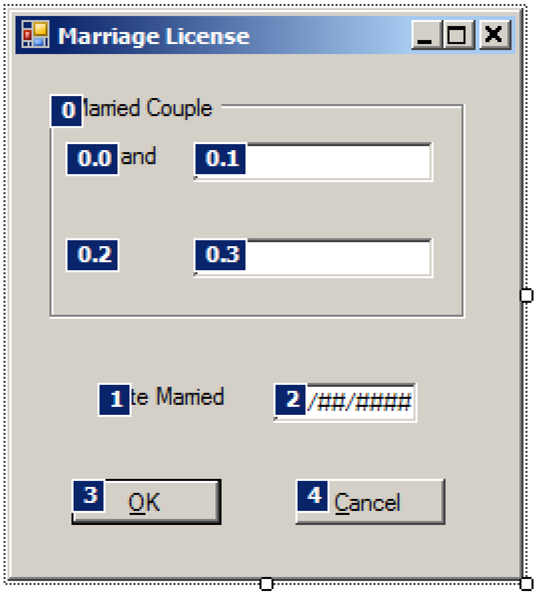


Designing "Easy-to-Use" Forms

- **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.



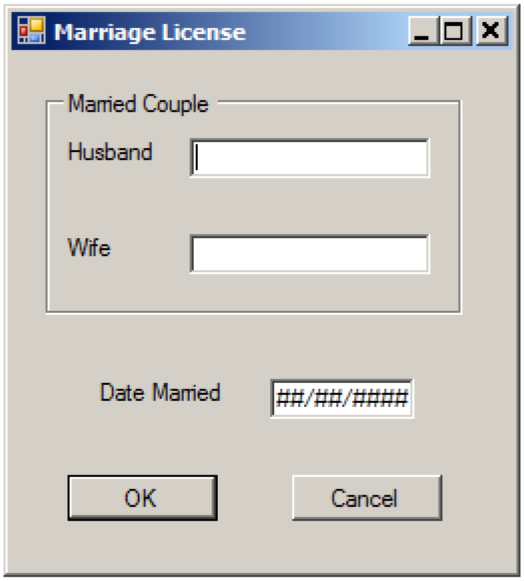
  - You should always include your labels in the tab order directly before the control they label. The reason will be apparent on the next page!

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 OK 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).

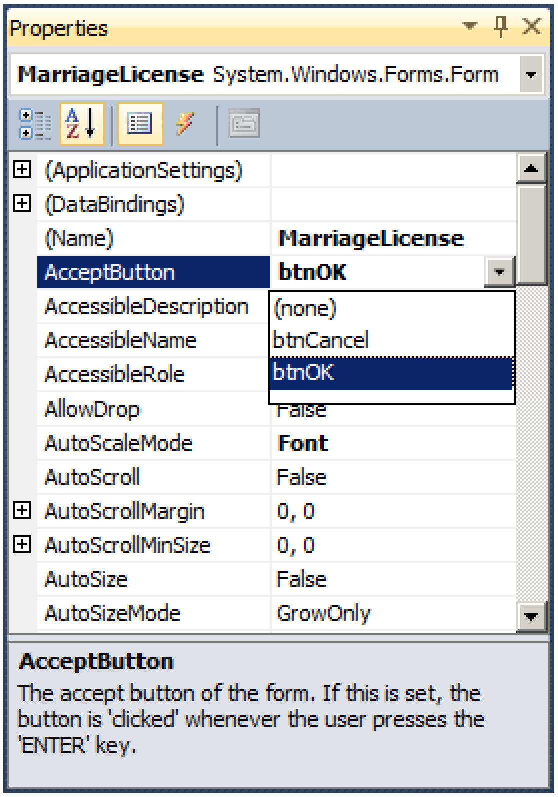
Defining Keyboard Shortcuts (Cont'd)

- **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.



Properties

MarriageLicense System.Windows.Forms.Form

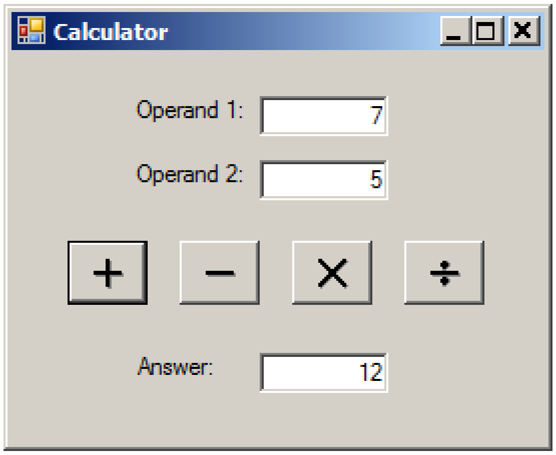
⊞ (ApplicationSettings)	
⊞ (DataBindings)	
(Name)	**MarriageLicense**
AcceptButton	**btnOK**
AccessibleDescription	(none)
AccessibleName	btnCancel
AccessibleRole	btnOK
AllowDrop	False
AutoScaleMode	**Font**
AutoScroll	False
⊞ AutoScrollMargin	0, 0
⊞ AutoScrollMinSize	0, 0
AutoSize	False
AutoSizeMode	GrowOnly

AcceptButton
The accept button of the form. If this is set, the button is 'clicked' whenever the user presses the 'ENTER' key.

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

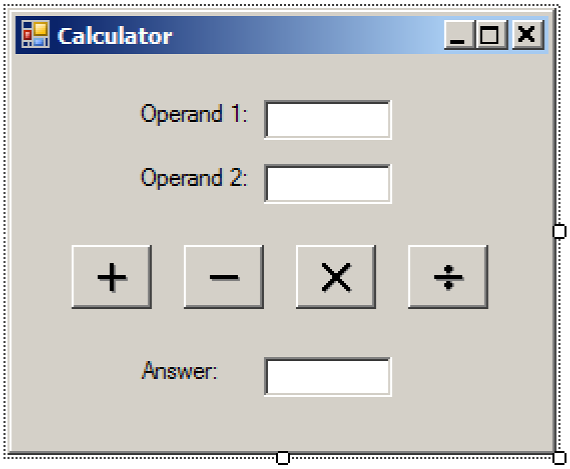
- **Visual Studio makes it easy for programmers to build Windows Forms applications.**
- **The Forms Designer allows programmers to drag controls from a toolbox and visually place them on a form.**
- **The Properties Window can be used to specify values for form and control properties.**
- **The Forms Designer generates code based on the programmer's drag-drop actions and property settings.**
- **C# 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\WinCs**

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

Instructions

1. Create a new C# 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.

4. Set the properties of each control to the following values:

Control	Property	Value
Label	Name	lblOperand1
	Text	Operand 1:
TextBox	Name	txtOperand1
	Text	(blank)
	TextAlign	Right
Label	Name	lblOperand2
	Text	Operand 2:
TextBox	Name	txtOperand2
	Text	(blank)
	TextAlign	Right
Button	Name	btnAdd
	Text	(blank)
	Image	MISC18.ICO
Button	Name	btnSubtract
	Text	(blank)
	Image	MISC19.ICO
Button	Name	btnMultiply
	Text	(blank)
	Image	MISC20.ICO
Button	Name	btnDivide
	Text	(blank)
	Image	MISC21.ICO
Label	Name	lblAnswer
	Text	Answer:
TextBox	Name	txtAnswer
	Text	(blank)
	TextAlign	Right

5. Trap the **Click** event for each of the four buttons that specify math operations.

6. In each handler, write code to convert the string data in each textbox to a floating point value. Perform the appropriate math operation for the button. Finally, place the result back in the textbox that holds the answer. Compile and run the program.

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 static methods.**

Starting and Stopping Applications

- **As you have seen thus far, Windows Forms applications have a *Main* function 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.

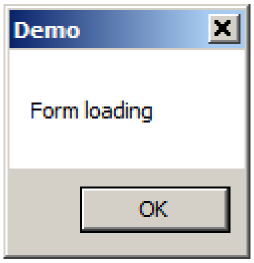
```
static void Main()
{
    ...
    Application.Run(new MainForm ());
}
```

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

```
private void mnuExit_Click(object sender,
EventArgs e)
{
    Application.Exit();
}
```

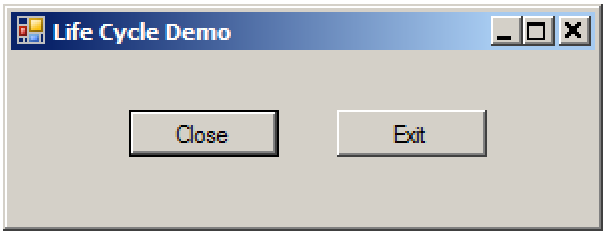
Life Cycle Demonstration

- **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.



Form loading

  - 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 static void OnExit(object sender,
EventArgs e)
{
   Log.WriteLine("OnExit called");
   MessageBox.Show("Exiting application", "Demo");
}
```

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

```
private void MainForm_Load(object sender,
EventArgs e)
{
   Log.Clear(); Log.WriteLine("Form loading");
   MessageBox.Show("Form loading", "Demo");
   Application.ApplicationExit
      += new EventHandler(OnExit);
}
```

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 static void WriteLine(string str)
   {
      StreamWriter writer =
         new StreamWriter(@"c:\OIC\log.txt", true);
      writer.WriteLine(str);
      writer.Close();
   }
   ...
}
```

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

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

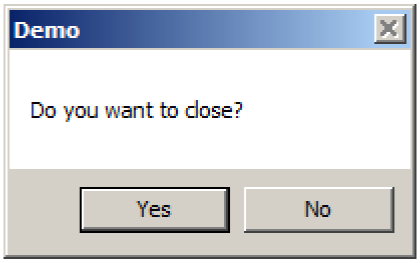
- **The log file *log.txt* for the simple run shows that the event handler was indeed called.**

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 void btnClose_Click(object sender,
EventArgs e)
{
   Log.WriteLine("Close clicked");
   Close();
}
```

- **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.**



Do you want to close?

Closing a Window (Cont'd)

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

```
private void MainForm_FormClosing(object sender,
FormClosingEventArgs e)
{
   Log.WriteLine("Form closing");
   DialogResult status = MessageBox.Show(
      "Do you want to close?", "Demo",
      MessageBoxButtons.YesNo);
   if (status == DialogResult.No)
      e.Cancel = true;
}

private void MainForm_FormClosed(object sender,
FormClosedEventArgs e)
{
   Log.WriteLine("Form closed");
}
```

- **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 void mnuDoSomethingLong_Click(object
sender, EventArgs e)
{
   for (i=Int32.MinValue;i<=Int32.MaxValue;i++)
   {
      // do something computationally intensive
      if ((Math.Abs(i) % 10000) == 0)
         Application.DoEvents();
   }
}
```

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
class FilterMouseMessages : IMessageFilter
{
   public bool PreFilterMessage(ref Message m)
   {
      // Filter right mouse button clicks
      if (m.Msg == 0x204)
         return true;
      else
         return false;
   }
}
```

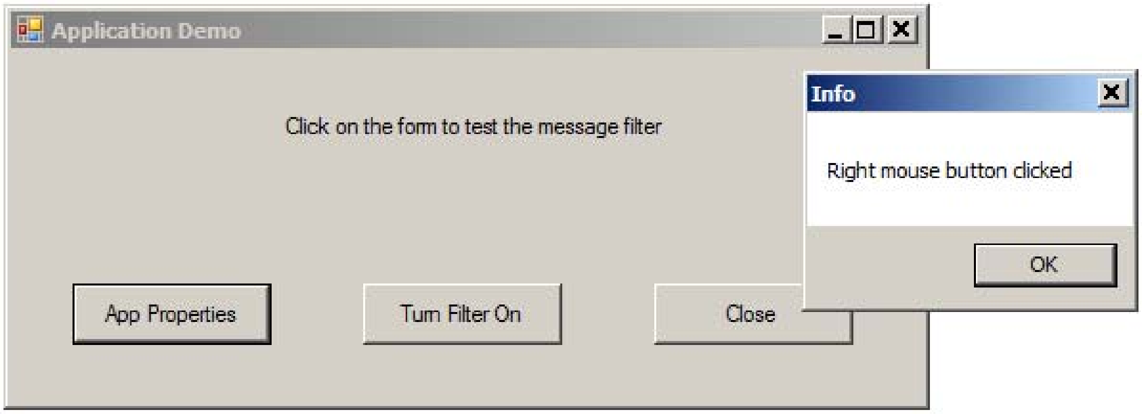
Filtering Messages (Cont'd)

- **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 IMessageFilter msgFilter = null;

private void btnFilter_Click(object sender,
EventArgs e)
{
   // Toggle filter
   if (msgFilter == null)
   {
      msgFilter = new FilterMouseMessages();
      btnFilter.Text = "Turn Filter Off";
      Application.AddMessageFilter(msgFilter);
   }
   else
   {
      Application.RemoveMessageFilter(msgFilter);
      msgFilter = null;
      btnFilter.Text = "Turn Filter On";
   }
}
```

Example: Using Application Class

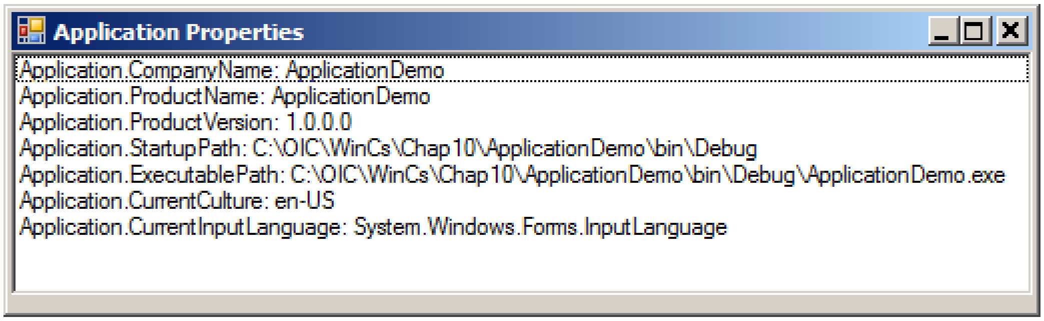
- **The example program *ApplicationDemo* demonstrates the use of the application class.**



Click on the form to test the message filter

Right mouse button clicked

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



```
Application.CompanyName: ApplicationDemo
Application.ProductName: ApplicationDemo
Application.ProductVersion: 1.0.0.0
Application.StartupPath: C:\OIC\WinCs\Chap10\ApplicationDemo\bin\Debug
Application.ExecutablePath: C:\OIC\WinCs\Chap10\ApplicationDemo\bin\Debug\ApplicationDemo.exe
Application.CurrentCulture: en-US
Application.CurrentInputLanguage: System.Windows.Forms.InputLanguage
```

- **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.

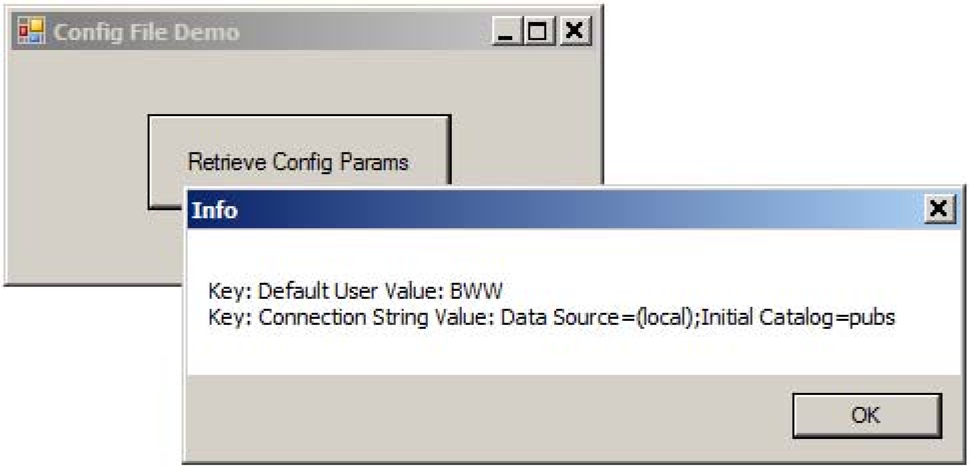
Reading Configuration Files

- **The System.Configuration namespace contains classes which can be used to read the .config file, including:**
  - The **ConfigurationManager**[1] 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.

[1] 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.



Key: Default User Value: BWW
Key: Connection String Value: Data Source=(local);Initial Catalog=pubs

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

```
private void btnRetrieve_Click(object sender,
EventArgs e)
{
    NameValueCollection parms;
    parms = ConfigurationManager.AppSettings;
    string msg = "";
    foreach (string key in parms.AllKeys)
    {
       msg += string.Format(
       "Key: {0} Value: {1}\n",key,parms.Get(key));
    }
    MessageBox.Show(msg, "Info");
}
```

Example: Using Config Files (Cont'd)

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

```
using System.Collections.Specialized;
using 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

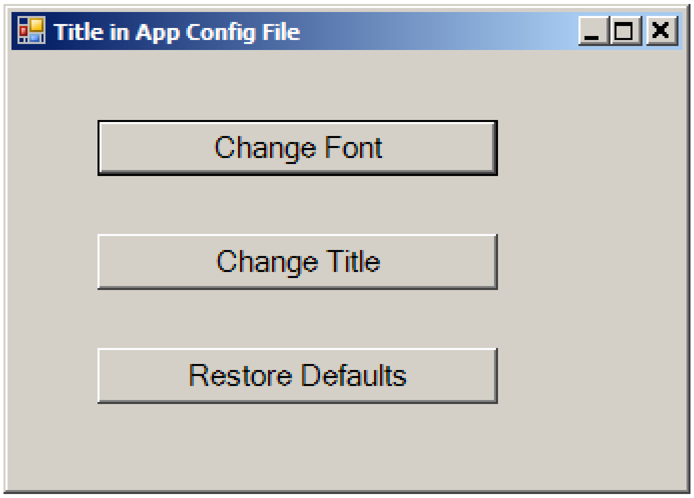
- **The .NET Framework has always provided the capability to store application settings as an XML fragment in a configuration file.**
  - Before .NET 2.0, configuration setting information pertained only to the application as a whole (not individual users) and was read-only.
- **Beginning in .NET 2.0 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.cs* 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**.

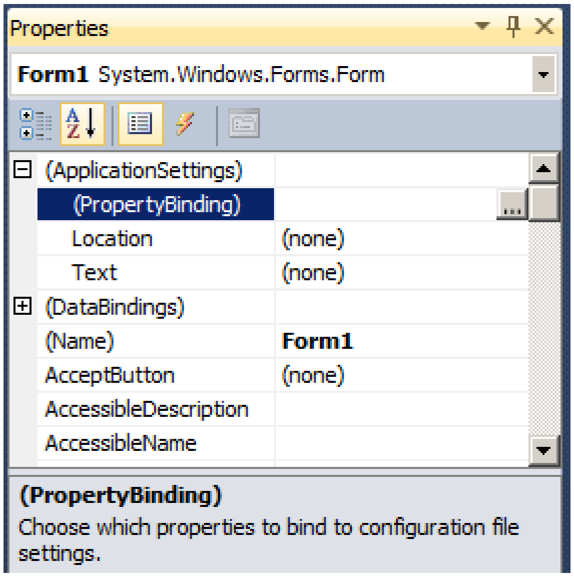
4. Add a handler for the Change Font button. Provide the following code.

```
private void btnFont_Click(object sender,
EventArgs e)
{
   if (fontDlg.ShowDialog() == DialogResult.OK)
   {
      Font f = fontDlg.Font;
      this.Font = f;
   }
}
```

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. Expand the (Application Settings) group and click on (PropertyBinding).

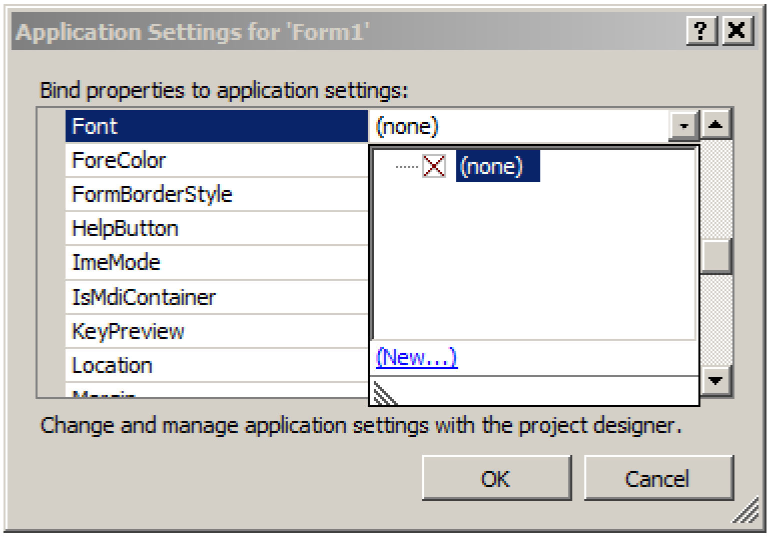


Form1 System.Windows.Forms.Form

⊟ (ApplicationSettings)	
(PropertyBinding)	...
Location	(none)
Text	(none)
⊞ (DataBindings)	
(Name)	**Form1**
AcceptButton	(none)
AccessibleDescription	
AccessibleName	

(PropertyBinding)
Choose which properties to bind to configuration file settings.

7. Click the Ellipsis button .... Select Font in the left-hand list, and click the drop-down arrow on the right side.



Bind properties to application settings:

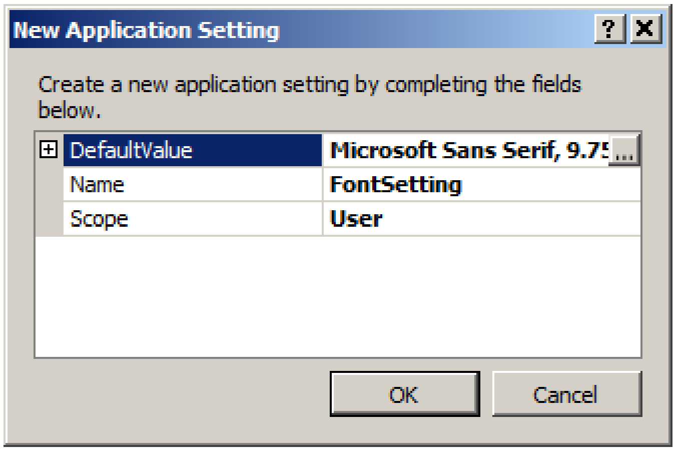
Font	(none)
ForeColor	☒ (none)
FormBorderStyle	
HelpButton	
ImeMode	
IsMdiContainer	
KeyPreview	
Location	(New...)
Margin	

Change and manage application settings with the project designer.

8. Click New ...

Application Settings Demo (Cont'd)

9. The New Application Setting dialog comes up. For Scope, leave it at User. For Name, type "FontSetting." For DefaultValue click on the Ellipsis button. The Font common dialog will come up. Leave the font name as "Microsoft Sans Serif" and select 10 as the font size (it will actually become 9.75).

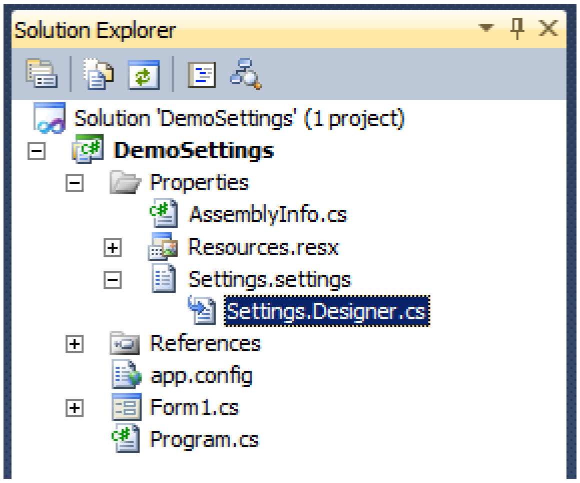


New Application Setting

Create a new application setting by completing the fields below.

DefaultValue	Microsoft Sans Serif, 9.75
Name	FontSetting
Scope	User

10. Click OK and OK. You should now see the new default font size reflected by the Form Designer. In Solution Explorer there will also be new files **Settings.Designer.cs** and **app.config**.



Solution Explorer

```
Solution 'DemoSettings' (1 project)
⊟ DemoSettings
    ⊟ Properties
        AssemblyInfo.cs
        ⊞ Resources.resx
        ⊟ Settings.settings
            Settings.Designer.cs
    ⊞ References
    app.config
    ⊞ Form1.cs
    Program.cs
```

Application Settings Demo (Cont'd)

11. Examine the **Settings.Designer.cs** file. There is a class **Settings** derived from **ApplicationSettingsBase**. It is in the namespace **DemoSettings.Properties**.

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

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

```
...
using DemoSettings.Properties;

namespace DemoSettings
{
   public partial class Form1 : Form
   {
      ...

      private void Form1_FormClosing(object sender,
      FormClosingEventArgs e)
      {
         Settings set = Settings.Default;
         set.Save();
      }

      private void Form1_Load(object sender,
      EventArgs e)
      {
         Settings set = Settings.Default;
         set.Reload();
      }
   }
}
```

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

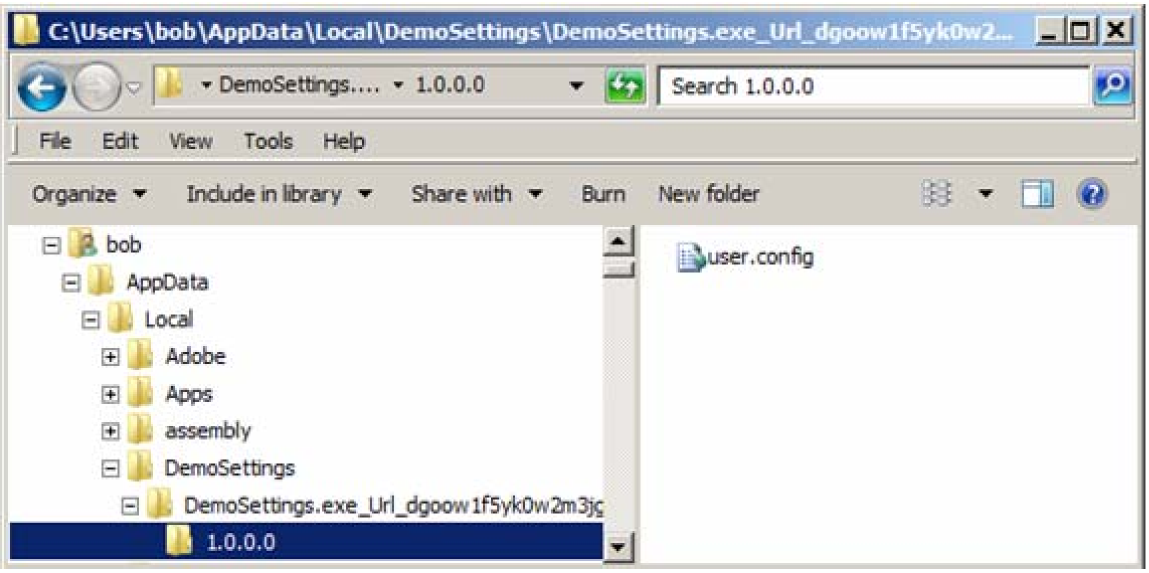
Application Configuration File

- **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.cs** to your project, defining a class **ManualSetting**, derived from **ApplicationSettingsBase**.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Configuration;

namespace DemoSettings
{
   class ManualSettings : ApplicationSettingsBase
   {
      [UserScopedSetting()]
      [DefaultSettingValue("Default Title")]
      public string TitleSetting
      {
         get
         {
            return (string) this["TitleSetting"];
         }
         set
         {
            this["TitleSetting"] = value;
         }
      }
   }
}
```

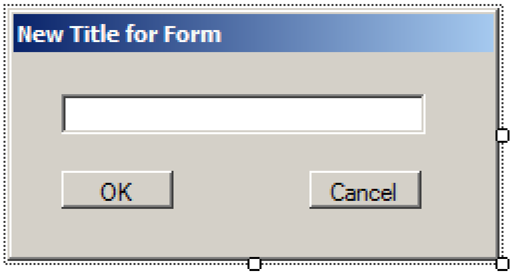
Manual Application Settings (Cont'd)

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.

```
TitleDialog dlg = new TitleDialog();
dlg.txtTitle.Text = this.Text;
if (dlg.ShowDialog() == DialogResult.OK)
{
   this.Text = dlg.txtTitle.Text;
   ManualSettings ms = new ManualSettings();
   ms.TitleSetting = this.Text;
   ms.Save();            // not bound, so save now
}
```

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.

```
Settings set = Settings.Default;
set.Reload();
ManualSettings ms = new ManualSettings();
ms.Reload();
this.Text = (string)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.

```
Settings set = Settings.Default;
set.Reset();
ManualSettings ms = new ManualSettings();
ms.Reset();
this.Text = (string)ms.TitleSetting;
ms.Save();                // not bound, so save now
```

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**.

```
RegistryKey reg = Registry.ClassesRoot;
```

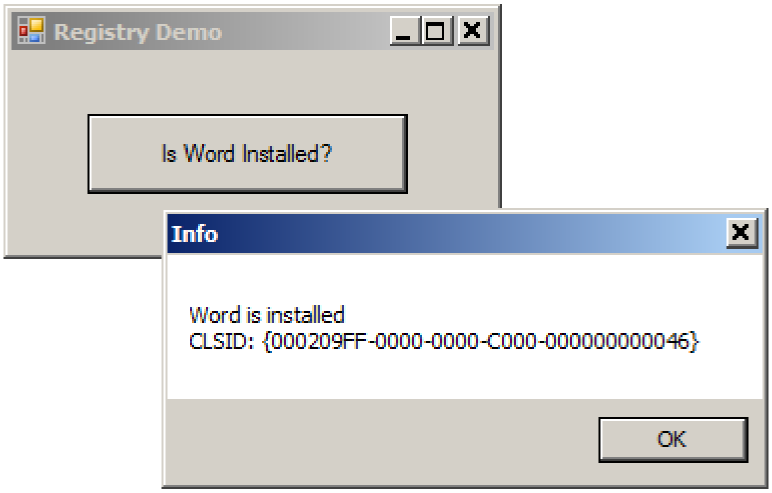
- **The RegistryKey class has three instance properties:**
  - The **Name** property is the name of the key.
  - The **ValueCount** property is the count of the values for the key.
  - The **SubKeyCount**, if greater than 0, is the number of subkeys for this key.

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.



Word is installed
CLSID: {000209FF-0000-0000-C000-000000000046}

- **The code that accomplishes this is shown below:**

```
private void btnWord_Click(object sender,
EventArgs e)
{
    // Get a reference to HKEY_CLASSES_ROOT
    RegistryKey hive = Registry.ClassesRoot;

    // Lookup the ProgID Word.Application
    RegistryKey wordProgID =
       hive.OpenSubKey("Word.Application");
```

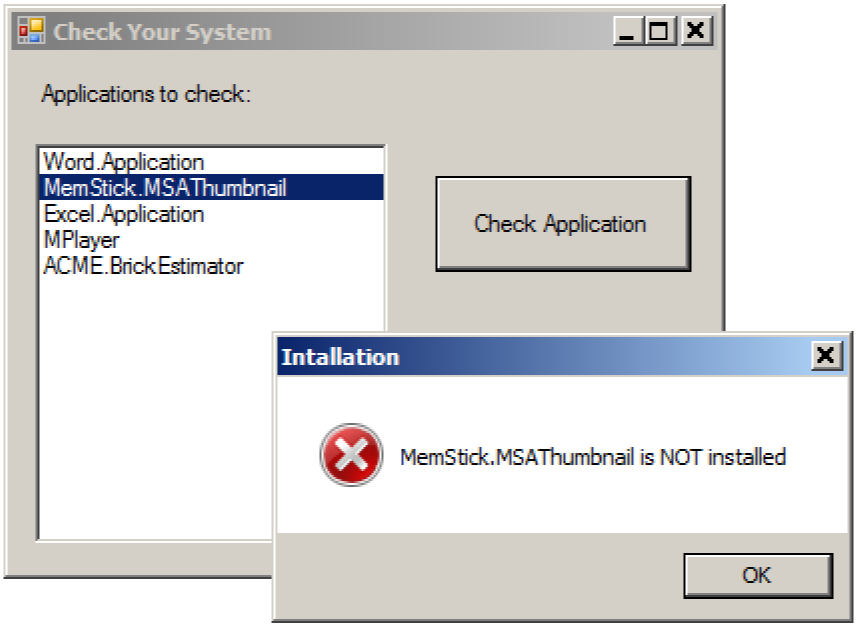
Example: Manipulating the Registry (Cont'd)

```
    if (wordProgID != null)
    {
       string msg = "Word is installed";
       ShowWordInformation(msg, wordProgID);
       wordProgID.Close();
    }
    else
    {
       MessageBox.Show("Word is not installed");
    }
}

private void ShowWordInformation(string msg,
RegistryKey progID)
{
    // Get a reference to HKEY_CLASSES_ROOT
    RegistryKey hive = Registry.ClassesRoot;

    // Lookup the value of CLSID
    RegistryKey clsid = progID.OpenSubKey("CLSID");
    string strCLSID = (string) clsid.GetValue("");
    clsid.Close();

    MessageBox.Show(msg + "\nCLSID: " + strCLSID);
}
```

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.



MemStick.MSAThumbnail is NOT installed

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

Suggested time: 45 minutes

Summary

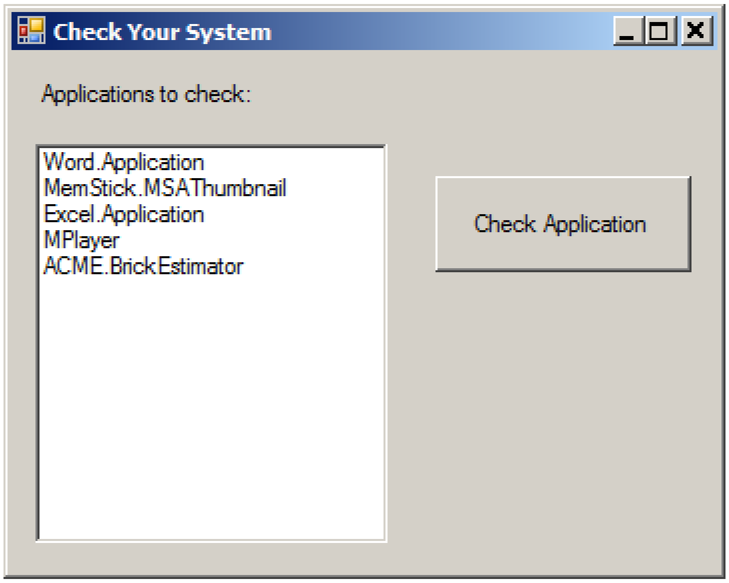
- **The Application class represents the class that manages a Windows Forms application.**
  - It contains methods to start and stop a Windows Forms application.
- **.NET applications use configuration files encoded using XML.**
  - The **ConfigurationManager** class provides access to the AppSettings 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.



Word.Application
MemStick.MSAThumbnail
Excel.Application
MPlayer
ACME.BrickEstimator

Suggested Time: 45 minutes

Root Directory: **OIC\WinCs**

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 void MainForm_Load(object sender, EventArgs e)
{
   NameValueCollection parms = ConfigurationManager.AppSettings;
   int count = Convert.ToInt32(parms.Get("Count"));
   string key = "";
   string value = "";
   for (int i = 1; i <= count; i++)
   {
      key = "App" + i.ToString();
      value = parms.Get(key);
      lstApplications.Items.Add(value);
   }
}
```

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 void btnCheckSelected_Click(object sender, EventArgs e)
{
   // Get a reference to HKEY_CLASSES_ROOT
   RegistryKey hive = Registry.ClassesRoot;

   // Get the ProgID from the listbox
   string progID = "";
   try
   {
      progID = lstApplications.SelectedItem.ToString();
      if (progID == "") return;
   }
   catch
   {
      return;
   }

   // Lookup the ProgID
   RegistryKey key = hive.OpenSubKey(progID);
   if (key != null)
   {
      string msg = progID + " is installed";
      MessageBox.Show(msg, "Intallation", MessageBoxButtons.OK,
         MessageBoxIcon.Information);
      key.Close();
   }
   else
   {
      string msg = progID + " is NOT installed";
```