msdn® training

Module 8: Creating Windows–based Applications

Contents

Microsoft®

Instructor Notes

Presentation:
105 minutes

Lab:
60 minutes

This module describes how to create menus, common and custom dialog boxes, status bars, and toolbars to enhance the usability of an application based on Microsoft® Windows®. The purpose of this module is to allow the students to apply their newly acquired C# language skills and develop useful Windows-based applications.

Important Each lesson of this module is preceded by an instructor-led demonstration, and followed by a Guided Practice.

Two advanced topics, Printing from an Application, and Implementing Drag and Drop, are available for your students' reference in Appendix B, "Advanced Topics."

After completing this module, students will be able to:

- Create the main menu.
- Create and use common dialog boxes.
- Create and use custom dialog boxes.
- Create and use toolbars.
- Create the status bar.
- Create and use combo boxes.

Required materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2609A_08.ppt
- Module 8, "Creating Windows-based Applications"

Preparation tasks

To prepare for this module:

- Read all of the materials for this module.
- Complete the practices and lab.
- Practice the instructor-led demonstrations.

How to Teach This Module

This section contains information that will help you to teach this module.

Lesson: Creating the Main Menu

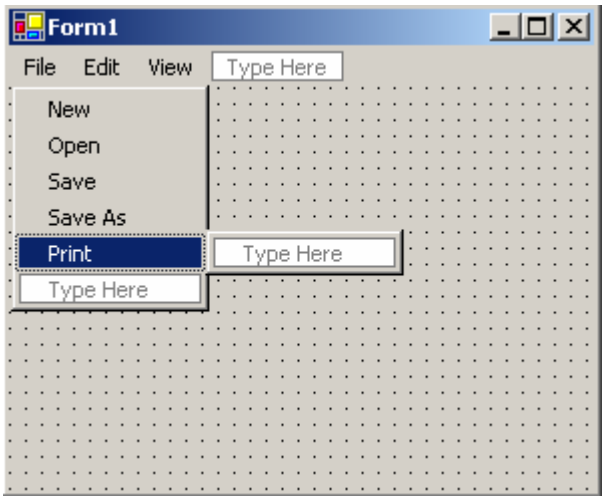
This section describes the instructional methods for teaching each topic in this lesson.

In this section, the students learn how to create the main window menu, add items to the menu, and associate methods with menu items. The following instructor-led demonstration will help you teach these skills effectively.

Demonstration

► **How to create the main menu**

1. Open Microsoft Visual Studio® .NET and create a new Windows Application project named **MyForm**.
2. Add a **MainMenu** control to the form.
3. Add some menu items, for example **File**, **Edit**, and **View**.



4. Add some submenus, for example, **New**, **Open**, **Save**, **Save As**, and **Print**.
5. Add an event handler to a menu item.

Lesson: Creating and Using Common Dialog Boxes

In this lesson, the students examine some of the most common dialog boxes that are provided in the **Forms** namespace and learn how to use them in a Windows-based application. The following instructor-led demonstration will help you teach these skills effectively.

► How to create and use a common dialog box

1. Open Visual Studio .NET and create a new Windows Application project named **MyForm**.
2. Create an instance of any one of the common dialog boxes, for example, **OpenFileDialog**, **SaveFileDialog**, **FontDialog**, or **PrintDialog**.
3. Set properties of the common dialog box. The following table shows properties of an **OpenFileDialog** dialog box.

Property	Description
Filter	The file filters to display in the dialog box, for example, C# files (*.cs); all files (*.*)
Multiselect	Controls whether multiple files can be selected in the dialog box
ShowHelp	Enables the **Help** button

4. Add an event handler to open the dialog box.
5. Use the **ShowDialog( )** method to display the dialog box.
6. Test the application.

Lesson: Creating and Using Custom Dialog Boxes

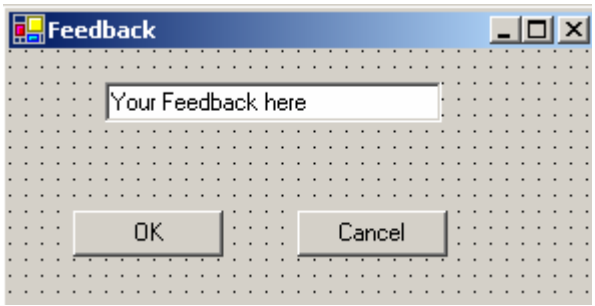
This lesson explains how to work with tabbed dialog boxes by using the development environment. The following instructor-led demonstration will help you teach these skills effectively.

► How to create and use a custom dialog box

1. Open Visual Studio .NET and create a new Windows Application project named **MyForm**.
2. Add a dialog box to the project by right-clicking the project in Solution Explorer.
3. Set the dialog box properties as shown in the following table.

Property	Setting
FormBorderStyle	FixedDialog
ControlBox	False
MinimizeBox	False
MaximizeBox	False
ShowInTaskbar	False

4. Provide a way for users to dismiss the dialog box by adding two buttons to the form. Change the **Text** property of one button to **OK** and the other button to **Cancel**, as shown in the following illustration:



5. Set the **DialogResult** property of the **OK** and **Cancel** buttons to **OK** and **Cancel** respectively.
6. Instantiate and display the dialog box from the event handler of any menu item, using the **ShowDialog** method.

► How to create a custom tabbed dialog box

1. Open Visual Studio .NET, create a new Windows Application project named **MyForm**.
2. Add a dialog box to the project by right-clicking the project in Solution Explorer.
3. Add a **TabControl** to the dialog box.
4. Add or remove tabs by using the TabPage Collection Editor.

Lesson: Creating and Using Toolbars

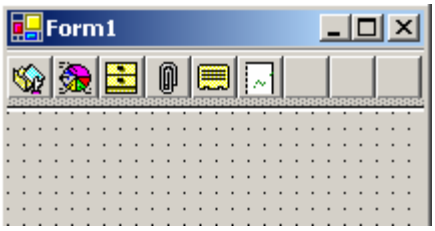
The toolbar is a standard feature in many Windows-based applications. In this lesson, the students learn how to display a row of buttons and drop-down menus that activate commands, and how to write the code for the **Button-Click** event. The following instructor-led demonstration will help you teach these skills effectively.

► How to create a toolbar

1. Open Visual Studio .NET and create a new Windows Application project named **MyForm**.
2. Add a **Toolbar** control to the form.
3. Set toolbar properties.

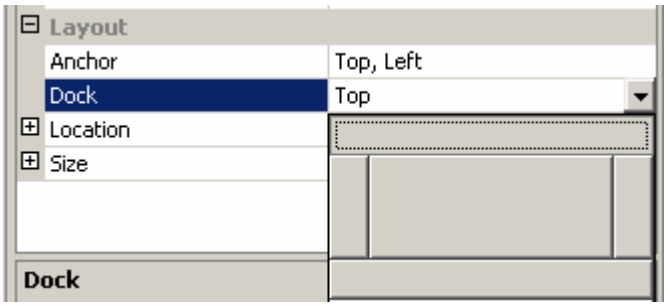
Property	Description
Appearance	Sets the appearance of the **ToolBar** control: **Normal** for a three-dimensional and raised view. **Flat** for a flat button that rises to a three-dimensional view.
Button size	Suggests the size of buttons of the tool bar.
Buttons	**(Collection)** editor allows adding and removing tool bar buttons.
Cursor	The cursor that appears when the mouse passes over the control.
ImageList	The imageList from which this tool bar will get all of the button images.
Dock	Determines the docking location of the tool bar.

4. Add icons to the toolbar.



Tip Visual Studio .NET provides a library of icons in the Program_Files\Visual Studio.NET\Common7\Graphics\icons\ folder.

5. Set docking options for a toolbar.



Lesson: Creating the Status Bar

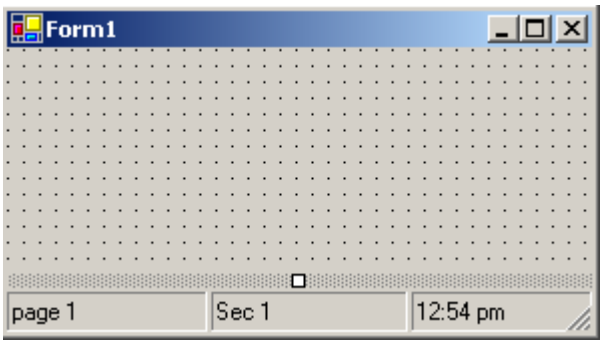
In this lesson, the students learn how to add a status bar to a form and customize it to provide useful information, such as the name of a file that is currently open or the current date or time. Students also learn how to add panels to a status bar. The following instructor-led demonstration will help you teach these skills effectively.

► How to create a status bar

1. Open Visual Studio .NET and create a new Windows Application project named **MyForm**.
2. Add a **StatusBar** control to the form.
3. Set the status bar properties.

Properties	Description
ShowPanel	Determines whether a status bar displays panels or a single line of text
Panels	**(Collection) Editor** allows adding and removing panels to the status bar

4. Add panels to a status bar at design time.

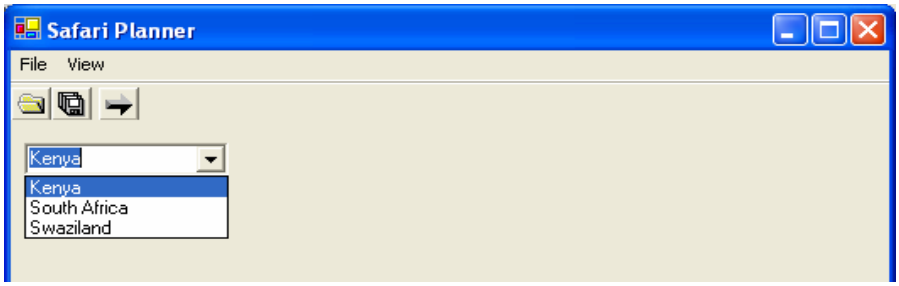


Lesson: Creating and Using Combo Boxes

The Windows Forms **ComboBox** control is used to display data in a drop-down combo box. In this lesson, the students learn how to create a combo box, and how to associate objects with it. The following instructor-led demonstration will help you teach these skills effectively.

► **How to create and use a combo box**

1. Open Visual Studio .NET and create a new Windows Application project named **MyForm**.
2. Add a **ComboBox** control to the form.



3. Add strings to the **items** collection by using the **Add** and **AddRange** methods.

Review

The review questions are based mostly on conceptual understanding and procedures that were covered thoroughly in the module. You can use a discussion format to answer the questions so that everyone gets the benefit of knowing the right answers.

Lab 8.1: Building Windows Applications

Before beginning this lab, students should have completed all of the practices and answered the review questions. Students must be able to perform most of the tasks that they learned in the lessons and the practices. The lab is simple but comprehensive. It leads students through the entire process of creating a Windows-based application as described in the lessons of this module.

Overview

- Creating the Main Menu
- Creating and Using Common Dialog Boxes
- Creating and Using Custom Dialog Boxes
- Creating and Using Toolbars
- Creating the Status Bar
- Creating and Using Combo Boxes

Introduction

Menus, dialog boxes, status bars, and toolbars are tools that enable you to expose functionality to your users or alert them to important information in your application. Menus contain commands that are grouped by a common theme. You can use dialog boxes to interact with the user and retrieve user input. Status bars indicate application state or provide information about the entity in the application that has focus, such as a menu command. Toolbars provide buttons that make frequently used commands available.

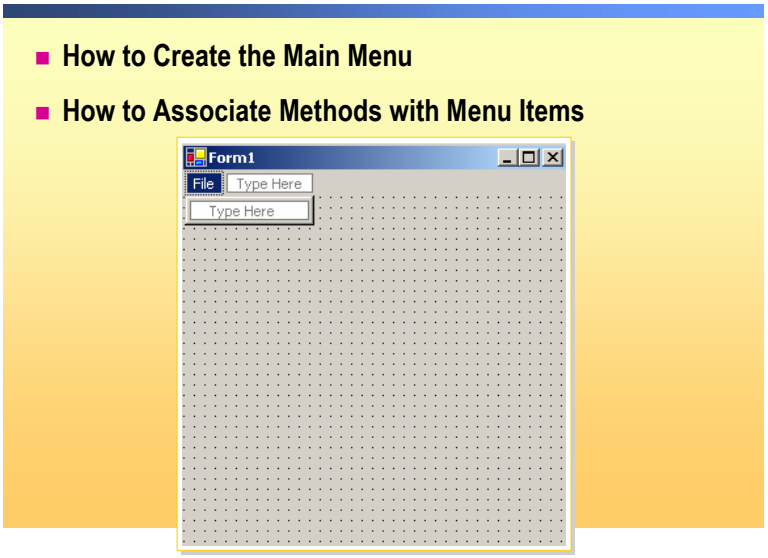
This module describes how to create menus, common and custom dialog boxes, status bars, and toolbars to enhance the usability of your application.

Objectives

After completing this module, you will be able to:

- Create the main menu.
- Create and use common dialog boxes.
- Create and use custom dialog boxes.
- Create and use toolbars.
- Create the status bar.
- Create and use combo boxes.

Lesson: Creating the Main Menu



Introduction

This lesson explains how to create the main window menu, add items to the menu, and associate methods with menu items.

Lesson objectives

After completing this lesson, you will be able to:

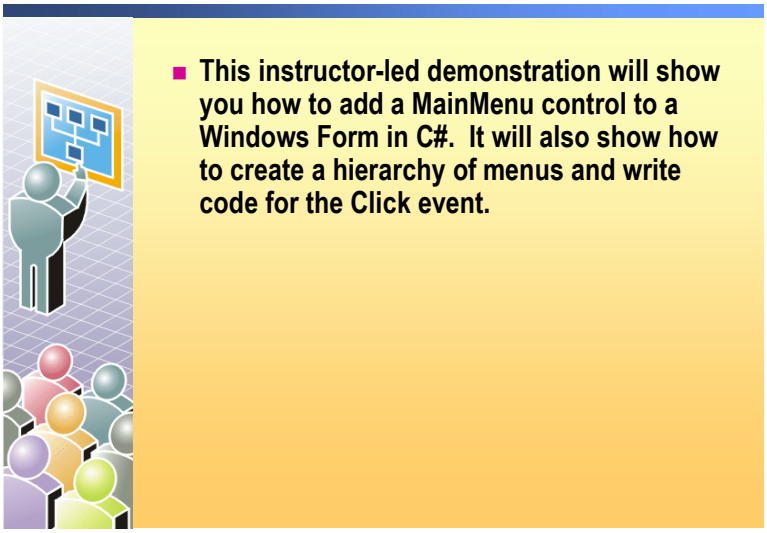
- Create the main menu.
- Add items to the menu.
- Associate methods with menu items.

Lesson agenda

This lesson includes the following topics and activities:

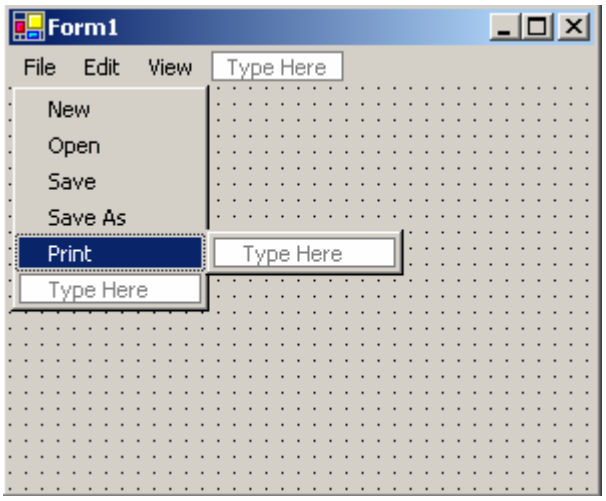
- Demonstration: Creating the Main Menu
- How to Create the Main Menu
- How to Associate Methods with Menu Items
- Practice: Creating the Main Menu

Demonstration: Creating the Main Menu



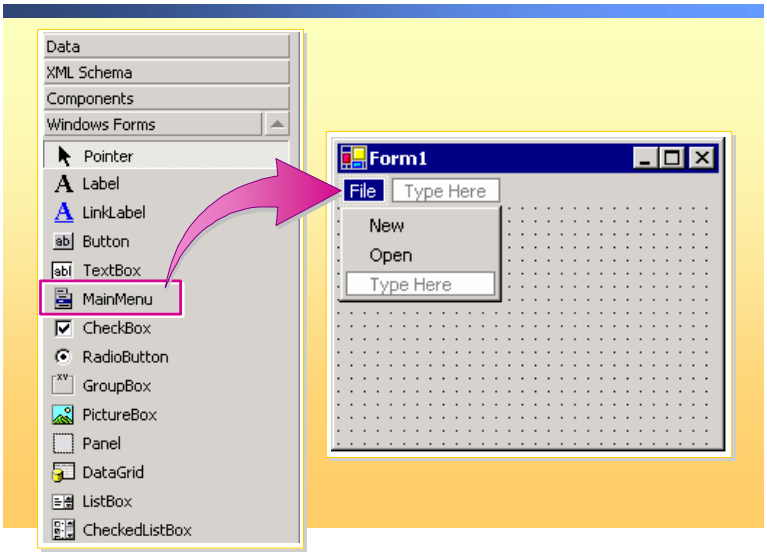
This instructor-led demonstration will show you how to add a **MainMenu** control to a Microsoft® Windows® Form in C#. It will also show how to create a hierarchy of menus and write code for the **Click** event. The instructor will:

1. Open Microsoft Visual Studio® .NET and create a new Windows Application project named **MyForm**.
2. Add a **MainMenu** control to the form.
3. Add some menu items, for example **File**, **Edit**, and **View**.



4. Add some submenus, for example, **New**, **Open**, **Save**, **Save As**, and **Print**.
5. Add an event handler to a menu item.

How to Create the Main Menu



Introduction

The purpose of a menu is to make using your application easy and intuitive for the user. You can create a menu in Windows Forms by adding a **MainMenu** object from the Toolbox to the form at design time. You can then create the menu items and submenus for your menu by using the Menu Designer.

Creating a menu, menu items, and submenus

To add a menu and menu items to a form:

1. In the Windows Forms Designer, open the form to which you want to add a menu.
2. In the Toolbox, double-click the **MainMenu** control.

   A menu is added to the form. The menu displays the text "Type Here."
3. Click the text **Type Here**, and then type the name of the desired menu item to add it to the menu.
4. To provide access to the built-in keyboard shortcut feature of Microsoft Windows, type an ampersand (&) in front of the letter that will be the shortcut. These shortcuts are indicated by an underlined letter, and can be activated by pressing ALT plus the underlined letter.

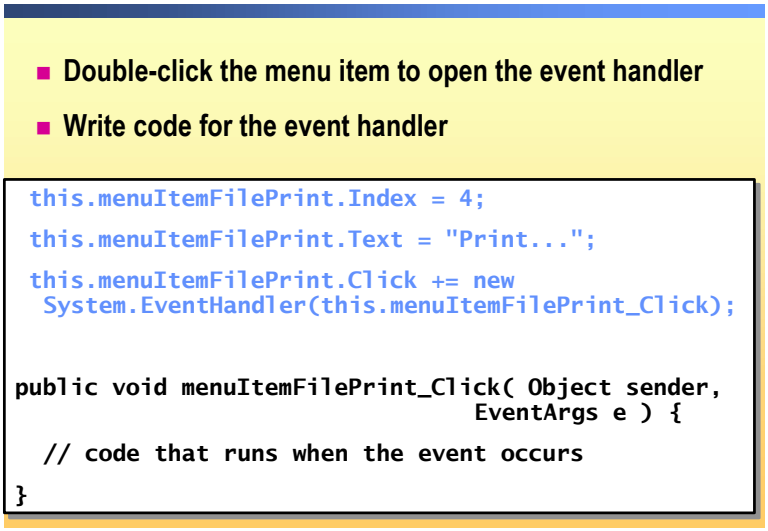
   For example, enter the string **&File** to place **File** on the menu, or **E&xit** for **E**x**it**. If you need to place an ampersand in a menu, type **&&**
5. The name you type is listed as the **Text** property of the menu item.

Note By default, the **MainMenu** object contains no menu items, so the first menu item that you add to the menu becomes the menu heading.

---

6. To add another menu item, click another "Type Here" area in the Menu Designer.
   a. To add a submenu, click the area to the right of the current menu item. You cannot add a submenu to a menu heading.
   b. To add another item to the same menu, click the area below the current menu item.
7. Change the **Name** property of the menu item to something meaningful. For example, if you added a menu item labeled **Open** to the **File** menu, name the item **fileOpen**.
8. You can insert a horizontal line into a menu by setting the text property to the minus symbol (–).

How to Associate Methods with Menu Items



Introduction

After you create the menu, you can add functionality to the menu by associating methods with the menu items.

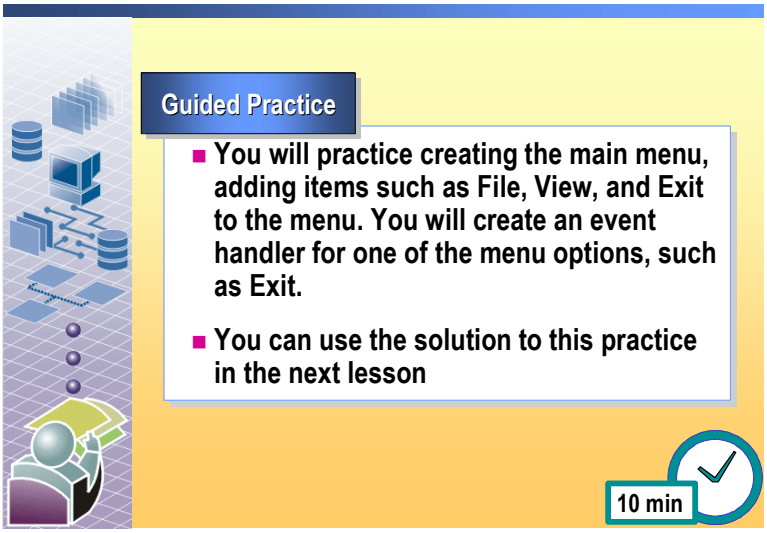
Associating methods with menu items

You can add functionality to menu items by associating methods with the **Click** event of the menu item. The **Click** event occurs when the user clicks the menu item, when the user selects the item by using the keyboard and presses ENTER, or when the user clicks an access key or shortcut key that is associated with the menu item.

To associate methods, and therefore functionality, with a menu item:

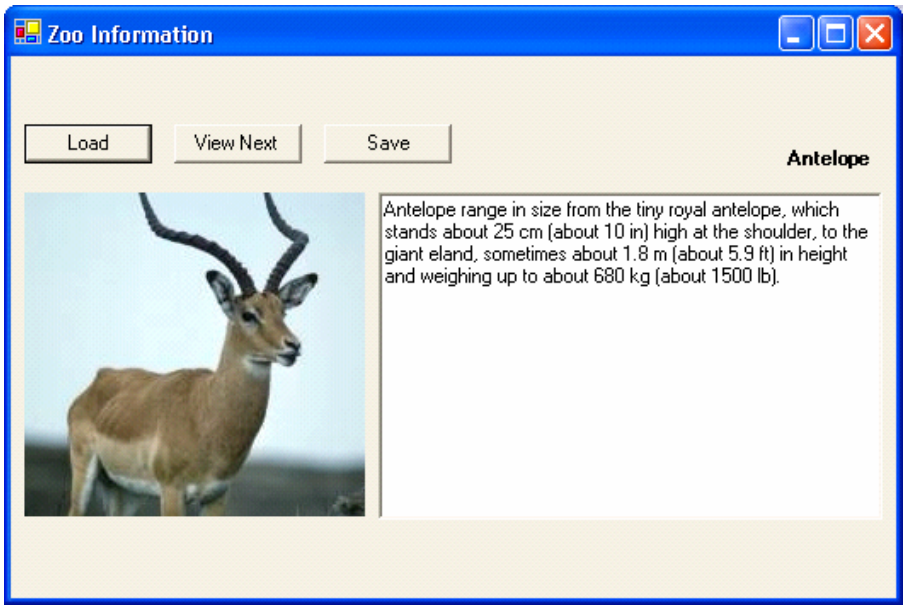
1. In the Menu Designer, click the menu item you want to add functionality to.
2. In the Properties window, rename the **Name** property of the item by using the naming convention that you will use for all items on the menu. For example, name the **Open** menu item **OpenItem**.
3. Double-click the menu item to open an event handler for its **Click** event.
4. Write the code for the event handler.

Practice: Creating the Main Menu



In this practice, you will open an application, add a main menu to it, and then implement menu functions.

The existing application reads animal information from an XML file and displays it in a simple Windows-based application. Currently, the application appears as follows:



The solution for this practice is located in *install_folder*\Practices\Mod08\MainMenu_Solution. Start a new instance of Visual Studio .NET before opening the solution.

Your task is to use menu functions to replace the buttons.

Note that the solution contains the following three C# files:

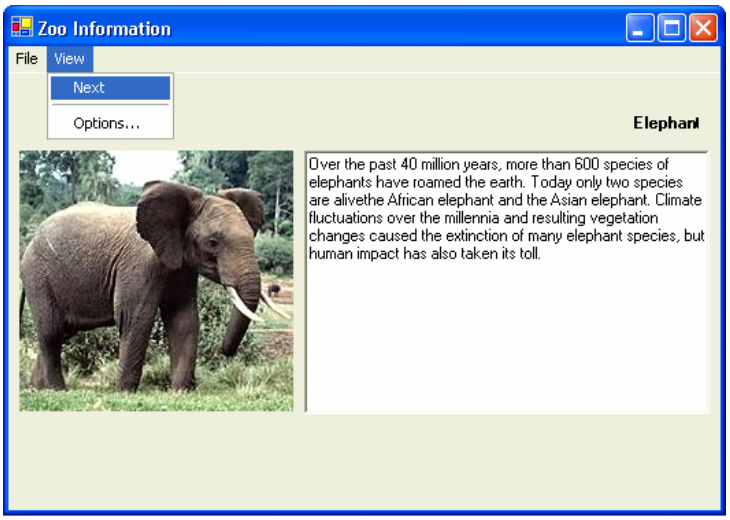
- Animals.cs—contains the definitions for the animal types. You will not need to change anything in this file.
- Form1.cs—the Windows interface code. You will write your code in this file.
- Zoo.cs—the class that contains the collection of animals. You will not need to change anything in this file.

The application expects to find the data files in the solution folder.

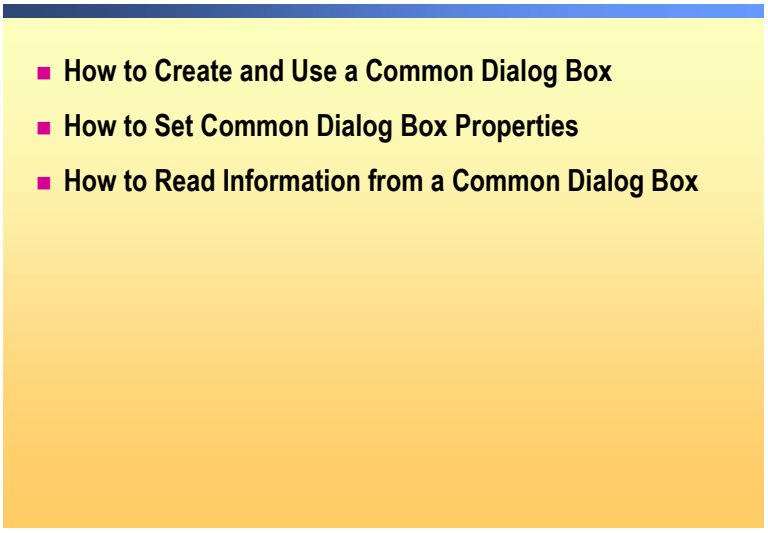
The solution for this practice is in *install_folder*\Practices\Mod08\MainMenu_Solution \Animals.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder*\Practices \Mod08\MainMenu\Animals.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder* \Practices\Mod08\MainMenu, click **Animals.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. Build and run the solution, and familiarize yourself with it.	**a.** In Visual Studio .NET, press F5. **b.** In the Zoo Information window, click **Load**, and then click **View Next**. **c.** Close the Zoo Information window.
3. Add a **MainMenu** control to the Form.	▪ For detailed information on how to complete this task, see How to Create the Main Menu in this lesson.
4. Add the following menu items, and create event handlers for them as specified in the following tables:	▪ For detailed information on how to complete this task, see How to Create the Main Menu and How to Associate Methods with Menu Items in this lesson.

Menu	Action
File	None
Open	Same as **Load** button: call the **Zoo.Load** method and Initialize the form
Save	Same as **Save** button: call the **Save** method in the **Zoo** class
-	Insert a horizontal line
Exit	(Optional: use **Application.Exit()** to exit the application.)
View	None
Next	Same as View Next button: display the next animal
-	Insert a horizontal line
Options	Do nothing in the event handler

Tasks	Detailed steps
5. Build and test your solution.	**a.** On the **Build** menu, click **Build Solution**. **b.** If necessary, use breakpoints and the debugging tool to check your application.
6. Delete the **Load**, **View Next**, and **Save** buttons, and their event handlers, and test your application again.	▪ After you load the data, your application should appear as follows: 
7. Save your solution.	▪ On the **File** menu, click **Save All**.

Lesson: Creating and Using Common Dialog Boxes



Introduction

This lesson examines some of the most common dialog boxes that are provided in the **Forms** namespace and explains how to use them in a Windows-based application.

Lesson objectives

After completing this lesson, you will be able to:

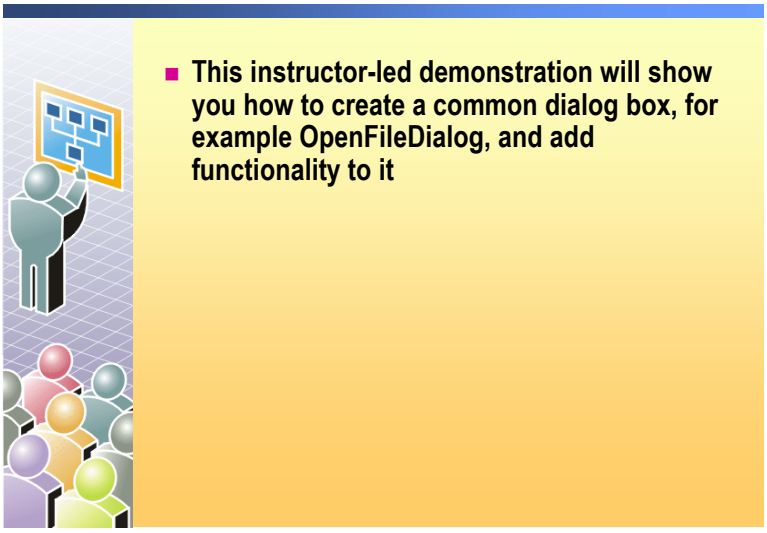
- Use the **OpenFileDialog** control.
- Use the **SaveFileDialog** control.
- Use the **PrintDialog** control.
- Use the **FontDialog** control.

Lesson agenda

This lesson includes the following topics and activities:

- Demonstration: Creating and Using a Common Dialog Box
- How to Create and Use a Common Dialog Box
- How to Set Common Dialog Box Properties
- How to Read Information from a Common Dialog Box
- Practice: Using a Common Dialog Box

Demonstration: Creating and Using a Common Dialog Box



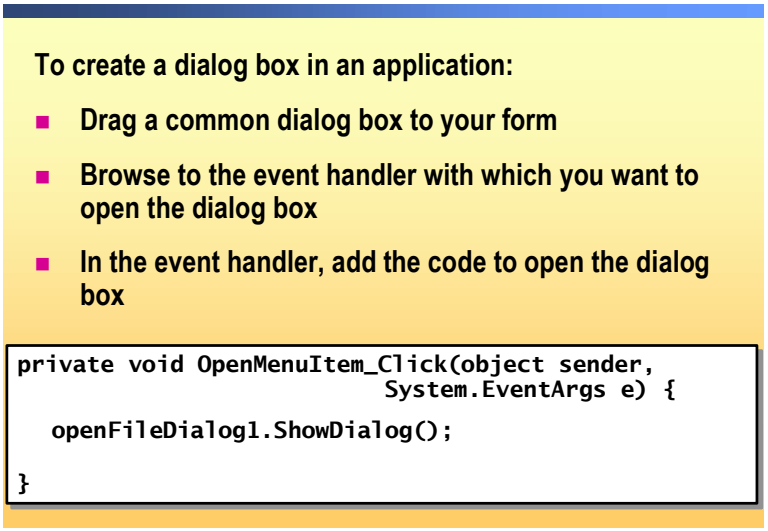
This instructor-led demonstration will show you how to create a common dialog box, for instance **OpenFileDialog**, and add functionality to it. The instructor will:

1. Open Visual Studio .NET and create a new Windows Application project named **MyForm**.
2. Create an instance of any one of the common dialog boxes, such as **OpenFileDialog**, **SaveFileDialog**, **FontDialog**, or **PrintDialog**.
3. Set the properties of the common dialog box. The following table shows properties of an **OpenFileDialog** dialog box.

Property	Description
Filter	The file filters to display in the dialog box, for example, C# files (*.cs); all files (*.*)
Multiselect	Controls whether multiple files can be selected in the dialog box
ShowHelp	Enables the **Help** button.

4. Add an event handler to open the dialog box.
5. Use the **ShowDialog( )** method to display the dialog box.
6. Test the application.

How to Create and Use a Common Dialog Box



Introduction

In this topic, you will learn how to add a common dialog box to a form, how to configure it, and how to write code to display it.

The Microsoft .NET Framework provides six classes that provide the common user interface functions of opening files, saving files, selecting fonts, setting page printing values, printing, and selecting a color. These classes, **OpenFileDialog**, **SaveFileDialog**, **FontDialog**, **PageSetupDialog**, **PrintDialog**, and **ColorDialog**, are implemented as dialog boxes.

Creating and displaying a dialog box

To add a common dialog box to an application, you select it from the Toolbox and drag it onto your form. To display a dialog box, you must create the object and then invoke the **ShowDialog** method on the object.

It is normal to display the dialog box in response to an event, such as a menu selection or a button click from the user of the application.

To display a dialog box:

1. Create an instance of the common dialog box.
2. Browse to the event handler within which you want to open the dialog box.
3. Use the **ShowDialog()** method to show the dialog box.

Example

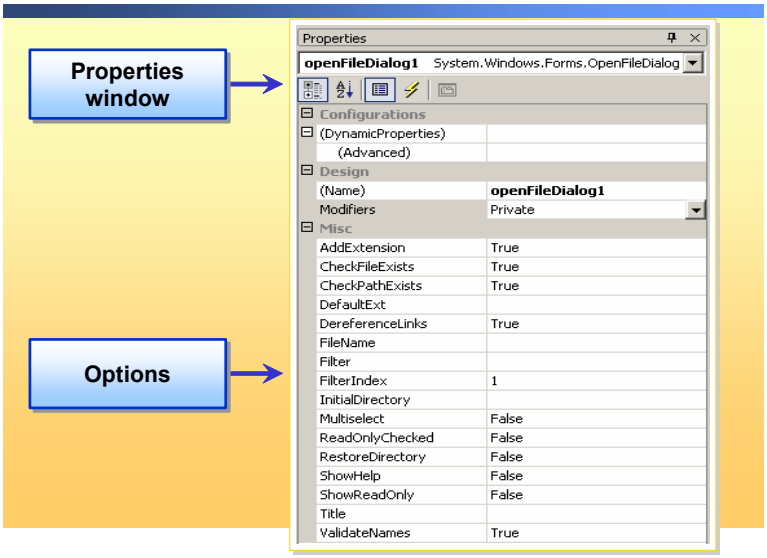
For example, Visual Studio .NET inserts the following line of code when you add an **OpenFileDialog** control to your project:

```
private System.Windows.Forms.OpenFileDialog openFileDialog1;
```

OpenMenuItem_Click, an event handler that is called when the user clicks **Open** on the **File** menu, is shown in the following example. This method displays the common Windows **Open File** dialog box.

```
private void OpenMenuItem_Click( object sender,
                                 System.EventArgs e) {
  openFileDialog1.ShowDialog();
}
```

How to Set Common Dialog Box Properties



Introduction

Visual Studio .NET includes a set of preconfigured dialog boxes, which you can adapt for your own applications. By relying on standard Windows dialog boxes, you create applications whose basic functionality is immediately familiar to users.

Common dialog boxes

The following table shows some of the most common dialog boxes, their functions, and some of their associated properties.

Options	Description	Properties
OpenFileDialog	Allows users to open files by using a preconfigured dialog box.	**Multiselect:** You can enable users to multi-select files to be opened with the **Multiselect** property **Filter** property sets the current file name filter string, which determines the options that appear in the **Files of type** box in the dialog box.
SaveFileDialog	Selects files to save and the location where they are saved.	**FileName:** The file first shown in the dialog box, or the last one selected by the user **Filter:** The file filters to display in the dialog box
FontDialog	Exposes the fonts that are currently installed on the system.	**Font:** The font selected by the user **MaxSize:** The maximum size that can be selected (or zero to disable) **MinSize:** The minimum size that can be selected (or zero to disable)
PrintDialog	Displays a document as it would appear when it is printed.	**AllowPrintToFile:** Enables and disables the **Print To File** check box

Example

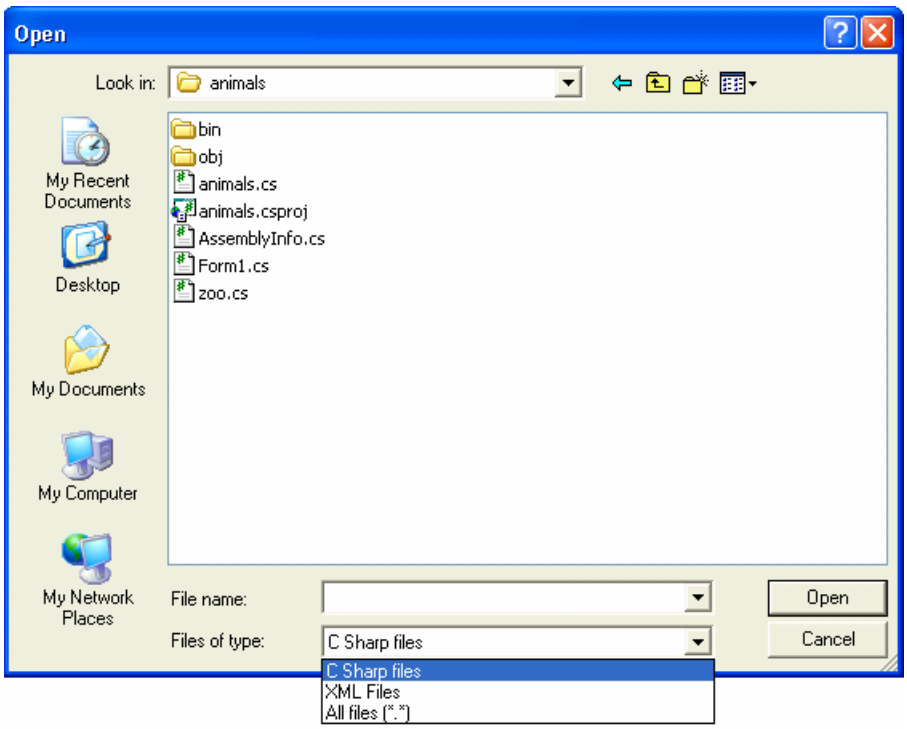
The **OpenFile** and **SaveFile** dialog boxes show all types of files unless the **Filter** property is set. To set this property, you pass a properly formatted string to the property. The string consists of pairs of labels and filters, separated by the vertical bar (**|**) character. The first pair is the default value.

For example, a graphics program may allow the user to browse for TIF files and JPEG files, expecting file extensions of .tif and .jpg. The Filter string could be "TIF files (*.tif) | *.tif | JPEG files (*.jpg) |*.jpg".

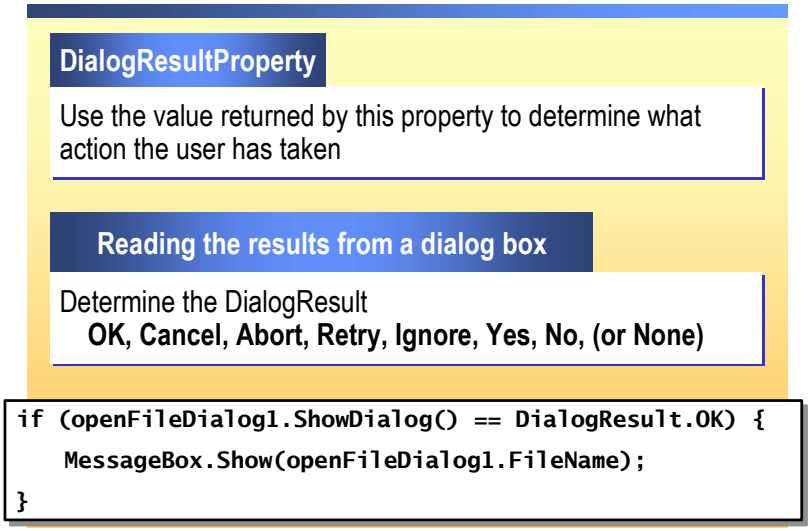
If you want to use more than one file extension filter with a label, separate the filters with a semicolon as shown in the following example:

```
openFileDialog1.Filter = @"C Sharp files|*.cs;*.csproj|XML
Files|*.xml|All files (*.*)|*.*";
```

When this dialog box is opened, it appears as follows:



How to Read Information from a Common Dialog Box



Introduction

When you display a dialog box in an application, it is very important to know what action the user took. For example, if you display a dialog box that prompts users to dismiss the dialog box, you must know whether the user clicked the **OK** button or the **Cancel** button.

Retrieving a result from a dialog box

When the user closes a dialog box, the **ShowDialog** method returns a **DialogResult** value to the calling method. The object that displayed the dialog box can check the **DialogResult** to determine if the user clicked **OK**, **Cancel**, or some other value.

For example:

```
if ( openFileDialog1.ShowDialog() == DialogResult.OK ) {
  MessageBox.Show( "You selected " +
                              openFileDialog1.FileName );
}
```

Possible **DialogResult** values are **OK**, **Cancel**, **Abort**, **Retry**, **Ignore**, **Yes**, and **No** (or **None**).

Reading the user information

The dialog box object maintains information about user selections and makes that information available through properties. For example, you can use the **Font** property of the **FontDialog** object to determine the font that the user selected. In the preceding examples, the **FileName** property of the **OpenFileDialog** object is used to determine the name of the file that the user selected.

Reading user information from a dialog box

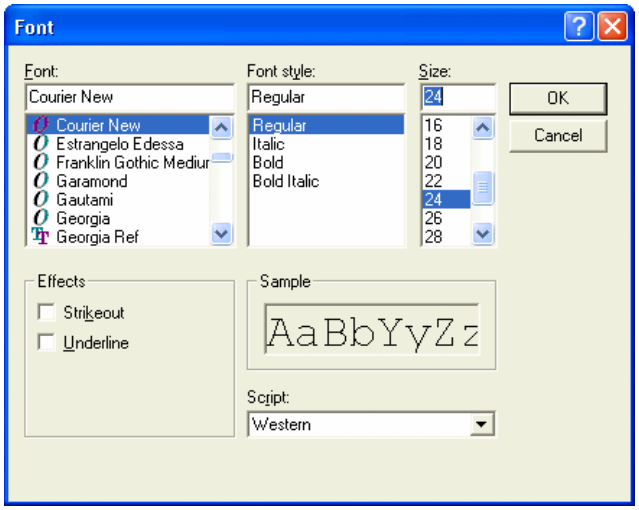
To read user information from a dialog box:

1. Browse to the event handler or the method for which you want to set the **DialogResult** property.
2. Add code to retrieve the **DialogResult** value.

The following examples illustrate how user input is derived from a **FontDialog** object:

```
fontDialog1 = new FontDialog();
if ( fontDialog1.ShowDialog() == DialogResult.OK ) {
  string fontInfo =  fontDialog1.Font.Name + " -- ems:"
                     + fontDialog1.Font.Size.ToString();
  MessageBox.Show("You selected: " + fontInfo );
}
```

The preceding code shows how to read the name of the selected font and the size of the font in ems. This code produces the following output:



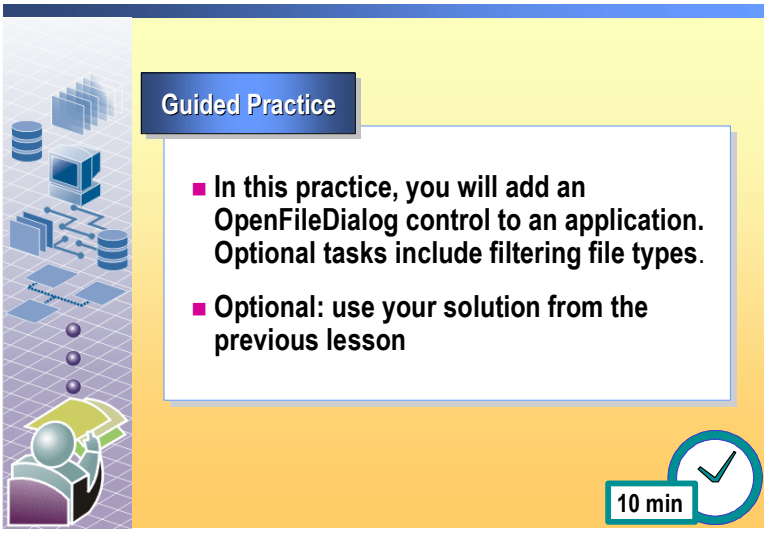
Setting the owner of the dialog box

The **Form.ShowDialog** method has an optional argument, *owner*, that you can use to specify a parent-child relationship for a form. Usually, you want the owner of the dialog box to be the object that created and uses the dialog box. When you specify that a dialog box is a child of your form, that dialog box always appears in front of the form, which is the correct Windows behavior for dialog boxes.

In your form code, you can use the **this** keyword to specify the calling object as the owner of the dialog box, as shown in the following code:

```
public class Form1 : System.Windows.Forms.Form {
  // . . .
  openFile_Click( object sender, EventArgs e ) {
      OpenfileDialog openFile = new OpenFileDialog();
      if ( openFile.ShowDialog( this ) == DialogResult.OK ) {
          // open the file
      }
   }
}
```

Practice: Using a Common Dialog Box



In this practice, you will add an **OpenFileDialog** control to the Zoo Information application.

If you have a working solution from the Creating the Main Menu lesson in this module and you want to build upon that, open that solution and skip steps 1 and 2 in this practice.

The solution for this practice is located in *install_folder*\Practices\Mod08\CommonDialog _Solution\Animals.sln. Start a new instance of Visual Studio .NET before opening the solution.

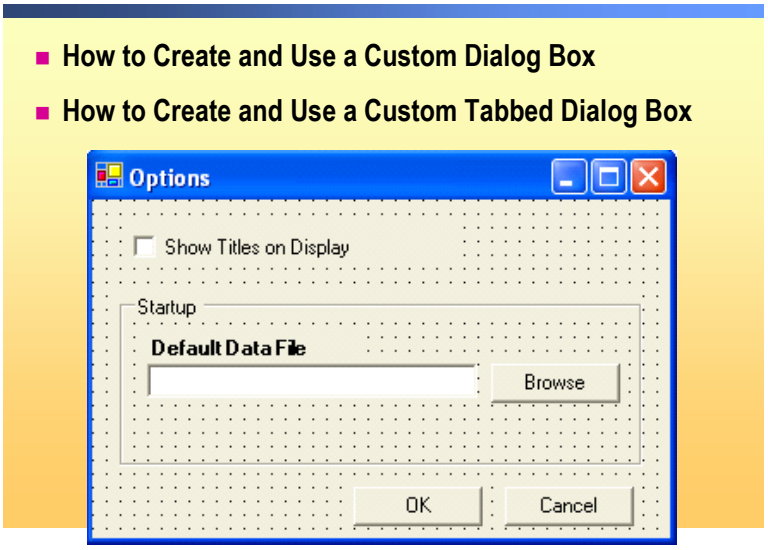
Tasks	Detailed steps
1. Start Visual Studio .NET, and then open *install_folder*\Practices\Mod08\CommonDialog\Animals.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Practices\Mod08\CommonDialog, click **Animals.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. *(Optional)* Build and run the solution, and familiarize yourself with it.	**a.** In Visual Studio .NET, press F5. **b.** Examine the Zoo Information application. **c.** Close the Zoo Information window.
3. Add an **OpenFileDialog** control to your project.	▪ Drag an **OpenFileDialog** control from the Toolbox onto your Form. By default, this is called **openFileDialog1**.

Tasks	Detailed steps
4. In the event handler for the **loadItem** menu item, add code so that the **OpenFileDialog** object shows XML files by default.	**a.** In Design view, on the **File** menu, double-click **Open**. **b.** In the **loadItem_Click** method, use the **Filter** property of **openFileDialog** to show XML files and all files: `openFileDialog1.Filter = @"XML Files (*.xml)|*.xml|All files (*.*)|*.*";`

5. Show the **OpenFileDialog** object, retrieve the filename selected by the user, and then load the file.	Note that the provided **LoadZoo** method loads the file that is specified in the **zooFile** string.
	a. Use the **ShowDialog** method to display **openFileDialog1**.
	b. If **ShowDialog** returns **DialogResult.OK**, then execute the following code:
	`zooFile = openFileDialog1.FileName;`
	`this.LoadZoo();`
	`InitializeDisplay();`
	c. An alternative solution to this task is to call the **Zoo.Load** method directly. If you do this, remember to handle any exceptions that may be thrown.

6. Test your application by loading the XML data file **AnimalData.xml**.	**a.** On the **Build** menu, click **Build Solution**. **b.** If necessary, use breakpoints and the debugger to check your application.
7. Save your solution.	▪ On the **File** menu, click **Save All**.

Lesson: Creating and Using Custom Dialog Boxes



Introduction

This lesson explains how to work with tabbed dialog boxes by using the development environment.

Lesson objectives

After completing this lesson, you will be able to:

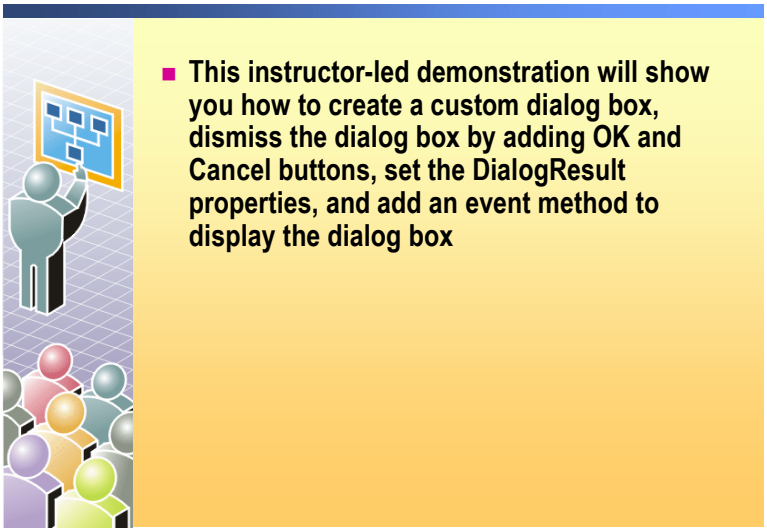
- Create and use a dialog box.
- Create and use a tabbed dialog box.

Lesson agenda

This lesson includes the following topics and activities:

- Demonstration: Creating and Using a Custom Dialog Box
- How to Create and Use a Custom Dialog Box
- Demonstration: Creating a Custom Tabbed Dialog Box
- How to Create and Use a Custom Tabbed Dialog Box
- Practice: Creating a Custom Dialog Box

Demonstration: Creating and Using a Custom Dialog Box

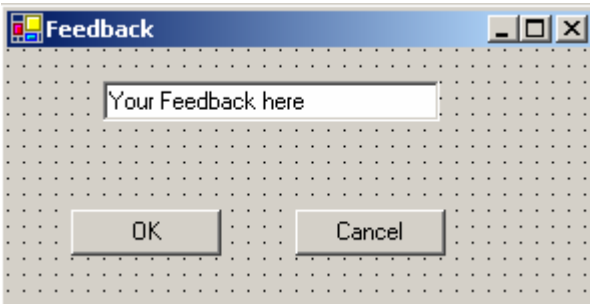


This instructor-led demonstration will show you how to create a custom dialog box, dismiss the dialog box by adding **OK** and **Cancel** buttons, set the **DialogResult** properties, and add an event method to display the dialog box. The instructor will:

1. Open Visual Studio .NET and create a new Windows Application project named **MyForm**.
2. Add a dialog box to the project by right-clicking the project in Solution Explorer.
3. Set the dialog box properties, as shown in the following table:

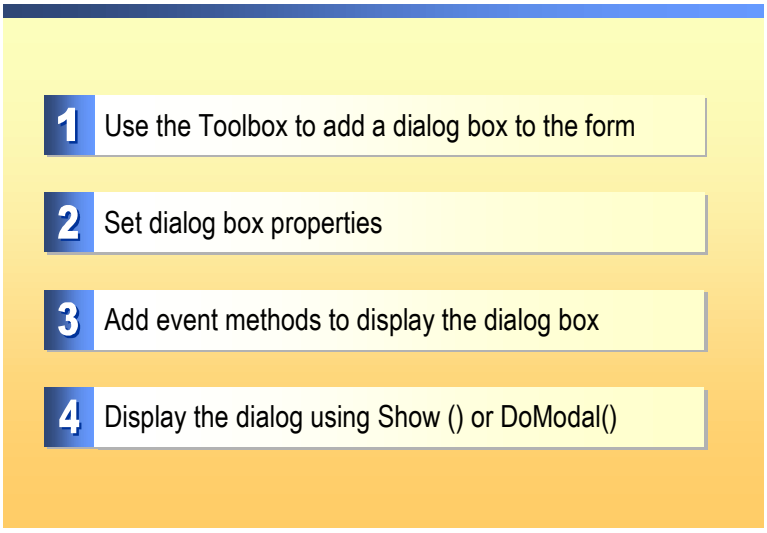
Property	Setting
FormBorderStyle	FixedDialog
ControlBox	False
MinimizeBox	False
MaximizeBox	False
ShowInTaskbar	False

4. Provide a way for users to dismiss the dialog box by adding two buttons to the form. Change the **Text** property of one button to **OK** and the other button to **Cancel** as follows:



5. Set the **DialogResult** property of the **OK** and **Cancel** buttons to **OK** and **Cancel** respectively.
6. Instantiate and display the dialog box from the event handler of any menu item by using the **ShowDialog** method.

How to Create and Use a Custom Dialog Box



Introduction

A dialog box is a form, **System.Windows.Forms.Form**, whose **FormBorderStyle Enumeration** property is set to **FixedDialog**. Dialog boxes are often used to provide or collect information from the user.

A dialog box is usually, but not always, characterized by a modal style of interaction with the user. (*Modal* means that the user is not able to use the rest of the software application while the modal dialog box is open.)

Dialog boxes are derived from the **System.Windows.Forms.Form** class, so when you create a new dialog box, you must inherit this class. You can do this easily by using the Visual Studio .NET Windows Forms Designer. The Windows Forms Designer automatically creates a new class for your dialog box that is derived from **System.Windows.Forms.Form**, and you can place controls on it just as you would on the main form.

Modal or modeless dialog boxes

A dialog box is either modal or modeless.

A *modal* dialog box, the most common type, must be explicitly closed before you can continue working with the rest of the application. To close the dialog box, you usually click **OK**, **Cancel**, or an equivalent. A *modeless* dialog box allows you to keep it open while you work in another window in the same application.

Use modal dialog boxes when you must obtain some information from the user before the program can proceed. For example, if the user wants to save a file, you must get a filename before you can create or open the file. So, you create a modal dialog box to obtain this information.

Use a modeless dialog boxes when an application must have multiple windows open at once. For example, a painting program may have a modeless dialog box that you can keep open so that you can adjust paintbrush properties.

Example

Any dialog box can be displayed in either manner.

To display a modal dialog box, use the following code:

```
userOptions.ShowDialog()
```

To display a modeless dialog box, use the following code:

```
userOptions.Show();
```

Show and **ShowDialog** take an optional parameter that allows you to specify the owner of the dialog box. Dialog boxes always layer on top of their owner. Normally, you can easily pass a reference to the main form by using **this** keyword.

```
printDialog.ShowDialog(this);
```

Adding a custom dialog box

To add a custom dialog box to your application from the Toolbox:

1. Add a form to your project by right-clicking the project in Solution Explorer, pointing to **Add**, and then clicking **Windows Form**.
2. Right-click the form in Solution Explorer, and then click **Rename** to rename the dialog box to something meaningful for your application.
3. In the Properties window, change the **FormBorderStyle** property to **FixedDialog**.
4. Set the **ControlBox**, **MinimizeBox**, and **MaximizeBox** properties to **false**. Dialog boxes do not usually include menu bars, window scroll bars, **Minimize** and **Maximize** buttons, status bars, or sizable borders.
5. Set the **ShowInTaskbar** property to **false**, because dialog boxes should not show in the Windows taskbar.

Dismissing the dialog box

Because a dialog box does not have a **Close** box, you must provide a way for users to dismiss the dialog box. Normally, you do this by add **OK** and **Cancel** buttons.

To add an **OK** button and a **Cancel** button:

1. Add two buttons to the form.
2. Change the **Text** property of one button to **OK** and change the **Name** property to something meaningful, for example **ok**.
3. Change the **Text** property of the other button to **Cancel**, and change the **Name** property to something meaningful, such as **cancel**.

Setting the DialogResult property of a button

Your dialog box must return a **DialogResult** property to the calling method so that the application can determine if the dialog box was accepted or canceled. Because buttons are used to perform this function, they have a **DialogResult** property that you can set to determine the value that is returned to the calling method. For example:

```
Button ok;
...
ok.DialogResult = DialogResult.OK;
```

Setting accept and cancel behavior

The dialog box must know which buttons provide **accept** and **cancel** types of behavior, so it can provide normal Windows behaviors, such as being dismissed when the user presses the **Escape** key. You set these behaviors with the **AcceptButton** and **CancelButton** properties.

- Set the dialog box properties **AcceptButton** and **CancelButton** to the **okOptions** and **cancelOptions** button objects.

Now you can add controls and code to the dialog box to implement the required functionality.

Adding an event method

You display custom dialog boxes in an application the same way you display a common dialog box or any other form, by using the **ShowDialog** method. Usually this is done in response to a user request, in an event handler.

For example, the following event handler is called when the user selects an **Options** menu item. It creates a custom dialog box **OptionsDialog** and calls the **ShowDialog** method to display the form.

```
private void OptionsItem_Click(object sender,
                               System.EventArgs e) {
  OptionsDialog userOptions = new OptionsDialog();
  if ( userOptions.ShowDialog() == DialogResult.OK ) {
      // User clicked OK
  }
}
```

The return value from the **ShowDialog** method is checked to discover whether the user clicked **OK** or **Cancel**.

Obtaining selections

The reason for using a dialog box is to allow the user to provide information or to change application settings. When the user clicks **OK** and closes the dialog box, your application needs a way to examine the settings of the dialog box. To do this, you create public properties for the dialog box form.

For example, if you have a **TextBox** called **myName** in a dialog box, you can encapsulate **myName.Text** in a public property called **Name**, as shown in the following code:

```
public class OptionsDialog : System.Windows.Forms.Form {
...
public string Name {
  get {
      return myName.Text;
  }
}
...
}
```

You can use the following code from the main form:

```
  if ( userOptions.ShowDialog() == DialogResult.OK ) {
      string username = userOptions.Name;
  }
```

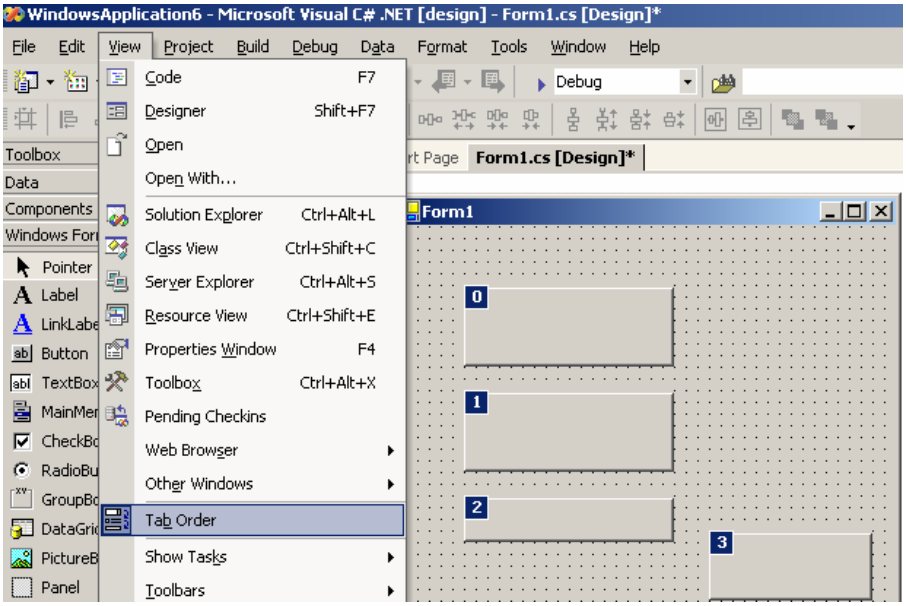
Setting the tab order of a form or dialog box

The tab order is the order in which a user moves focus from one control to another on a form by pressing the TAB key. Each form has its own tab order. By default, the tab order is the same as the order in which you created the controls. Tab-order numbering begins with zero.

Setting the tab order using the View menu

To set the tab order by using the **View** menu:

1. On the **View** menu, click **Tab Order**.
2. Click the controls sequentially to establish the tab order that you want.
3. When you are finished, on the **View** menu, click **Tab Order** again.



Alternatively, you can set tab order in the Properties window by using the **TabIndex** property. The **TabIndex** property of a control determines where it is positioned in the tab order. By default, the first control drawn has a TabIndex value of **0**; the second has a TabIndex of **1**, and so on.

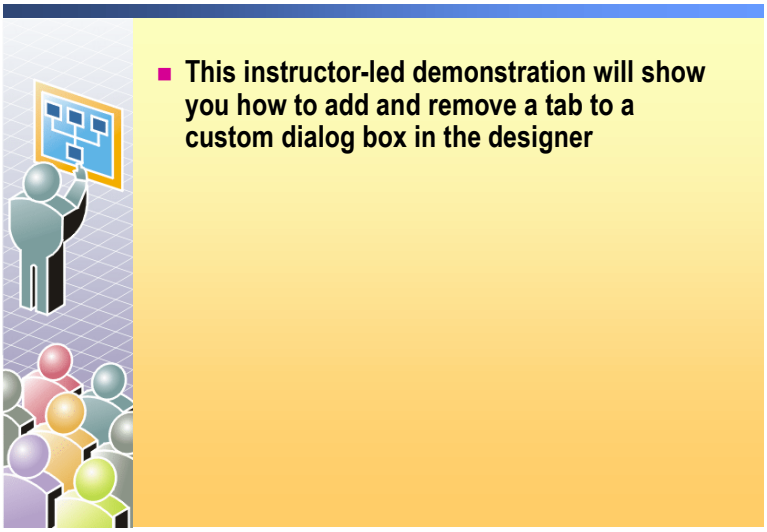
Setting the tab order using the TabIndex property

To set the tab order using the **TabIndex** property:

1. Select the control.
2. Set the **TabIndex** property to the required value.
3. Set the **TabStop** property to **True**.

   If you set the **TabStop** property to **False**, the control is passed over in the tab order of the form. A control whose **TabStop** property has been set to **False** still maintains its position in the tab order, even though the control is skipped when you cycle through the controls by using the TAB key.

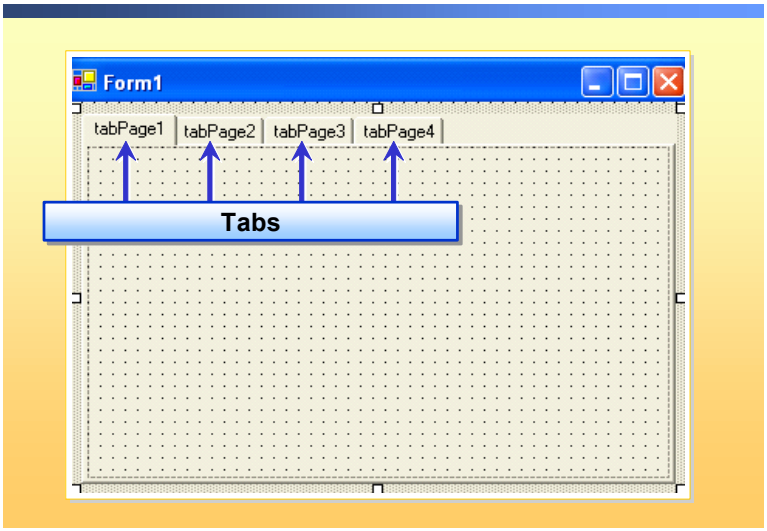
Demonstration: Creating a Custom Tabbed Dialog Box



This instructor-led demonstration will show you how to add and remove a tab to a custom dialog box in the designer. The instructor will:

1. Open Visual Studio .NET and create a new **Windows Application** project named **MyForm**.
2. Add a dialog box to the project by right-clicking the project in Solution Explorer.
3. Add a **TabControl** to the dialog box.
4. Add or remove tabs by using the TabPage Collection Editor.

How to Create and Use a Custom Tabbed Dialog Box



**********ILLEGAL FOR NON-TRAINER USE**********

Introduction

The .NET Framework provides many controls that you can add to your application so that it provides a standard Windows user interface. Most of these are simple to use, but others have some unique features. Some of the more interesting controls are described in this topic.

TabControl object

You can customize a dialog box by adding a Windows Forms **TabControl** object. The **TabControl** object displays multiple tabs, like dividers in a notebook or labels in a set of folders in a filing cabinet. The tabs can contain pictures and other controls. You can use the **TabControl** object to produce the kind of multiple-page dialog box that appears many places in the Windows operating system, such as the Display control panel.

TabControl properties

The most important property of the **TabControl** object is the **TabPages** collection, which contains the individual tabs. Each individual tab is a **TabPage** object. Clicking a tab raises the **Click** event for that **TabPage** object.

You can change the appearance of tabs in Windows Forms by using the properties of the **TabControl** and the **TabPage** objects that make up the individual tabs on the control. By setting these properties, you can display images on tabs, display tabs vertically instead of horizontally, have multiple rows of tabs, and enable or disable tabs programmatically.

Adding and removing a tab in the designer

To add a tab in the designer:

1. Drag a **TabControl** from the **Windows Forms** tab of the Toolbox to the designer.
2. In the Properties window, click the **Add Tab** link.

   - Or -

   In the Properties window, click the **Ellipsis** button ... next to the **TabPages** property to open the TabPage Collection Editor, and then click the **Add** button.

To remove a tab in the designer:

1. In the Properties window, click the **Ellipsis** button ... next to the **TabPages** property to open the TabPage Collection Editor.
2. In the left window, under **Members:**, select the tab to remove, and then click the **Remove** button.

Adding, removing, enabling, and disabling tabs programmatically

To add a tab programmatically:

- Use the **Add** method of the **TabPages** property.

```
TabPage myTabPage = new TabPage("Print Options");
tabControl1.TabPages.Add(myTabPage);
```

To remove a tab programmatically:

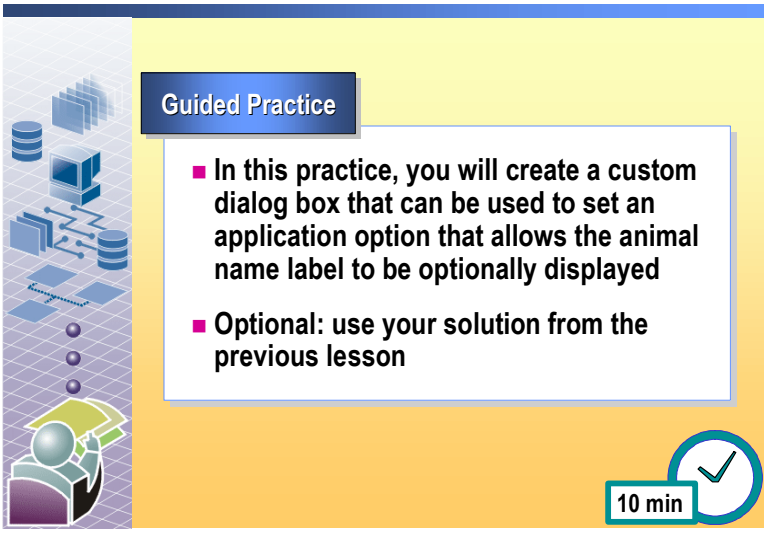
- To remove selected tabs, use the **Remove** method of the **TabPages** property. To remove all tabs, use the **Clear** method of the **TabPages** property.

```
tabControl1.TabPages.Remove(tabControl1.SelectedTab);
 // Removes all the tabs:
tabControl1.TabPages.Clear( );
```

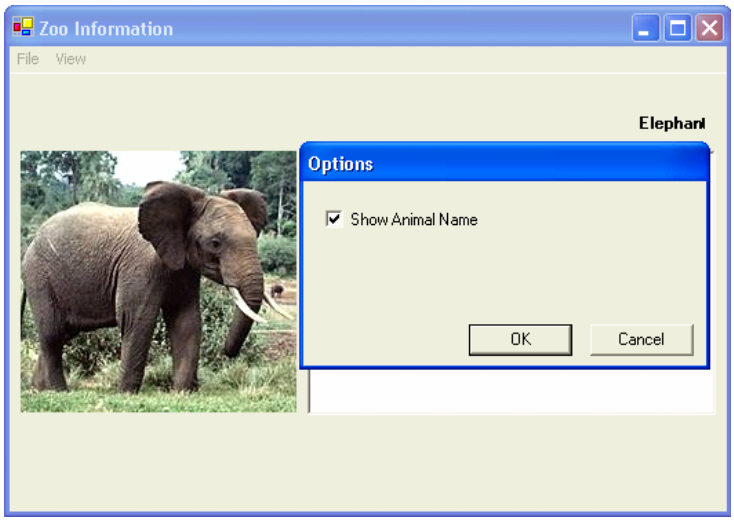
To enable or disable tabs programmatically:

```
tabPage1.Enabled = false;
```

Practice: Creating a Custom Dialog Box



In this practice, you will create a custom dialog box that can be used to set an application option that allows the animal name label to be optionally displayed. The **Options** dialog box is launched from the **Options** item on the **View** menu.



If you have a working solution from the Using a Common Dialog Box lesson in this module and you want to build on that, open that solution and skip steps 1 and 2 in this practice.

The solution for this practice is located in *install_folder*\Practices\Mod08\Custom_Solution \Animals.sln. Start a new instance of Visual Studio .NET before opening the solution.

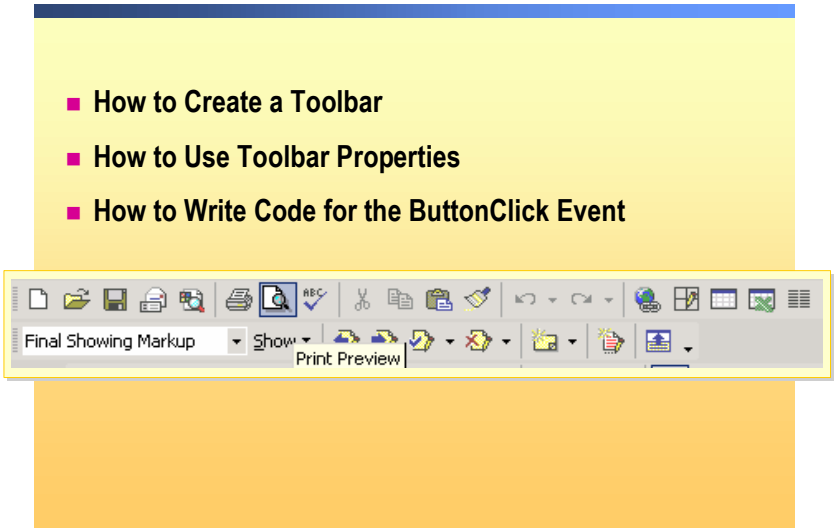
Tasks	Detailed steps
1. Start Visual Studio .NET and then open *install_folder*\Practices \Mod08\Custom\Animals.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder* \Practices\Mod08\Custom folder, click **Animals.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. *(Optional)* Build and run the solution, and familiarize yourself with it.	**a.** In Visual Studio .NET, press F5. **b.** Examine the Zoo Information application. **c.** Close the Zoo Information window.
3. Add a new custom dialog box to your project.	**a.** In Solution Explorer, use the **Add New Item** option on the **Add** menu to add a new Window Form to your project. • Name the form **Options.cs**. **b.** Set the following property values: • FormBorderStyle: **FixedDialog** • ControlBox, MinimizeBox, MaximizeBox and ShowInTaskbar: **false** **Note:** • **FormBorderStyle** is located under **Appearance** in the Properties window. • **ControlBox**, **MinimizeBox**, **MaximizeBox,** and **ShowInTaskbar** are located under **Window Style** in the Properties window.
4. Add **OK** and **Cancel** buttons to your dialog box.	▪ Drag two buttons from the Toolbox to the **Options** form. • Name the **OK** button **ok**, and change the **Text** property to **OK**. • Name the **Cancel** button **cancel**, and change the **Text** property to **Cancel**. • Set the **DialogResult** properties for the buttons to the correct values.
5. Add a **CheckBox** control to your Options form, and set the following properties: • Name: **showLabel** • Text: **Show animal name** • Checked: **true**	**a.** Drag a **CheckBox** control from the Toolbox to the **Options.cs** form. **b.** Change the **Name, Text,** and **Checked** properties to the values shown in the left column.

Tasks	Detailed steps

6. Write a Property **bool Options.ShowLabel** that returns **true** when the CheckBox is checked.
This value is retrieved from the **CheckBox.Checked** property.
	a. Switch to the Code Editor for the Options form.
	b. Write a property named **ShowLabel** that returns the value of **CheckBox.Checked**. Place this code between the class member variable declarations and the **Options** constructor.
	Example code:
	`public bool ShowLabel {`
	`get {`
	`return showLabel.Checked;`
	`}`
	`}`

7. In the event handler for the **Options** menu item, write code to create and display your new dialog box.	**a.** If necessary, create an event handler for the **Options** menu item by switching to Design view and double-clicking the **Options** menu item. **b.** In the event handler, create your dialog box by using the following code: `Options zooOptions = new Options();` **c.** Use the **ShowDialog** method to display the dialog box.
8. In the **Options** event handler, set **animalName.Visible** to **false** if the user cleared the **Show animal name** check box on your dialog box, and clicked **OK**.	▪ In the **Options** event handler, write code that sets the **animalName.Visible** property to the value of the **showLabel** property on your custom dialog box if the **DialogResult** is **DialogResult.OK**: `animalName.Visible = zooOptions.ShowLabel;`
9. Test your application by loading the XML data file **AnimalData.xml**, and changing the **Show Animal Name** option.	**a.** On the **Build** menu, click **Build Solution**. **b.** If necessary, use breakpoints and the debugger to check your application.
10. Save your solution.	▪ On the **File** menu, click **Save All**.

Lesson: Creating and Using Toolbars



Introduction

The toolbar is a standard feature in many Windows-based applications. Toolbars display a row of buttons and drop-down menus that activate commands. Typically, the buttons and drop-down menus correspond to items in the menu structure of an application, providing a graphical interface through which the user has quick access to the application's most frequently used functions and commands.

Lesson objectives

After completing this lesson, you will be able to:

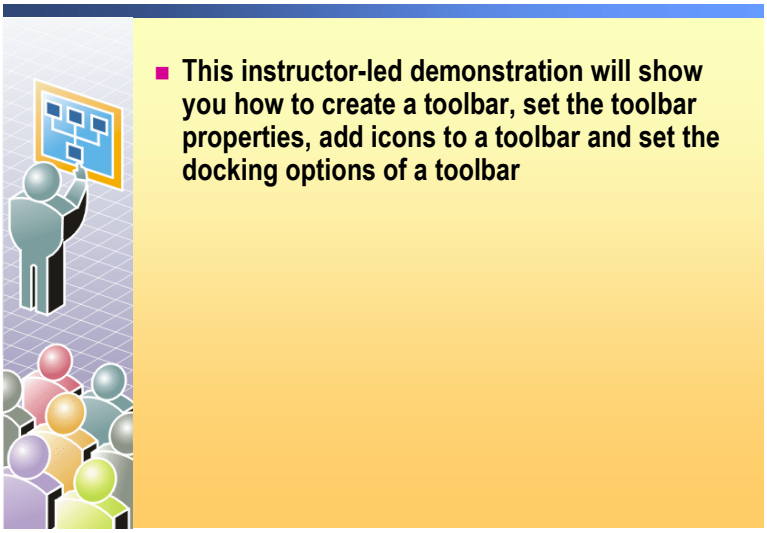
- Create a toolbar.
- Set toolbar icons and docking options.
- Write an event handler for the **ButtonClick**.event.

Lesson agenda

This lesson includes the following topics and activities:

- Demonstration: Creating a Toolbar
- How to Create a Toolbar
- How to Use Toolbar Properties
- How to Write Code for the **ButtonClick** Event
- Practice: Creating and Using a Toolbar

Demonstration: Creating a Toolbar



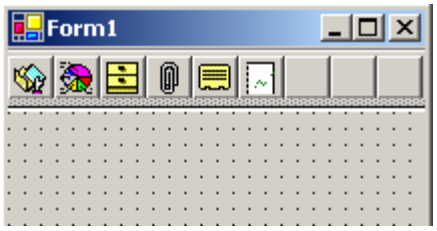
This instructor-led demonstration will show you how to create a toolbar, set the toolbar properties, add icons to a toolbar, and set the docking options of a toolbar.

The instructor will:

1. Open Visual Studio .NET and create a new **Windows Application** project named **MyForm**.
2. Add a **Toolbar** control to the form.
3. Set toolbar properties.

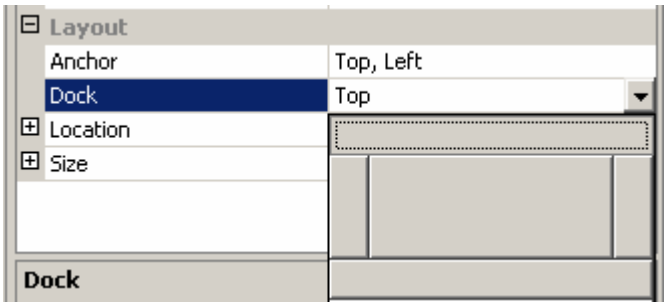
Property	Description
Appearance	Controls the appearance of the **ToolBar** control: **Normal** for a three-dimensional and raised view **Flat** for a flat button that rises to a three-dimensional view
Button size	Suggests the size of buttons of the toolbar
Buttons	**(Collection)** editor allows adding and removing tool bar buttons
Cursor	The cursor that appears when the mouse passes over the control
ImageList	The imageList from which this tool bar will get all of the button images
Dock	Determines the docking location of the tool bar

4. Add icons to the toolbar.

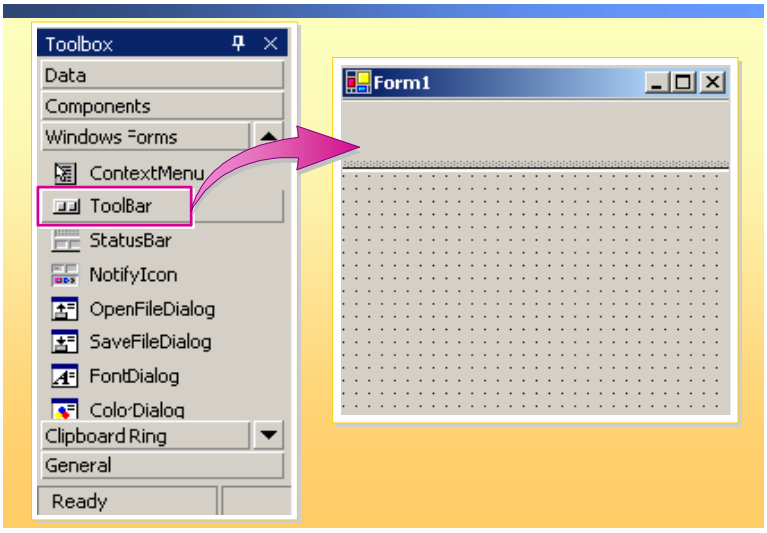


Tip Visual Studio .NET provides a library of icons in the Program_Files\Visual Studio.NET\Common7\Graphics\icons\ folder.

5. Set docking options for a toolbar.



How to Create a Toolbar



Introduction

To create a toolbar for an application, you use the Windows Forms **ToolBar** control.

Creating a toolbar

To create and add a toolbar to a form:

1. In the Windows Forms Designer, open the form to which you want to add a toolbar.
2. In the Toolbox, double-click the **Toolbar** control. A toolbar is added to the form.

When you want to add a toolbar that uses images to your application, you must:

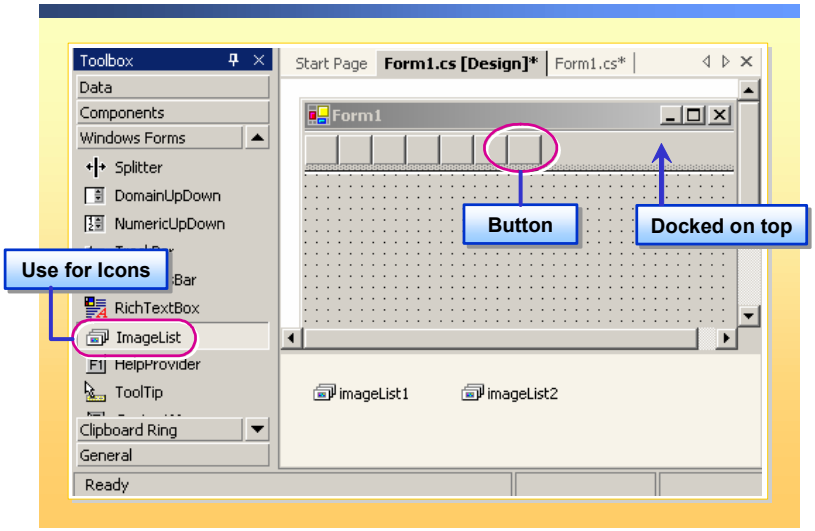
- Add a **ToolBar** control to the form.
- Add **ToolBarButton** objects to the **Buttons** collection of the toolbar.
- Add an **ImageList** control to the form.
- Load icon images into the **Images** collection of the **ImageList** object.
- Assign an index to the **Images** collection of the **ImageIndex** property of each **ToolBarButtons** object.
- Set docking options.
- Write an event handler for the toolbar.

Toolbar properties

The following table shows **ToolBar** properties that you will frequently use.

Property	Description
Appearance	Affects the appearance of the buttons that are assigned to the toolbar. **Normal:** The toolbar buttons appear three-dimensional and raised. **Flat:** The toolbar buttons have a flat appearance. As the mouse pointer moves over the flat buttons, they appear raised and three-dimensional.
Buttons	Holds all the **ToolBarButton** controls that are assigned to the toolbar. The **Buttons** property is a zero-based indexed collection. Use this property to add buttons to or remove buttons from the toolbar.
ButtonSize	Sets the size of the **ToolBarButton** controls on the toolbar. If the **ButtonSize** property is not set, it will either be set to a default size or large enough to accommodate the image and text, whichever is greater.
ImageList	If you instantiate an **ImageList** object and assign it to the **ImageList** property, you can assign an image from the list to the **ToolBarButton** controls.
ShowToolTips	Determines whether ToolTips will be visible to the user. ToolTips allow you to provide help to users when they rest the mouse pointer on a **ToolBarButton** control. The default value is **True**.

How to Use Toolbar Properties



Introduction

After you add a **ToolBar** control to your form, you must add buttons to it. You can add icons to the buttons to represent various functions, and set docking options for the toolbar.

Adding buttons to a toolbar

To add buttons to a toolbar:

1. In Visual Studio .NET, open the Properties window for the **ToolBar** control.
2. Click the **Buttons** property to select it, and then click the **Ellipsis** button ... to open the ToolBarButton Collection Editor.
3. Use the **Add** and **Remove** buttons to add buttons to and remove buttons from the **ToolBar** control.
4. Set the properties of the individual buttons in the Properties window that appears in the pane to the right of the editor.
5. Click **OK** to close the dialog box and create the buttons that you specified.

Button properties

The following table shows toolbar button properties that you will frequently use.

Property	Description
DropDownMenu	Sets the menu that is to appear in the drop-down toolbar button. The **Style** property of the toolbar button must be set to **DropDownButton**.
Pushed	Sets whether a toggle-style toolbar button is currently in the pushed state. The **Style** property of the toolbar button must be set to **ToggleButton** or **PushButton**.
Style	Sets the style of the toolbar button.
	DropDownButton: A drop-down control that displays a menu or other window when clicked.
	PushButton: A standard three-dimensional button.
	Separator: A space or line between toolbar buttons.
	ToggleButton: A toggle button that appears sunken when clicked and retains the sunken appearance until it is clicked again.
Text	Specifies the text string displayed by the button.
ToolTipText	Specifies the text that appears as a ToolTip for the button. ToolTips allow you to provide help to users when they rest the mouse pointer on a toolbar button.

Adding icons to toolbar buttons

Toolbars usually have buttons that use icons to represent a function of the application. The icons provide easy identification for users. For example, an icon of a floppy disk is commonly used to represent a **File Save** function. Each button should have text or an image assigned to it; you can also assign both.

To display images on your toolbar, you must first add the images to the **ImageList** component and then associate the **ImageList** component with the **ToolBar** control.

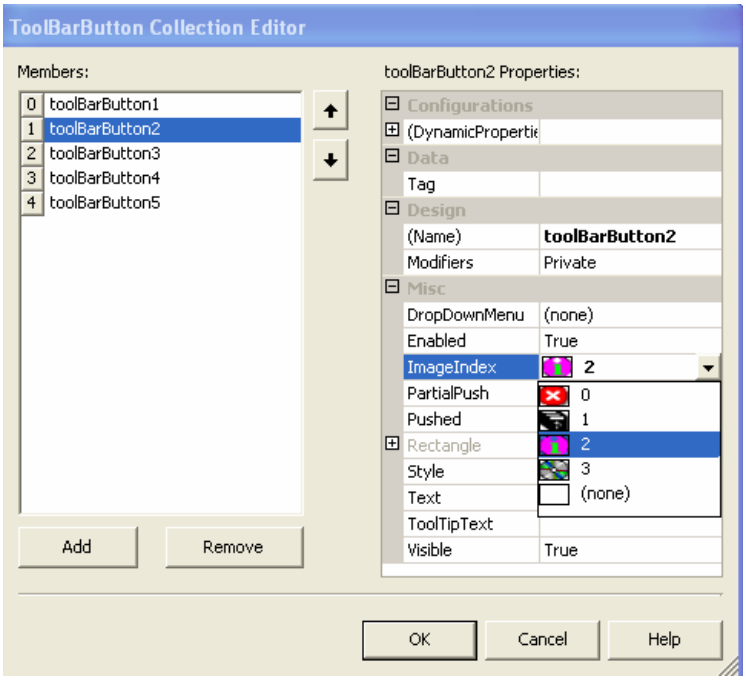
To add an icon for a toolbar button at design time:

1. Add an **ImageList** control from the Toolbox to your form.
2. In the Properties window for the **ImageList** component, click the **Images** property to select it, and then click the **Ellipsis** button ... to open the Image Collection Editor.
3. Use the **Add** button to add images to the **ImageList** component, and then click **OK** to close the Image Collection Editor.

Tip Visual Studio .NET provides a library of icons in the Program_Files\Visual Studio.NET\Common7\Graphics\icons\ folder.

---

4. In the Properties window for of the **ToolBar**, set the **ImageList** property to the **ImageList** component that you added earlier.
5. Click the **Buttons** property of the **ToolBar** control to select it, and then click the **Ellipsis** button ... to open the ToolBarButton Collection Editor.
6. Select and click a button. Then, in the Properties window that appears in the pane to the right of the ToolBarButton Collection Editor, set the **ImageIndex** property of each toolbar button to one of the values in the list, which is drawn from the images that you added to the **ImageList** component. Click **OK** to close the ToolBarButton Collection Editor.



Docking the toolbar

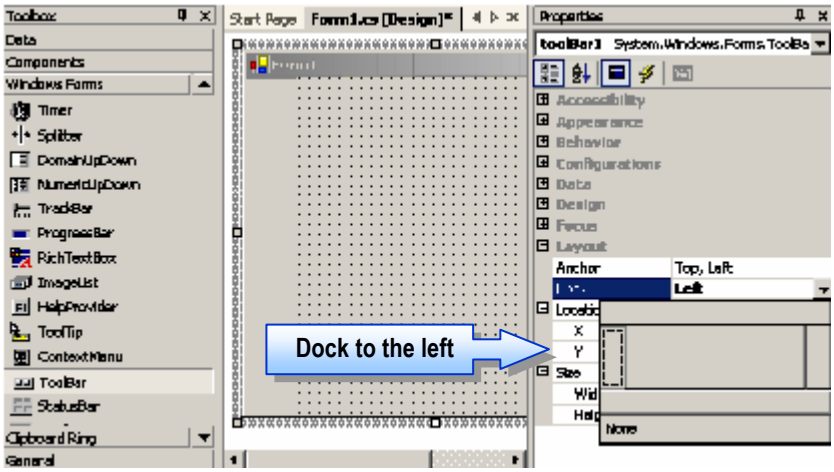
You can dock toolbars to the edges of your form, either on the top, bottom, right, or left.

Setting docking options for a toolbar

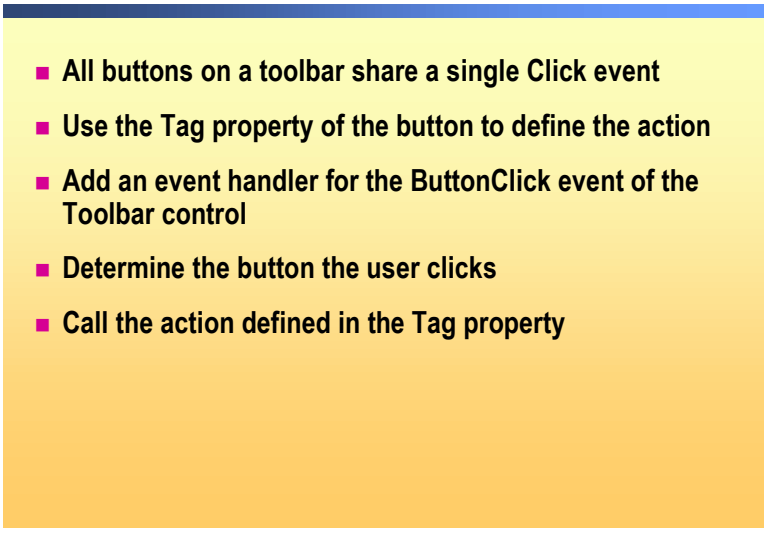
To set docking options for a toolbar on a form:

1. Drag a **Toolbar** control onto your form.
2. In the Properties window, click the arrow to the right of the **Dock** property.

   An editor is displayed that shows a series of boxes representing the edges and the center of the form.
3. Click the button that represents the edge of the form where you want to dock the toolbar. In the Properties window, click the arrow to the right of the **Dock** property.



How to Write Code for the ButtonClick Event



Introduction

All buttons on a toolbar share a single **Click** event. To add functionality to your toolbar, you must know which button the user clicks.

Determining which button is clicked

Toolbar buttons normally duplicate events that can be raised in some other manner, usually from menu items. Therefore, when you handle an event on the toolbar, you identify the button that was pressed and then call the event handler that it is associated with that button.

Tag property

The **ToolBarButton** class provides a **Tag** property that makes this task easy. When you create the **ToolBarButton** object, set the **Tag** property to the object whose behavior you are duplicating. When the toolbar event handler is called, you can use this property to send a **Click** event to the original object.

Example

For example, if the user clicks the icon to open a file, you call the same event handler that you would call in response to the user selecting **Open** on the **File** menu. The following code, which assumes that you have a menu item, **openFile**, illustrates this example.

```
MenuItem openFile = new MenuItem();
openFile.Click += new EventHandler(openFile_Click);
...
// create the ToolBar and ToolBarButton objects
...
ToolBarButton openButton = new ToolBarButton();
openButton.Tag = openFile;
```

The event handler for the toolbar can handle any button with the following code:

```
private void toolBar1_ButtonClick(
                        object sender,
                        ToolBarButtonClickEventArgs e) {
      ToolBarButton tbb = e.Button;
      MenuItem mItem = (MenuItem) tbb.Tag;
      mItem.PerformClick();
}
```

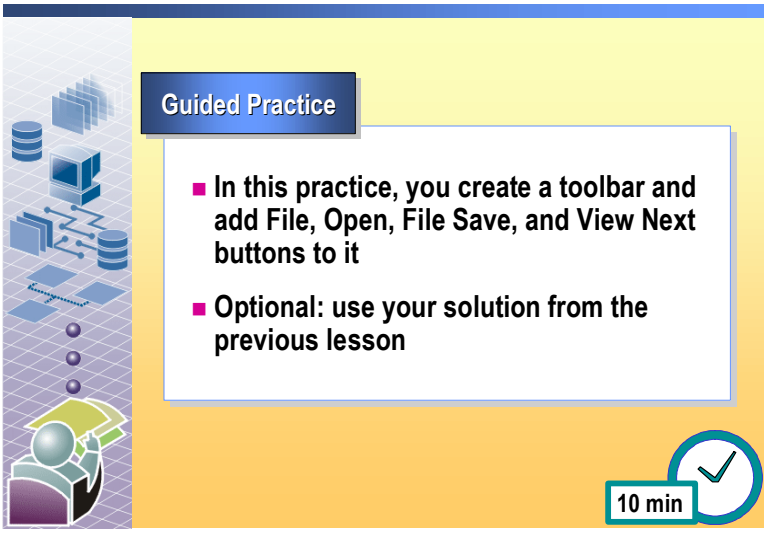
From the *ToolBarButtonClickEventArgs* parameters, the event handler for the toolbar retrieves a reference to the button that was clicked. The button uses the **Tag** property to reference the menu item that has the duplicate functionality, so the event handler can call the **PerformClick** method of the menu item, similar to a click. The **PerformClick** method causes the event handler for the menu item to be called.

The advantage of this technique is that the event handler can handle any button.

Note that although the code in the example is broken onto three lines, it normally would be written as follows:

```
((MenuItem)(e.Button.Tag)).PerformClick();
```

Practice: Creating and Using a ToolBar



In this practice, you will create a toolbar and add **File Open**, **File Save** and **View Next** buttons to it. At the end of this practice, your solution should appear similar to the following illustration:

If you have a working solution from the Creating a Custom Dialog Box lesson in this module and you want to build upon that, open that solution and skip steps 1 and 2 in this practice.

The solution for this practice is located at *install_folder*\Practices\Mod08\ ToolBar_Solution\Animals.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET and then open *install_folder*\Practices \Mod08\ToolBar\Animals.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder* \Practices\Mod08\ToolBar, click **Animals.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. *(Optional)* Build and run the solution, and familiarize yourself with it.	**a.** In Visual Studio .NET, press F5. **b.** Examine the Zoo Information application. **c.** Close the Zoo Information window.
3. Load icons for buttons by adding an **ImageList** control to the form and then loading three appropriate images from Program Files \Microsoft Visual Studio .NET \Common7\Graphics\icons.	**a.** Press SHIFT+F7 to switch to Design view and then drag an **ImageList** control from the Toolbox to **Form1**. **b.** Use the **Images** collection and the **Image Collection Editor** to add 3 images to the **ImageList**. Samples images can be found in the folders under Program Files \Microsoft Visual Studio .NET\Common7\Graphics\icons.
4. Add a **ToolBar** control to your form, and set the ImageList property to the ImageList that you created in the previous task.	**a.** Drag the **ToolBar** control from the **Toolbox** to your form. **b.** Set the **ImageList** property to the **ImageList** that you created earlier. By default, this will be **imageList1**.
5. Add four buttons to the Toolbar, using the information in the following table.	▪ Use the **Buttons** collection and the **ToolBarButton Collection Editor** to add the buttons.

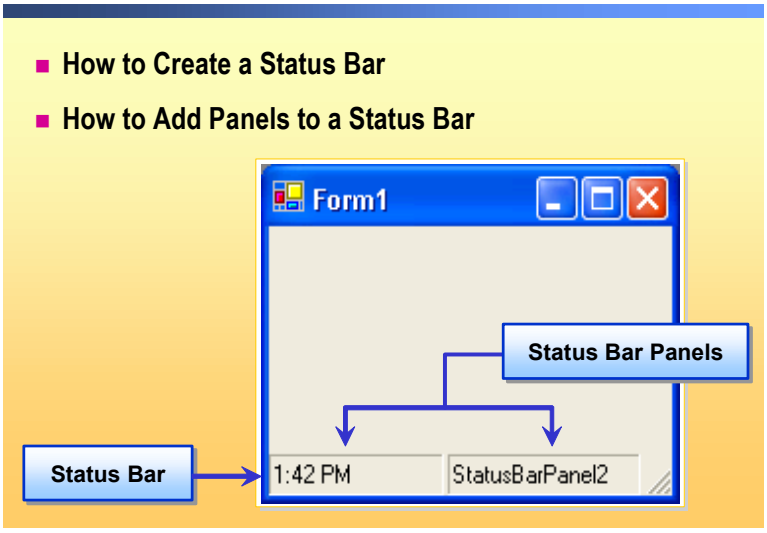
Button	Style	Name	ImageIndex
First button	PushButton	openFile	Select an icon
Second button	PushButton	saveFile	Select an icon
Third button	Separator	default	none
Fourth button	PushButton	viewNext	Select an icon

Tasks	Detailed steps
6. Write code to set the **Tag** property of each **PushButton** to reference the menu item that the button is equivalent to, using the table to the right.	▪ Place the code in the main form constructor, after the **InitializeComponents()** method. **Button name** — **Set the tag property value to:** openFile — Name of the **File Load** menu item saveFile — Name of the **File Save** menu item viewNext — Name of the **View Next** menu item For example: `openFile.Tag = loadItem;`

7. Create an event handler for the **ToolBar** control that calls the **PerformClick** method on the menu item associated with the button.	**a.** In Design view, double-click the toolbar.
	b. Use the following code to invoke the desired method:
	`ToolBarButton anyButton = e.Button;`
	`MenuItem anyMenuItem =`
	`(MenuItem) anyButton.Tag;`
	`anyMenuItem.PerformClick();`

8. Test your application by clicking the **ToolBar** button that loads the XML data file **AnimalData.xml**.	**a.** On the **Build** menu, click **Build Solution**. **b.** If necessary, use breakpoints and the debugger to check your application.
9. Save your solution.	▪ On the **File** menu, click **Save All**.

Lesson: Creating the Status Bar



Introduction

A **StatusBar** control can be added to a form and customized to provide useful information, such as the name of a file that is currently open, the current date or time, or the status of certain keys on the keyboard. In this lesson, you will learn how to enhance the interface of an application by using the **StatusBar** control.

Lesson objectives

After completing this lesson, you will be able to:

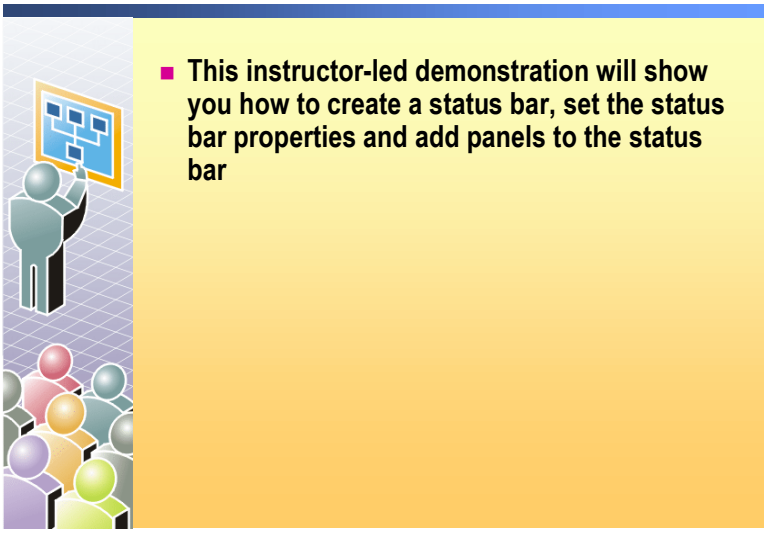
- Create the status bar.
- Set the status bar properties.

Lesson agenda

This lesson includes the following topics and activities:

- Demonstration: Creating a Status Bar
- How to Create a Status Bar
- How to Add Panels to a Status Bar
- Practice: Creating a Status Bar

Demonstration: Creating a Status Bar

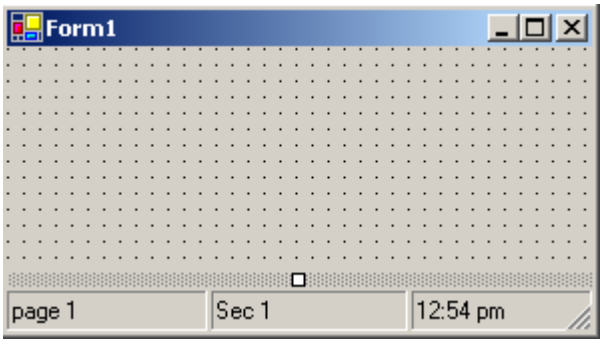


This instructor-led demonstration will show you how to create a status bar, add panels to it, and set the panel properties. The instructor will:

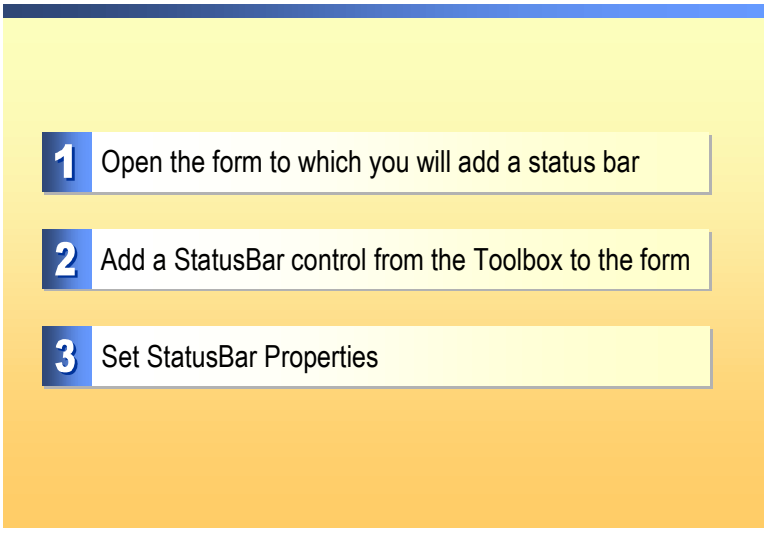
1. Open Visual Studio .NET and create a new **Windows Application** project named **MyForm**.
2. Add a **StatusBar** control to the form.
3. Set the status bar properties, as shown in the following table:

Properties	Description
ShowPanel	Determines if a status bar displays panels, or if it displays a single line of text.
Panels	**(Collection) Editor** allows adding and removing panels to the status bar.

4. Add panels to a status bar at design time.



How to Create a Status Bar



Introduction

You can add a **StatusBar** control to a form and customize the control to provide useful information, such as the name of a file that is currently open, the current date or time, or the status of certain keys on the keyboard. The Windows Forms **StatusBar** control appears on forms as an area, usually displayed at the bottom of a window, in which an application can display various kinds of status information.

Adding a status bar

To add a status bar to your form:

1. Open the form to which you want to add a status bar.
2. Use the Toolbox to add a **StatusBar** control to the form.
3. Set the appropriate **StatusBar** properties, including **ShowPanels**.

StatusBar properties

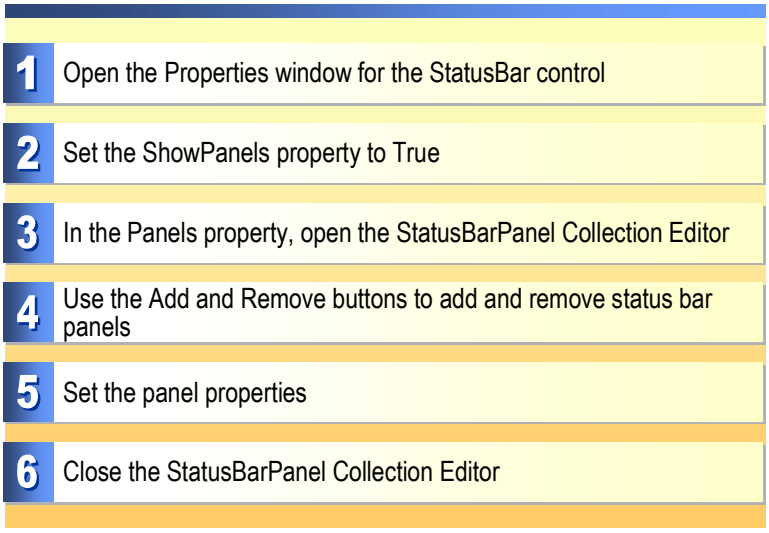
The following table shows **StatusBar** properties that you will frequently use.

Property	Description
Panels	By default, a status bar has no panels. Use the **Panels** property to add panels to or remove panels from the StatusBarPanels collection.
ShowPanels	If set to **False** (default), displays only the value of the **Text** property of the control. If set to **True**, enables you to display panels in your status bar.
SizingGrip	Displays a sizing grip on the lower right corner of the form to indicate to users that the form can be resized. Use only on a form that can be resized.
Text	Contains the text string displayed in the status bar.

If you will not add panels to the status bar, set the **ShowPanels** property to **False** (the default), and then set the **Text** property to the text that you want to appear in the status bar.

To display more than one type of information in the status bar, set the **ShowPanels** property to **True**, and then add the desired number of **StatusBarPanel** objects to the **Panels** collection.

How to Add Panels to a Status Bar



Introduction

The programmable area in a **StatusBar** control consists of instances of the **StatusBarPanel** class. You can display more than one type of information in the status bar by setting the **ShowPanels** property to **True** and adding panels to the status bar.

You can use status bar panels to display text or icons to indicate state, or to display a series of icons in an animation to indicate that a process is working. For example, a status bar panel in Microsoft Word displays a small icon to indicate when a document is being saved.

Adding panels to a status bar

To add panels to a status bar at design time:

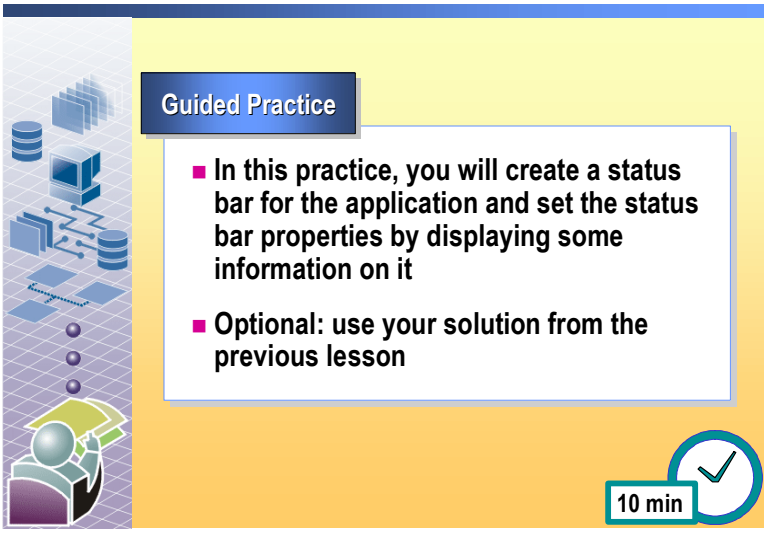
1. Open the Properties window for the StatusBar control.
2. In the Properties window for the status bar, set the **ShowPanels** property to **True**.
3. Click the **Panels** property to select it, and then click the **Ellipsis** button ... to open the StatusBarPanel Collection Editor.
4. Use the **Add** and **Remove** buttons to add panels to and remove panels from the **StatusBar** control.
5. Configure the properties of the individual panels in the Properties window that appears in the pane to the right of the editor.
6. Click **OK** to close the dialog box and create the panels that you specified.

Panel properties

The following table shows **StatusBar** panel properties that you will frequently use.

Property	Description
AutoSize	Sets the resizing behavior of the panel. **Contents:** The width of the panel is determined by its contents. **None:** The panel does not change size when the status bar control is resized. **Spring:** The panel shares the available space on the status bar with other panels that have their **AutoSize** property set to **Spring**.
Alignment	Sets the alignment of the panel in the **StatusBar** control. Options include **Center**, **Left**, and **Right**.
BorderStyle	Sets the type of border that is displayed at the edges of the panel. **None:** No border is displayed. **Raised:** The panel is displayed with a three-dimensional raised border. **Sunken:** The panel is displayed with a three-dimensional sunken border.
Icon	Sets the icon (.ico file) that is displayed in the panel.
MinWidth	Sets the minimum width of the panel in the status bar.
Style	Sets the style of the panel. **OwnerDraw:** Supports the display of images or the use of a different font than the rest of the panel objects on a status bar. **Text:** The panel displays text in the standard font.
Text	Sets the text string displayed in the panel.
Width	Sets the width of the panel, in pixels. This property may change when the form is resized, depending on the setting of the **AutoSize** property.

Practice: Creating the Status Bar



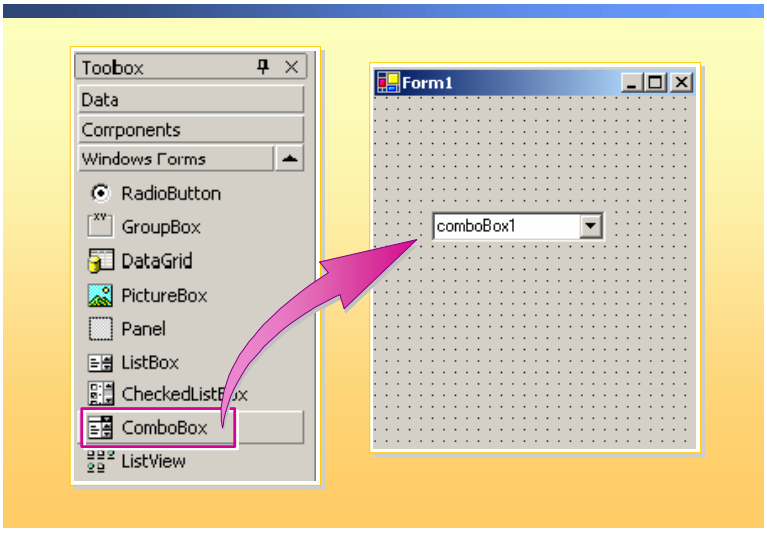
In this practice, you will create a status bar that shows the name of the file that is loaded. When you are finished, your solution should look similar to the following illustration:

If you have a working solution from the Creating and Using ToolBars lesson in this module that you want to develop, open that solution and skip steps 1 and 2 in this practice.

The solution for this practice is located in *install_folder*\Practices\Mod08\ StatusBar_Solution \Animals.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET and then open *install_folder*\Practices \Mod08\StatusBar\Animals.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder* \Practices\Mod08\StatusBar\, click **Animals.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. *(Optional)* Build and run the solution, and familiarize yourself with it.	**a.** In Visual Studio .NET, press F5. **b.** Examine the Zoo Information application. **c.** Close the Zoo Information window.
3. Add a **StatusBar** control to your application, and set the **ShowPanels** property to true.	**a.** Press SHIFT+F7 to switch to Design view. **b.** Use the Toolbox to add a **StatusBar** control to your form. **c.** Set the **ShowPanels** property to **true**.
4. Add a panel to the **StatusBar** object, naming it **filePanel**.	**a.** Use the **Panels** property and the StatusBarPanel Collection Editor to add a panel to the status bar. **b.** Set the **Name** property of the panel to **filePanel**.
5. Write code to assign the filename to **filePanel.Text** when a file is loaded.	**a.** Locate the **LoadZoo** method in Form1, and then under the call to **Zoo.Load**, assign the name of the file being loaded to the **filePanel.Text** property. **b.** **zooFile** holds the full path and file name of the file being loaded. Use the following code to extract the file name.
`int lastFilemarkerIndex = zooFile.LastIndexOf('\\');` `filePanel.Text = zooFile.Substring( lastFilemarkerIndex + 1 );`	
6. Test your application by loading the XML data file **AnimalData.xml**.	**a.** Press F5 to build and run your application. **b.** If necessary, use breakpoints and the debugger to check your application.
7. Save your solution.	▪ On the **File** menu, click **Save All**.

Lesson: Creating and Using Combo Boxes



Introduction

The Windows Forms **ComboBox** control is used to display data in a drop-down combo box. This lesson explains how to create a combo box, and how to associate objects with it.

Lesson objectives

After completing this lesson, you will be able to:

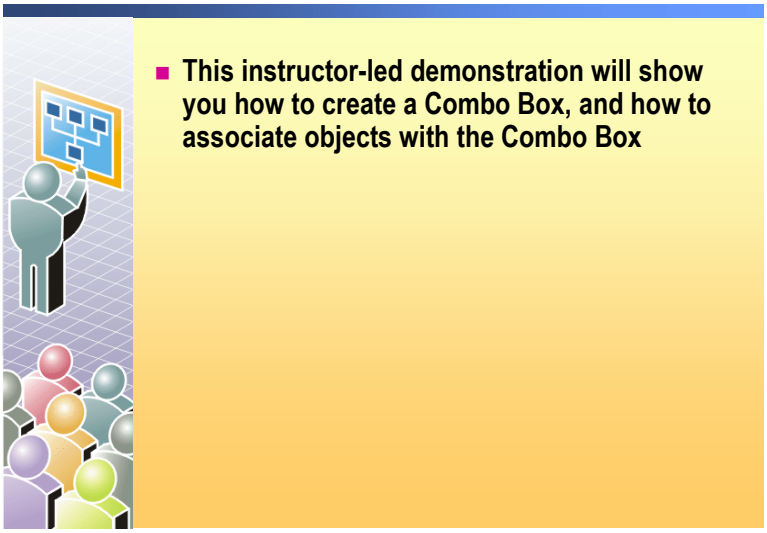
- Use a **ComboBox** control.
- Associate objects with the combo box.
- Add an event handler for the combo box.

Lesson agenda

This lesson includes the following topic and activities:

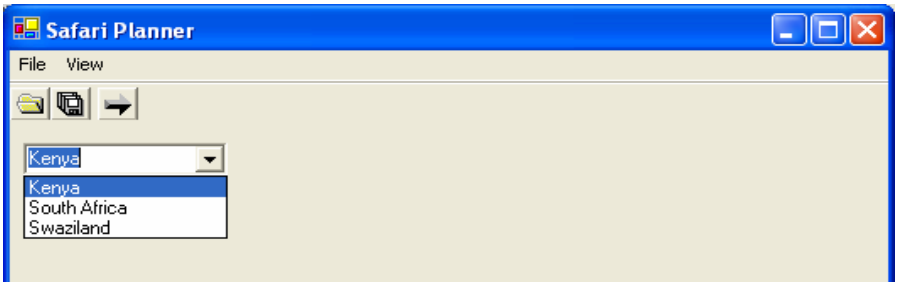
- Demonstration: Creating and Using a Combo Box
- How to Use a Combo Box
- Practice: Using a **ComboBox** Control

Demonstration: Creating and Using a Combo Box



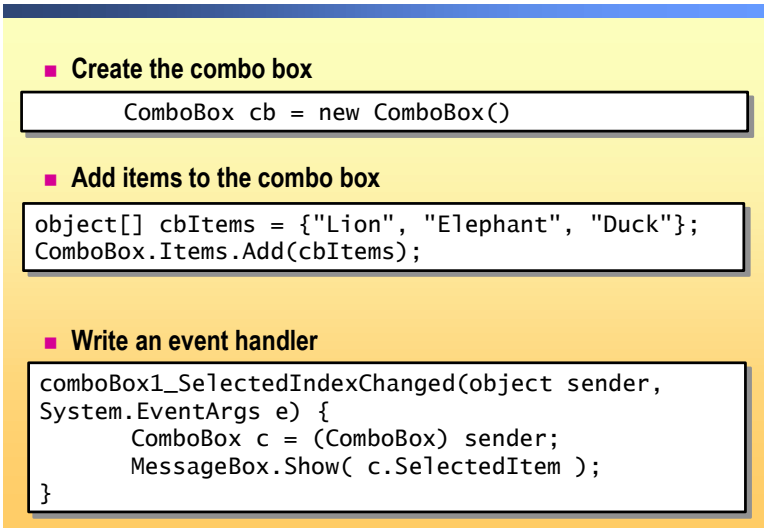
This instructor-led demonstration will show you how to create a combo box , and how to associate objects with the **ComboBox** control. The instructor will:

1. Open Visual Studio .NET and create a new **Windows Application** project named **MyForm**.
2. Add a **ComboBox** control to the form.



3. Add strings to the **items** collection using the **Add** and **AddRange** method.

How to Use a Combo Box



Introduction

Generally, a combo box is appropriate when there is a list of *suggested* selection. A combo box contains a text box field, so users can type selections that are not on the list. Also, combo boxes save space on a form. Because the full list is not displayed until the user clicks the down arrow, a combo box can easily fit in a small space. By default, the **ComboBox** control contains two parts:

- A text box at the top that allows the user to type a list item.
- A list box on the bottom that displays a list of items that the user can select from.

After you create a combo box, you can add to and remove items from it, write an event handler for it, and associate methods with it.

Creating a combo box

As with other controls, you can create a **Combobox** control by dragging the control from the Toolbox onto your form. You can also create it with code as shown in the following example:

```
ComboBox cb = new ComboBox();
```

Adding items to a combo box

You can add items to a combo box in a variety of ways, because these controls can be bound to a variety of data sources. The simplest way to add items to a combo box is to add strings to the **Items** collection by using the **Add** or **AddRange** method, as shown in the following code:

```
string[] animalNames = { "Antelope", "Bear", "Elephant",
"Lion" };
comboBox1.Items.AddRange( animalNames );
```

A much more useful way to add items to a combo box is to add the objects themselves, as shown in the following code:

```
animalList = new object[] {
                 new Antelope(),
                 new Bear(),
                 new Elephant(),
                 new Lion() };
comboBox1.Items.AddRange( animalList );
```

Or:

```
for (int i = 0; i < object.Length; i++ ) {
  comboBox1.Items.Add( animalist[i] );
}
```

The advantage of adding an object is that when you retrieve the user's selection, you can get a reference to the selected object rather than a string.

The combo box uses the **ToString** method to generate the label for the drop-down list, so you must often override the **ToString** method.

Finally, you can bind a combo box to a data source, as shown in the following code:

```
comboBox1.DataSource = animalist;
```

Writing an event handler

Because a combo box has a test entry element and a menu element, you normally must write event handlers for both of these components. The most useful events for the menu are **SelectedIndexChanged** and **SelectionChangeCommitted**. The **SelectedIndexChanged** event is sent when the index changes, including when the user scrolls through the menu. The **SelectionChangeCommitted** event is sent when the selection is made, such as when the user closes the menu.

For example, to show the selected item in a message box, you can use the following code:

```
private void comboBox1_SelectionChangeCommitted (object
sender, System.EventArgs e) {
  // cast the object sender parameter to a combo box
  ComboBox c = (ComboBox) sender;
  MessageBox.Show(c.SelectedItem);
}
```

The text box component of the combo box generates a **TextChanged** event when the user types in the text box. You can use this event to retrieve the **Text** property from the combo box and, for example, match it against the contents of the menu.

```
private void comboBox1_TextChanged(object sender,
                                   System.EventArgs e) {
  ComboBox c = (ComboBox) sender;
  MessageBox.Show(this, "You typed " + c.Text );
}
```

Note Often you will want a combo box to use the menu items to automatically complete the text typed into the text box. This is achieved by using the **FindString** method of the combo box. A working sample is provided on the Student Materials compact disc, in the file Samples\Mod08\ComboBoxSample\ComboBoxSample.sln.

Example

The following example shows how to associate objects with the combo box. In this example, the combo box holds a collection of objects that are derived from the **Animal** class.

```
public abstract class Animal { }

public class Elephant : Animal {
  public override string ToString() { return "Elephant"; }
}

public class Lion : Animal {
  public override string ToString() { return "Lion"; }
}

public class Bear : Animal {
  public override string ToString() { return "Bear"; }
}

public class Antelope : Animal {
  public override string ToString() { return "Antelope"; }
}

// ...create a combobox, name it comboBox1...
```

The preceding code is written as an array. Under normal circumstances, this code is created somewhere else in the program and copied into an array, as shown in the following code:

```
object[] animalList = {
             new Antelope(),
             new Bear(),
             new Elephant(),
             new Lion()
         };
```

You can then add these items to the **ComboBox** object. To improve performance when you use the **Add** method to add the objects, call the **BeginUpdate()** method before you add and the **EndUpdate()** method after you add.

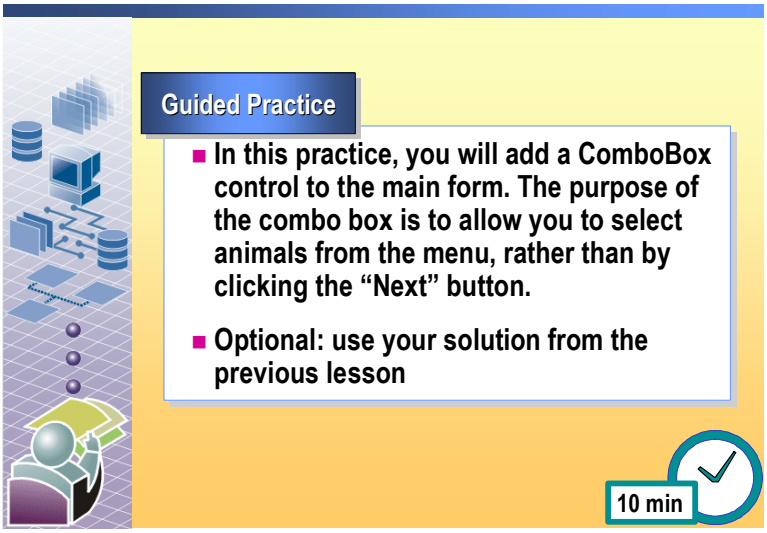
```
comboBox1.Items.AddRange( animalList );
```

The event handler for this is shown in the following code:

```
private void comboBox1_SelectionChangeCommitted (
                        object sender, System.EventArgs e) {
     ComboBox c = (ComboBox) sender;
     Animal a = (Animal) c.SelectedItem;
     MessageBox.Show(this, "You selected " + a.ToString() );
}
```

Note that the event handler is able to get a reference to the **Animal** object.

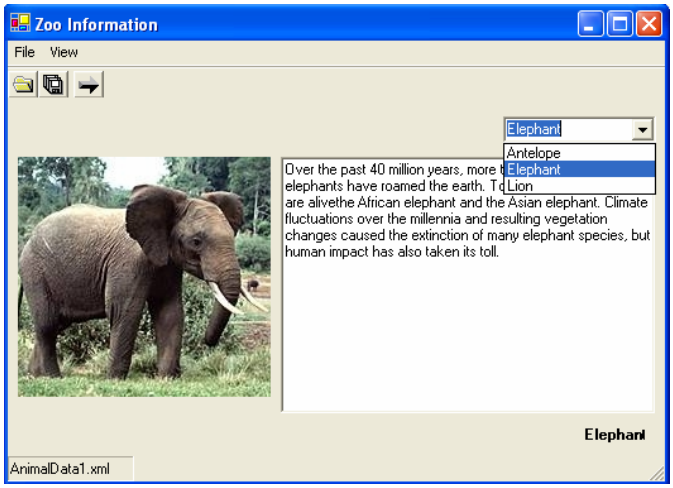
Practice: Using a ComboBox Control



In this practice, you will add a combo box to the main form. The purpose of the combo box is to allow the user to select animals from the menu, rather than by clicking the **Next** button.

If you have a working solution from the Creating the Status Bar lesson in this module that you want to develop, open that solution and skip steps 1 and 2 in this practice.

Your solution should appear as shown in the following illustration:



The solution for this practice is located in *install_folder*\Practices\Mod08\ComboBox _Solution\Animals.sln. Start a new instance of Visual Studio .NET before opening the solution.

Tasks	Detailed steps
1. Start Visual Studio .NET and then open *install_folder* \Practices\Mod08\ComboBox \Animals.sln.	**a.** Start a new instance of Visual Studio .NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\Practices \Mod08\ComboBox , click **Animals.sln**, and then click **Open**. **d.** In Solution Explorer, click **Form1.cs**, and then press F7 to open the Code Editor.
2. (Optional) Build and run the solution, and familiarize yourself with it.	**a.** In Visual Studio .NET, press F5. **b.** Examine the Zoo Information application. **c.** Close the Zoo Information window.
3. Add a **ComboBox** control to Form1, and name it **animalSelection**.	**a.** Press SHIFT+F7 to enter Designer mode. **b.** Drag the **ComboBox** control from the **Toolbox** onto **Form1**. **c.** Change the **Name** property to **animalSelection**.
4. In Form1, write a method named **InitializeAnimalSelection**. The purpose of this method is to add the **Animal** objects to the **Items** collection in the **animalSelection** object.	**a.** In the **Form1** class, write a method named **InitializeAnimalSelection**. This method does not return a value and takes no parameters. **b.** The method adds the **Animal** objects to the **Items** collection in the combo box. Sample code is provided below. The **for** loop checks **myZoo.Count** and adds a reference to each animal in the **Zoo** to the **Items** collection in the **animalSelection** object. Note that **myZoo** has an indexer.

```
public void InitializeAnimalSelection() {
  for ( int i = 0; i < myZoo.Count; i++ ) {
        animalSelection.Items.Add( (object) myZoo[i] );
  }
}
```

5. Ensure that **InitializeAnimalSelection** is called after the data file is loaded.	▪ Insert a call to **InitializeAnimalSelection** in the **loadItem_Click** event handler, after the call to the **Zoo.Load()** method.

Tasks	Detailed steps
6. Write an event handler for the **SelectionChangeCommitted** event that gets the reference to the selected animal from the sender parameter, and calls **DisplayAnimal** to display it. 	**a.** Press SHIFT+F7 to switch to Design view. **b.** Click the **ComboBox** control, click the **Events** button (shown to the left) in the **Properties** window, and then double-click **SelectionChangeCommitted**. **c.** Write code that converts the sender parameter to a **ComboBox**, then convert the **SelectedItem** in the **ComboBox** object to an **Animal**. **d.** Call **DisplayAnimal** with the **Animal** object.

`ComboBox c = (ComboBox) sender;`
`Animal a = (Animal) c.SelectedItem;`
`DisplayAnimal( a );`	

7. Test your application by loading the XML data file **AnimalData.xml**.	**a.** On the **Build** menu, click **Build Solution**. **b.** If necessary, use breakpoints and the debugger to check your application.
8. Save your solution and quit Visual Studio .NET.	**a.** On the **File** menu, click **Save All**. **b.** Quit Visual Studio .NET.

Review

- Creating the Main Menu
- Creating and Using Common Dialog Boxes
- Creating and Using Custom Dialog Boxes
- Creating and Using Toolbars
- Creating the Status Bar
- Creating and Using Combo Boxes

1. What namespace contains menus, dialog boxes, status bars, and toolbars?

   System.Windows.Forms

2. What is the difference between a form and a dialog box?

   A dialog box is a form that has its FormBorderStyle set to FixedDialog, and its ControlBox, MinimizeBox, MaximizeBox, and ShowInTaskbar properties set to false.

3. Which of the following statements are true?

   Images for a toolbar's buttons are:

   a. Assigned an index number in the Image Collection Editor.

   b. Automatically attached to the toolbar button based on function.

   c. Maintained in the ToolBarButton Image Collection Editor.

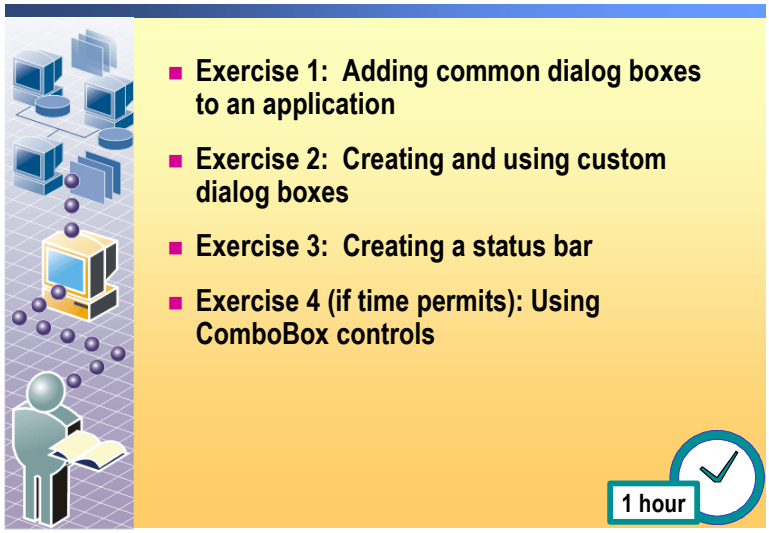
   d. Maintained in a separate **ImageList** control.

   Both a and d are true.

4. Name two methods by which you can add items to a combo box.

   The simplest way to add items to a combo box is to add strings to the Items collection by using the Add or the AddRange method.

Lab 8.1: Building Windows Applications



**********ILLEGAL FOR NON-TRAINER USE**********

Objectives

After completing this lab, you will be able to create an application that uses standard Windows controls to create a user interface.

Note This lab focuses on the concepts in this module and, as a result, may not comply with Microsoft security recommendations.

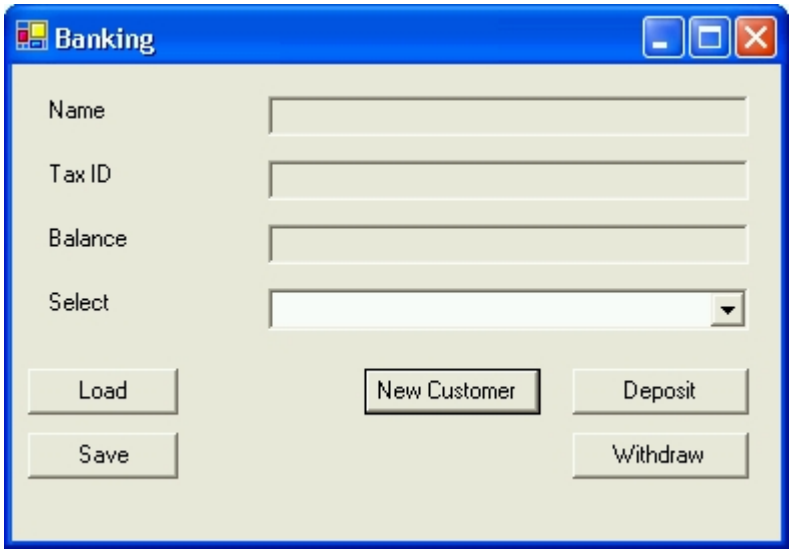
Prerequisites

Before working on this lab, you must have the ability to add Windows controls to an application.

Scenario

In this lab, you will add typical Windows functionality to an existing bank teller application.

The existing application is a very simple example, so that you can quickly understand it in this lab.



The provided application has the following files:

- *Account.cs*. Contains **CheckingAccount** and **SavingAccount** classes, both derived from **BankAccount**. These have properties, such as **Balance**, and methods, such as **Withdraw** and **Deposit**. This class is complete and you will not need to change the code in this file to complete the lab.
- *Customer.cs*. Contains the **Customer** class. Each customer contains a list of bank accounts, although this example creates only one bank account per customer. Customers have properties, such as **Name**, and methods, such as **AddAccount**. This class is complete and you will not need to change the code in this file to complete the lab.
- *Bank.cs*. Contains the list of bank customers. Important methods in the Bank class are **Load** and **Save**, which load and save the bank data, and **Add**, which adds a new customer to the bank. This class is complete and you will not need to change the code in this file to complete the lab.
- *Form1.cs*. The main application window, shown in the preceding illustration. In this lab, you will modify this class so that it uses Windows controls.
- *NewCustomer.cs*. A dialog box that allows the user to create a new customer account. This class is complete and you will not need to change the code in this file to complete the lab.

In the starter code:

- The **Load** button loads the bank data from a file.
- The **Save** button saves the bank data to the same file.
- The **New Customer** button opens a dialog box that allows the user to enter new customer information.
- The **Deposit** and **Withdraw** buttons do nothing.

Estimated time to complete this lab: 60 minutes

Exercise 0
Lab Setup

The Lab Setup section lists the tasks that you must perform before you begin the lab.

Task	Detailed steps
▪ Log on to Windows using your Student account.	▪ Log on to Windows using the following account information: • User name: **Student** • Password: **P@ssw0rd** Note that the 0 in the password is a zero.

Note that by default the *install_folder* is C:\Program Files\Msdntrain\2609.

The solution code for this lab is located in *install_folder*\Labfiles\Lab08_1\Exercise1 \Solution_Code\Bank.sln. Start a new instance of Visual Studio .NET before opening the solution.

Note that exercises 2 and 3 do not have separate starter code. If necessary, you can use the starter code listed in exercise 1, step 1, to start these exercises.

Exercise 1
Adding Common Dialog Boxes to an Application

In this exercise, you will modify the bank teller application so that it uses the **OpenFileDialog** and **SaveFileDialog** controls instead of the current **Load** and **Save** buttons.

The application uses a data file to store all the information. By default, this is called bankdata.bnk and is located in the same folder as the starter code.

Allow users of the application to use a typical **File Open** menu selection to browse to any file location, and load the file. Also, allow them to use a typical **File Save** menu selection to save the file to any location.

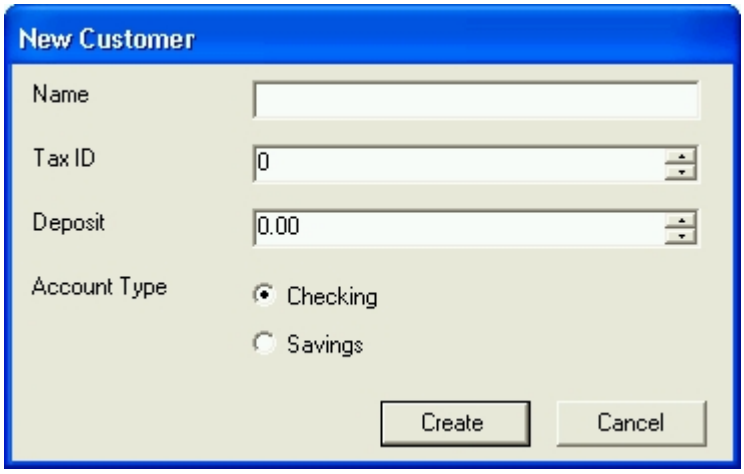
Tasks	Detailed steps
1. Start Visual Studio.NET and then open *install_folder* \Labfiles\Lab08_1\Exercise1 \Bank.sln.	**a.** Start a new instance of Visual Studio.NET. **b.** On the **Start Page**, click **Open Project**. **c.** In the **Open Project** dialog box, browse to *install_folder*\ Labfiles\ Lab08_1\Exercise1, click **Bank.sln**, and then click **Open**.
2. Add a main menu to the application, with a **File** menu that contains an **Open** option.	**a.** Add a main menu to the application. **b.** Add a **File** menu to the main menu. **c.** Add an **Open** menu item to the **File** menu. Remember to change the name properties of the menu items to meaningful values.
3. Add an **OpenFileDialog** control, display it when the user selects **Open** from the **File** menu, and then load the selected data file.	**a.** Add an **OpenFileDialog** control to your application. **b.** Add an event handler to the **Open** menu item by double-clicking it in the design window. **c.** In the event handler, write code that uses the **OpenFileDialog** to locate a data file, and then load it. The data file is called **bankdata.bnk**, and it located in *install_folder*\ Labfiles\ Lab08_1\Exercise1. **d.** If the result returned from the **OpenFileDialog** is **DialogResult.OK**, then use the method **Load** in the **Bank** class to load the data file specified in the **FileName** property of the **OpenFileDialog** object.

Tasks	Detailed steps
4. Add a **Save** option to the **File** menu. Implement the expected functionality for this option.	**a.** Add a **Save** item to the **File** menu. **b.** Add a **SaveFileDialog** control to your application. **c.** Add an event handler to the Save menu item, and write code that uses the **SaveFileDialog** to define a file name, and then save the data. The method **Save** in the **Bank** class saves a data file. Remember to delete the **Load** and **Save** buttons from the form.
5. Test your code.	**a.** Copy the data file **bankdata.bnk** to your desktop. **b.** Press F5 to compile your application and then locate the copy of the data file that is on your desktop, and load it. **c.** Add a new customer record. **d.** Save the data, and exit your application. **e.** Restart your application and load the data file, to ensure that the new customer record is loaded.

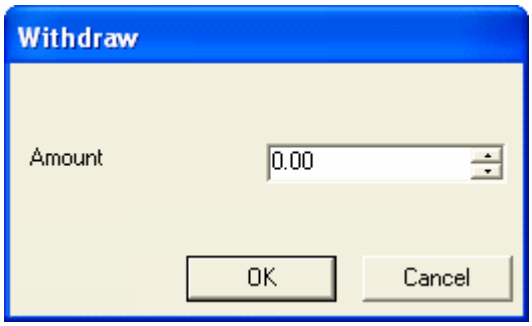
Exercise 2
Creating and Using Custom Dialog Boxes

In this exercise, you will invoke the **New Customer** dialog box, and create custom dialog boxes that allow the user of the application to withdraw and deposit amounts of money into the selected account.

The **New Customer** dialog box appears as follows:



The **Withdraw** and **Deposit** dialog boxes appear as follows:



Tasks	Detailed steps
1. Create a new menu labeled **Customers** with an item labeled **New**. Invoke the **New Customer** dialog box when this item is selected.	**a.** On the main menu, add a menu labeled **Customers** and a menu item labeled **New**. The purpose of this item is to allow the user to choose to create a new customer account. **b.** Add an event handler to the menu item and in the event handler, create a **NewCustomer** object, and then display it by using the **ShowDialog** method.
2. Create a new customer account and a bank account for that customer.	**a.** If the dialog box is closed by clicking the **Create** button, create a new customer record and a new bank account. Note that the value of the **DialogResult** property for the **Create** button is **DialogResult.OK**. Look in the **newCustomer_Click** method in the **Form1** class for sample code. **b.** Delete the **New Customer** button from **Form1**.
3. Add menu items that will invoke the **Withdraw** and **Deposit** dialog boxes.	**a.** Add a **Withdraw** menu item. **b.** Add a **Deposit** menu item. **c.** Create event handlers for these menu items.
4. Create a **Withdraw** dialog box.	**a.** Using Solution Explorer, add a new Windows form to the project, name it **Withdraw**, and configure it as a dialog box by setting the **FormBorderStyle** property to **FixedDialog**, and the **ControlBox**, **MaximizeBox**, **MinimizeBox**, and **ShowInTaskBar** properties to **false**. **b.** Add a **NumericUpDown** control to the **Withdraw** form. **c.** Create a public property called **WithdrawalAmount** that returns the value in the **Value** property of the **NumericUpDown** object. **d.** Add **OK** and **Cancel** buttons to the form.

Tasks	Detailed steps
5. Write code to open the **Withdraw** dialog box from the application menu.	**a.** In the **Withdraw** menu item event handler in **Form1**, write code to create and display the **Withdraw** dialog box. **b.** If the **Withdraw form** object returns a **DialogResult** of **DialogResult.OK**, then read the value in the **WithdrawalAmount** property, and perform the withdrawal from the currently selected customer's account. The currently selected customer is always referenced from the **currentCustomer** member of **Form1**. The currently selected customer's bank account can be accessed by using the following code: `BankAccount thisAccount = (BankAccount) currentCustomer.Accounts[0];` **c.** Use the **Withdraw** method of the **BankAccount** object to remove the correct amount from the currently selected account. For the purposes of this lab, you can assume that every customer has one bank account. **d.** Use the **SetCurrentCustomer** method to update the display on the main form after you have withdrawn the money.
6. Implement a **Deposit** dialog box.	**a.** Following the steps outlined in the previous two tasks, create a **Deposit** dialog box. **b.** Use the **BankAccount.Deposit** method to add the correct amount of money in the currently selected account.
7. Test your code.	**a.** Press F5 to compile your application. **b.** Withdraw money and deposit money to ensure that your application is working as expected.
8. Save your application and quit Visual Studio .NET.	**a.** Save your application. **b.** Quit Visual Studio .NET.

Exercise 3
Creating a Status Bar

In this exercise, you will add a status bar to the application.

Tasks	Detailed steps
1. Add a status bar to the application, with one status pane.	**a.** Add a status bar control to your application. **b.** Use the **Panels** property and the **StatusBarPanel Collection Editor** to add one panel to the status bar. **c.** Remember to set the **ShowPanels** property to **true**.
2. Display the total number of customers in the status bar pane.	▪ Write code that displays the total number of customers in the status bar. The following code returns the total number of customers: `theBank.Customers.Count.ToString();` The **SetCurrentCustomer** method is called every time a change is made to the data, so this is a good place to add the code that updates the contents of the status bar panel.
3. Test your application.	**a.** Press F5 to compile your application. **b.** Add a new customer to the list and make sure that the status bar updates correctly.
4. Save your application and quit Visual Studio .NET.	**a.** Save your application. **b.** Quit Visual Studio .NET.

If Time Permits
Using ComboBox Controls

Write a dialog box that transfers money from one account to another. Use a combo box on the transfer dialog box to select destination customer accounts.