msdn training

Module 6: Using Windows Forms

Contents

Microsoft

Instructor Notes

Presentation:
120 Minutes

Lab:
45 Minutes

This module provides students with the knowledge needed to create Microsoft® Windows®-based applications.

In the lab, students will continue working with the Cargo system. The **Customer** class from Lab 5.1 has been enhanced for students, and a **CustomerList** class has been provided to iterate through the customers. The basic Customer form has been provided, but it requires further development.

After completing this module, students will be able to:

- Describe the benefits of Windows Forms.
- Use the new properties and methods of Windows Forms.
- Write event-handling code.
- Use the new controls and control enhancements.
- Add and edit menus.
- Create a form that inherits from another form.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2373A_06.ppt
- Module 6, "Using Windows Forms"
- Lab 6.1, Creating the Customer Form

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Read the instructor notes and the margin notes for the module.
- Practice the demonstrations.
- Complete the practice.
- Complete the lab.

Demonstrations

This section provides demonstration procedures that will not fit in the margin notes or are not appropriate for the student notes.

Manipulating Windows Forms

To examine the startup code

1. Open the FormsDemo.sln solution in the *install folder*\DemoCode\Mod06\FormsDemo folder.
2. Open the Code Editor window for the modMain.vb file.
3. Examine the code in the **Sub Main** procedure.

To test the application

1. Run the project.
2. Click the **Center** button on the form, and step through the code when execution halts at the preset breakpoint.
3. Click the **Auto Scroll** button on the form, and step through the code when execution halts at the preset breakpoint. When execution resumes, resize the form vertically so that the scroll bars appear at the side of the form, and show that all of the buttons are still accessible. Resize the form again back to its original size.
4. Click the **Change Font** button on the form, and step through the code when execution halts at the preset breakpoint. When execution resumes, point out that the button fonts have changed automatically.
5. Click the **Add Owned Form** button on the form, and step through the code when execution halts at the preset breakpoint. When execution resumes, point out that the new form is owned, so that it is always displayed in front of its owner and is minimized and closed along with the owner.
6. Click the **Top Level** button on the form, and step through the code when execution halts at the preset breakpoint. When execution resumes, point out that the new form is enclosed within the boundaries of the parent form.
7. Close the application and Microsoft Visual Studio® .NET.

Using Controls

To examine the frmControls code

1. Open the ControlsDemo.sln solution in the *install folder*\DemoCode\Mod06\ControlsDemo\Starter folder.
2. Open the design window for frmControls.vb. The purpose of the form is to show the effects of the anchor and dock styles to a **Button** control. Point out the **Anchoring** and **Docking** menus and their menu items. Also, point out the **FlatStyle** property of the btnClose button in the Properties window.
3. View the code window for the frmControls.vb form.
4. View the **Sub ProcessAnchorMenus** procedure and examine the code, pointing out that multiple menus are being handled by the one procedure.
5. View the **Sub ApplyAnchoring** procedure and examine the code.
6. View the **Sub ApplyDocking** procedure and examine the code.

To test the frmControls form

1. Run the project.
2. Resize the form to show that the **Close** button does not move. This is because the **Top** and **Left** menu items are checked by default. Resize the form to its original size.
3. Use the **Anchoring** menu to demonstrate the following:

Checked Anchor menu items	Purpose
No items checked	**Close** button stays approximately in the middle of the form.
Bottom and **Right** checked	**Close** button stays close to the bottom and right of the form.
Top, **Left**, **Bottom**, and **Right** checked	**Close** button grows in direct relation to the form.

4. Using the **Docking** menu, test each docking option, beginning with the **Top** menu item and finishing with the **Fill** menu item. Show the effects these settings have on the **Close** button.
5. Click the **Close** button to quit the application.

To examine the frmDemo code

1. Open the design window of the **frmDemo.vb** form.
2. In the Properties window, view the **Anchor** properties for each control on the form.
3. Open the code window for frmDemo.vb.
4. Locate the **Sub New** procedure and explain the following line:

```
Me.MinimumSize = Me.Size
```

To test the frmDemo form

1. Run the project.
2. On the **Other** menu, click **Display** and resize the form to show how the controls behave and how the **MinimumSize** property affects the form.
3. Close both forms to quit the application.

To add ToolTip, Help, and NotifyIcon controls to frmControls

1. Open the design window for frmControls.vb, and add the following controls to the form, setting the appropriate properties.

Control	Property	Value
HelpProvider	**Name**	**hpHelp**
NotifyIcon	**Name**	**niTray**
	Icon	**C:\Program Files\Microsoft Visual Studio.NET\Common7\Graphics\icons\Elements\Cloud.ico** (or any available icon)
ToolTip	**Name**	**ttToolTip**

2. Open the code window for frmControls.vb.
3. Open the **Sub New** procedure, and uncomment and explain the lines of code beginning with the following:

```
'Dim cmTrayMenu As ContextMenu, mItem As MenuItem
```

To test the ToolTip, Help, and TrayIcon controls

1. Run the application.
2. Pause the mouse over the **Close** button control to view the ToolTip.
3. Press F1 to display the Help string.
4. Right-click the **TrayIcon** in the Windows System Tray, and then click **Exit**.
5. Close Visual Studio .NET.

Implementing Drag-and-Drop Functionality

To examine the code

1. Open the DragDrop.sln solution in the *install folder*\DemoCode\Mod06\DragDrop folder.
2. View the form design for Form1.vb. The purpose of the form is to drag an item from the list box on the left to the list box on the right.
3. View the code window for the Form1.vb form.
4. Locate the **lstListBoxFrom_MouseMove** event handler, and explain each line of code. Point out that the **DoDragDrop** method will wait until the drop action has occurred before processing the remainder of the procedure.
5. Locate the **lstListBoxTo_DragOver** event handler, and explain each line of code.
6. Locate the **lstListBoxTo_DragDrop** event handler, and explain the single line of code.

To test the application

1. Run the project.
2. Click an item in the ListBoxFrom control. Hold down the left mouse button to drag the item to the ListBoxTo control. Point out that the icon is displayed as the Move operation when over the control. Release the item over the control, and debug the code at the preset breakpoint to show the removal of the item from ListBoxFrom.
3. Press F5 to allow execution to continue.
4. Repeat the drag process from the ListBoxFrom control to the ListBoxTo control, but hold down the CTRL key while dragging the control. Point out that the icon is now displayed as the Copy operation when over the ListBoxTo control. Release the item over the list box and debug the code at the preset breakpoint to show that the item is not removed from ListBoxFrom.
5. Click the **Close** button on the form to close the debugging session.
6. Close the Visual Studio **.**NET integrated development environment (IDE).

Using Windows Forms Inheritance

To view the base form

1. Open the Visual Inheritance.sln solution in the *install folder*\DemoCode\Mod06\Visual Inheritance\Starter folder.
2. Open the design window for the frmBase.vb form, pointing out the different controls that already exist.

To test the base form application

1. Run the project.
2. Click the **System Info** button on the form and close the resulting message box.
3. Click **OK** to close the application.

To modify the base form

1. In the design window for the frmBase.vb form, set the **Modifiers** property of the following control. All other controls should remain **Assembly**.

Control	Modifiers value
lblProductName	**Public**

2. Open the code window for frmBase.vb.
3. Locate the **btnOK_Click** event handler, and change the procedure declaration to be as follows:

```
Protected Overridable Sub btnOk_Click(…)
```

4. On the **Build** menu, click **Rebuild All**.

To create the inherited form

1. On the **Project** menu, click **Add Inherited Form** to create a new inherited form. Rename the form **frmInherited.vb** and click **Open**. Select the **frmBase.vb** form as the base form, and click **OK**.
2. Open the design window for the frmInherited.vb form and point out the differences between the Public and Assembly controls by resizing the lblProductName label and attempting to resize the btnSysInfo button (which should not resize).
3. Open the code window for frmInherited.vb.
4. Locate the **Sub New** procedure, and add the following lines before the **End Sub**:

```
Me.Text = "About My Application"
txtLicensedTo.Text = "Karen Berge"
lblProductName.Text = "My Wonderful Application"
btnSysInfo.Visible = False
```

5. Open the modMain.vb file and change the startup form to frmInherited.

```
Application.Run(New frmInherited( ))
```

To test the application

- Run the project, and test the **OK** button that will close the application.

To override the OK button

1. Open the code window for frmInherited.vb.
2. Click **(Overrides)** in the **Class Name** box at the top of the code window. Select the **btnOk_Click** method from the **Method Name** box.
3. Add the following code to the event handler:

```
MessageBox.Show("Closing the inherited form.", "Close")
MyBase.btnOk_Click(sender, e)
```

To test the overridden OK button

- Run the project and test the **OK** button, confirming that the message box is displayed and the form is then closed, returning to the IDE.

To modify the base form

1. Open the design window for the frmBase.vb form.
2. From the Toolbox, add a **Button** control just beneath the btnSysInfo button, setting the following properties of the new button.

Property	Value
Name	**btnCall**
Text	“” (Empty string)
FlatStyle	**Popup**
Image	**C:\Program Files\Microsoft Visual Studio.NET\Common7\Graphics\icons\Comm\Phone01.ico**

3. Double-click the new button to edit its **btnCall_Click** event handler, adding the following line:

```
MessageBox.Show("Dialing help line.", "Help Line")
```

To test the modified application

1. Run the project and confirm that the new button is automatically displayed in the inherited form before exiting the application.
2. Close Visual Studio **.**NET.

Module Strategy

Use the following strategy to present this module:

- Why Use Windows Forms?

  This lesson shows some of the benefits of using Windows Forms rather than Microsoft Visual Basic® forms. This lesson provides an overview because some of these topics are addressed in more detail later in the module. GDI+, printing support, accessibility, and the extensible object model are not covered in any more detail in this module. Extensibility will be covered when user controls are discussed in Module 9, "Developing Components in Visual Basic .NET," in Course 2373, *Programming with Microsoft Visual Basic .NET (Prerelease)*.

- Structure of Windows Forms

  This lesson shows how to use the new Windows Forms object model to create Windows-based applications. Emphasize the object model hierarchy because it is this object-oriented approach that provides the power of Windows Forms. Ensure that students are comfortable with the anatomy of the form code because this may confuse or concern students whose background is limited to Visual Basic.

  The lesson also discusses various form properties, methods, and events, as well as how to handle those events. Finally, dialog boxes are examined to point out the subtle differences between Visual Basic .NET dialog boxes and those of previous versions of the Visual Basic language.

- Using Controls

  This lesson shows how to use some of the new and enhanced controls in a Windows Form. Creating menus with the Menu Designer is not explicitly covered, although you may want to quickly demonstrate the new designer if students have not already used it. The lesson also examines some of the user assistance controls and the new technique for drag-and-drop operations.

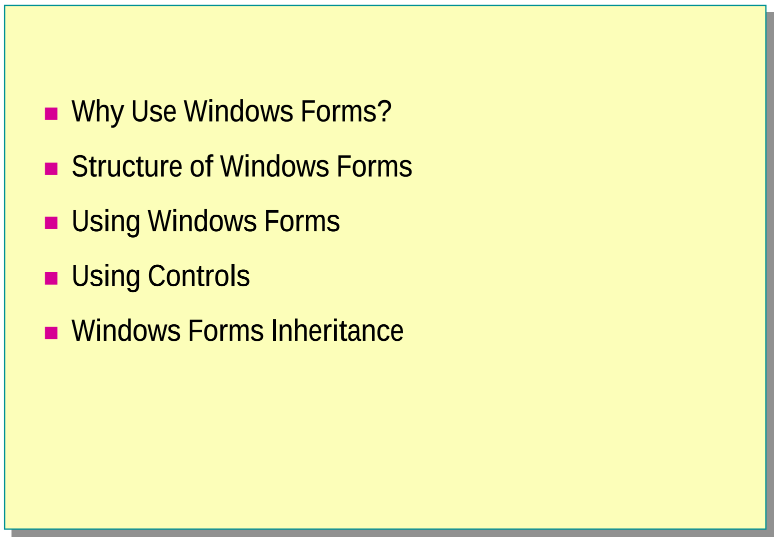
- Windows Form Inheritance

  This lesson shows how to use visual inheritance through the Windows Forms class library. If students are comfortable with the concept of class inheritance, they will have no trouble extending that knowledge to visual inheritance. The process for creating base forms and inherited forms is examined, as are the effects that modification to a base form can have on those that inherit from it.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn about Visual Basic .NET Windows Forms.



- Why Use Windows Forms?
- Structure of Windows Forms
- Using Windows Forms
- Using Controls
- Windows Forms Inheritance

This module describes the new Microsoft® Windows® Forms that are provided by the Microsoft .NET Framework. Windows Forms are the Microsoft Visual Basic® .NET version 7.0 equivalent to Visual Basic forms.

You will learn about the new features available in Windows Forms and how to make changes to forms and controls, and their properties, methods, and events. You will also learn how to create some of the standard Windows dialog boxes. Finally, you will learn about visual inheritance, which allows you to use object-oriented techniques within your forms.

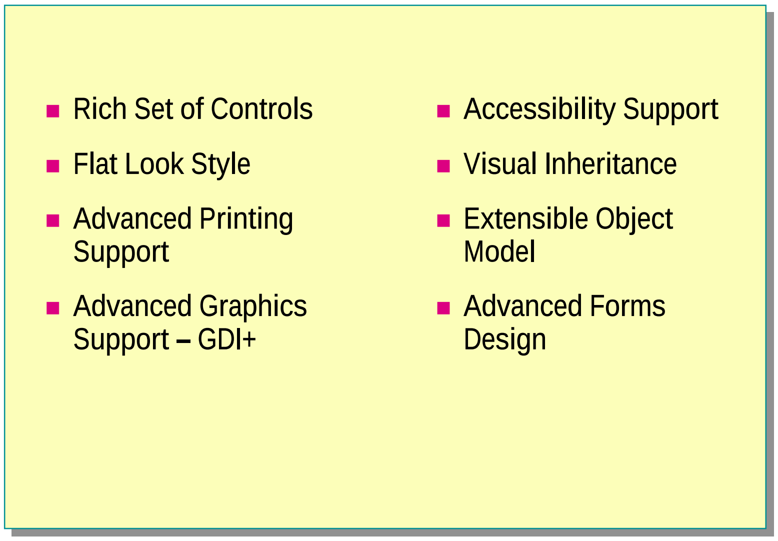
After completing this module, you will be able to:

- Describe the benefits of Windows Forms.
- Use the new properties and methods of Windows Forms.
- Write event-handling code.
- Use the new controls and control enhancements.
- Add and edit menus.
- Create a form that inherits from another form.

Why Use Windows Forms?

Topic Objective
To explain some of the benefits of Windows Forms.

Lead-in
Windows Forms provide many enhancements over previous versions of Visual Basic forms.



- Rich Set of Controls
- Flat Look Style
- Advanced Printing Support
- Advanced Graphics Support – GDI+
- Accessibility Support
- Visual Inheritance
- Extensible Object Model
- Advanced Forms Design

Windows Forms provide many enhancements over standard Visual Basic forms, including:

- Rich set of controls

  By using classes in the **System.Windows.Forms** namespace, you can create Visual Basic .NET applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system. This namespace provides the **Form** class and many other controls that can be added to forms to create user interfaces. Many additional controls are included that were previously only available through external libraries (.ocx's) or third-party products. Some existing controls now allow simple access to properties and methods from the object model instead of requiring complex application programming interfaces (APIs) to perform extended functionality.

- Flat look style

  Windows Forms allow you to create applications that use the new flat look style as seen previously in Microsoft Money 2000.

- Advanced printing support

  Windows Forms provide advanced printing support through the **PageSetupDialog**, **PrintPreviewControl**, **PrintPreviewDialog**, and **PrintDialog** controls.

Delivery Tip
GDI+ allows high-powered graphics to be used in Windows Forms–based applications. This allows developers to use any type of graphics file within their applications and to use techniques such as alpha blending, color gradients, and anti-aliasing.

- Advanced graphics support—GDI+

  The **System.Drawing** namespace provides access to GDI+ basic graphics functionality. GDI+ provides the functionality for graphics in Windows Forms that are accessible in the .NET Framework. More advanced functionality is provided in the **System.Drawing.Drawing2D**, **System.Drawing.Imaging**, and **System.Drawing.Text** namespaces.

  You can take full advantage of these system classes to create applications that provide the user with a richer graphical environment.

- Accessibility support

  Windows Forms provide accessibility properties for controls so that you can develop applications that people with disabilities can use.

- Visual inheritance

  Windows Forms are classes and can benefit from inheritance. Windows Forms can be inherited in derived forms that automatically inherit the controls and code defined by the base form. This adds powerful reuse possibilities to your applications.

- Extensible object model

  The Windows Forms class library is extensible, so you can enhance existing classes and controls with your own functionality.

  You can also create your own designers, similar to the docking or anchoring designers, that will work in the Microsoft Visual Studio® .NET integrated development environment (IDE).

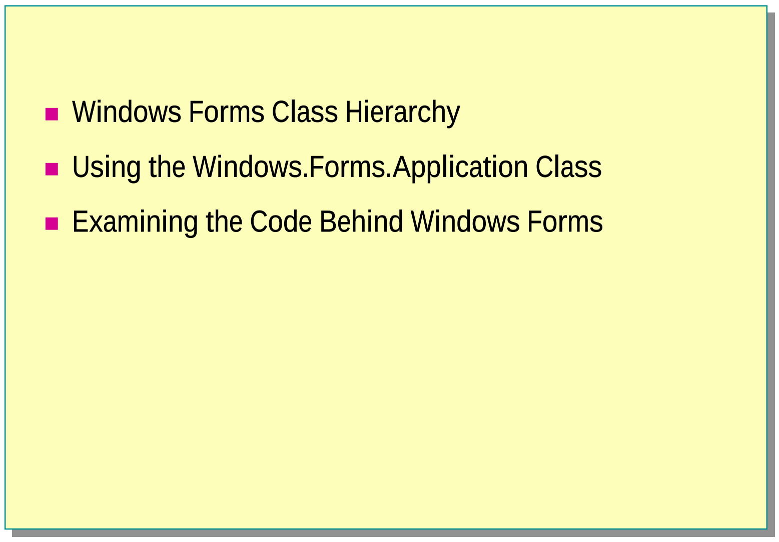
- Advanced forms design

  Developers have traditionally spent much time writing code to handle form resizing, font changes, and scrolling. Windows Forms provide much of this functionality with built-in properties for docking, anchoring, automatic sizing, and automatic scrolling. These new features allow you to concentrate on the functions of your applications.

◆ Structure of Windows Forms

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
Windows Forms at first appear to be similar to Visual Basic 6.0 forms, but there are structural differences.



- Windows Forms Class Hierarchy
- Using the Windows.Forms.Application Class
- Examining the Code Behind Windows Forms

Windows Forms appear similar to Visual Basic 6.0 forms, but the structure of the Windows Form code is different from previous Visual Basic forms. This is because the Windows Forms library in the .NET Framework is object oriented.

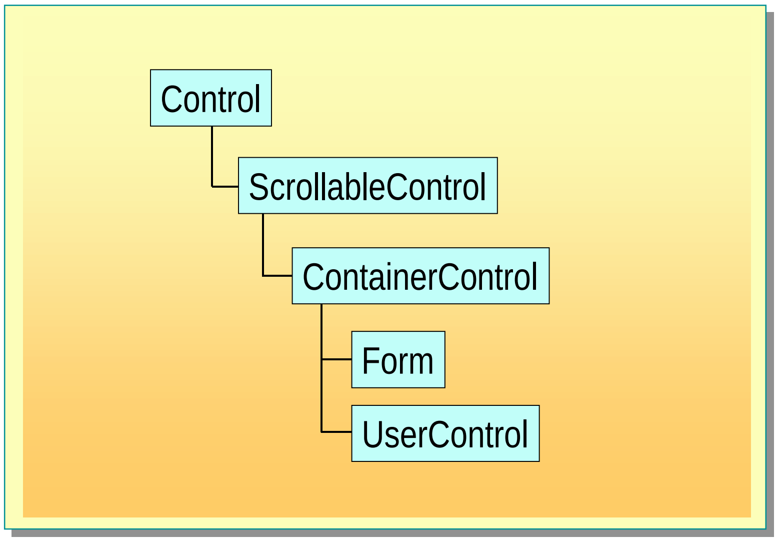
After completing this lesson, you will be able to:

- Describe several of the classes in the Windows Forms class hierarchy.
- Use the **Windows.Forms.Application** class to manage your application at run time.
- Interpret the code generated by Windows Forms.

Windows Forms Class Hierarchy

Topic Objective
To explain some significant classes in the Windows Forms inheritance hierarchy.

Lead-in
The .NET Framework provides many built-in classes for Windows Forms development.



The .NET Framework provides all of the classes that make up Windows Forms–based applications through the **System.Windows.Forms** namespace. The inheritance hierarchy provides many common features across the .NET Windows Forms classes, providing a consistent set of properties and methods for many controls and forms. Some of the classes are examined below.

Delivery Tip
This is not a definitive list of classes, just the relevant ones for this module.

Control

The **Control** class is the fundamental base class for other controls. It provides the basic functionality for a control, such as sizing, visibility, and tab order.

ScrollableControl

The **ScrollableControl** class inherits directly from the **Control** class and provides automatic scrolling capabilities for any control that requires scroll bars.

ContainerControl

The **ContainerControl** class inherits directly from the **ScrollableControl** class and adds tabbing and focus management functionality for controls that can host other controls.

Form

The **Form** class inherits directly from the **ContainerControl** class and represents any window displayed in the application. The properties and methods provided by the **Form** class allow you to display many different types of forms, including dialog boxes and multiple-document interface (MDI) forms. All Windows Forms are derived from this class because it provides the basic functionality required by forms.

UserControl

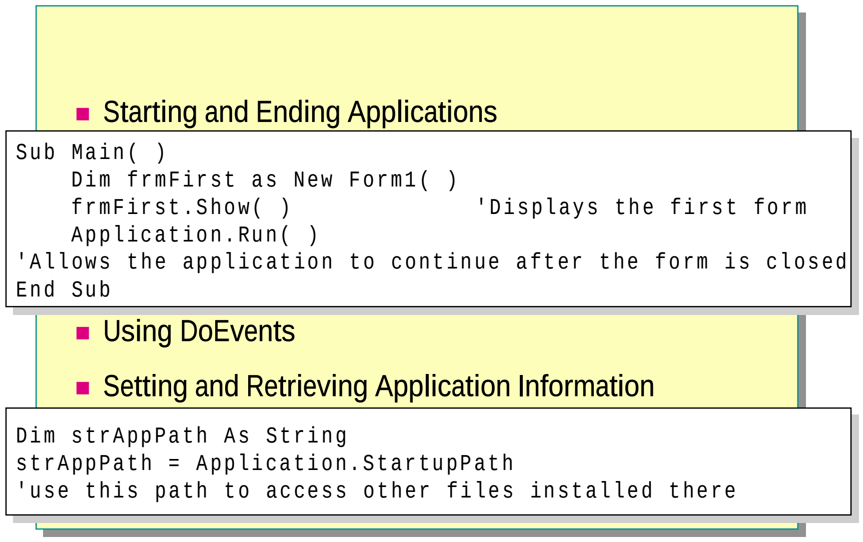
The **UserControl** class also inherits directly from the **ContainerControl** class and provides an empty control that you can use to create your own controls by using the Windows Forms Designer.

Note For information about creating controls, see Module 9, "Developing Components in Visual Basic .NET," in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease).*

Using the Windows.Forms.Application Class

Topic Objective
To explain the purpose of the Windows.Forms. Application class.

Lead-in
The Windows.Forms. Application class manages the application at run time.



- Starting and Ending Applications

```
Sub Main( )
    Dim frmFirst as New Form1( )
    frmFirst.Show( )             'Displays the first form
    Application.Run( )
'Allows the application to continue after the form is closed
End Sub
```

- Using DoEvents
- Setting and Retrieving Application Information

```
Dim strAppPath As String
strAppPath = Application.StartupPath
'use this path to access other files installed there
```

You can use the **Windows.Forms.Application** class for managing your application at run time, in a similar way to using the **App** object in Visual Basic 6.0. You cannot instantiate this class in your code because a single instance exists for the duration of your application at run time.

Starting and Ending Applications

The **Application** object provides methods that you use to start and end your applications. Use the **Run** method to start an application and the **Exit** method to terminate an application.

The **Run** method has an optional parameter that specifies the form to be displayed. If you specify this parameter, the application will end when that form is closed. To enable your application to continue running after the initial form has closed, use the **Show** method of the form before calling the **Run** method of the **Application**. When you use the **Show** method before calling the **Run** method, you must use the **Exit** method to explicitly end your application. Calling this does not run the Close event on your forms, but simply ends the application.

The following example shows how to use the **Application** class to start your application, keep it running after the first form is closed, and end the application. You must remember to change the **Startup Object** property of the project to **Sub Main** for this to work.

```
Sub Main( )
    Dim frmFirst as New Form1( )
    frmFirst.Show( )    ' Displays the first form
    Application.Run( )
' Allows the application to continue after the form is closed
End Sub

Private Sub LastForm_Closing (ByVal sender As Object, ByVal e
As System.ComponentModel.CancelEventArgs) Handles
MyBase.Closing
  ' Any cleanup code for the application
  Application.Exit
End Sub
```

Using DoEvents

The **Application** class also provides the **DoEvents** method. This method is similar to the **DoEvents** function in Visual Basic 6.0, but it is now implemented as a method of the **Application** object.

You use this method to allow other messages in the message queue to be processed during the handling of a single event in your code. By default, when your form handles an event, it processes all code in that event handler and will not respond to other events that may be occurring. If you call the **DoEvents** method in that code, your application will have a chance to handle these other events, such as the repainting of a form that has had another window dragged over it. You will typically use this method within loops to ensure that other messages are processed.

Warning When you use the **DoEvents** method, be careful not to re-enter the same code. This will cause your application to stop responding.

Setting and Retrieving Application Information

The **Application** class contains many useful properties that you can use to set and retrieve application-level information.

You can use the **CommonAppDataRegistry** and **UserAppDataRegistry** properties to set the keys to which shared and user-specific registry information for your application will be written when you are installing an application. After your application is installed, both of these properties are read-only.

The **StartUpPath** property specifies where the running executable file is stored, just like the **App.Path** property in Visual Basic 6.0. You can use this information to access other files that will be installed into the same folder.

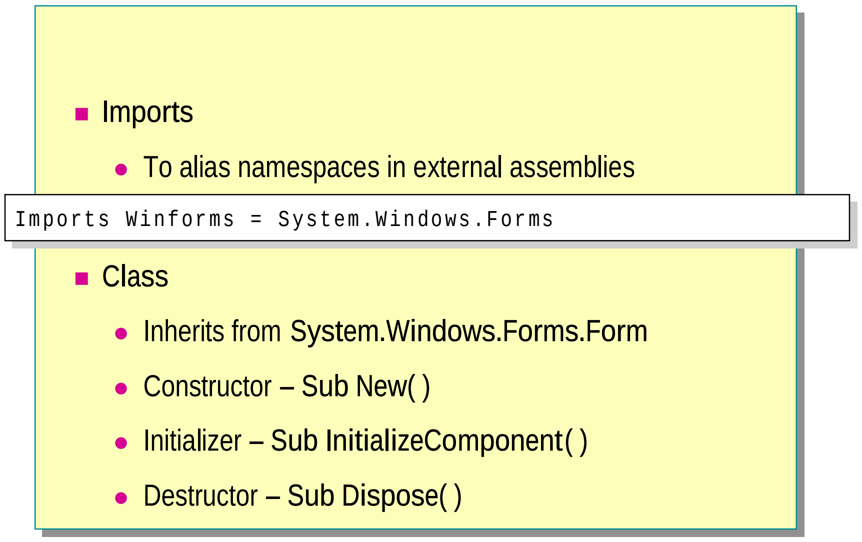
The following example shows how you can use the **StartUpPath** property to provide the installation path of the application:

```
Dim strAppPath As String
strAppPath = Application.StartupPath
'Use this path to access other files installed there
```

Examining the Code Behind Windows Forms

Topic Objective
To explain the basic anatomy of Windows Form code.

Lead-in
The structure of the code behind a Windows Form is different from that of a Visual Basic 6.0 form.



- Imports
  - To alias namespaces in external assemblies

```
Imports Winforms = System.Windows.Forms
```

- Class
  - Inherits from System.Windows.Forms.Form
  - Constructor – Sub New( )
  - Initializer – Sub InitializeComponent( )
  - Destructor – Sub Dispose( )

The structure of the code behind a Windows Form differs from the structure of the code behind a Visual Basic 6.0 form because of the object-orientation of the .NET Framework.

Delivery Tip
It may be useful to open a blank form in Visual Basic .NET and discuss these items in more depth. However, the first demonstration can also be used to perform this task.

Imports

At the top of the code, you may find a list of **Imports** statements, which you can use to provide access to functionality contained within namespaces in referenced external assemblies. If you do not use an **Imports** statement, then all references to classes in external assemblies must use fully qualified names. Using **Imports** allows you to specify an alias to be used for the namespace.

The following example shows how to use the **Imports** statement to declare an alias of *Winforms* for the **System.Windows.Forms** namespace. This statement allows you to use the alias in place of the full name for the rest of the form's code.

```
Imports Winforms = System.Windows.Forms
```

Class

A form is an instance of a class in Visual Basic .NET, so all the code belonging to the form is enclosed within a **Public Class** definition. This structure allows you to implement visual inheritance by creating forms that inherit from other forms.

Inherits System.Windows.Forms.Form

Forms inherit from the **System.Windows.Forms.Form** class. If you create a form in Visual Studio .NET, this inheritance is automatic, but if you create forms elsewhere, you must manually add the **Inherits** statement. This gives you the standard functionality of a form but allows you to override methods or properties as required.

Constructor

In Visual Basic 6.0, you use the **Form_Initialize** and **Form_Load** events to initialize your forms. In Visual Basic .NET, the **Form_Initialize** event has been replaced with the class constructor **Public Sub New**.

Initializer

As in previous versions of Visual Basic, you can assign many property values at design time. These design-time values are used by the run-time system to provide initial values. In Visual Basic 6.0, properties are initialized through the run-time system, and the code is not visible to the developer. In Visual Basic .NET, the Windows Form Designer creates a subroutine called **InitializeComponent** that contains the settings you define in the properties window at design time. This subroutine is called from the class constructor code.

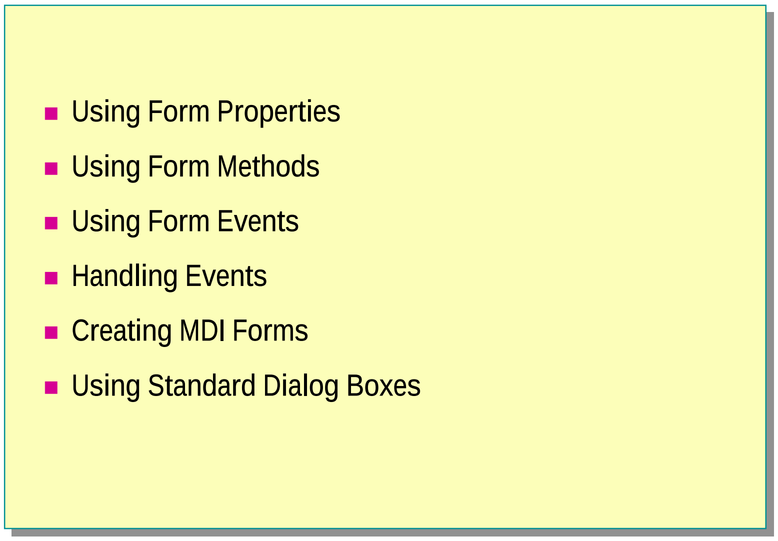
Destructor

In previous versions of Visual Basic, you use the **Form_Terminate** and **Form_Unload** events to provide finalization code. In Visual Basic .NET, these events have been replaced with the class destructor **Public Sub Dispose** and the **Form_Closed** event. When a form is shown non-modally, Dispose is called when the form is closed. When you show forms modally, you must call the Dispose method yourself.

◆ Using Windows Forms

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
Windows Forms supply you with some new functionality.



- Using Form Properties
- Using Form Methods
- Using Form Events
- Handling Events
- Creating MDI Forms
- Using Standard Dialog Boxes

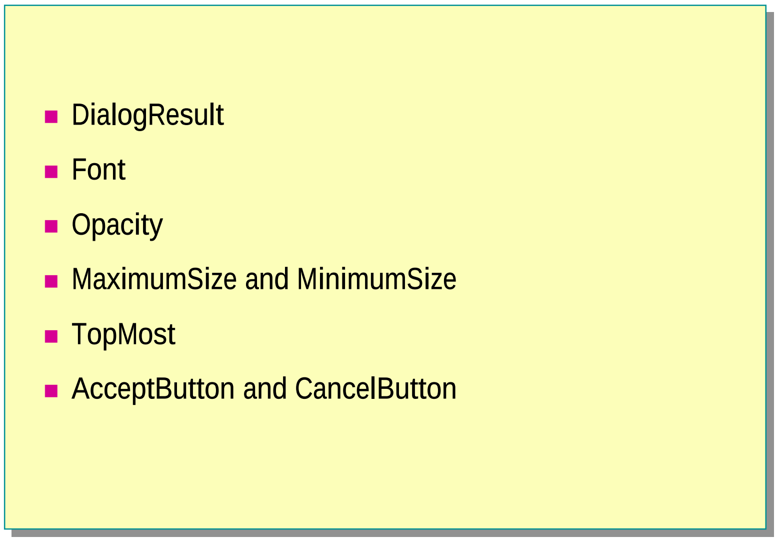
Using Windows Forms is similar to using Visual Basic 6.0 forms, but there are a number of new properties, methods, and events.

In this lesson, you will learn how to use the new form properties, methods, and events. You will also learn how to use MDI forms and standard Windows dialog boxes.

Using Form Properties

Topic Objective
To explain some of the new properties that Windows Forms support.

Lead-in
Windows Forms include many new properties.



- DialogResult
- Font
- Opacity
- MaximumSize and MinimumSize
- TopMost
- AcceptButton and CancelButton

Windows Forms have many new powerful properties that previously would have required API calls to achieve a similar functionality. Many properties are inherited from classes such as the **Control**, **ScrollableControl**, and **ContainerControl** classes, and some properties are defined by the **Form** class itself.

DialogResult

Windows Forms allow you to easily create your own customized dialog boxes. You can create customized dialog boxes by setting the **DialogResult** property for buttons on your form and displaying the form as a dialog box. Once the form is closed, you can use the **DialogResult** property of the form to determine which button was clicked.

The following example shows how to use the **DialogResult** property of a Windows Form:

```
Form1.ShowDialog( )
'The DialogResult property is updated when a button is pressed
and the form closed
If Form1.DialogResult = DialogResult.Yes Then
   'Do something
End If
Form1.Dispose( )
```

Font

The **Font** property of a Windows Form behaves slightly differently than that of a Visual Basic 6.0 form. Controls inherit **Font.BackColor** and **Font.ForeColor** from their parent control. If the font is not set on a control, then the control inherits the font from the parent. This allows you to change the font on the form, and have all controls on the form automatically pick up that new font.

Opacity

By default, all Windows Forms are 100% opaque. In Windows 2000, it is possible to create forms that are transparent or translucent. You can do this by changing the **Opacity** property of a form. This holds a double value between 0 and 1, with 1 being opaque and 0 being transparent.

The following example shows how to make a form 50% opaque:

```
Me.Opacity = 0.5
```

MaximumSize and MinimumSize

These two properties allow you to define maximum and minimum sizes of a form at run time. Their data type is **Size**, which has a **Height** property and a **Width** property to define a total size of the form.

The following example shows how to use these properties:

```
Dim MaxSize As New Size( )
Dim MinSize As New Size( )
MaxSize.Height = 500
MaxSize.Width = 500
MinSize.Height = 200
MinSize.Width = 200
Me.MaximumSize = MaxSize
Me.MinimumSize = MinSize
```

TopMost

The **TopMost** property allows your form to remain on top of all other windows, even when it does not have the focus. This is what the Windows Task Manager does by default. In previous versions of Visual Basic, this frequently used feature can be achieved only by using API calls. In Visual Basic .NET, you simply assign a **Boolean** property of a Windows Form.

The following example shows how to toggle the **TopMost** property:

```
Me.TopMost = Not Me.TopMost
```

AcceptButton and CancelButton

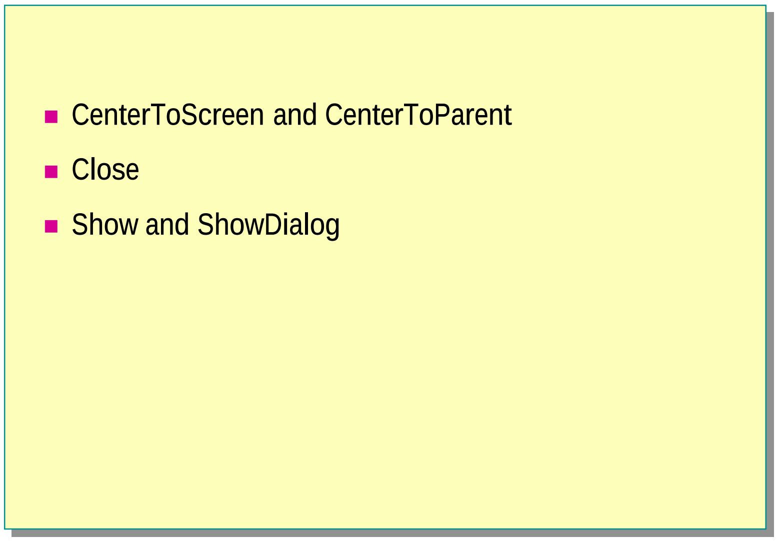
The **AcceptButton** and **CancelButton** properties of a Windows Form allow you to specify which buttons should be activated when the ENTER and ESC keys are pressed, like setting the **Default** and **Cancel** properties of **CommandButtons** in Visual Basic 6.0. The following example shows how to specify your **OK** and **Cancel** buttons as the **AcceptButton** and **CancelButton**:

```
Me.AcceptButton = btnOK
Me.CancelButton = btnCancel
```

Using Form Methods

Topic Objective
To explain how to use some Windows Forms methods.

Lead-in
Windows Forms support several new methods.



- CenterToScreen and CenterToParent
- Close
- Show and ShowDialog

Windows Forms provide several new methods in addition to supporting some existing methods from previous versions of Visual Basic, such as **Hide** and **Refresh**.

CenterToScreen and CenterToParent

You can use these two methods to center your forms on the screen or on the parent of the form. If you use the **CenterToParent** method on a top-level form, it will be centered to the screen. Neither method takes any arguments.

Close

This method is similar to the **Unload** method in Visual Basic 6.0. You can use it to close the current form and release any resources it is holding. The following example shows how to use the **Close** method of a Windows Form:

```
If blnEndApp = True Then
  Me.Close( )
End If
```

Show and ShowDialog

You can use these methods to display a form on the screen. The **Show** method simply displays the form by setting its **Visible** property to **True**. The **ShowDialog** method displays the form as a modal dialog box.

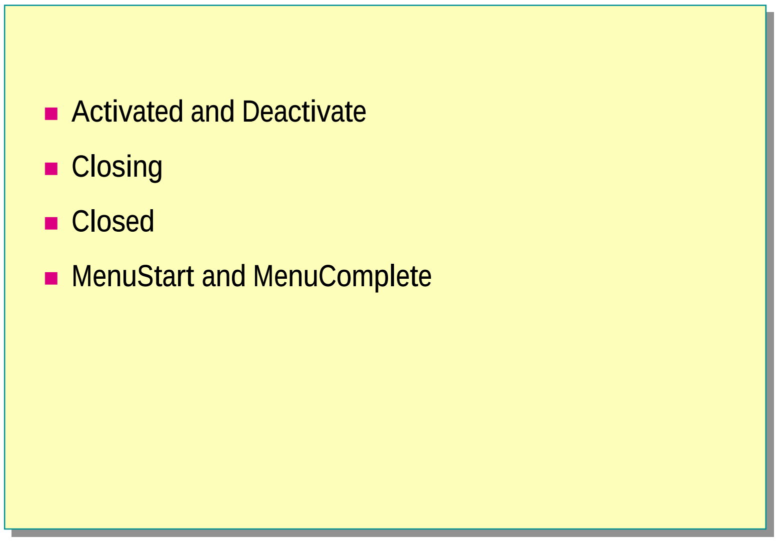
The following example shows how to display a form as a modal dialog box and how to use the **DialogResult** property of the form to determine the action to be taken:

```
Dim frm2 As New Form2( )
frm2.ShowDialog( )
If frm2.DialogResult = DialogResult.OK Then
    MessageBox.Show("Processing request")
ElseIf frm2.DialogResult = DialogResult.Cancel Then
    MessageBox.Show("Cancelling request")
End If
frm2.Dispose( )
```

Using Form Events

Topic Objective
To explain some of the form events that are in the Windows Forms framework.

Lead-in
The Windows Forms framework provides a series of familiar and not–so-familiar form events.



- Activated and Deactivate
- Closing
- Closed
- MenuStart and MenuComplete

Many events from previous versions of Visual Basic are unchanged in Visual Basic .NET, such as mouse and focus events (although these events do have different parameters). Several events have been replaced with slightly different events from the .NET Framework to become standard across the .NET-compatible languages. A number of new events have also been added to allow further flexibility when designing Windows Forms–based applications. In this topic, you will take a closer look at these changes.

Activated and Deactivate

The **Activated** event is raised when the form is activated by code or by user interaction, and the **Deactivate** event is raised when the form loses focus. In Visual Basic 6.0, the **Activate** event was raised only when the form was activated from within the same application. In Visual Basic .NET, it is raised whenever the form is activated, regardless of where it is activated from. You can use this event to ensure that a particular control is always selected when you activate a form.

The following example shows how to use the **Activated** event to select the text in a text box:

```
Private Sub Form2_Activated(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles MyBase.Activated
  TextBox1.Focus( )
  TextBox1.SelectAll( )
End Sub
```

Closing

This event is similar to the Visual Basic 6.0 **Unload** event. It occurs when the form is being closed and allows you to cancel the closure through the use of the **CancelEventArgs** argument.

The following example shows how to use the **Closing** event to query whether the user wants to end the application:

```
Private Sub Form1_Closing(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
  If MessageBox.Show("Do you really want to close this form?",
  "Closing", MessageBoxButtons.YesNo) = DialogResult.No Then
      e.Cancel( ) = True
  End If
End Sub
```

Closed

The **Closed** event occurs after the **Closing** event but before the **Dispose** method of a form. You can use it to perform tasks such as saving information from the form.

The following example shows how to use the **Closed** event to store information in a global variable:

```
Private Sub Form2_Closed(ByVal sender As System.Object, ByVal
e As System.EventArgs) Handles MyBase.Closed
  strName = "Charlie"
End Sub
```

MenuStart and MenuComplete

These two events are raised when a menu receives and loses focus. You can use these events to set properties of the menu items, such as the **Checked** or **Enabled** property.

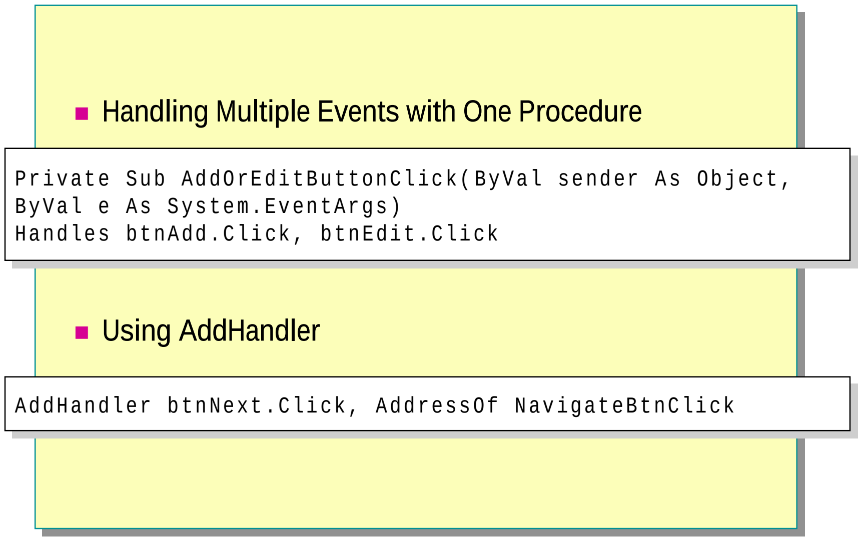
The following example shows how to enable and disable menu items based on the type of control on the form that currently has the focus:

```
If TypeOf (ActiveControl) Is TextBox Then
  mnuCut.Enabled = True
  mnuCopy.Enabled = True
Else
  mnuCut.Enabled = False
  mnuCopy.Enabled = False
End If
```

Handling Events

Topic Objective
To explain how to handle events in Windows Forms.

Lead-in
Event handling has changed significantly since Visual Basic 6.0.



- Handling Multiple Events with One Procedure

```
Private Sub AddOrEditButtonClick(ByVal sender As Object,
ByVal e As System.EventArgs)
Handles btnAdd.Click, btnEdit.Click
```

- Using AddHandler

```
AddHandler btnNext.Click, AddressOf NavigateBtnClick
```

Delivery Tip
Point out the benefits of the different approaches, particularly that of the ability to handle multiple control events from the one handler.

Ensure that students understand the two common arguments that are sent to event handlers:
The sender argument: the object that raised the event.
The e argument: common types of arguments, such as EventArgs or MouseEventArgs.

In previous versions of Visual Basic, you create event handlers by selecting the object and event from the **Object** and **Procedure** boxes in the Code Editor. You can create event handlers in Visual Basic .NET the same way, although to create some of the common event handlers for forms, you need to access the (Base Class Events) group in the **Object** box. You can also add event handlers programmatically by using the **AddHandler** keyword.

Handling Multiple Events with One Procedure

Events can also be handled by any procedure that matches the argument list of the event, also referred to as its *signature*. This allows you to handle events from multiple controls within one procedure, reducing code duplication in a similar way that control arrays do in previous versions of Visual Basic.

You can achieve this functionality by using the **Handles** keyword in conjunction with controls that are declared using the **WithEvents** keyword.

The following example shows how to handle multiple events programmatically with one procedure:

```
Private Sub AddOrEditButtonClick(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
btnAdd.Click, btnEdit.Click
  btnFirst.Enabled = False
  btnLast.Enabled = False
  btnNext.Enabled = False
  btnPrevious.Enabled = False
  btnSave.Enabled = True
  btnCancel.Enabled = True
End Sub
```

Note The *signature* of an event is the list of variables passed to an event-handling procedure. For a procedure to handle multiple events, or to handle events from multiple controls, the argument list must be identical for each event or else a compilation error will occur.

Using AddHandler

The **AddHandler** keyword allows you to add event handling to your form or control at run time by using one of two techniques, as is described for classes in Module 5, “Object-Oriented Programming in Visual Basic .NET,” in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease)*. It is similar to the **Handles** keyword in that it also allows you to use one event-handling procedure for multiple events or multiple controls. With **AddHandler**, however, you do not need to declare the control variable by using the **WithEvents** modifier. This allows a more dynamic attaching of events to handlers.

The following example shows how to use the **AddHandler** keyword to assign control events to procedure:

```
Private Sub NavigateBtnClick(ByVal sender As System.Object,
ByVal e As System.EventArgs)
  MessageBox.Show("Moving record")
End Sub

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles MyBase.Load
  AddHandler btnNext.Click, AddressOf NavigateBtnClick
  AddHandler btnPrevious.Click, AddressOf NavigateBtnClick
End Sub
```

Note The **RemoveHandler** keyword removes an event handler from a form or control’s event. For more information about **RemoveHandler**, see Module 5, “Object-Oriented Programming in Visual Basic .NET,” in Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease).*

Practice: Using Form Events

Topic Objective
To practice using form events.

Lead-in
In this practice, you will create a simple form in order to observe the events of the form.

In this practice, you will create a Windows-based application containing a single form that displays event information in the Debug Output window.

To create the application

1. Open Visual Studio .NET.
2. On the **File** menu, select **New**, and then click **Project**. Set the location to *install folder*\Practices\Mod06, and rename the solution FormEvents.
3. Create event handlers for the following form events, and enter the specified code in the code window.

Event	Code
Form1_Activated	`Debug.WriteLine("Activated")`
Form1_Closed	`Debug.WriteLine("Closing")`
Form1_Deactivate	`Debug.WriteLine("Deactivated")`
Form1_SizeChanged	`Debug.WriteLine("Size changed")`

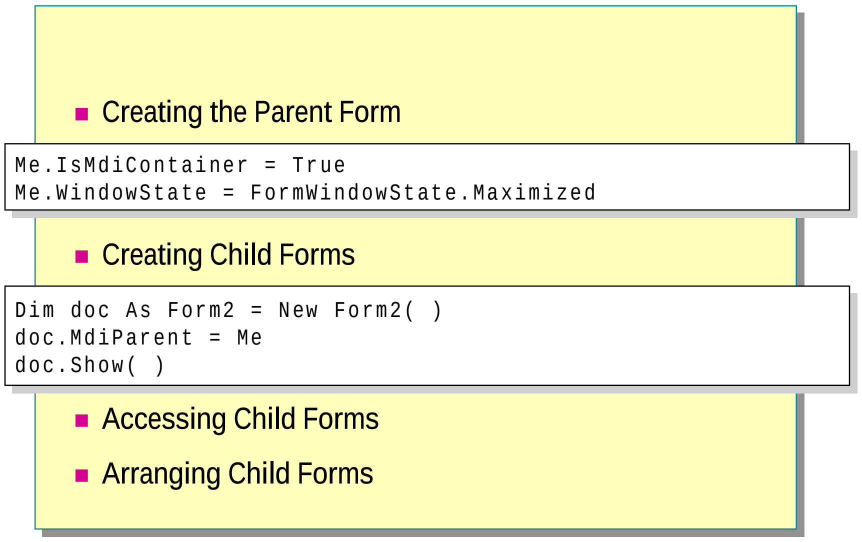
To test the application

1. On the **Debug** menu, click **Start**.
2. On the **View** menu, point to **Other Windows**, and then click **Output** to display the Debug Output window.
3. Perform the following actions on the form: **Resize**, **Minimize**, **Restore**, and **Close**. (Ensure that you can view the activity in the Debug Output window as you perform each action.)
4. Close Visual Studio .NET.

Creating MDI Forms

Topic Objective
To explain how to create MDI applications.

Lead-in
Creating MDI applications is a common task for a Visual Basic developer.



- Creating the Parent Form

```
Me.IsMdiContainer = True
Me.WindowState = FormWindowState.Maximized
```

- Creating Child Forms

```
Dim doc As Form2 = New Form2( )
doc.MdiParent = Me
doc.Show( )
```

- Accessing Child Forms
- Arranging Child Forms

Creating multiple-document interface (MDI) applications is a common task for Visual Basic developers. There have been a number of changes to this process in Visual Basic .NET, although the basic concepts of a parent forms and child forms remain the same.

Creating the Parent Form

You can use the **IsMdiContainer** property of a form to make it an MDI parent form. This property holds a **Boolean** value and can be set at design time or run time.

The following example shows how to specify a form as an MDI parent and maximize it for easy use.

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles MyBase.Load
  Me.IsMdiContainer = True
  Me.WindowState = FormWindowState.Maximized
End Sub
```

Creating Child Forms

You can create child forms by setting the **MdiParent** property of a form to the name of the already-created MDI parent.

The following example shows how to create an MDI child form. This procedure could be called from the **Form_Load** procedure, and a New Document menu item. It uses a global variable to store the number of child windows for use in the caption of each window.

```
Private Sub AddDoc( )
  WindowCount = WindowCount + 1
  Dim doc As Form2 = New Form2( )
  doc.MdiParent = Me
  doc.Text = "Form" & WindowCount
  doc.Show( )
End Sub
```

Accessing Child Forms

It is common to use menus on the MDI parent form to manipulate parts of the MDI child forms. When you use this approach, you need to be able to determine which is the active child form at any point in time. The **ActiveMdiChild** property of the parent form identifies this for you.

The following example shows how to close the active child form:

```
Private Sub mnuFileClose_Click(ByVal sender As Object, ByVal e
As System.EventArgs) Handles mnuFileClose.Click
  Me.ActiveMdiChild.Close( )
End Sub
```

Arranging Child Forms

You can use the **LayoutMdi** method of the parent form to arrange the child forms in the main window. This method takes one parameter that can be one of the following:

- **MdiLayout.Cascade**
- **MdiLayout.ArrangeIcons**
- **MdiLayout.TileHorizontal**
- **MdiLayout.TileVertical**

The following examples show how to use these settings:

```
Private Sub mnuWindowCascade_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
mnuWindowCascade.Click
  Me.LayoutMdi(MdiLayout.Cascade)
End Sub

Private Sub mnuWinArrIcons_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
mnuWinArrIcons.Click
  Me.LayoutMdi(MdiLayout.ArrangeIcons)
End Sub

Private Sub mnuWinTileHoriz_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
mnuWinTileHoriz.Click
  Me.LayoutMdi(MdiLayout.TileHorizontal)
End Sub

Private Sub mnuWinTileVert_Click(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
mnuWinTileVert.Click
  Me.LayoutMdi(MdiLayout.TileVertical)
End Sub
```

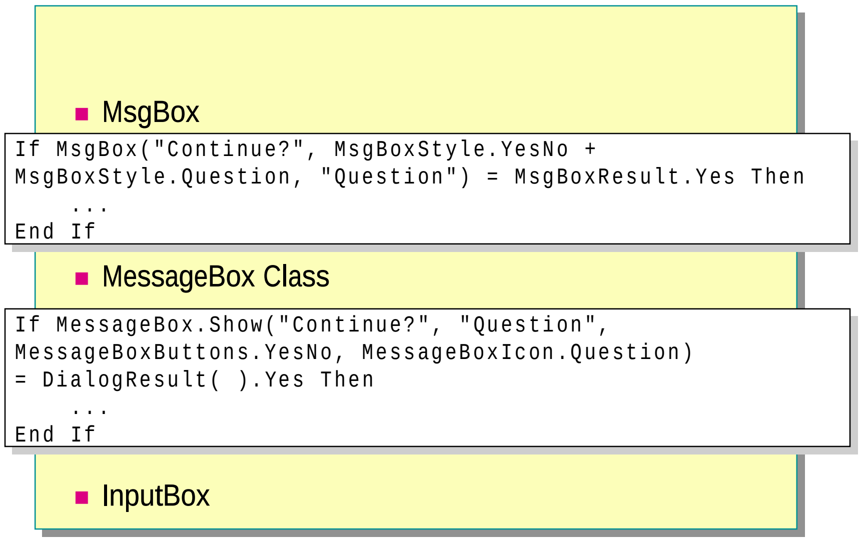
Creating a Window List

In previous versions of Visual Basic, you can set the **WindowList** property of a menu to create a list of child forms at the bottom of that menu. In Visual Basic .NET, you can achieve this functionality by setting the **MdiList** property of a menu.

Using Standard Dialog Boxes

Topic Objective
To explain how to display standard dialog boxes.

Lead-in
There are several ways to display a dialog box and retrieve input from a user.



- MsgBox

```
If MsgBox("Continue?", MsgBoxStyle.YesNo +
MsgBoxStyle.Question, "Question") = MsgBoxResult.Yes Then
    ...
End If
```

- MessageBox Class

```
If MessageBox.Show("Continue?", "Question",
MessageBoxButtons.YesNo, MessageBoxIcon.Question)
= DialogResult( ).Yes Then
    ...
End If
```

- InputBox

Modal forms or dialog boxes require that users close the window before they can continue interacting with other windows in the application. You can create them in any of three different ways.

MsgBox

The traditional **MsgBox** function used by Visual Basic developers is still provided in the .NET Framework. You use the same syntax that you used in previous versions, except you define the display style by the **MsgBoxStyle** enumeration and the resulting user decision by the **MsgBoxResult** enumeration. The following example shows how to use the **MsgBox** function:

```
If MsgBox("Continue?", _
         MsgBoxStyle.YesNo + MsgBoxStyle.Question, _
         "Question") _
      = MsgBoxResult.Yes Then
    ...
End If
```

MessageBox Class

In the .NET Framework, you use the **MessageBox** class for displaying a simple message in a dialog box. It provides a **Show** method and integer constants for controlling the display style of the message box. You can compare the resulting user decision to the **System.Windows.Forms.DialogResult** enumeration, as shown in the following example:

```
If MessageBox.Show("Continue?", "Question", _
      MessageBoxButtons.YesNo, MessageBoxIcon.Question) _
      = DialogResult.Yes Then
    ...
End If
```

The **Show** method allows extra flexibility by allowing you to optionally specify a different form as the owner of the dialog box.

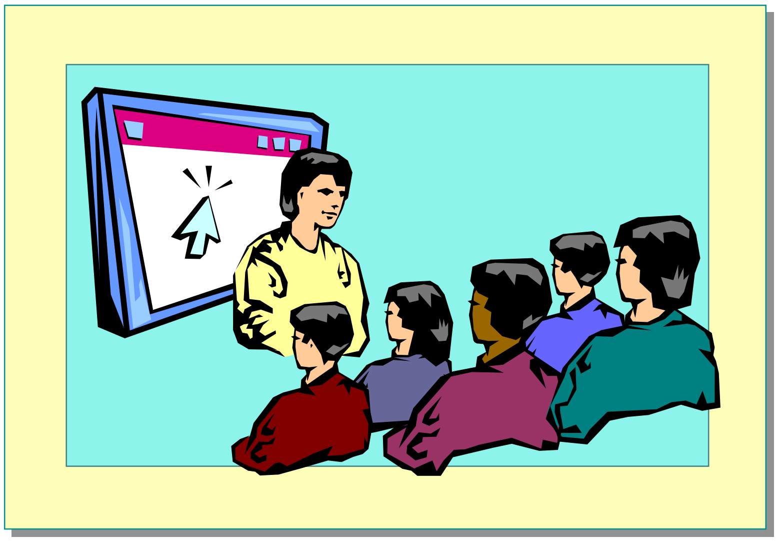
InputBox

The **InputBox** function is still supported in Visual Basic .NET and has not changed from previous versions of Visual Basic.

Demonstration: Manipulating Windows Forms

Topic Objective
To demonstrate how to manipulate Windows Forms.

Lead-in
This demonstration shows how to use properties and methods of a Windows Form.



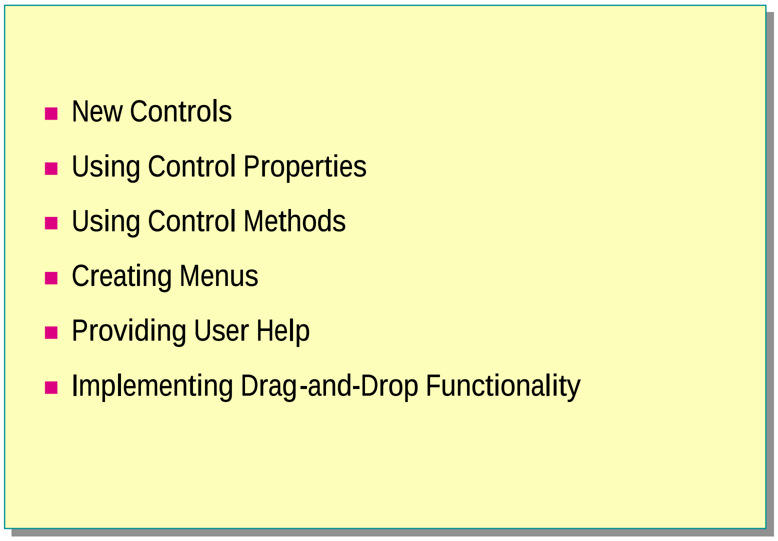
Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to use the properties and methods of a Windows Form, including owner forms, opacity, and automatic scrolling.

◆ Using Controls

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
Many controls have undergone enhancements in Visual Basic .NET, and new controls have been added to the Toolbox.



- New Controls
- Using Control Properties
- Using Control Methods
- Creating Menus
- Providing User Help
- Implementing Drag-and-Drop Functionality

Visual Basic .NET introduces several new controls and many enhancements to the way you use existing controls.

After completing this lesson, you will be able to:

- Describe the new controls in the developer's Toolbox.
- Apply new properties and methods to existing controls.
- Use menus to improve user interaction with your application.
- Implement a Help system for your application.
- Create drag-and-drop operations.

New Controls

Topic Objective
To explain some of the new controls in Visual Basic .NET.

Lead-in
There are several new controls available in Visual Basic .NET.

- CheckedListBox
- LinkLabel
- Splitter
- ToolTip
- NotifyIcon

Visual Basic .NET provides many controls that will be familiar to Visual Basic developers, in addition to some new controls to help you create your Windows Forms–based applications. There are also some controls provided in the default Toolbox that are only available by using ActiveX® controls in Visual Basic 6.0, such as the **CommonDialog** controls and the Windows common controls library.

CheckedListBox

The **CheckedListBox** control allows you to use a list box with check boxes beside each item. This is a commonly used control in Windows and was previously available through the **Style** property of a standard **ListBox**.

The following example shows how you can use the **CheckedItems** property to access the selected items in the list:

```
Dim intTotalChecked As Integer
For intTotalChecked = 0 To CheckedListBox1.CheckedItems.Count
- 1
  Messagebox.Show(CheckedListBox1.CheckedItems(intTotalChecked
).ToString)
Next
```

LinkLabel

Using the **LinkLabel** control, you can display hyperlinks on a form. You can specify the **Text** of the hyperlink and the **VisitedLinkColor** and **LinkColor** of links. The default event for a **LinkedLabel** control is the **LinkClicked** event. The following example shows how you can use this to display a Web page in a **WebBrowser** control:

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles MyBase.Load
  LinkLabel1.Text = "www.microsoft.com"
  LinkLabel1.LinkColor = Color.Blue
  LinkLabel1.VisitedLinkColor = Color.Purple
End Sub

Private Sub LinkLabel1_LinkClicked(ByVal sender As
System.Object, ByVal e As
System.Windows.Forms.LinkLabelLinkClickedEventArgs) Handles
LinkLabel1.LinkClicked
  AxWebBrowser1.Navigate(LinkLabel1.Text)
End Sub
```

Splitter

Splitter controls have become a common feature of Microsoft applications over the last few years. Visual Basic .NET provides a built-in control to allow the user to resize the different sections of your form without any need for resizing code.

To use the **Splitter** control, you must perform the following steps:

1. Add the control to be resized to a container.
2. Dock the control to one side of the container.
3. Add the **Splitter** to the container.
4. Dock the **Splitter** to the side of the control to be resized.

After completing these steps, when you rest the mouse pointer on the edge of the control, the pointer will change shape and the control can be resized.

ToolTip

In Visual Basic 6.0, most built-in controls have a **ToolTip** property that allows you to attach textual Help to a control. This is implemented by means of the **ToolTip** control in Visual Basic .NET. You can use one **ToolTip** control to implement ToolTips on many controls on your form. The following example shows how to link the ToolTip text to be used with a particular control in the **Form_Load** event:

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles MyBase.Load
  ToolTip1.SetToolTip(Button1, "Click to confirm")
  ToolTip1.SetToolTip(Button2, "Click to cancel")
End Sub
```

NotifyIcon

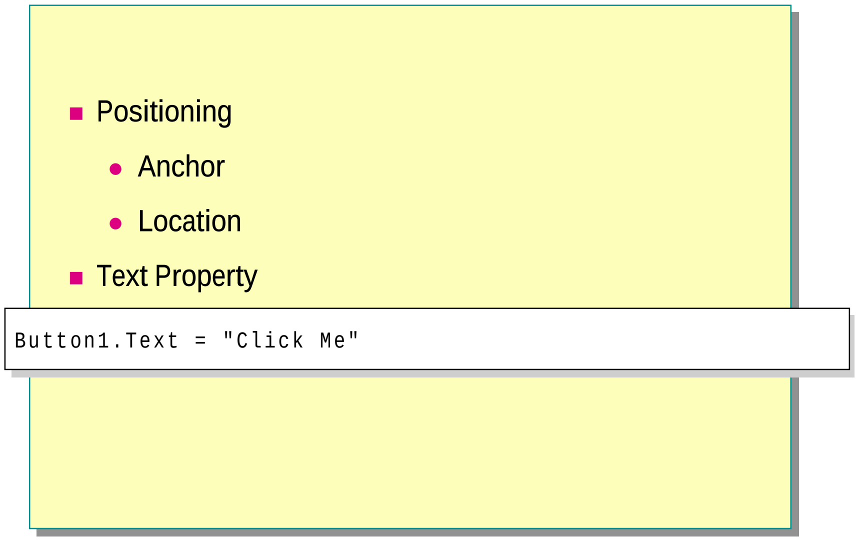
The **NotifyIcon** control is a component that displays an icon in the notification area of the Windows taskbar, like the Windows Volume Control icon. The component is placed in the component tray of the Windows Forms Designer for a particular form. When that form is displayed at run time, the icon will display automatically in the notification area and will be removed when the **Dispose** method of the **TrayIcon** component is called. A **ContextMenu** can be associated with the component so that users can right-click on the icon and select options from the menu.

Note For more information about other new controls, search for "Controls" in the Visual Basic .NET documentation.

Using Control Properties

Topic Objective
To explain some of the common properties that many controls share.

Lead-in
Many controls share some new common properties.



- Positioning
  - Anchor
  - Location
- Text Property

```
Button1.Text = "Click Me"
```

Many of the Windows Forms controls share some new common properties because they inherit from the same base classes.

Positioning

In Visual Basic 6.0, you regularly have to write code to cope with the resizing of a form. If a user maximizes a form at run time, the controls will stay in their original position relative to the top left corner of a form. This means that if you have a set of command buttons—for example, **OK** and **Cancel**—positioned either in the top right corner of a form or across the bottom of a form, you need to write your own code to reposition these controls. In Visual Basic .NET, this type of functionality is built into the controls and form classes.

- **Anchor** property

  In Visual Basic .NET, you can anchor a control to the top, bottom, left, or right side of a form (or any combination). This means that at design time you can use the Properties window to anchor a control, and you no longer need to write repositioning code in the **Resize** event of a form.

- Resizing

  Because you can anchor any or all of the sides of a control, you can effectively resize a control to correspond to the resizing of a form. If you have a form containing a picture box that you want to fill the form, you can anchor it to all sides, and it will remain the same distance from the edges of the form at all times. This feature cannot override the size restrictions applied to some of the Visual Basic .NET controls, such as the height of a combo box.

- **Location** property

  This property allows you to specify the location of a control with respect to the top left corner of its container. The property takes a **Point** data type, which represents an x and y coordinate pair. This property replaces the Top and Left properties used in Visual Basic 6.

Text Property

In earlier versions of Visual Basic, you used different methods to set the text displayed in the various controls. For instance, **Forms** and **Label** controls have a **Caption** property, whereas **TextBox** controls have a **Text** property. In Visual Basic .NET, any textual property of a control is determined by the **Text** property. This provides consistency within Visual Basic, and with the other .NET-compatible languages.

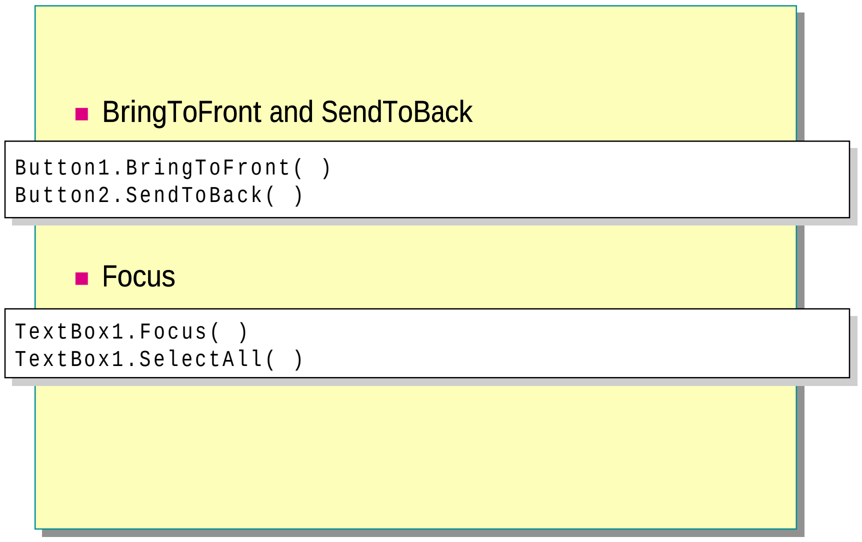
The following example shows how to initialize a **Button** control in the **Form_Load** or **InitializeComponent** procedures.

```
Button1.Top = 20
Button1.Height = 50
Button1.Left = 20
Button1.Width = 120
Button1.Text = "Click Me"
```

Using Control Methods

Topic Objective
To explain some of the new methods common to many controls.

Lead-in
Many controls share some new methods.



- BringToFront and SendToBack

```
Button1.BringToFront( )
Button2.SendToBack( )
```

- Focus

```
TextBox1.Focus( )
TextBox1.SelectAll( )
```

Many of the Windows Forms controls share some new common methods because they inherit from the same base classes.

BringToFront and SendToBack

You can use the **BringToFront** method of a control to place it in front of other controls, and the **SendToBack** method to place it behind all other controls. In earlier versions of Visual Basic, you can achieve this functionality by setting the **ZOrder** property of a control. The following example shows how to rearrange the order of controls at run time:

```
Button1.BringToFront( )
Button2.SendToBack( )
```

Focus

You can use this method to set the focus to a specific control. It is similar to the **SetFocus** method used in Visual Basic 6.0. The following example shows how to check the **Text** property of a **TextBox** control and return focus to the control if the text is not valid:

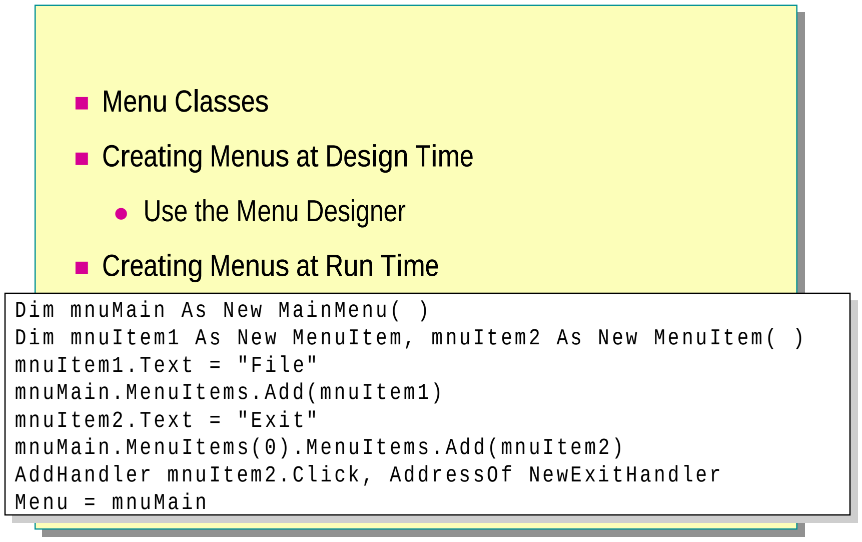
```
If TextBox1.Text <> "password" Then
  MessageBox.Show("Incorrect password")
  TextBox1.Focus( )
  TextBox1.SelectAll( )
End If
```

When trapping focus events, you should use the Enter and Leave events, rather than the GotFocus and LostFocus events.

Creating Menus

Topic Objective
To explain the new changes to menu creation.

Lead-in
Menus have been greatly enhanced in Visual Basic .NET.



- Menu Classes
- Creating Menus at Design Time
  - Use the Menu Designer
- Creating Menus at Run Time

```
Dim mnuMain As New MainMenu( )
Dim mnuItem1 As New MenuItem, mnuItem2 As New MenuItem( )
mnuItem1.Text = "File"
mnuMain.MenuItems.Add(mnuItem1)
mnuItem2.Text = "Exit"
mnuMain.MenuItems(0).MenuItems.Add(mnuItem2)
AddHandler mnuItem2.Click, AddressOf NewExitHandler
Menu = mnuMain
```

In Visual Basic .NET, the process of creating menus is very different from that of Visual Basic 6.0. You can have more than one menu system per form, which reduces the complexity of creating dynamic menus, and you can create **ContextMenus** directly without designing them as top-level menus first.

Menu Classes

There are three main classes that you will use when creating menus:

- **MainMenu**

  You use the **MainMenu** class to create a standard Windows menu bar at the top of a form.

- **ContextMenu**

  You use the **ContextMenu** class to define pop-up menus associated with particular controls.

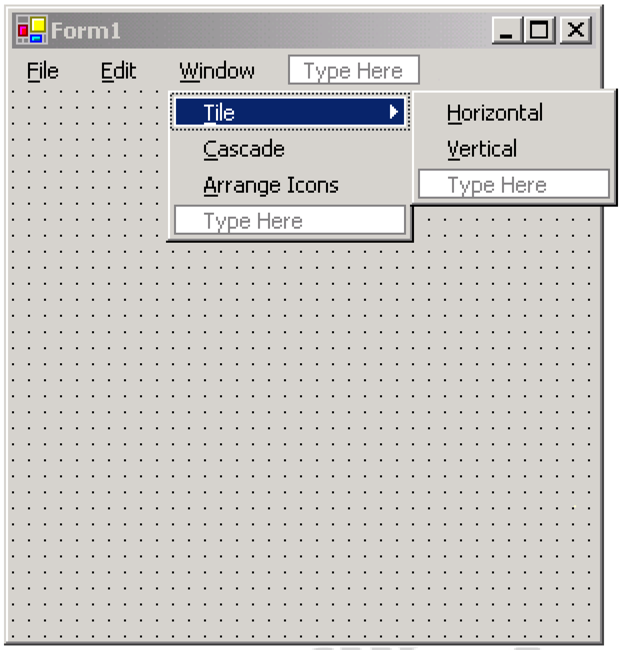
- **MenuItem**

  You use the **MenuItem** class to define menu items within a **MainMenu** or a **ContextMenu**.

Creating Menus at Design Time

Delivery Tip
Demonstrate how to add a menu to a form by using the Menu Designer.
Most students should understand this concept quickly, so this should not be a lengthy discussion.

You can use the Menu Designer to create your menus at design time, which is something you cannot do in Visual Basic 6.0. You can also design and edit your menus in-place, rather than in a separate dialog box.



Creating Menus at Run Time

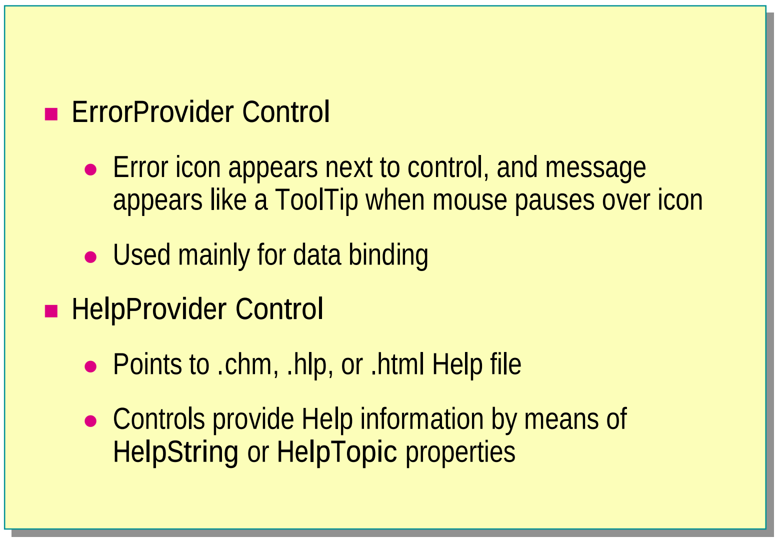
You can add or edit menus at run time by using the **MainMenu**, **ContextMenu**, and **MenuItem** classes. Each of these classes contains a **MenuItems** collection that has **Add** and **Remove** methods. The following example shows how to dynamically create menus:

```
Dim mnuMain As New MainMenu( )
Dim mnuItem1 As New MenuItem( )
Dim mnuItem2 As New MenuItem( )

mnuItem1.Text = "File"
mnuMain.MenuItems.Add(mnuItem1)

mnuItem2.Text = "Exit"
mnuMain.MenuItems(0).MenuItems.Add(mnuItem2)
AddHandler mnuItem2.Click, AddressOf NewExitHandler

Menu = mnuMain
```

Providing User Help

Topic Objective
To explain the ways in which to provide Help.

Lead-in
There are several ways to provide Help in Visual Basic .NET.



- ErrorProvider Control
  - Error icon appears next to control, and message appears like a ToolTip when mouse pauses over icon
  - Used mainly for data binding
- HelpProvider Control
  - Points to .chm, .hlp, or .html Help file
  - Controls provide Help information by means of HelpString or HelpTopic properties

Visual Basic .NET allows you to create user Help in a number of ways by using controls. Each of these controls is placed in the component tray for an individual form.

Delivery Tip
Discuss the relevant methods of each control that are used to set information, as mentioned in the student notes.

ErrorProvider Control

The **ErrorProvider** control indicates to the user that a control has an error associated with it by displaying a small icon near the control. When the user pauses the mouse over the icon, a ToolTip showing the error message appears. **ErrorProvider** can also be used with bound data.

You can set your own error messages manually, as shown in the following example, or when working with bound data, you set the **DataSource** property of the **ErrorProvider** to automatically pick error messages up from the database.

```
Public Sub TextBox1_Validating(ByVal sender As Object, _
  ByVal e As System.ComponentModel.CancelEventArgs) Handles
TextBox1.Validating

    If TextBox1.Text = "" Then
        ErrorProvider1.SetError(TextBox1, _
         "Please enter a value for the text box")
    Else
        ErrorProvider1.SetError(TextBox1, "")
    End If
End Sub
```

The **Validating** event is raised whenever the next control receives focus, providing that the next control has **CausesValidation** property set to **True**, allowing the **Text** property of the control to be tested. If this property contains an empty string, the **ErrorProvider** will display an exclamation icon next to the control and update the ToolTip for the error. If the error message is an empty string, the icon does not appear.

HelpProvider Control

You can use the **HelpProvider** control to display a simple pop-up Help window or online Help from a Help file specified by the **HelpProvider.HelpNamespace** property. This Help is automatically activated when the user presses the F1 Help key while a control has focus.

Implementing Pop-up Help

You can specify pop-up Help at design time by using the **HelpString** property in the Properties window for each control. Each control can have more than one **HelpString** property if the form has more than one **HelpProvider** control, if you use the format *HelpString on HelpProviderControlName*. You can also set the Help string programmatically by using the **SetHelpString** method of the **HelpProvider** control, passing in the control reference and the Help string.

Implementing Online Help

If you specify a Help file, each control can specify the relevant Help topic with the **HelpTopic** property. As for the **HelpString** property, each control can have more than one **HelpTopic** property if the form has more than one **HelpProvider** control, if you use the format *HelpTopic on HelpProviderControlName*. The Help topic can also be set programmatically by using the **SetTopicString** method of the **HelpProvider** control, passing in the control reference and the Help topic string.

Using SetShowHelp

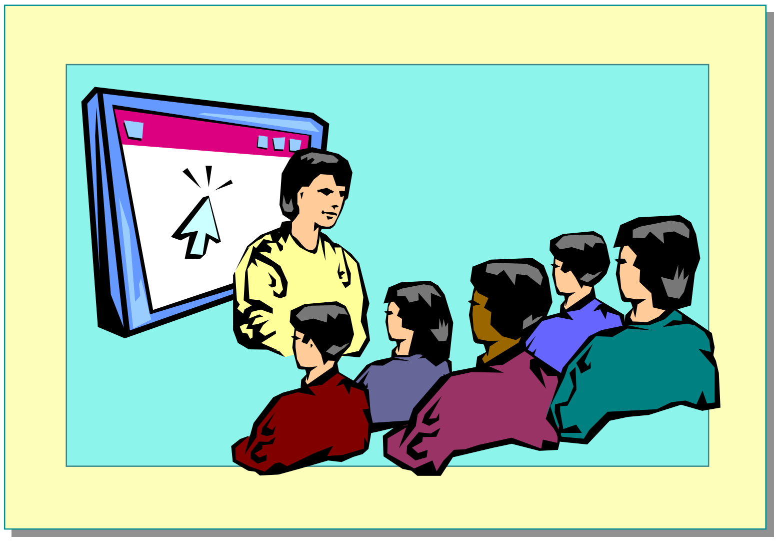
You can also turn Help on or off for an individual control by using the **SetShowHelp** method of the **HelpProvider** control as shown in this example:

```
Sub SetTextboxHelp( )
    HelpProvider1.SetHelpString(TextBox1, "This is my help")
    HelpProvider1.SetShowHelp(TextBox1, True) 'True = On
End Sub
```

Demonstration: Using Controls

Topic Objective
To demonstrate various layout properties and methods of controls, and how to provide simple user assistance.

Lead-in
This demonstration shows the effects of the layout properties of controls, and various other aspects of using controls in Windows Forms.



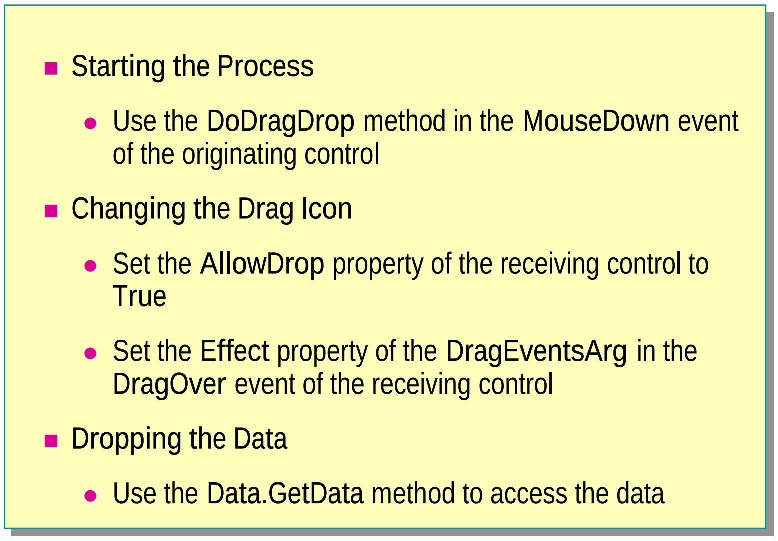
Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to use the layout properties of a **Button** control. You will also learn how to handle control events from multiple controls in one event handler. Finally, you will learn how to provide simple user assistance with the **HelpProvider** and **ToolTip** controls, and how to programmatically create a context menu.

Implementing Drag-and-Drop Functionality

Topic Objective
To explain drag-and-drop drop techniques.

Lead-in
Drag-and-drop techniques have changed significantly in Visual Basic .NET.



- Starting the Process
  - Use the **DoDragDrop** method in the **MouseDown** event of the originating control
- Changing the Drag Icon
  - Set the **AllowDrop** property of the receiving control to **True**
  - Set the **Effect** property of the **DragEventsArg** in the **DragOver** event of the receiving control
- Dropping the Data
  - Use the **Data.GetData** method to access the data

Drag-and-drop techniques in Visual Basic .NET are significantly different from those of previous versions of Visual Basic.

Starting the Process

You can use the **DoDragDrop** method of a control to initiate the dragging and to halt the execution of code until the item is dropped. This method takes two parameters: *data*, which defines the information that is to be dropped, and *allowedEffects* which defines which operations are valid, such as Copy, Move, Link and so on.

The following example shows how to use the **DoDragDrop** method to begin the dragging process:

```
Private Sub TextBox1_MouseDown(ByVal sender As System.Object,
ByVal e As MouseEventArgs) Handles TextBox1.MouseDown
  Dim DragDropResult As DragDropEffects
  If e.Button = MouseButtons.Left Then
      DragDropResult = TextBox1.DoDragDrop(TextBox1.Text, _
      DragDropEffects.All)
      If DragDropResult = DragDropEffects.Move Then
          TextBox1.Text = ""
      End If
  End If
End Sub
```

Changing the Drag Icon

For a control to receive drag-drop notifications, you must set its the **AllowDrop** property to **True**. Without this setting, the **DragDrop**, **DragOver**, **DragEnter**, and **DragLeave** events will not execute.

You can use the **KeyState** property of the **DragEventsArg** argument passed to controls in the **DragOver** event of a control to change the drag icon to an appropriate symbol. This helps the user to know what action they are about to perform. This property is an integer property that specifies which keys (such as SHIFT and CONTROL) are being held down during the drag process.

The following example shows how to set the appropriate icon:

```
Private Sub TextBox2_DragOver(ByVal sender As Object, ByVal e
As DragEventArgs) Handles TextBox2.DragOver
  Select Case e.KeyState
      Case 1
          'No key pressed
          e.Effect = DragDropEffects.Move
      Case 9
          'CONTROL key pressed
          e.Effect = DragDropEffects.Copy
      Case Else
          e.Effect = DragDropEffects.None
  End Select
End Sub
```

Dropping the Data

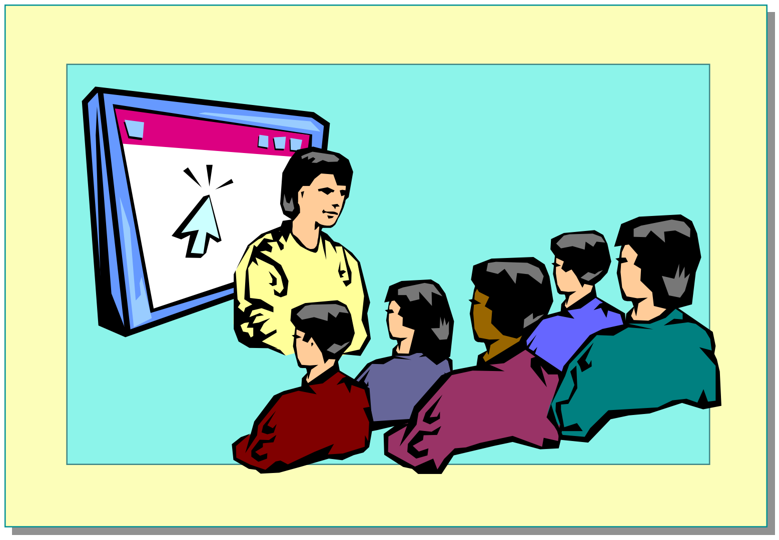
You can drop the data in the **DragDrop** event of the receiving control. The following example shows how to write code to accept textual data from another **TextBox** control:

```
Public Sub TextBox2_DragDrop(ByVal sender As Object, ByVal e
As DragEventArgs) Handles TextBox2.DragDrop
  TextBox2.Text = e.Data.GetData(DataFormats.Text).ToString
End Sub
```

Demonstration: Implementing Drag-and-Drop Functionality

Topic Objective
To demonstrate simple drag-and-drop functionality within an application.

Lead-in
This demonstration shows at how to use drag-and-drop functionality within an application.



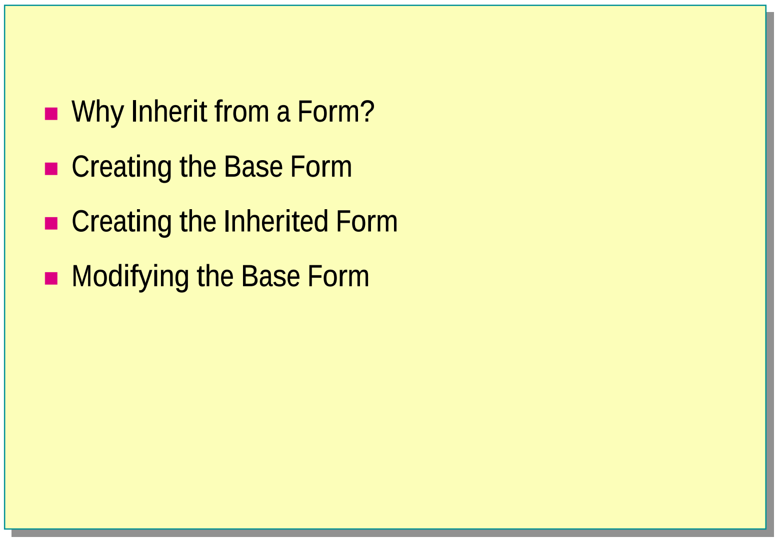
Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to use drag-and-drop operations within a simple application.

◆ Windows Forms Inheritance

Topic Objective
To provide an overview of the topics covered in this lesson.

Lead-in
Visual Basic .NET introduces the concept of visual inheritance to Visual Basic developers.



- Why Inherit from a Form?
- Creating the Base Form
- Creating the Inherited Form
- Modifying the Base Form

Visual Basic .NET introduces the concept of visual inheritance to Visual Basic developers. This type of inheritance can improve code reuse in your applications and provide them with a standard appearance and behavior.

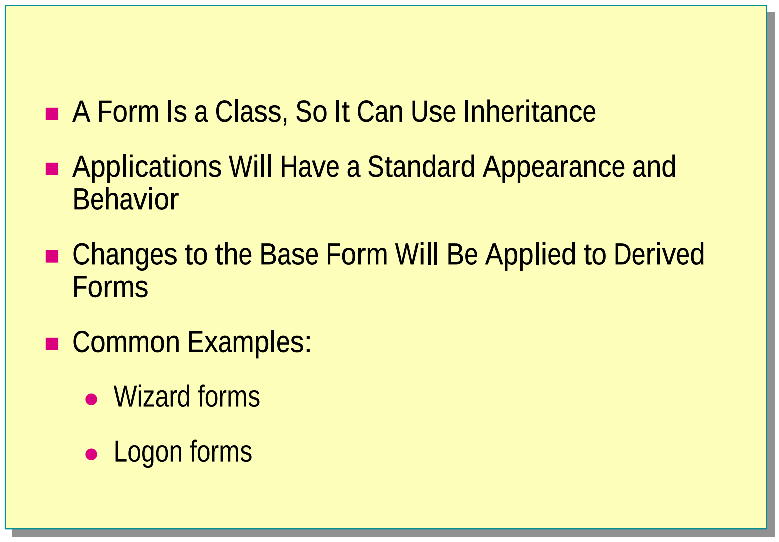
After you complete this lesson, you will be able to use visual inheritance to:

- Create a form that inherits from a given base form.
- Modify a base form from which other forms have inherited.

Why Inherit from a Form?

Topic Objective
To explain the reasons for visual inheritance.

Lead-in
You can improve code reuse by using visual inheritance.



- A Form Is a Class, So It Can Use Inheritance
- Applications Will Have a Standard Appearance and Behavior
- Changes to the Base Form Will Be Applied to Derived Forms
- Common Examples:
  - Wizard forms
  - Logon forms

You will likely need to create forms that are similar to forms you have created before. In previous versions of Visual Basic, you can create templates on which to base your forms. In Visual Basic .NET, you can inherit from existing forms.

Inheriting from a form is as simple as deriving one class from another, because a form is simply a class with an extra visual component. This technique allows you to define a base form that can be derived from in order to create a standard appearance and behavior of your applications. It also shares the same benefits as class inheritance, in that code can be reused from the base form in all of the derived forms.

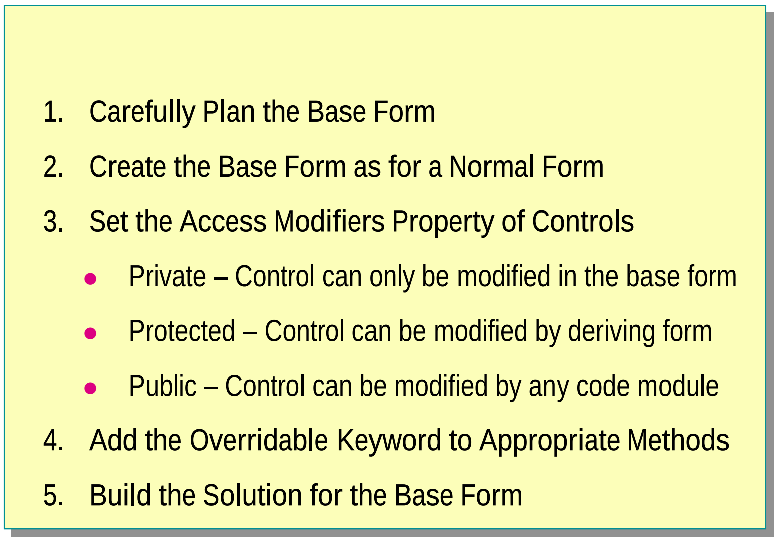
Any changes that you make to the base form can be applied to any of the derived forms, making simple updates to multiple forms easy.

You can use visual inheritance whenever forms behave in a similar way or need to have a standard appearance. Common examples of these types of forms are wizards and logon forms.

Creating the Base Form

Topic Objective
To explain the process for creating a base form.

Lead-in
Creating a base form requires a bit of careful planning.



1. Carefully Plan the Base Form
2. Create the Base Form as for a Normal Form
3. Set the Access Modifiers Property of Controls
   - Private – Control can only be modified in the base form
   - Protected – Control can be modified by deriving form
   - Public – Control can be modified by any code module
4. Add the Overridable Keyword to Appropriate Methods
5. Build the Solution for the Base Form

The base form serves as the template for your standard form. You design and code the form in the usual way, to perform whatever functionality you want to be inherited. After you have created the base form, you can build your solution to make the form accessible, and then inherit from it.

When creating a base form, use the following process:

1. Carefully plan the base form.

   Changes are easier to make before any forms inherit from your base form because making changes afterwards will require extra retesting.

2. Create the base form as you would a normal form.

   Create the base form using the same techniques you would use to create a normal form.

3. Set the access modifiers property of controls.

   - Private controls cannot have their properties modified outside of the base form.

   - Public controls can have their properties modified by any form or code module without restriction.

4. Add the **Overridable** keyword to appropriate methods.

   Any method that can be overridden in a derived form must be marked as overridable in the base form.

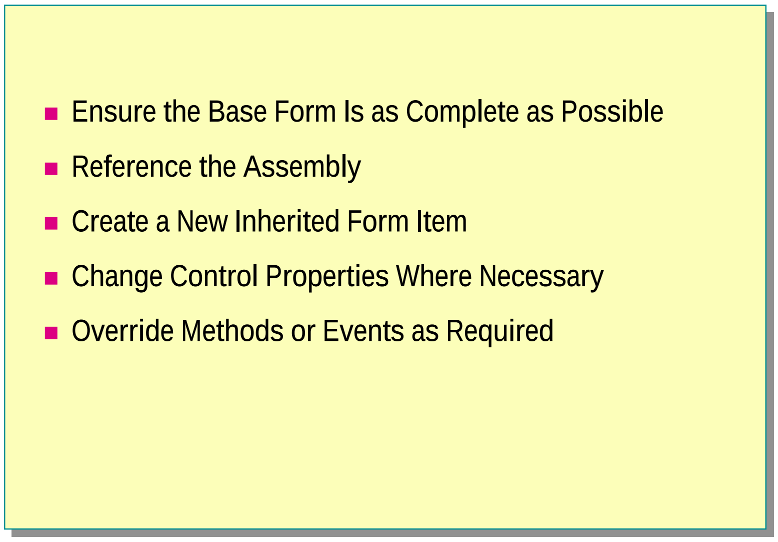
5. Build the solution for the base form.

   You cannot create a form that inherits from a base form until the base form has been built.

Creating the Inherited Form

Topic Objective
To explain how to create an inherited form.

Lead-in
Creating an inherited form is simple.



- Ensure the Base Form Is as Complete as Possible
- Reference the Assembly
- Create a New Inherited Form Item
- Change Control Properties Where Necessary
- Override Methods or Events as Required

After you have designed your base form and built the solution, you are ready to begin deriving forms. To do this, you simply add a new item to the project by clicking **Inherited Form** in the **Add New Item** window. This will run the Inheritance Picker for you.

When inheriting from a base Windows Form, consider the following guidelines carefully:

- Ensure that the base form is as complete as possible.

  Make any last minute changes to the base form before inheriting from it.
- Reference the assembly.

  If the base form is not in the same project, you must make a reference to the appropriate assembly.
- Create a new Inherited Form item.

  Add a new Inherited Form item to your project, selecting the base form in the **Inheritance Picker** dialog box. A list of available base forms is shown, and you can browse for other assemblies.
- Change control properties where necessary.

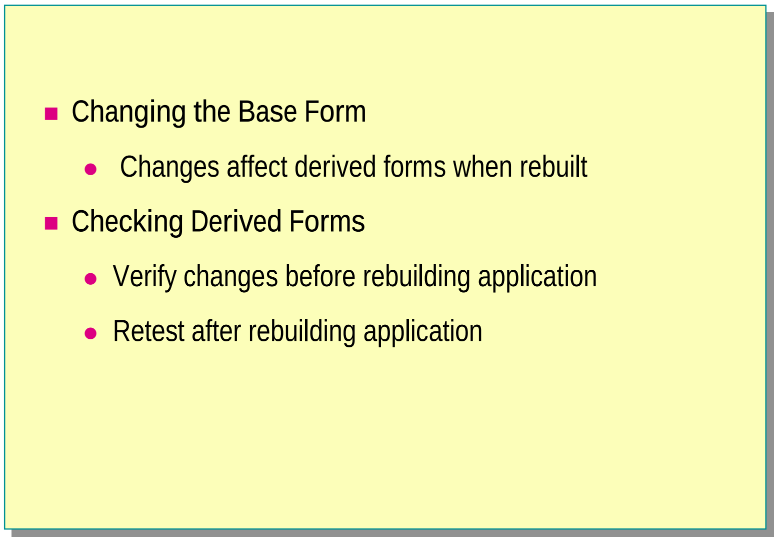
  You can programmatically change public and protected controls, and you can use the Properties window of the Windows Forms Designer for a derived form. Private controls cannot be altered outside of the base form.
- Override methods or events as required.

  If methods or event handlers have been marked as overridable, you can implement your own code in the derived form.

Modifying the Base Form

Topic Objective
To explain the impact of modifying a base form after it has been inherited.

Lead-in
You can modify your base form after it has been inherited.



- Changing the Base Form
  - Changes affect derived forms when rebuilt
- Checking Derived Forms
  - Verify changes before rebuilding application
  - Retest after rebuilding application

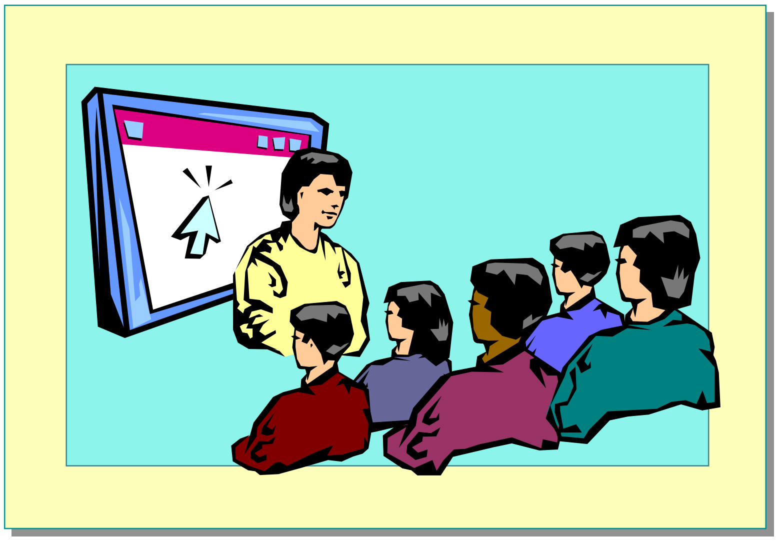
The derived form is linked directly to the base form; it is not a copy of the base form. This means that changes you make to the base form will be reflected in the derived form when the project is rebuilt. You can quickly update a series of forms that contain the same code or visual elements by making the changes in the base form. However, you may find that changes that are valid in the base form can introduce errors into the derived forms.

For example, any overridden method that calls a method on the **MyBase** object may expect a certain behavior, and careful retesting is needed to validate this expectation. This is true of all types of inheritance, not just visual inheritance.

Demonstration: Using Windows Forms Inheritance

Topic Objective
To demonstrate visual inheritance.

Lead-in
This demonstration shows how to achieve visual inheritance by using Windows Forms.



Delivery Tip
The step-by-step instructions for this demonstration are in the instructor notes for this module.

In this demonstration, you will learn how to create a base form specifically for inheritance purposes. You will learn how to inherit from the form and how to override properties and methods of the base form controls. Finally, you will learn how to modify the base form after it has been used for inheritance and learn the effects the base form modifications have on the derived form.

Lab 6.1: Creating the Customer Form

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will complete the Customer form to display a list of customers.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Use Windows Forms in an application.
- Use the layout properties of controls.
- Create menus.
- Provide user assistance by means of ToolTips.

Prerequisites

Before working on this lab, you must have designed forms in previous versions of Visual Basic.

Scenario

In this lab, you will continue working with the Cargo system. The **Customer** class from Lab 5.1 of Course 2373A, *Programming with Microsoft Visual Basic .NET (Prerelease),* has been enhanced for you, and a **CustomerList** class has been provided so you can iterate through the customers. The basic Customer form has been provided for you, but it requires further development.

Starter and Solution Files

There are starter and solution files associated with this lab. The starter files are in the *install folder*\Labs\Lab061\Starter folder, and the solution files are in the *install folder*\Labs\Lab061\Solution folder.

Estimated time to complete this lab: 45 minutes

Exercise 1
Extending the Customer Form

In this exercise, you will enhance the existing Customer form by using the layout properties of the controls and form. The form is currently only intended to retrieve customer information.

To open the starter project

1. Open Visual Studio .NET.
2. On the **File** menu, point to **Open**, and click **Project**. Set the folder location to *install folder*\Labs\Lab061\Starter, click **Lab061.sln**, and then click **Open**.

To add the Sub Main procedure

1. On the **Project** menu, click **Add Module**, and rename the file **modMain.vb**.
2. Open the **modMain.vb** code Editor.
3. Insert the following code.

```
Sub Main( )
    Application.Run(New frmCustomer( ))
End Sub
```

4. In Solution Explorer, right-click **Lab061**, and then click **Properties**.
5. In the **Startup object** box, click **Sub Main**, and then click **OK**.

To view the frmCustomer form

1. Open the frmCustomer.vb design window, and examine the layout of the controls.
2. Open the frmCustomer.vb code window, and examine the existing code.

To test the application

1. Open the **Sub New** procedure in frmCustomer.vb, and set a breakpoint on the following line by using the F9 key:

```
custList = New CustomersList( )
```

2. On the **Debug** menu, click **Start**.
3. Step through the code by using the F11 key (to step into procedures) and the F10 key (to step over procedures) until you understand how the application loads the customer information into the list. Press F5 to resume execution.
4. Click different customers in the list, and observe the resulting behavior.
5. Click **Close** to quit the application, and remove the breakpoint from the **Sub New** procedure.

To set the layout properties of the controls

1. Open the frmCustomer.vb design window, and set the **Anchor** properties of the following controls to the following values in the Properties window.

Control	Anchor value
lstCustomers	**Top, Bottom, Left**
txtID	**Top, Left, Right**
txtEmail	**Top, Left, Right**
txtTitle	**Top, Left, Right**
txtFName	**Top, Left, Right**
txtLName	**Top, Left, Right**
txtAddress	**Top, Left, Right**
txtCompany	**Top, Left, Right**
btnClose	**Bottom, Right**

2. Set the following properties for the form and controls in the Properties window.

Object	Property	Value
txtAddress	**MultiLine**	**True**
txtAddress	**AcceptsReturn**	**True**
txtAddress	**Size.Height**	**60**
frmCustomer	**CancelButton**	**btnClose**

3. Open the frmCustomer.vb code window.
4. Add the following line immediately before the end of the **Sub New** procedure:

```
Me.MinimumSize = Me.Size
```

To test the project

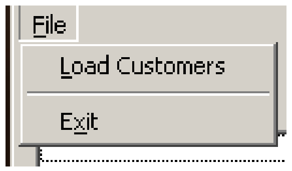
1. Run the project.
2. Resize the form to confirm that all controls are anchored correctly and that the **MinimumSize** property limits the form size so that all controls are visible.
3. Click the **Close** button to quit the application.

Exercise 2
Adding a Menu and ToolTips

In this exercise, you will add a menu and ToolTips to the frmCustomer form.

To add a menu

1. Open the frmCustomer.vb design window.
2. Using the Toolbox, add a **MainMenu** control, renaming it **mnuMain**.
3. Using the Menu Designer, add menu items as shown in the following illustration.



4. Use the following table to name the menu items.

Caption	Name
&File	**mnuFile**
&Load Customers	**mnuFileLoad**
-	**mnuFileSeparator**
E&xit	**mnuFileExit**

5. Create the **Click** event handler for the mnuFileLoad menu item.

6. From the **Sub New** procedure, cut the existing code for loading customers, and paste it into the new event handler (making sure to leave the **MinimumSize** code that was added in the previous exercise as it is). Your code should now look as follows:

```
Public Sub New( )
    MyBase.New( )

    'This call is required by the Windows Forms Designer
    InitializeComponent( )
    Me.MinimumSize = Me.Size
End Sub

Private Sub mnuFileLoad_Click(ByVal sender As _
System.Object, ByVal e As System.EventArgs)
   'Create the customerlist object
   custList = New CustomersList( )

   'Load the customers
   custList.LoadCustomers( )

   'Populate the list box with customers
   PopulateListBox( )
   LoadCustomer(0)
End Sub
```

7. Locate the existing **btnClose_Click** event handler, and rename the procedure **CloseForm**, leaving the arguments unchanged, and adding the following statement after the existing **Handles** clause. This allows both events to be handled by the same procedure.

```
, mnuFileExit.Click
```

8. Save the project.

To add ToolTip user assistance

1. Open the frmCustomer.vb design window.
2. Using the Toolbox, add a **ToolTip** control, renaming it **ttCustomerList**.
3. Using the Properties window, set the **ToolTip** property for lstCustomers to "Select a customer to display the full details."

To test the application

1. On the **Debug** menu, click **Start**.
2. On the **File** menu, click **Load Customers**.
3. Rest the mouse pointer on the **Customer** list to confirm that the ToolTip appears.
4. On the **File** menu, click **Exit** to quit the application.

Exercise 3
Adding a Shortcut Menu

In this exercise, you will programmatically add a shortcut menu for the customer **ListBox** control.

To create the context menu

1. Open the frmCustomer.vb code window, and locate the **Sub New** procedure.
2. After the call to the **InitializeComponent** procedure, declare a **ContextMenu** variable called *cmListBox*, and a **MenuItem** variable called *mItem*.
3. Instantiate the *cmListBox* context menu object by using the default **New** constructor.
4. Add a menu item to the context menu, as shown in the following code:

```
mItem = cmListBox.MenuItems.Add("&Delete")
```

5. Disable this menu item until there are entries in the list box, as shown in the following code:

```
mItem.Enabled = False
```

6. Add an event handler for the new *mItem* object by using the **AddHandler** function, as shown in the following code:

```
AddHandler mItem.Click, AddressOf onDeleteClick
```

7. Assign the new context menu to the **ContextMenu** property of the lstCustomers control, as shown in the following code:

```
lstCustomers.ContextMenu = cmListBox
```

8. Before the **Catch** statement in the **LoadCustomer** procedure, enable the context menu as shown in the following code:

```
lstCustomers.ContextMenu.MenuItems(0).Enabled = True
```

To create the event handler for the context menu item

1. At the end of the form definition, create a new private subroutine called **onDeleteClick** that accepts the following arguments:

```
ByVal sender As Object, ByVal e As System.EventArgs
```

2. Display a message box with the following options specified.

Argument	Value
Text	**Are you sure you want to delete this customer?**
Caption	**Confirm**
Buttons	**MessageBoxButtons.YesNo**
Icon	**MessageBoxIcon.Question**

3. Use an **If** statement to test the result of the **MessageBox.Show** method against the value **DialogResult.Yes**. In the **True** section, enter the following code:

```
custlist.RemoveAt(lstCustomers.SelectedIndex)
PopulateListBox( )
```

4. Insert an **If** statement into the procedure to test to see whether the number of items in lstCustomers is zero. (Hint: Use the **lstCustomers.Items.Count** property).
5. In the **True** section, disable the **Delete** menu item.
6. Save the project.

To test the application

1. On the **Debug** menu, click **Start**.
2. On the **File** menu, click **Load Customers**.
3. Right-click a customer and click **Delete**.
4. When the confirmation message appears, click **Yes**.
5. On the **File** menu, click **Exit** to quit the application.
6. Close and exit Visual Studio .NET.

If Time Permits
Creating an About Box Form Using Visual Inheritance

In this optional exercise, you will create an About box form by inheriting from an existing base form.

To add the base form to the project

1. Open your solution to the previous exercise.
2. On the **Project** menu, click **Add Existing Item**.
3. Click **frmBase.vb** in the starter folder, and click **Open**.
4. On the **Build** menu, click **Rebuild All**.

To inherit the base form

1. On the **Project** menu, click **Add Inherited Form**, renaming the file **frmAbout.vb**.
2. In the **Inheritance Picker** dialog box, click **frmBase**, and then click **OK**.
3. Open the frmAbout.vb design window.
4. Change the **Text** property of the lblProductName control to **Cargo**.

To display the About box form

1. Open the frmCustomer.vb design window.
2. Add the following menus to the mnuMain control.

Caption	Name
&Help	**mnuHelp**
&About…	**mnuHelpAbout**

3. Create the **Click** event handler for the mnuHelpAbout menu item, and add the following code:

```
Dim aboutForm As New frmAbout( )
aboutForm.ShowDialog( )
```

4. Save the project.

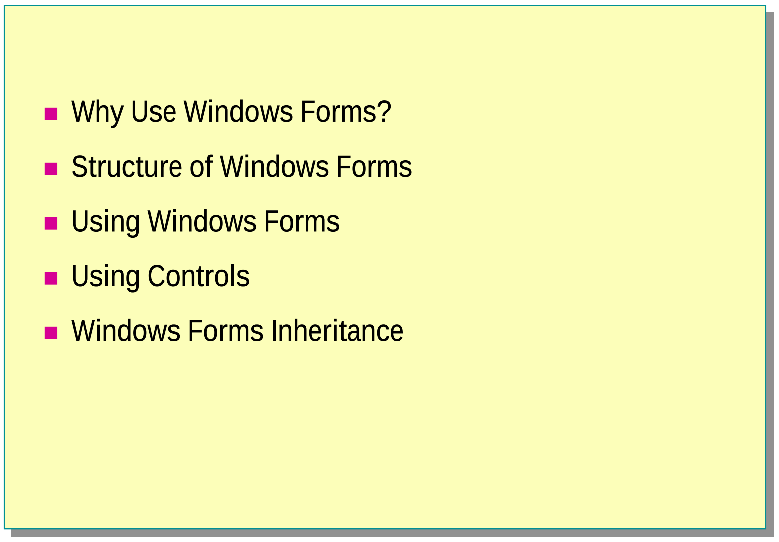
To test the About box form

1. On the **Debug** menu, click **Start**.
2. On the **Help** menu, click **About**.
3. Click OK to close the About Form dialog box.
4. Click **Close** to quit the application.

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover some of the key concepts taught in the module.



- Why Use Windows Forms?
- Structure of Windows Forms
- Using Windows Forms
- Using Controls
- Windows Forms Inheritance

1. Identify some of the benefits of Windows Forms.

   Rich set of controls, GDI+ support, advanced layout possibilities, accessibility support, advanced printing support, visual inheritance, extensibility.

2. The **ContainerControl** class is the fundamental base class for all other controls. True or false?

   False. The Control class is the fundamental base class for all other controls.

3. Write the code to access the path from which an executable is running.

```
Dim strAppPath as String
strAppPath = Application.StartupPath
```

4. Describe an owned form.

   An owned form is always displayed on top of its owner. It is minimized or closed when the owner is minimized or closed.

5. Write code to make the code behind a button called btnOK execute when a user presses RETURN.

```
Me.AcceptButton = btnOK
```

6. List two ways to provide Help to the user.

   ErrorProvider, HelpProvider, or ToolTip controls.

7. Write code to create a Help menu with one menu item—**About**— at run time.

```
Dim mnuMain As New MainMenu( )
Dim mnuItem1 As New MenuItem( )
Dim mnuItem2 As New MenuItem( )

mnuItem1.Text = "Help"
mnuMain.MenuItems.Add(mnuItem1)

mnuItem2.Text = "About"
mnuMain.MenuItems(0).MenuItems.Add(mnuItem2)
AddHandler mnuItem2.Click, AddressOf AboutClick

Menu = mnuMain
```