Chapter 15 Graphical User Interfaces with Windows Forms: Part 2

Visual C# 2012 How to Program

15.2 Menus

- **Menus** provide groups of related commands for Windows `Form`s apps (Fig. 15.1).
- Menus organize commands without "cluttering" the GUI.

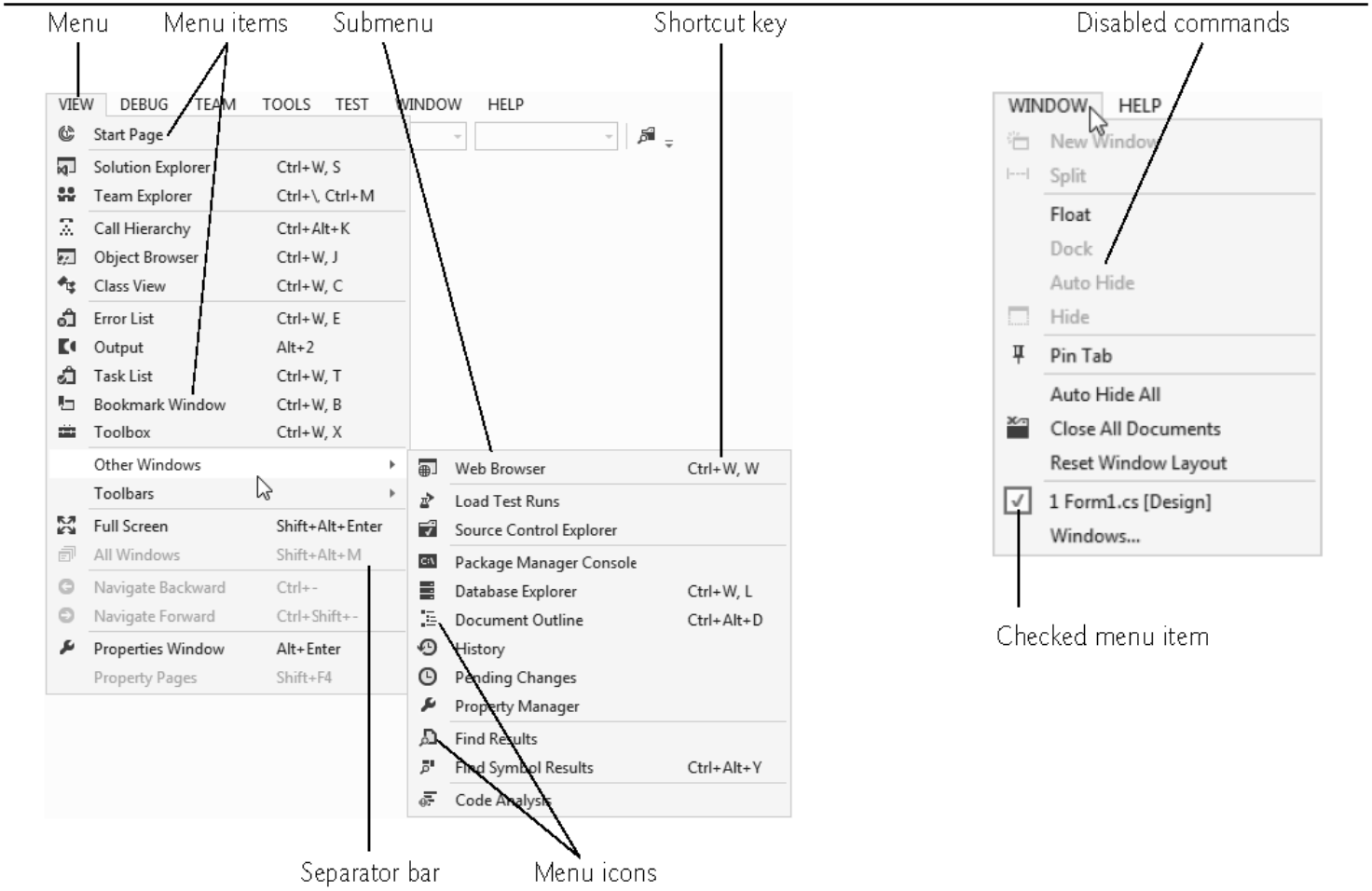

Fig. 15.1 | Menus, submenus and menu items.

15.2 Menus (Cont.)

- To create a menu, open the **Toolbox** and drag a `MenuStrip` control onto the `Form`.
- To add menu items to the menu, click the **Type Here** `TextBox` (Fig. 15.2) and type the menu item's name.

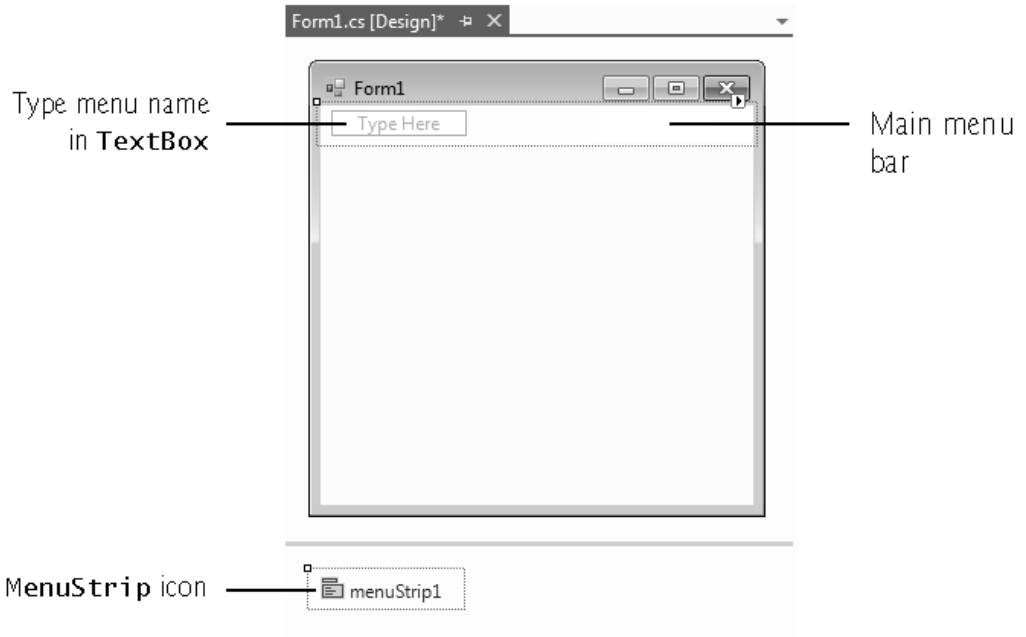

Fig. 15.2 | Editing menus in Visual Studio.

15.2 Menus (Cont.)

- After you press the *Enter* key, the menu item is added.
- More **Type Here** `TextBox`es allow you to add more items (Fig. 15.3).

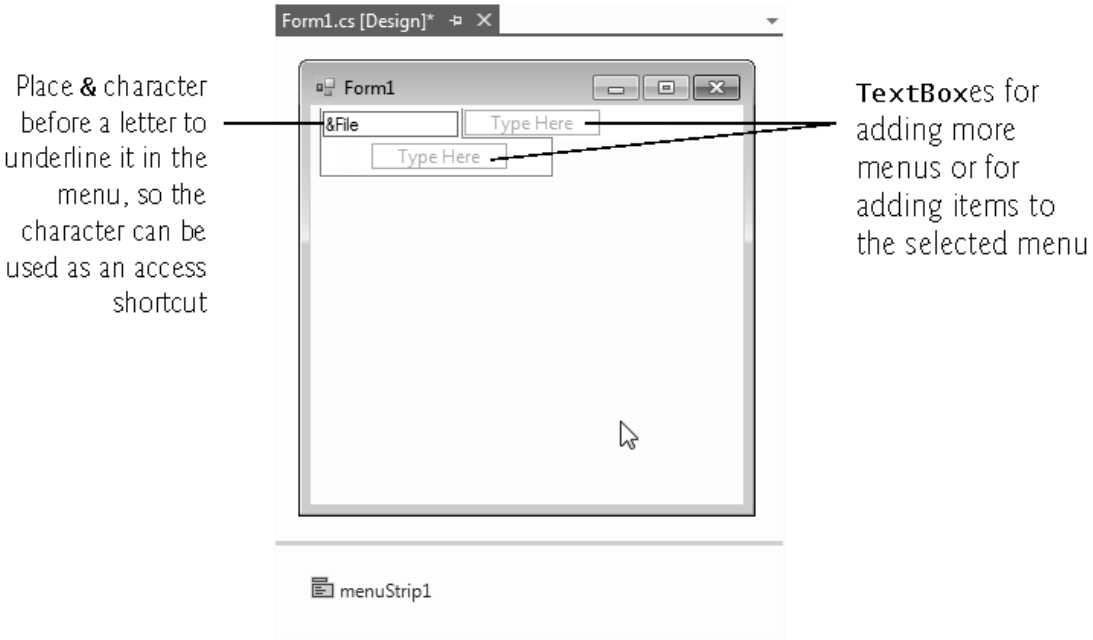

Fig. 15.3 | Adding `ToolStripMenuItems` to a `MenuStrip`.

15.2 Menus (Cont.)

- Menus can have *Alt* key shortcuts which are accessed by pressing *Alt* and the underlined letter.
- To make the **File** menu item have a key shortcut, type `&File`.
- The letter **F** is underlined to indicate that it's a shortcut.

15.2 Menus (Cont.)

- Menu items can have shortcut keys as well (*Ctrl*, *Shift*, *Alt*, *F1*, *F2*, letter keys, and so on).
- To add other shortcut keys, set the `ShortcutKeys` property (Fig. 15.4).

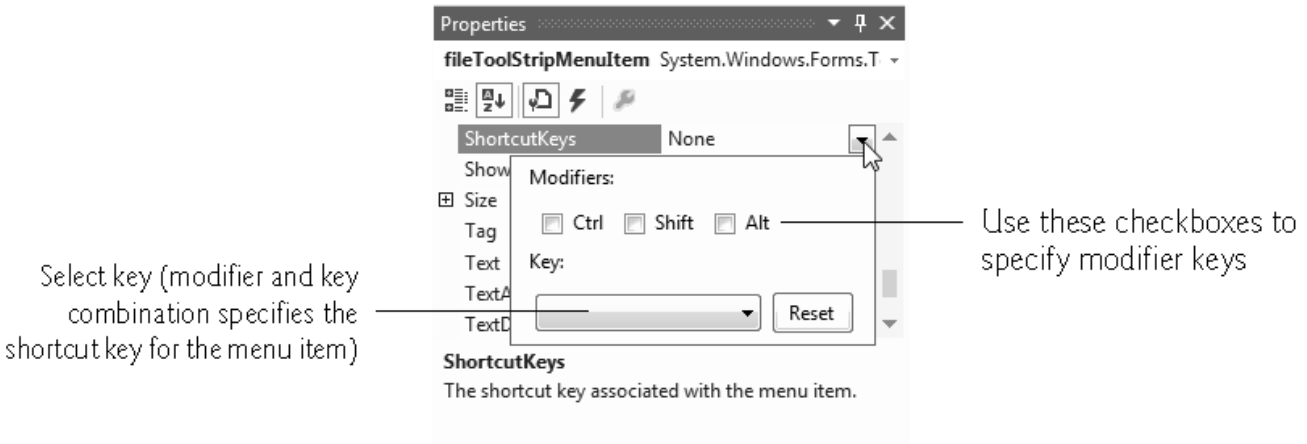

Fig. 15.4 | Setting a menu item's shortcut keys.

15.2 Menus (Cont.)

- You can remove a menu item by selecting it with the mouse and pressing the *Delete* key.
- Menu items can be grouped by **separator bars**, which are inserted by right clicking and selecting **Insert > Separator** or by typing "-" for the text of a menu item.

15.5 LinkLabel Control

- The `LinkLabel` control displays *links* to other resources, such as files or web pages (Fig. 15.12).

Fig. 15.12 | LinkLabel control in running program.

15.5 LinkLabel Control (Cont.)

- When clicked, the `LinkLabel` generates a `LinkClicked` event (Fig. 15.13).
- Class `LinkLabel` is derived from class `Label` and therefore inherits all of class `Label`'s functionality.

`LinkLabel` properties and an event	Description
Common Properties	
`ActiveLinkColor`	Specifies the color of the active link when the user is in the process of clicking the link. The default color (typically red) is set by the system.
`LinkArea`	Specifies which portion of text in the `LinkLabel` is part of the link.
`LinkBehavior`	Specifies the link's behavior, such as how the link appears when the mouse is placed over it.
`LinkColor`	Specifies the original color of the link before it's been visited. The default color (typically blue) is set by the system.
`LinkVisited`	If `true`, the link appears as though it has been visited (its color is changed to that specified by property `VisitedLinkColor`). The default value is `false`.
`Text`	Specifies the control's text.
`UseMnemonic`	If `true`, the `&` character in the `Text` property acts as a shortcut (similar to the *Alt* shortcut in menus).
`VisitedLinkColor`	Specifies the color of a visited link. The default color (typically purple) is set by the system.

Fig. 15.13 | `LinkLabel` properties and an event. (Part 1 of 2.)

`LinkLabel` properties and an event	Description
Common Event (Event arguments `LinkLabelLinkClickedEventArgs`)	
`LinkClicked`	Generated when the link is clicked. This is the default event when the control is double clicked in **Design** mode.

Fig. 15.13 | `LinkLabel` properties and an event. (Part 2 of 2.)

15.6 ListBox Control

- The `ListBox` control allows the user to view and select from multiple items in a list.
- The `CheckedListBox` control extends a `ListBox` by including `CheckBox`es next to each item in the list.
- Figure 15.15 displays a `ListBox` and a `CheckedListBox`.

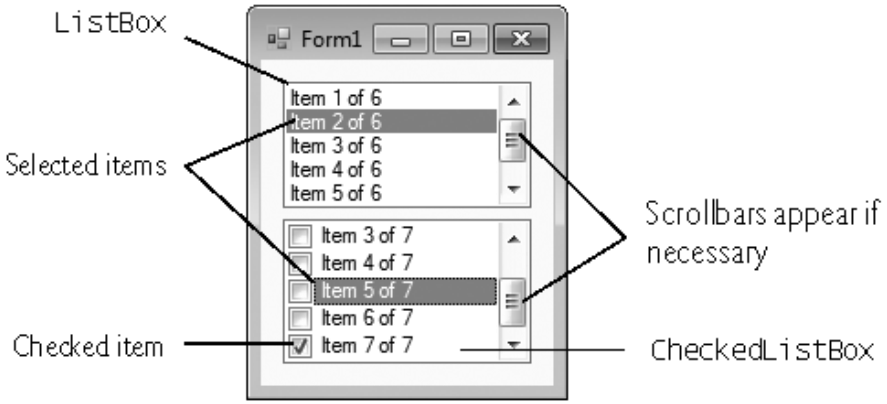

Fig. 15.15 | ListBox and CheckedListBox on a Form.

`ListBox` properties, methods and an event	Description
Common Properties	
`Items`	The collection of items in the `ListBox`.
`MultiColumn`	Indicates whether the `ListBox` can display multiple columns. Multiple columns eliminate vertical scrollbars from the display.
`SelectedIndex`	Returns the index of the selected item. If no items have been selected, the property returns `-1`. If the user selects multiple items, this property returns only one of the selected indices. If multiple items are selected, use property `SelectedIndices`.
`SelectedIndices`	Returns a collection containing the indices for all selected items.
`SelectedItem`	Returns a reference to the selected item. If multiple items are selected, it can return any of the selected items.
`SelectedItems`	Returns a collection of the selected item(s).

Fig. 15.16 | `ListBox` properties, methods and an event. (Part 1 of 2.)

`ListBox` properties, methods and an event	Description
`SelectionMode`	Determines the number of items that can be selected and the means through which multiple items can be selected. Values `None`, `One` (the default), `MultiSimple` (multiple selection allowed) or `MultiExtended` (multiple selection allowed using a combination of arrow keys or mouse clicks and *Shift* and *Ctrl* keys).
`Sorted`	Indicates whether items are sorted *alphabetically*. Setting this property's value to `true` sorts the items. The default value is `false`.
Common Methods	
`ClearSelected`	Deselects every item.
`GetSelected`	Returns `true` if the item at the specified index is selected.
Common Event	
`SelectedIndexChanged`	Generated when the selected index changes. This is the default event when the control is double clicked in the designer.

Fig. 15.16 | `ListBox` properties, methods and an event. (Part 2 of 2.)

15.6 ListBox Control (Cont.)

Adding Items to ListBoxes and CheckedListBoxes

- To add items to a `ListBox` or to a `CheckedListBox`, we must add objects to its `Items` collection.

  myListBox`.Items.Add(` *myListItem* `);`

- You can add items to `ListBox`es and `CheckedListBox`es visually by examining the `Items` property in the `Properties` window (Fig. 15.17).

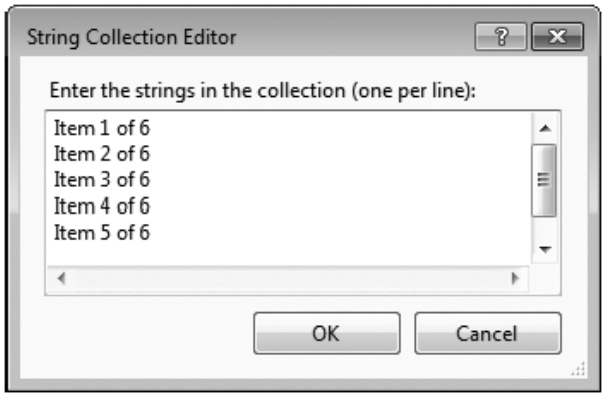

Fig. 15.17 | String Collection Editor.

15.8 ComboBox Control

- The **ComboBox** control combines TextBox features with a **drop-down list**.
- Figure 15.21 shows a sample ComboBox in three different states.

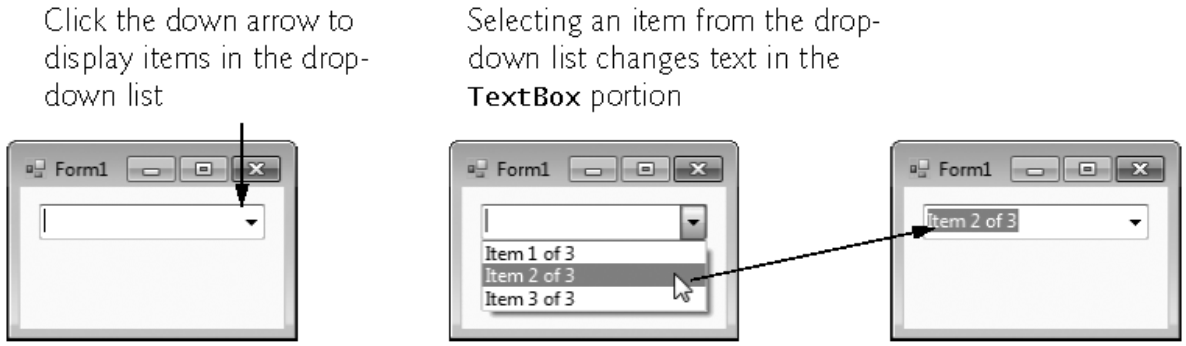

Fig. 15.21 | ComboBox demonstration.

ComboBox properties and an event	Description
Common Properties	
`DropDownStyle`	Determines the type of `ComboBox`. Value `Simple` means that the text portion is editable and the list portion is always visible. Value `DropDown` (the default) means that the text portion is editable but the user must click an arrow button to see the list portion. Value `DropDownList` means that the text portion is not editable and the user must click the arrow button to see the list portion.
`Items`	The collection of items in the `ComboBox` control.
`MaxDropDownItems`	Specifies the maximum number of items (between `1` and `100`) that the drop-down list can display. If the number of items exceeds the maximum number of items to display, a scrollbar appears.
`SelectedIndex`	Returns the index of the selected item, or `-1` if none are selected.
`SelectedItem`	Returns a reference to the selected item.
`Sorted`	Indicates whether items are sorted alphabetically. Setting this property's value to `true` sorts the items. The default is `false`.

Fig. 15.22 | ComboBox properties and an event. (Part 1 of 2.)

ComboBox **properties and an event**	**Description**
Common Event	
`SelectedIndexChanged`	Generated when the selected index changes (such as when a different item is selected). This is the default event when control is double clicked in the designer.

Fig. 15.22 | ComboBox properties and an event. (Part 2 of 2.)

15.11 TabControl Control

- The `TabControl` creates tabbed windows, such as those in Visual Studio (Fig. 15.32)

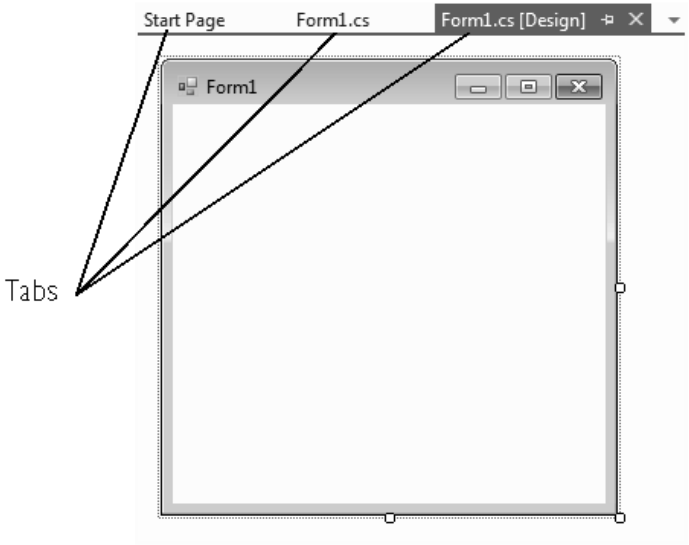

Fig. 15.32 | Tabbed windows in Visual Studio.

15.11 TabControl Control (Cont.)

- TabControls contain TabPage objects, which are similar to Panels.
- First add controls to the TabPage objects, then add the TabPages to the TabControl.

  myTabPage.Controls.Add(*myControl*);
  myTabControl.TabPages.Add(*myTabPage*);
- We can use method AddRange to add an array of TabPages or controls to a TabControl or TabPage.

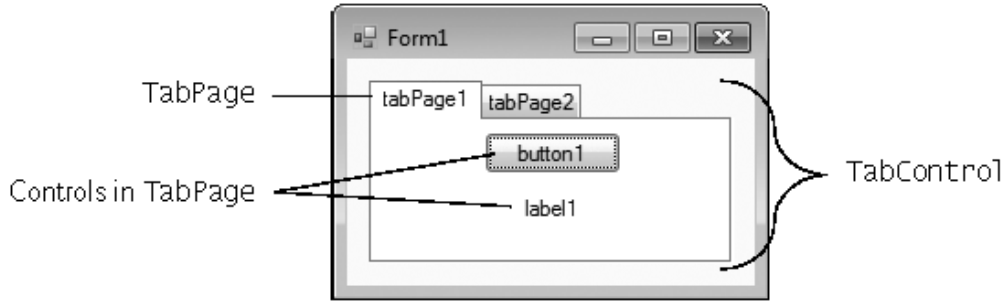

Fig. 15.33 | TabControl with TabPages example.

15.11 TabControl Control (Cont.)

- Add `TabControl`s visually by dragging and dropping them onto a `Form` in **Design** mode.
- To add `TabPage`s in **Design** mode, click the top of the `TabControl`, open its *smart tasks menu* and select **Add Tab** (Fig. 15.34).
- To select a `TabPage`, click the control area underneath the tabs.
- Common properties and a common event of `TabControl`s are described in Fig. 15.35.

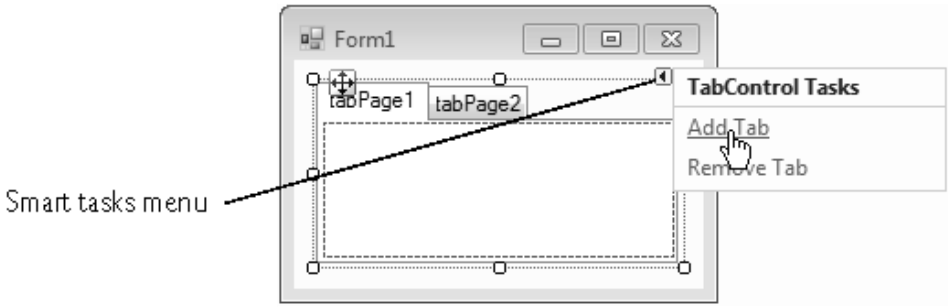

Fig. 15.34 | TabPages added to a TabControl.

TabControl properties and an event	Description
Common Properties	
`ImageList`	Specifies images to be displayed on tabs.
`ItemSize`	Specifies the tab size.
`Multiline`	Indicates whether multiple rows of tabs can be displayed.
`SelectedIndex`	Index of the selected `TabPage`.
`SelectedTab`	The selected `TabPage`.
`TabCount`	Returns the number of tab pages.
`TabPages`	Returns the collection of `TabPages` within the `TabControl` as a `TabControl.TabPageCollection`.
Common Event	
`SelectedIndexChanged`	Generated when `SelectedIndex` changes (i.e., another `TabPage` is selected).

Fig. 15.35 | `TabControl` properties and an event.

15.12 Multiple Document Interface (MDI) Windows

- Many complex apps are **multiple document interface (MDI)** programs, which allow users to edit multiple documents at once.
- An MDI program's main window is called the **parent window**, and each window inside the app is referred to as a **child window**.
- Figure 15.37 depicts a sample MDI app with two child windows.
- To create an MDI Form, create a new `Form` and set its `IsMdiContainer` property to `true` (Fig. 15.38).

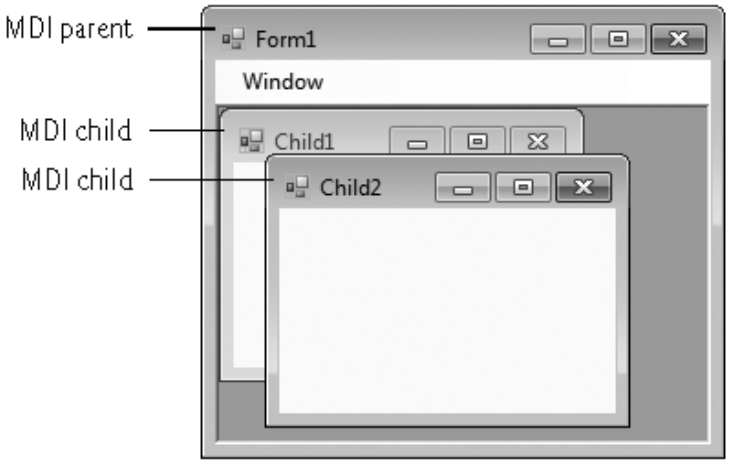

Fig. 15.37 | MDI parent window and MDI child windows.

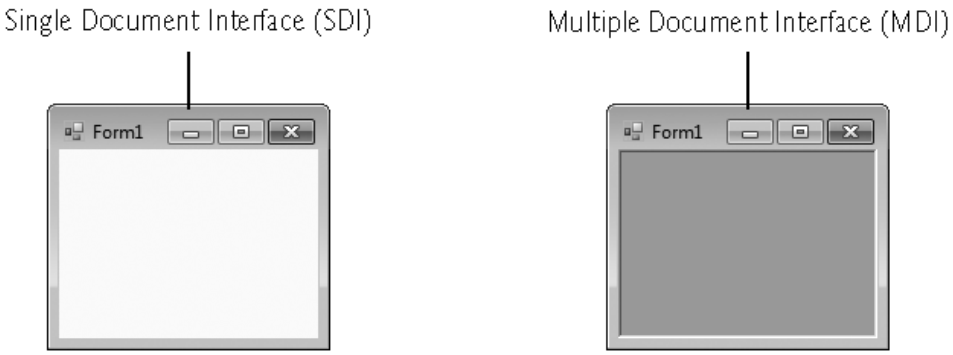

Fig. 15.38 | SDI and MDI forms.

15.12 Multiple Document Interface (MDI) Windows (Cont.)

- Right click the project in the **Solution Explorer**, select **Project > Add Windows Form...** and name the file.
- Set the `Form`'s `MdiParent` property to the parent `Form` and call the child `Form`'s `Show` method.

```
ChildFormClass childForm = New ChildFormClass();

childForm.MdiParent = parentForm;
childForm.Show();
```

- In most cases, the parent `Form` creates the child, so the *parentForm* reference is `this`.

MDI `Form` properties, a method and an event	**Description**
Common MDI Child Properties	
`IsMdiChild`	Indicates whether the `Form` is an MDI child. If `true`, `Form` is an MDI child (read-only property).
`MdiParent`	Specifies the MDI parent `Form` of the child.
Common MDI Parent Properties	
`ActiveMdiChild`	Returns the `Form` that's the currently active MDI child (returns `null` if no children are active).
`IsMdiContainer`	Indicates whether a `Form` can be an MDI parent. If `true`, the `Form` can be an MDI parent. The default value is `false`.
`MdiChildren`	Returns the MDI children as an array of `Forms`.

Fig. 15.39 | MDI parent and MDI child properties, a method and an event. (Part 1 of 2.)

MDI Form properties, a method and an event	**Description**
Common Method	
`LayoutMdi`	Determines the display of child forms on an MDI parent. The method takes as a parameter an `MdiLayout` enumeration with possible values `ArrangeIcons`, `Cascade`, `TileHorizontal` and `TileVertical`. Figure 15.42 depicts the effects of these values.
Common Event	
`MdiChildActivate`	Generated when an MDI child is closed or activated.

Fig. 15.39 | MDI parent and MDI child properties, a method and an event. (Part 2 of 2.)

15.12 Multiple Document Interface (MDI) Windows (Cont.)

- Property `MdiWindowListItem` of class `MenuStrip` specifies which menu, if any, displays a list of open child windows.
- When a new child window is opened, an entry is added to the end of the list (Fig. 15.41).

15.12 Multiple Document Interface (MDI) Windows (Cont.)

- MDI containers allow you to organize the placement of its child windows.
- Method `LayoutMdi` takes a value of the `MdiLayout` enumeration (Fig. 15.42).
- Class `UsingMDIForm` (Fig. 15.43) demonstrates MDI windows.

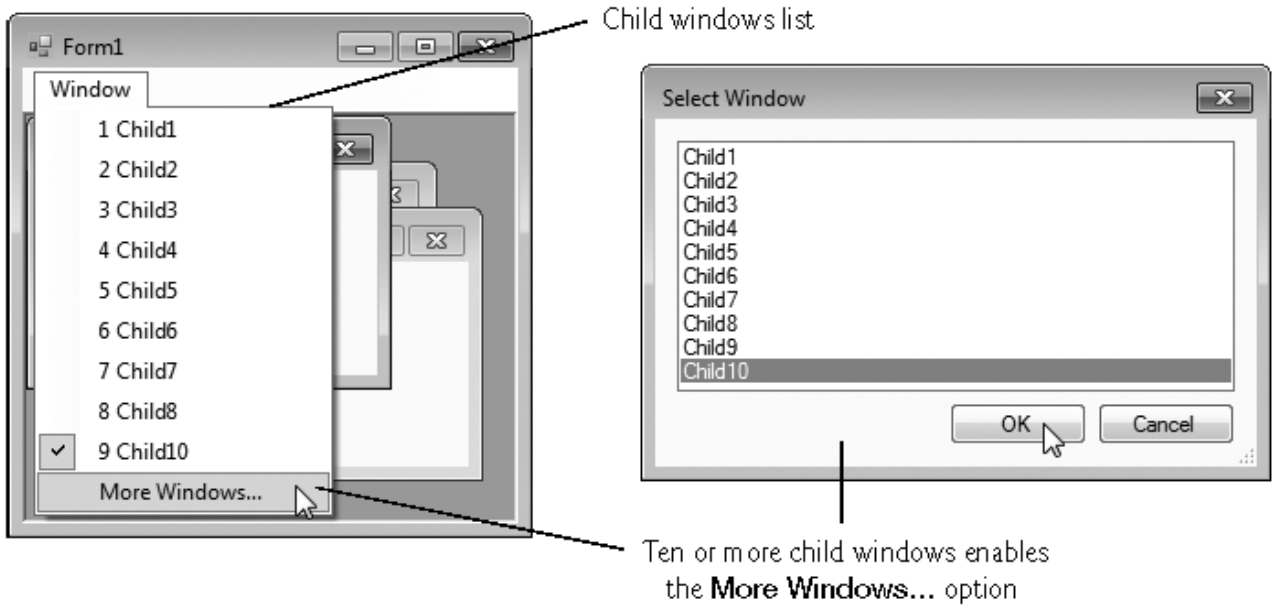

Fig. 15.41 | `MenuStrip` property `MdiWindowListItem` example.

15.12 Multiple Document Interface (MDI) Windows (Cont.)

- Define the MDI child class by right clicking the project in the Solution Explorer and selecting **Add** > **Windows Form…**.
- Name the new class `ChildForm` (Fig. 15.44).

```
1  // Fig. 15.44: ChildForm.cs
2  // Child window of MDI parent.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace UsingMDI
8  {
9     public partial class ChildForm : Form
10    {
11       public ChildForm( string title, string resourceName )
12       {
13          // Required for Windows Form Designer support
14          InitializeComponent();
15
16          Text = title; // set title text
17
18          // set image to display in PictureBox
19          displayPictureBox.Image =
20             ( Image ) ( Properties.Resources.ResourceManager.GetObject(
21                resourceName );
22       } // end constructor
23    } // end class ChildForm
24 } // end namespace UsingMDI
```

Fig. 15.44 | MDI child `ChildForm`.

15.13 Visual Inheritance

- **Visual inheritance** enables you to achieve visual consistency.
- For example, you could define a base `Form` that contains a product's logo and a specific background color.
- You then could use the base `Form` throughout an app for uniformity and branding.

15.13 Visual Inheritance (cont.)

- Class `VisualInheritanceBaseForm` (Fig. 15.45) derives from Form.
- We use the public class `VisualInheritanceBaseForm`.
- Use the namespace declaration that was created for us by the IDE.
- Right click the project name in the **Solution Explorer** and select **Properties**, then choose the **Application** tab.
- In the **Output** type drop-down list, change **Windows Application** to **Class Library**.
- Building the project produces the **.dll**.

```
1  // Fig. 15.45: VisualInheritanceBaseForm.cs
2  // Base Form for use with visual inheritance.
3  using System;
4  using System.Windows.Forms;
5
6  namespace VisualInheritanceBase
7  {
8     // base Form used to demonstrate visual inheritance
9     public partial class VisualInheritanceBaseForm : Form
10    {
11       // constructor
12       public VisualInheritanceForm()
13       {
14          InitializeComponent();
15       } // end constructor
16
```

Fig. 15.45 | Class `VisualInheritanceBaseForm`, which inherits from class `Form`, contains a `Button` (**Learn More**). (Part 1 of 2.)

```
17         // display MessageBox when Button is clicked
18         private void learnMoreButton_Click( object sender, EventArgs e )
19         {
20            MessageBox.Show(
21               "Bugs, Bugs, Bugs is a product of deitel.com",
22               "Learn More", MessageBoxButtons.OK,
23               MessageBoxIcon.Information );
24         } // end method learnMoreButton_Click
25      } // end class VisualInheritanceBaseForm
26   } // end namespace VisualInheritanceBase
```

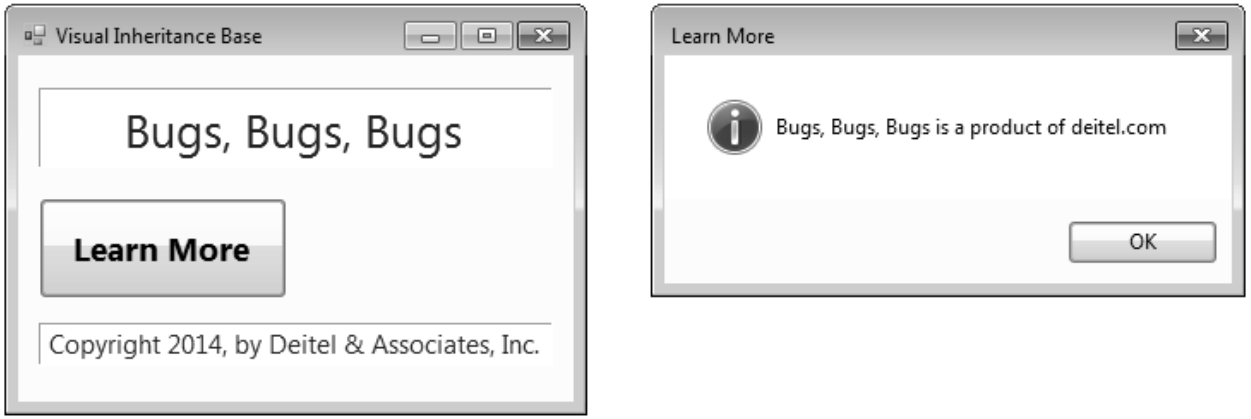

Fig. 15.45 | Class `VisualInheritanceBaseForm`, which inherits from class `Form`, contains a `Button` (**Learn More**). (Part 2 of 2.)

15.13 Visual Inheritance (Cont.)

- To visually inherit from `VisualInheritanceBaseForm`, create a new Windows `Forms` app.
- In this app, add a reference to the `.dll` you just created.
- Modify the line that defines the class:

```
public partial class VisualInheritanceTestForm :
   VisualInheritanceBase.VisualInheritanceBaseForm
```

15.13 Visual Inheritance (Cont.)

- In **Design** view, the new app's `Form` should now display the controls inherited from the base `Form` (Fig. 15.46).
- Class `VisualInheritanceTestForm` (Fig. 15.47) is a derived class of `VisualInheritanceBaseForm`.

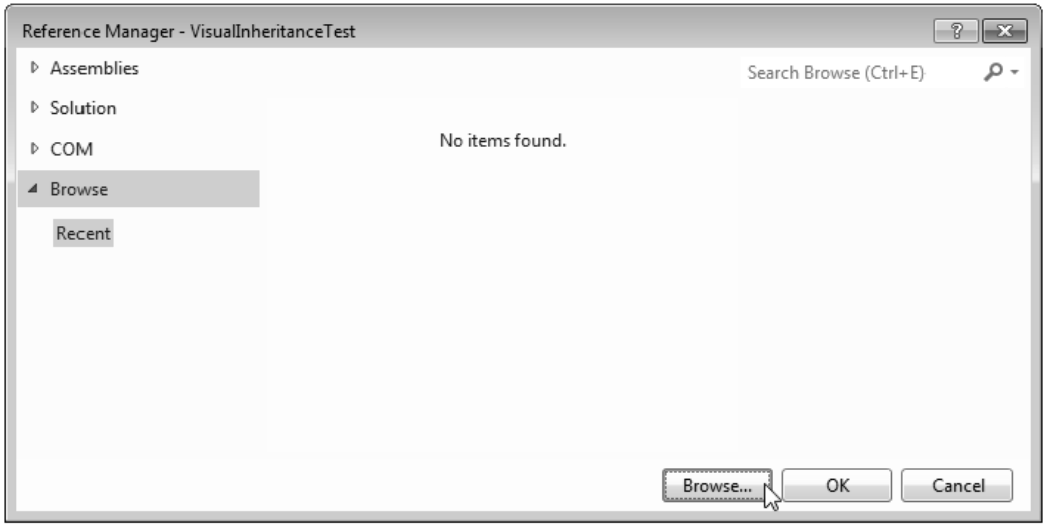

Fig. 15.46 | Using the Reference Manager dialog to browse for a DLL. (Part 1 of 2.)

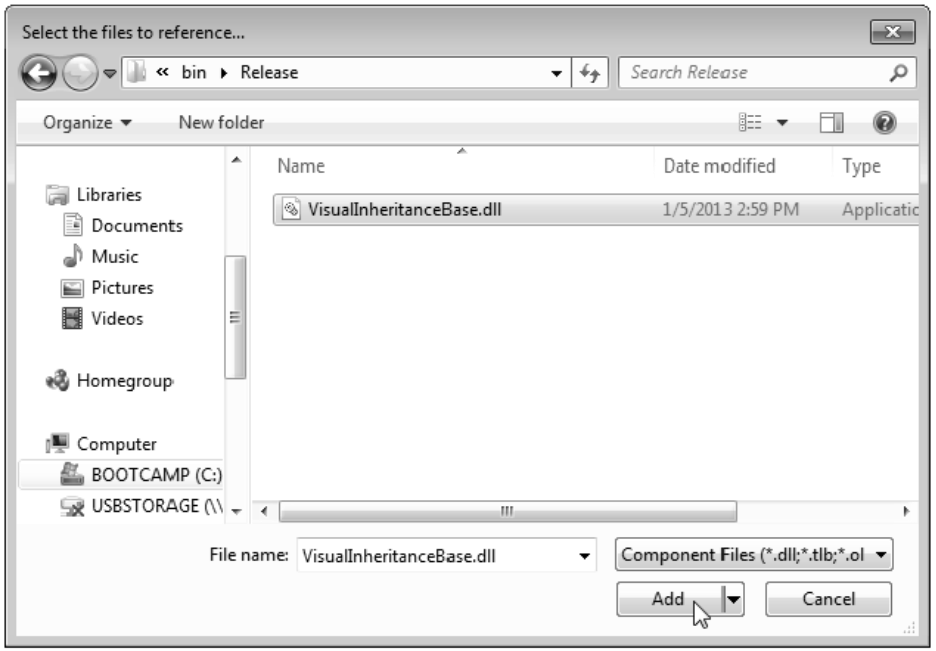

Fig. 15.46 | Using the Reference Manager dialog to browse for a DLL. (Part 2 of 2.)

Fig. 15.47 | Form demonstrating visual inheritance.

```
1  // Fig. 15.48: VisualInheritanceTestForm.cs
2  // Derived Form using visual inheritance.
3  using System;
4  using System.Windows.Forms;
5
6  namespace VisualInheritanceTest
7  {
8     // derived form using visual inheritance
9     public partial class VisualInheritanceTestForm :
10       VisualInheritanceBase.VisualInheritanceBaseForm
11    {
12       // constructor
13       public VisualInheritanceTestForm()
14       {
15          InitializeComponent();
16       } // end constructor
17
```

Fig. 15.48 | Class `VisualInheritanceTestForm`, which inherits from class `VisualInheritanceBaseForm`, contains an additional `Button`. (Part 1 of 3.)

```
18         // display MessageBox when Button is clicked
19         private void aboutButton_Click(object sender, EventArgs e)
20         {
21            MessageBox.Show(
22               "This program was created by Deitel & Associates.",
23               "About This Program", MessageBoxButtons.OK,
24               MessageBoxIcon.Information );
25         } // end method aboutButton_Click
26      } // end class VisualInheritanceTestForm
27   } // end namespace VisualInheritanceTest
```

Fig. 15.48 | Class `VisualInheritanceTestForm`, which inherits from class `VisualInheritanceBaseForm`, contains an additional `Button`. (Part 2 of 3.)

Fig. 15.48 | Class `VisualInheritanceTestForm`, which inherits from class `VisualInheritanceBaseForm`, contains an additional `Button`. (Part 3 of 3.)

15.14 User-Defined Controls

- The .NET Framework allows you to create **custom controls**.
- Custom controls appear in the user's **Toolbox**.
- There are multiple ways to create a custom control, depending on the level of customization that you want (Fig. 15.49).

Custom-control techniques and `PaintEventArgs` properties	Description
Custom-Control Techniques	
Inherit from Windows `Forms` control	You can do this to add functionality to a preexisting control. If you override method `OnPaint`, call the base class's `OnPaint` method. You only can add to the original control's appearance, not redesign it.
Create a `UserControl`	You can create a `UserControl` composed of multiple preexisting controls (e.g., to combine their functionality). You place drawing code in a `Paint` event handler or overridden `OnPaint` method.
Inherit from class `Control`	Define a brand new control. Override method `OnPaint`, then call base-class method `OnPaint` and include methods to draw the control. With this method you can customize control appearance and functionality.

Fig. 15.49 | Custom-control creation. (Part 1 of 2.)

Custom-control techniques and `PaintEventArgs` properties	Description
`PaintEventArgs` Properties	
`Graphics`	The control's graphics object, which is used to draw on the control.
`ClipRectangle`	Specifies the rectangle indicating the boundary of the control.

Fig. 15.49 | Custom-control creation. (Part 2 of 2.)

15.14 User-Defined Controls (Cont.)

- `Timers` are non-visual components that generate `Tick` events at a set interval.
  - The `Timer`'s **Interval** property defines the number of milliseconds between events.
- Create a `UserControl` class for the project by selecting **Project > Add User Control ...**.
  - We name the file (and the class) `ClockUserControl`.
  - Add a `Label` and a `Timer` to the `UserControl`.
  - Set the `Timer` interval to `1000` milliseconds.
  - `clockTimer` must be enabled by setting `Enabled` to `true`.

15.14 User-Defined Controls (Cont.)

- Figure 15.50 shows the output of `Clock`, which contains our `ClockUserControl`.

```
1  // Fig. 15.50: ClockUserControl.cs
2  // User-defined control with a timer and a Label.
3  using System;
4  using System.Windows.Forms;
5
6  namespace ClockExample
7  {
8     // UserControl that displays the time on a Label
9     public partial class ClockUserControl : UserControl
10    {
11       // constructor
12       public ClockUserControl()
13       {
14          InitializeComponent();
15       } // end constructor
16
17       // update Label at every tick
18       private void clockTimer_Tick(object sender, EventArgs e)
19       {
20          // get current time (Now), convert to string
21          displayLabel.Text = DateTime.Now.ToLongTimeString();
22       } // end method clockTimer_Tick
23    } // end class ClockUserControl
24 } // end namespace ClockExample
```

Fig. 15.50 | UserControl-defined clock. (Part 1 of 2.)

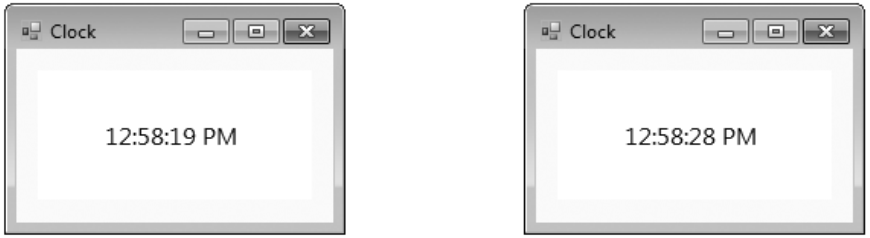

Fig. 15.50 | `UserControl`-defined clock. (Part 2 of 2.)

15.14 User-Defined Controls (Cont.)

- To create a `UserControl` that can be exported to other solutions, do the following:
  - Create a new **Class Library** project.
  - Delete `Class1.cs`, initially provided with the app.
  - Right click the project in the **Solution Explorer** and select **Add > User Control...**
  - Add controls and functionality to the `UserControl` (Fig. 15.51).
  - Build the project. Visual Studio creates a `.dll` file for the `UserControl` in the output directory (`bin/Release` or `bin/Release`).

Fig. 15.51 | Custom-control creation.

15.14 User-Defined Controls (Cont.)

- Create a new Windows app.
- Right click the **ToolBox** and select **Choose Items...**.
- In the **Choose Toolbox Items** dialog, click **Browse...**
- Select the `.dll` file that you created.
- The item will then appear in the **Choose Toolbox Items** dialog (Fig. 15.52).
- Check this item and click **OK** to add the item to the **Toolbox**.

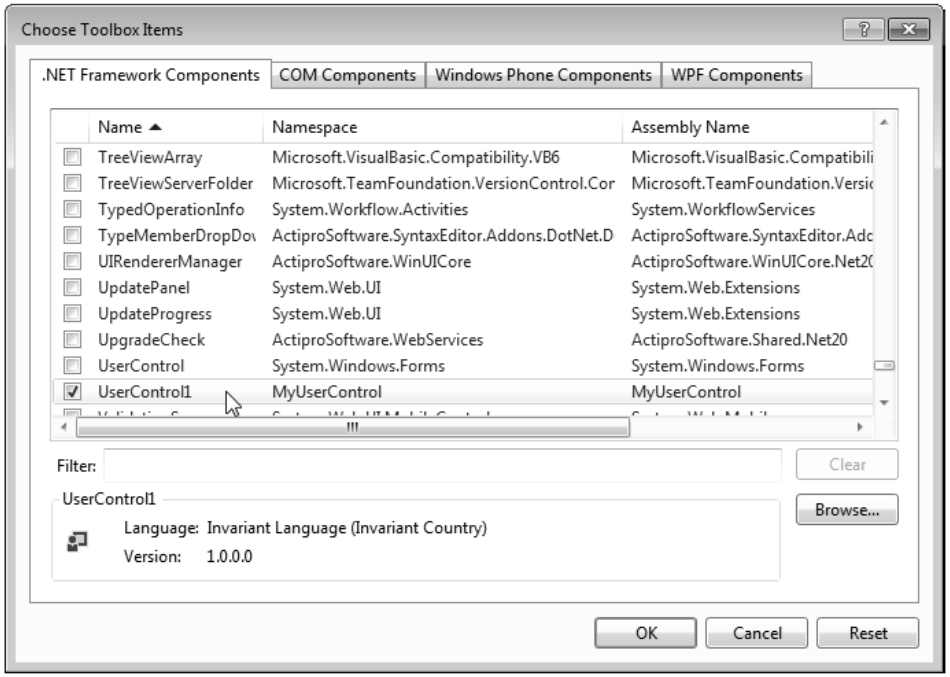

Fig. 15.52 | Custom control added to the ToolBox.