`Form` properties, methods and an event	Description
Common Properties	
`AcceptButton`	`Button` that is clicked when *Enter* is pressed.
`AutoScroll`	`bool` value that allows or disallows scrollbars when needed.
`CancelButton`	`Button` that is clicked when the *Escape* key is pressed.
`FormBorderStyle`	Border style for the `Form` (e.g., none, single, three-dimensional).
`Font`	Font of text displayed on the `Form`, and the default font for controls added to the `Form`.
`Text`	Text in the `Form`'s title bar.
Common Methods	
`Close`	Closes a `Form` and releases all resources, such as the memory used for the `Form`'s contents. A closed `Form` cannot be reopened.

Fig. 14.4 | Common `Form` properties, methods and an event. (Part 1 of 2.)

`Form` **properties, methods and an event**	**Description**
`Hide`	Hides a `Form`, but does not destroy the `Form` or release its resources.
`Show`	Displays a hidden `Form`.
Common Event	
`Load`	Occurs before a `Form` is displayed to the user. The handler for this event is displayed in the Visual Studio editor when you double click the `Form` in the Visual Studio designer.

Fig. 14.4 | Common `Form` properties, methods and an event. (Part 2 of 2.)

Class `Control` properties and methods	Description
Common Properties	
`BackColor`	The control's background color.
`BackgroundImage`	The control's background image.
`Enabled`	Specifies whether the control is enabled (i.e., if the user can interact with it). Typically, portions of a disabled control appear "grayed out" as a visual indication to the user that the control is disabled.
`Focused`	Indicates whether the control has the focus.
`Font`	The `Font` used to display the control's text.
`ForeColor`	The control's foreground color. This usually determines the color of the text in the `Text` property.
`TabIndex`	The tab order of the control. When the *Tab* key is pressed, the focus transfers between controls based on the tab order. You can set this order.
`TabStop`	If `true`, then a user can give focus to this control via the *Tab* key.

Fig. 14.11 | Class `Control` properties and methods. (Part 1 of 2.)

Class `Control` properties and methods	**Description**
`Text`	The text associated with the control. The location and appearance of the text vary depending on the type of control.
`Visible`	Indicates whether the control is visible.
Common Methods	
`Hide`	Hides the control (sets the `Visible` property to `false`).
`Select`	Acquires the focus.
`Show`	Shows the control (sets the `Visible` property to `true`).

Fig. 14.11 | Class `Control` properties and methods. (Part 2 of 2.)

CheckBox **properties and events**	**Description**
Common Properties	
`Appearance`	By default, this property is set to `Normal`, and the `CheckBox` displays as a traditional checkbox. If it's set to `Button`, the `CheckBox` displays as a `Button` that looks pressed when the `CheckBox` is checked.
`Checked`	Indicates whether the `CheckBox` is checked (contains a check mark) or unchecked (blank). This property returns a `bool` value. The default is `false` (unchecked).
`CheckState`	Indicates whether the `CheckBox` is checked or unchecked with a value from the `CheckState` enumeration (`Checked`, `Unchecked` or `Indeterminate`). `Indeterminate` is used when it's unclear whether the state should be `Checked` or `Unchecked`. When `CheckState` is set to `Indeterminate`, the `CheckBox` is usually shaded.
`Text`	Specifies the text displayed to the right of the `CheckBox`.

Fig. 14.25 | CheckBox properties and events. (Part 1 of 2.)

CheckBox **properties and events**	**Description**
`ThreeState`	When this property is `true`, the `CheckBox` has three states—checked, unchecked and indeterminate. By default, this property is `false` and the `CheckBox` has only two states—checked and unchecked.
Common Events	
`CheckedChanged`	Generated when the `Checked` property changes. This is a `CheckBox`'s default event. When a user double clicks the `CheckBox` control in design view, an empty event handler for this event is generated.
`CheckStateChanged`	Generated when the `CheckState` property changes.

Fig. 14.25 | CheckBox properties and events. (Part 2 of 2.)

CheckBoxes and RadioButtons (Cont.)

- To change the font style on a `Label`, you must set its `Font` property to a new **Font object**.
- The `Font` constructor we used takes the current font and the new style as arguments.
- Styles can be combined via **bitwise operators**—operators that perform manipulation on bits of information.
- We needed to set the `FontStyle` so that the text appears in bold if it was not bold originally, and vice versa
  - The logical exclusive OR operator makes toggling the text style simple.

`RadioButton` properties and an event	Description
Common Properties	
`Checked`	Indicates whether the `RadioButton` is checked.
`Text`	Specifies the `RadioButton`'s text.
Common Event	
`CheckedChanged`	Generated every time the `RadioButton` is checked or unchecked. When you double click a `RadioButton` control in design view, an empty event handler for this event is generated.

Fig. 14.27 | `RadioButton` properties and an event.

```
1  // Fig. 14.30: PictureBoxTestForm.cs
2  // Using a PictureBox to display images.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace PictureBoxTest
8  {
9     // Form to display different images when PictureBox is clicked
10    public partial class PictureBoxTestForm : Form
11    {
12       private int imageNum = -1; // determines which image is displayed
13
14       // default constructor
15       public PictureBoxTestForm()
16       {
17          InitializeComponent();
18       } // end constructor
19
20       // change image whenever Next Button is clicked
21       private void nextButton_Click( object sender, EventArgs e )
22       {
23          imageNum = ( imageNum + 1 ) % 3; // imageNum cycles from 0 to 2
24
```

Fig. 14.30 | Using a `PictureBox` to display images. (Part 1 of 3.)

```
            // retrieve image from resources and load into PictureBox
            imagePictureBox.Image = ( Image )
               ( Properties.Resources.ResourceManager.GetObject(
               string.Format( "image{0}", imageNum ) ) );
         } // end method nextButton_Click
      } // end class PictureBoxTestForm
   } // end namespace PictureBoxTest
```

Fig. 14.30 | Using a `PictureBox` to display images. (Part 2 of 3.)

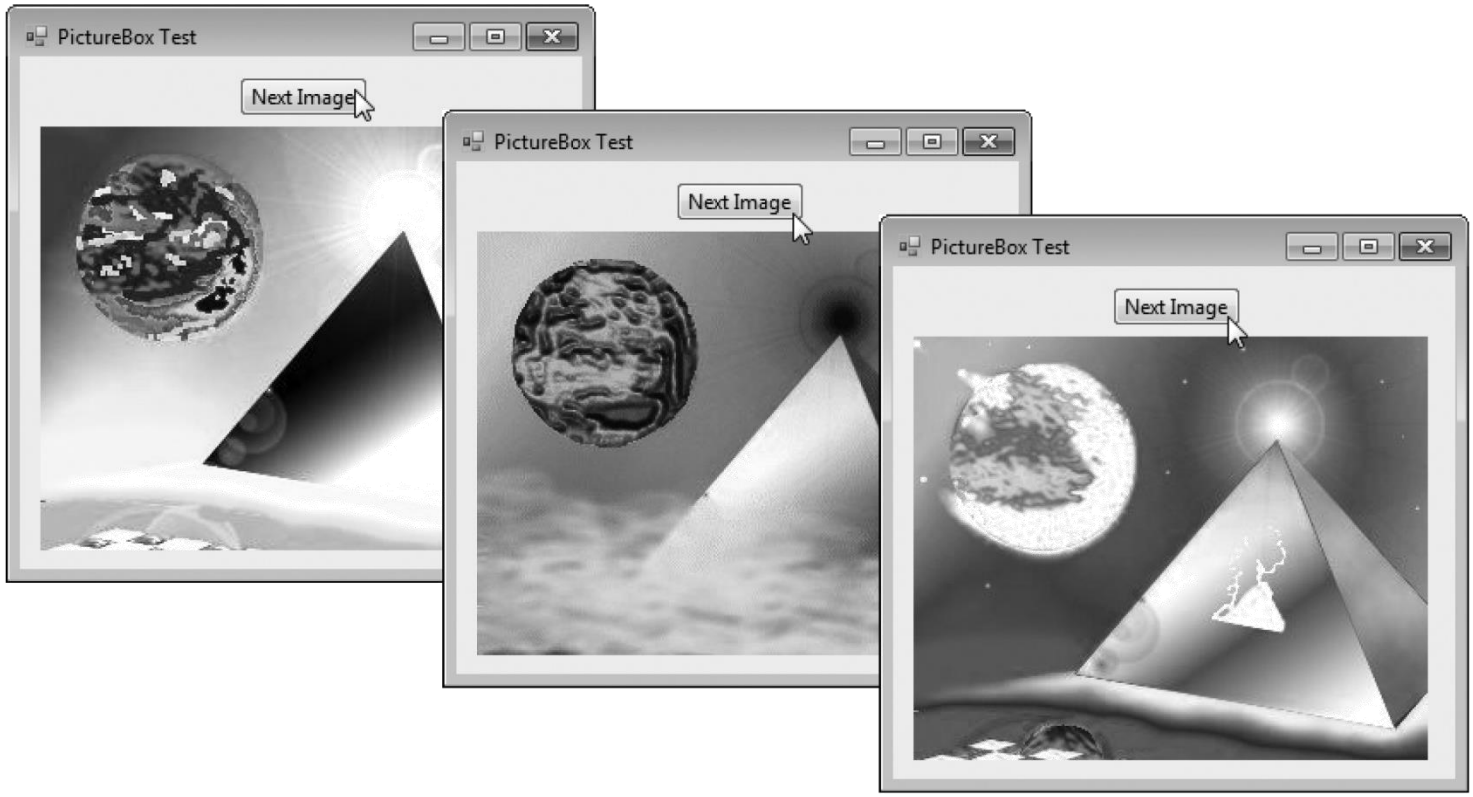

Fig. 14.30 | Using a `PictureBox` to display images. (Part 3 of 3.)

PictureBoxes (Cont.)

- Embedding the images in the application prevents problems of using several separate files.
- To add a resource:
  - Double click the project's **Properties** node in the **Solution Explorer**.
  - Click the **Resources** tab.
  - At the top of the **Resources** tab click the down arrow next to the **Add Resource** button and select **Add Existing File...**
  - Locate the files you wish to add and click the **Open** button.
  - Save your project.

PictureBoxes (Cont.)

- A project's resources are stored in its `Resources` class.
- Its `ResourceManager` object allows interacting with the resources programmatically.
- To access an image, use the method `GetObject`, which takes as an argument the resource name as it appears in the **Resources** tab.
- The `Resources` class also provides direct access with expressions of the form `Resources.`*resourceName*.

Mouse-Event Handling

- **Mouse events** are generated when the user interacts with a control via the mouse.
- Information about the event is passed through a `MouseEventArgs` object, and the delegate type is `MouseEventHandler`.
- `MouseEventArgs` *x*- and *y*-coordinates are relative to the control that generated the event.
- Several common mouse events and event arguments are described in Figure 14.37.

Mouse events and event arguments

Mouse Events with Event Argument of Type `EventArgs`

`MouseEnter`	Occurs when the mouse cursor enters the control's boundaries.
`MouseHover`	Occurs when the mouse cursor hovers within the control's boundaries.
`MouseLeave`	Occurs when the mouse cursor leaves the control's boundaries.

Mouse Events with Event Argument of Type `MouseEventArgs`

`MouseDown`	Occurs when a mouse button is pressed while the mouse cursor is within a control's boundaries.
`MouseMove`	Occurs when the mouse cursor is moved while in the control's boundaries.
`MouseUp`	Occurs when a mouse button is released when the cursor is over the control's boundaries.

Class `MouseEventArgs` *Properties*

`Button`	Specifies which mouse button was pressed (`Left`, `Right`, `Middle` or `None`).
`Clicks`	The number of times that the mouse button was clicked.
`X`	The *x*-coordinate within the control where the event occurred.
`Y`	The *y*-coordinate within the control where the event occurred.

Fig. 14.37 | Mouse events and event arguments.

```
1  // Fig. 14.38: PainterForm.cs
2  // Using the mouse to draw on a Form.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace Painter
8  {
9     // creates a Form that is a drawing surface
10    public partial class PainterForm : Form
11    {
12       bool shouldPaint = false; // determines whether to paint
13
14       // default constructor
15       public PainterForm()
16       {
17          InitializeComponent();
18       } // end constructor
19
```

Fig. 14.38 | Using the mouse to draw on a Form. (Part 1 of 4.)

```
20          // should paint when mouse button is pressed down
21          private void PainterForm_MouseDown(
22             object sender, MouseEventArgs e )
23          {
24             // indicate that user is dragging the mouse
25             shouldPaint = true;
26          } // end method PainterForm_MouseDown
27
28          // stop painting when mouse button is released
29          private void PainterForm_MouseUp( object sender, MouseEventArgs e )
30          {
31             // indicate that user released the mouse button
32             shouldPaint = false;
33          } // end method PainterForm_MouseUp
34
```

Fig. 14.38 | Using the mouse to draw on a Form. (Part 2 of 4.)

```
35         // draw circle whenever mouse moves with its button held down
36         private void PainterForm_MouseMove(
37            object sender, MouseEventArgs e )
38         {
39            if ( shouldPaint ) // check if mouse button is being pressed
40            {
41               // draw a circle where the mouse pointer is present
42               using ( Graphics graphics = CreateGraphics() )
43               {
44                  graphics.FillEllipse(
45                     new SolidBrush( Color.BlueViolet ), e.X, e.Y, 4, 4 );
46               } // end using; calls graphics.Dispose()
47            } // end if
48         } // end method PainterForm_MouseMove
49      } // end class PainterForm
50   } // end namespace Painter
```

Fig. 14.38 | Using the mouse to draw on a Form. (Part 3 of 4.)

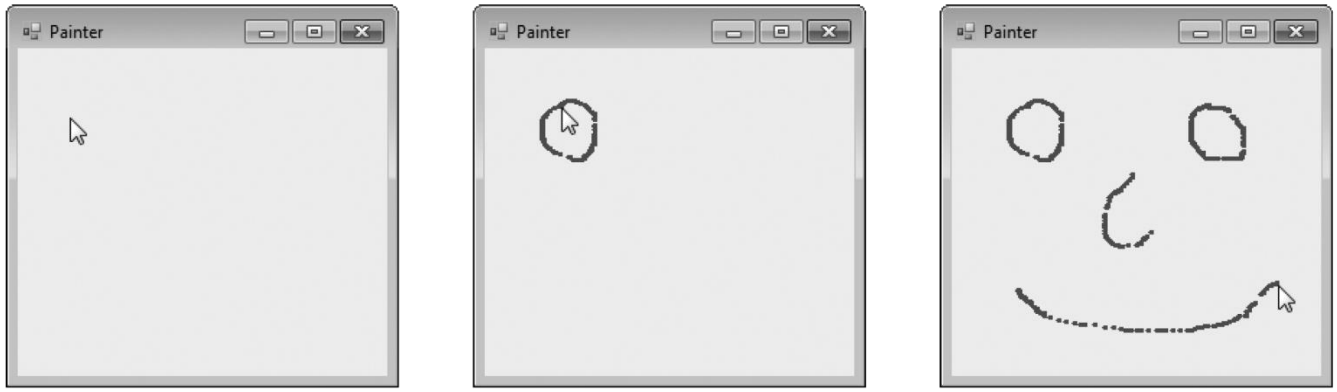

Fig. 14.38 | Using the mouse to draw on a `Form`. (Part 4 of 4.)

Keyboard-Event Handling

- There are three key events:
  - The `KeyPress` event occurs when the user presses a key that represents an ASCII character.
  - The `KeyPress` event does not indicate whether modifier keys (e.g., *Shift*, *Alt* and *Ctrl*) were pressed; if this information is important, the `KeyUp` or `KeyDown` events can be used.

Keyboard events and event arguments	
Key Events with Event Arguments of Type `KeyEventArgs`	
`KeyDown`	Generated when a key is initially pressed.
`KeyUp`	Generated when a key is released.
Key Event with Event Argument of Type `KeyPressEventArgs`	
`KeyPress`	Generated when a key is pressed. Raised after `KeyDown` and before `KeyUp`.
Class `KeyPressEventArgs` *Properties*	
`KeyChar`	Returns the ASCII character for the key pressed.
Class `KeyEventArgs` *Properties*	
`Alt`	Indicates whether the *Alt* key was pressed.
`Control`	Indicates whether the *Ctrl* key was pressed.
`Shift`	Indicates whether the *Shift* key was pressed.
`KeyCode`	Returns the key code for the key as a value from the `Keys` enumeration. This does not include modifier-key information. It's used to test for a specific key.

Fig. 14.39 | Keyboard events and event arguments. (Part 1 of 2.)

Keyboard events and event arguments	
`KeyData`	Returns the key code for a key combined with modifier information as a `Keys` value. This property contains all information about the pressed key.
`KeyValue`	Returns the key code as an `int`, rather than as a value from the `Keys` enumeration. This property is used to obtain a numeric representation of the pressed key. The `int` value is known as a Windows virtual key code.
`Modifiers`	Returns a `Keys` value indicating any pressed modifier keys (*Alt*, *Ctrl* and *Shift*). This property is used to determine modifier-key information only.

Fig. 14.39 | Keyboard events and event arguments. (Part 2 of 2.)

```
1  // Fig. 14.40: KeyDemo.cs
2  // Displaying information about the key the user pressed.
3  using System;
4  using System.Windows.Forms;
5
6  namespace KeyDemo
7  {
8     // Form to display key information when key is pressed
9     public partial class KeyDemo : Form
10    {
11       // default constructor
12       public KeyDemo()
13       {
14          InitializeComponent();
15       } // end constructor
16
17       // display the character pressed using KeyChar
18       private void KeyDemo_KeyPress(
19          object sender, KeyPressEventArgs e )
20       {
21          charLabel.Text = "Key pressed: " + e.KeyChar;
22       } // end method KeyDemo_KeyPress
23
```

Fig. 14.40 | Demonstrating keyboard events. (Part 1 of 3.)

```
24          // display modifier keys, key code, key data and key value
25          private void KeyDemo_KeyDown( object sender, KeyEventArgs e )
26          {
27             keyInfoLabel.Text =
28                "Alt: " + ( e.Alt ? "Yes" : "No" ) + '\n' +
29                "Shift: " + ( e.Shift ? "Yes" : "No" ) + '\n' +
30                "Ctrl: " + ( e.Control ? "Yes" : "No" ) + '\n' +
31                "KeyCode: " + e.KeyCode + '\n' +
32                "KeyData: " + e.KeyData + '\n' +
33                "KeyValue: " + e.KeyValue;
34          } // end method KeyDemo_KeyDown
35
36          // clear Labels when key released
37          private void KeyDemo_KeyUp( object sender, KeyEventArgs e )
38          {
39             charLabel.Text = "";
40             keyInfoLabel.Text = "";
41          } // end method KeyDemo_KeyUp
42       } // end class KeyDemo
43    } // end namespace KeyDemo
```

Fig. 14.40 | Demonstrating keyboard events. (Part 2 of 3.)

a) *H* pressed

b) *F7* pressed

c) *$* pressed

d) *Tab* pressed

Fig. 14.40 | Demonstrating keyboard events. (Part 3 of 3.)

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 `CheckBoxes` next to each item in the list.

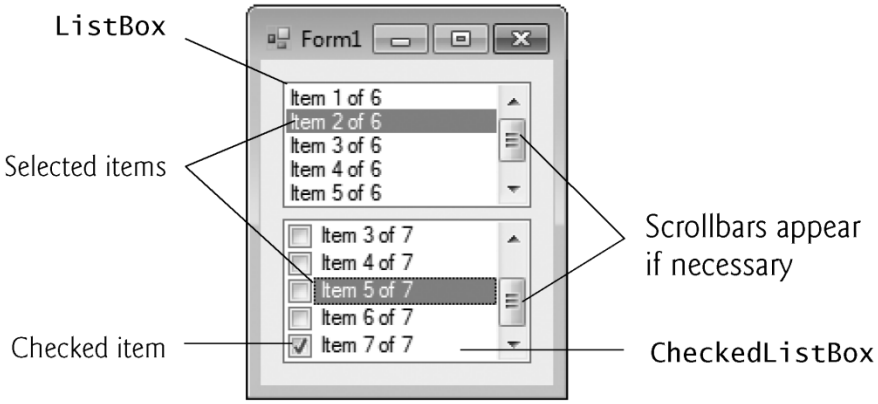

Fig. 15.15 | ListBox and CheckedListBox on a Form.

`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 returns the item with the lowest index number.
`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.)

ListBox Control (Cont.)

- 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 `ListBoxes` and `CheckedListBoxes` visually by examining the `Items` property in the Properties window (Fig. 15.17).

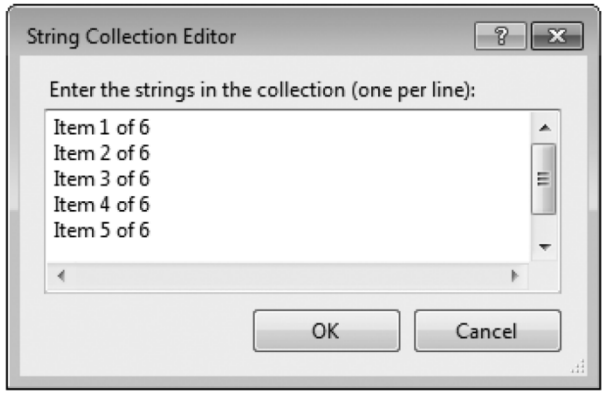

Fig. 15.17 | String Collection Editor.

ComboBox Control

- The `ComboBox` control combines `TextBox` features with a drop-down list.

Multiple Document Interface (MDI) Windows

- Many complex applications 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 application is referred to as a **child window**.
- Figure 15.37 depicts a sample MDI application with two child windows.
- To create an MDI Form, create a new `Form` and set its `IsMdiContainer` property to `true` (Fig. 15.38).

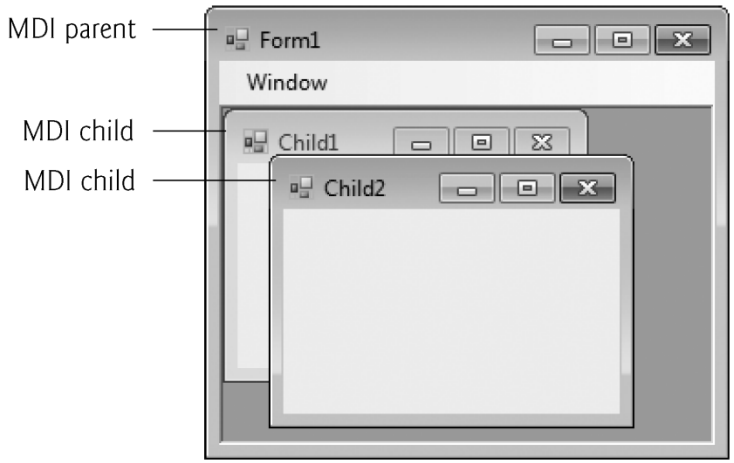

Fig. 15.37 | MDI parent window and MDI child windows.

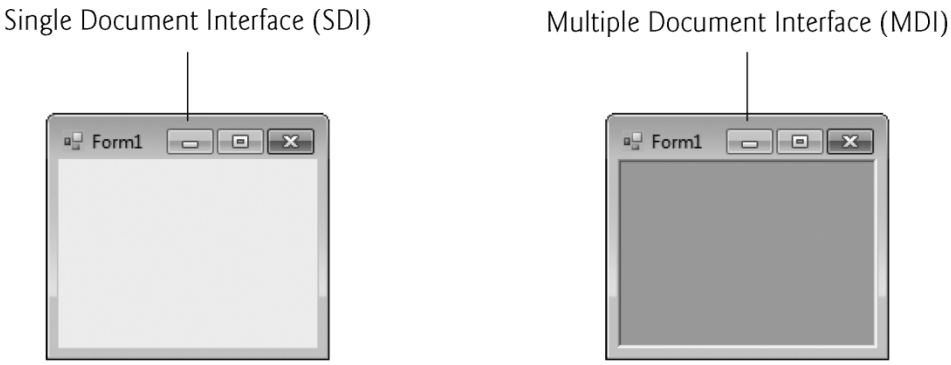

Fig. 15.38 | SDI and MDI forms.

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 is 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 `Form`s.

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.)

c) `TileHorizontal`

d) `TileVertical`

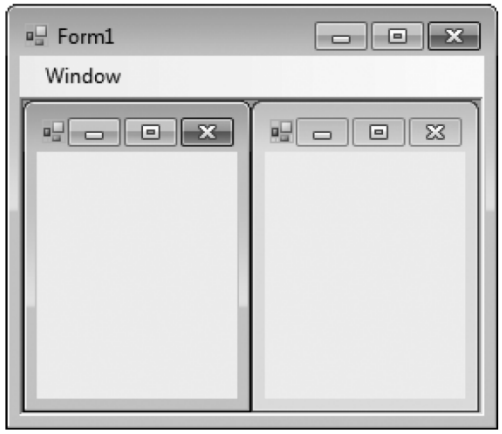

Fig. 15.42 | `MdiLayout` enumeration values. (Part 2 of 2.)