Second Edition of *Windows Forms Programming with C#*

SAMPLE CHAPTER

WINDOWS FORMS IN ACTION

Erik Brown

Windows Forms in Action

by Erik Brown

Chapter 18

Copyright 2006 Manning Publications

brief contents

Part 1 Hello Windows Forms 1

1 *Getting started with Windows Forms 3*
2 *Getting started with Visual Studio 33*

Part 2 Basic Windows Forms 63

3 *Menus 67*
4 *Context menu and status strips 92*
5 *Reusable libraries 116*
6 *Files and common dialog boxes 148*
7 *Dialog boxes 180*
8 *Text boxes 212*
9 *Buttons 240*
10 *Handling user input and encryption 268*
11 *List boxes 299*
12 *Combo boxes 327*
13 *Tab controls and pages 356*
14 *Dates, calendars, and progress bars 383*
15 *Bells and whistles 415*
16 *Tool strips 443*

Part 3 Advanced Windows Forms 475

17 Custom controls 477
18 Explorer interfaces and tree views 507
19 List views 541
20 Multiple document interfaces 575
21 Data binding 608
22 Two-way binding and binding sources 637
23 Odds and ends .NET 665

appendix A C# primer 701
appendix B .NET namespaces 735
appendix C Visual index 741
appendix D For more information 758

C H A P T E R 1 8

Explorer interfaces and tree views

18.1 Interface styles 508
18.2 Explorer interfaces in .NET 511
18.3 Tree nodes 518
18.4 Custom tree views 523
18.5 Recap 540

In computer programming, a style is a way of doing something, whether it's how a control is docked or anchored to a form or how a Windows interface is laid out. In this chapter we look at a specific application interface style known as the explorer interface.

An interface style, for our purposes, is just a convention for how a graphical user interface presents information to a user. The .NET Framework supports a number of interface styles. We introduce three in this chapter: single document interfaces, multiple document interfaces, and explorer interfaces. Chapter 20 looks at multiple document interfaces in more detail. In this chapter our focus is on explorer interfaces.

This chapter presents the following concepts:

- Understanding various interface styles
- Dividing containers with the `SplitContainer` control
- Representing a hierarchy with the `TreeView` control
- Adding `TreeNode` objects to a tree view
- Building a custom tree view control

These topics are covered as we progress through the chapter, beginning with interface styles.

18.1 INTERFACE STYLES

Sometimes programmers don't worry about what style of interface they are building. They build a program that accomplishes some task, and that's the end of it. The layout of controls in an application is a function of what needs to be done.

This typically results in a dialog interface style, where a bunch of controls are laid out on a form to accomplish a specific task. This works fine for small or very focused applications, but can be difficult to extend into a full-fledged Windows application with millions of users.

An alternate approach is to begin with a style in mind, and work toward this style as the application grows. Of course, it is even more desirable to have the completed application fully designed, and then incrementally build this application over a series of iterations. This is desirable but not always possible. My approach is to at least have a vision in mind, if not on paper, and work toward this vision as the application grows.

In this section we discuss three common styles of Windows interfaces that cover a broad range of possible applications. A style, in this sense, is simply an approach for how information is presented to the user. We examine these approaches:

- Single document interfaces
- Explorer interfaces
- Multiple document interfaces

We discuss each style separately.

18.1.1 Single document interfaces

A single document interface (SDI) is an interface that displays a single document or other encapsulated data within a single form. Our MyPhotos application, shown in figure 18.1, displays a single photo album to the user, and is a good example of this style. The contents of two albums cannot be compared unless two copies of the program are running simultaneously.

Figure 18.1 Our single document interface displays one photo album at a time.

More generally, a single document interface presents a single concept in a single window to the user, be it a photo album, a paper document, a tax form, or some other concept. Single document interfaces typically provide a menu bar to open, save, and

otherwise manipulate the concept; a status bar to summarize information related to the presented information; and one or more toolbars as an alternate way to manipulate the data.

18.1.2 Multiple document interfaces

A multiple document interface (MDI) is a logical extension of the single document interface style. MDI applications allow multiple views of one or more documents or other encapsulated data to be displayed at the same time. This permits alternate views of the same data, or separate instances of the same concept, within a single window. For example, a stock market MDI application might present different historical or graphical views of stock portfolios, each within its own separate window as part of a larger application. Alternately, such an application might display a single portfolio, with different views of the data in multiple windows.

In the traditional conception of this style, a single window acted as a container for other windows, where each contained window displayed a specific instance or view of a concept. Such an interface is shown in figure 18.2. More recently, well-known MDI applications such as Microsoft Word and Excel have taken the approach of displaying all of their windows directly on the desktop, each within a separate application window, while still preserving an MDI look and feel from the menu bar and other parts of the interface. This relatively new style, the multiple single document interface (MSDI), is consistent with the manner in which web browsers have historically worked.

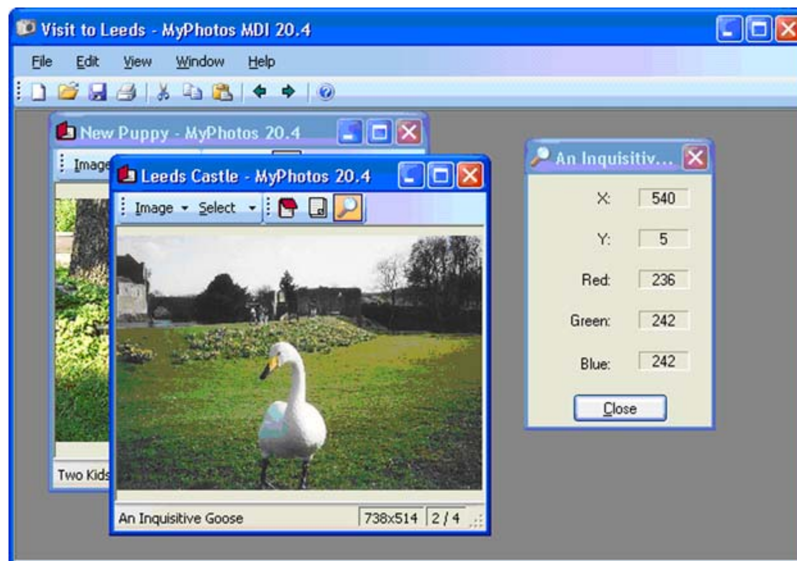


Figure 18.2 Our multiple document interface displays a selected set of photo albums within a single window.

Also note that Visual Studio, while providing an MDI-like interface, uses more of a TabControl look and feel for the set of displayed windows, or what might be called a tabbed document interface (TDI). In this style, multiple sets of windows are displayed as horizontal or vertical groups of tabs. The Mozilla Firefox browser, available at www.mozilla.org/products/firefox, uses this approach rather well.

Both the MSDI and MTDI approaches can be created using the .NET Framework as an alternative to the traditional MDI interface, although there is no direct support for these newer styles. As a result, implementing such interfaces requires a bit more effort from the developer and is therefore a bit beyond the scope of this book.

For our purposes, a traditional MDI application provides the means to discuss and demonstrate the manner in which the .NET Framework supports such applications. In chapter 20, we convert the existing MyPhotos application into an MDI application, as shown here in figure 18.2. As you can see, this application incorporates the Form classes we created in part 2 of this book.

The reuse of our existing classes is possible because of the manner in which the Form class in general and MDI support in particular is integrated into the Windows Forms hierarchy. As we discussed in chapter 7, a Form object is a Control instance that happens to display an application window. For MDI applications, Form controls are contained by a parent Form. The contained forms can be resized and moved within their container, just like any other control within a container, and yet can still display menus, status bars, and other controls. The relationship between MDI parent and child forms is slightly different than the relationship between control containers and controls, as we see in a moment.

18.1.3 Explorer interfaces

In an explorer interface a hierarchy or other organization is imposed on a concept or set of concepts, with the hierarchy shown on one side and details related to a selected item shown in the main portion of the window. The classic example of this style is the Windows Explorer interface, which displays information about the data available on the computer, be it from disk, CD, network, or another source.

Explorer interfaces give an overview of the concept in question, and typically provide a tree control to allow the user to explorer the data at a high level or successive levels of detail. The user can select a specific item, or *node*, in the tree and see information about this item in the other portion of the interface, most commonly in the form of a list showing the selected item's contents.

In this chapter we start building the MyAlbumExplorer interface as an example of this style. The end result is shown in figure 18.3. The next section begins this discussion with an exploration of splitter controls and an introduction to the TreeView and ListView controls.

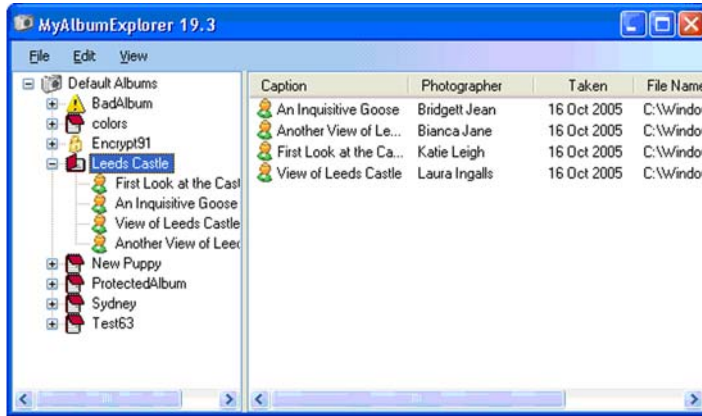


Figure 18.3 Our explorer interface allows a user to explore a collection of photo albums.

18.2 EXPLORER INTERFACES IN .NET

Now that we've reviewed the three main interface styles for Windows applications, let's take a look at the explorer interface in more detail. Part 1 of the book built a single document interface, and we examine multiple document interfaces in chapter 20.

We discuss the explorer interface style in this as well as the next chapter. This chapter focuses more on the interface itself, and the tree view and splitter controls commonly used for this interface. Chapter 19 elaborates on this discussion and examines the `ListView` control.

While building an explorer interface has always been possible in .NET, the release of .NET 2.0 made this task much easier. In the .NET 1.*x* releases, a `Splitter` class was provided that split a container into two distinct regions. The proper location of the splitter object was dependent on the order in which controls were added to the container, which could get rather confusing. This class is still available in .NET 2.0, but Microsoft no longer recommends its use.

Recognizing the error of their ways, the Windows Forms team has added the `SplitContainer` control. This class works much like the `JSplitPane` class in the Java Swing interface, for you Java aficionados out there. We discuss split containers in a moment.

Once a `SplitContainer` object is on a form, a `TreeView` control can be dropped into the left-hand side and a `ListView` control into the right-hand side. Explorer interfaces in Windows were never quite so easy.

We begin our discussion with the splitter controls.

18.2.1 The SplitContainer class

As already indicated, the `SplitContainer` class is the recommended mechanism for splitting a container into multiple sections. The members of this class are shown

in .NET Table 18.1. As you can see, the `SplitContainer` class provides a number of properties to control the appearance of the two resulting panels. The size of each panel can be adjustable or fixed, depending on the `FixedPanel` and `IsSplitterFixed` properties described in the table. Panel-specific properties define a minimum size for each panel and allow either panel to be hidden from view.

.NET Table 18.1 SplitContainer class

New in 2.0 The `SplitContainer` class is a container control that provides two resizable panels separated by a movable bar. This control composes two panels, encapsulated as `SplitterPanel` objects, divided by a movable bar, or splitter, into a single control. This class is part of the `System.Windows.Forms` namespace, and inherits from the `ContainerControl` class.

The `SplitterPanel` class is a panel suitable for placing within a split container control. This class is part of the `System.Windows.Forms` namespace, and inherits from the `Panel` class.

Public Properties	*BorderStyle*	Gets or sets the border style applied to the panels within the control.
	FixedPanel	Gets or sets which panel, if any, cannot be resized.
	IsSplitterFixed	Gets or sets whether the splitter is movable.
	Orientation	Gets or sets the orientation of the panels and splitter within the control.
	Panel1	Gets the left or top panel in the container as a `SplitterPanel` instance. The `Panel2` property gets the right or bottom panel.
	Panel2Collapsed	Gets or sets whether the right or bottom panel is visible or hidden. The `Panel1Collapsed` property gets or sets whether the left or top panel is collapsed.
	Panel1MinSize	Gets or sets the minimum size for the left or top panel. The `Panel2MinSize` property affects the right or bottom panel.
	SplitterDistance	Gets or sets the distance, in pixels, of the splitter from the left or top edge of the form.
	SplitterIncrement	Gets or sets the smallest distance, in pixels, that the splitter will move.
	SplitterRectangle	Gets the location and size of the splitter relative to the container.
	SplitterWidth	Gets or sets the width of the splitter in pixels.
Public Events	*SplitterMoved*	Occurs after the splitter has moved.
	SplitterMoving	Occurs when the splitter is about to move.

The movable bar within a split container, called the *splitter*, is affected by properties that define its location and size within the container. The `SplitterIncrement` property allows a splitter to move in increments larger than one pixel. Additional

tasks can be performed before or after a user moves the splitter by handling the `SplitterMoved` and `SplitterMoving` events.

Split containers can orient their panels in the horizontal or vertical direction, and can be nested to allow multiple display areas within a form. A quick example of these two concepts appears in listing 18.1, which creates the form shown in figure 18.4. This example displays a horizontal split container nested within the right panel of a vertical split container. Each panel assigns a different `BackColor` setting so the location of each splitter panel is immediately apparent.

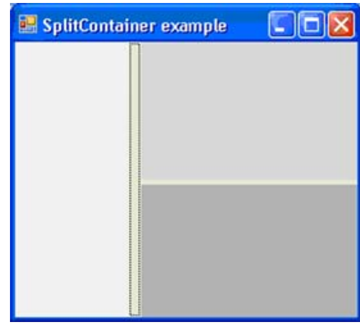


Figure 18.4 This sample application demonstrates nested SplitContainer controls.

Listing 18.1 Sample application using SplitContainer

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

namespace SplitContainerExample
{
  static class SplitContainerProgram
  {
    [STAThread]
    static void Main()
    {
      Application.EnableVisualStyles();

      SplitContainer split2 = new SplitContainer();
      split2.Dock = DockStyle.Fill;
      split2.Orientation = Orientation.Horizontal;
      split2.Panel1.BackColor = Color.LightGray;
      split2.Panel2.BackColor = Color.DarkGray;

      // Create split1 container to hold split2 container
      SplitContainer split1 = new SplitContainer();
      split1.Dock = DockStyle.Fill;
      split1.Panel1.BackColor = Color.WhiteSmoke;
      split1.SplitterWidth = 10;
      split1.Panel2.Controls.Add(split2);

      // Create a form to hold split1 container
      Form f = new Form();
      f.Text = "SplitContainer example";
      f.Controls.Add(split1);

      // Display the form
      Application.Run(f);
    }
  }
}
```

18.2.2 The TreeView class

The `SplitContainer` class is great for dividing a form or other container into two parts. The next question, of course, is what to place in each part. One of the more common controls for the left side of a split form is the `TreeView` control.

One of the better-known examples of the `TreeView` control is the Windows Explorer interface, where the Folders view provides a tree of the devices, folders, network locations, and other information about the local computer environment. The Windows Forms `TreeView` class supports this functionality, and is summarized in .NET Table 18.2.

Each item in a tree view is called a *tree node*, or just a *node*. Tree nodes can contain additional nodes, called *child nodes*, to arbitrary levels in order to represent a hierarchy of objects in a single tree. Tree nodes at the top level of a tree view control are called *root nodes*. Figure 18.5 shows some of concepts and classes used with the Windows Forms `TreeView` control.

The members in .NET Table 18.2 are just some of the members defined by the `TreeView` class explicitly. As you can see, members are provided to manage the nodes in the tree, including the selection, drawing, and contents of each node. We use many of these members as we build our sample explorer application.

A short summary of some members not mentioned in .NET Table 18.2 is probably worthwhile. The nodes in the tree can be sorted alphabetically by setting the `Sorted` property to `true`, or a custom `IComparer` interface can be assigned to the `TreeViewNodeSorter` property to specify a custom sort for the nodes.

An additional set of properties defines how the nodes in the tree are laid out. The `Indent` property indicates how many pixels to indent each level in the tree, while the color, existence, and appearance of lines between nodes are affected by the `LineColor`, `ShowLines`, `ShowPlusMinus`, and `ShowRootLines` properties.

As shown in figure 17.5, each node in a tree view can display an icon next to the node label. The `ImageList` property defines the images to use for this purpose.

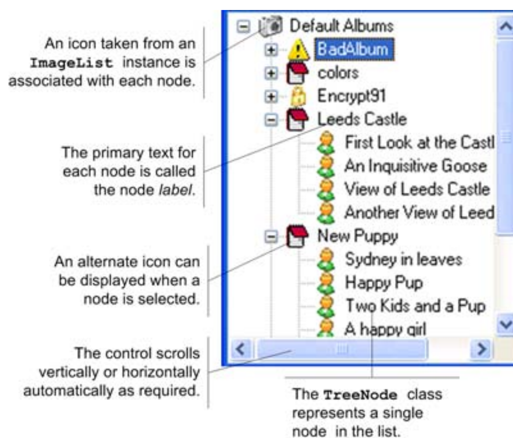


Figure 18.5
Icons in a tree view typically indicate additional information about each node. In this example a notebook indicates a valid album, a lock indicates an encrypted album, and an exclamation point indicates an album that could not be opened.

.NET Table 18.2 TreeView class

The `TreeView` class is a control that displays a collection of labeled items in a tree-style hierarchy. Typically an icon is displayed for each item in the collection to provide a graphical indication of the nature or purpose of the item. Items in a tree view are referred to as *nodes*, and each node is represented by a `TreeNode` class instance. The `TreeView` class is part of the `System.Windows.Forms` namespace, and inherits from the `Control` class.

Public Properties	BorderStyle	Gets or sets the border style for the tree view
	CheckBoxes	Gets or sets whether a check box appears next to each node
	DrawMode	Gets or sets the drawing mode for the tree view
	FullRowSelect	Gets or sets whether the selection of a node should span the width of the tree view
	HideSelection	Gets or sets whether the selected node is highlighted when the tree view loses focus
	HotTracking	Gets or sets whether a node label alters its appearance as the mouse passes over it
	ImageList	Gets or sets the image list that contains the images to use with the tree's nodes
	LabelEdit	Gets or sets whether node labels can be edited
	Nodes	Gets the tree's collection of `TreeNode` objects
	PathSeparator	Gets or sets the delimiter used for a tree node path
	SelectedNode	Gets or sets the selected tree node
	Sorted	Gets or sets whether nodes in the tree are sorted
	StateImageList	Gets or sets the image list used to indicate the state of each tree node
	TopNode	Gets or sets the first fully visible node in the tree
	VisibleCount	Gets the maximum number of nodes that can be fully visible in the tree
Public Methods	CollapseAll	Collapses all nodes in the tree
	ExpandAll	Expands all nodes in the tree
	GetNodeAt	Retrieves the tree node at the specified pixel location
	GetNodeCount	Returns the number of top-level nodes in the tree, or the total number of nodes in the entire tree
Public Events	AfterExpand	Occurs after a tree node has been expanded
	AfterLabelEdit	Occurs after a tree node label has been edited
	BeforeSelect	Occurs before a tree node is selected
	DrawNode	Occurs when a node should be drawn
	ItemDrag	Occurs when a user begins dragging a node
	NodeMouseClick	Occurs when a user clicks a tree node

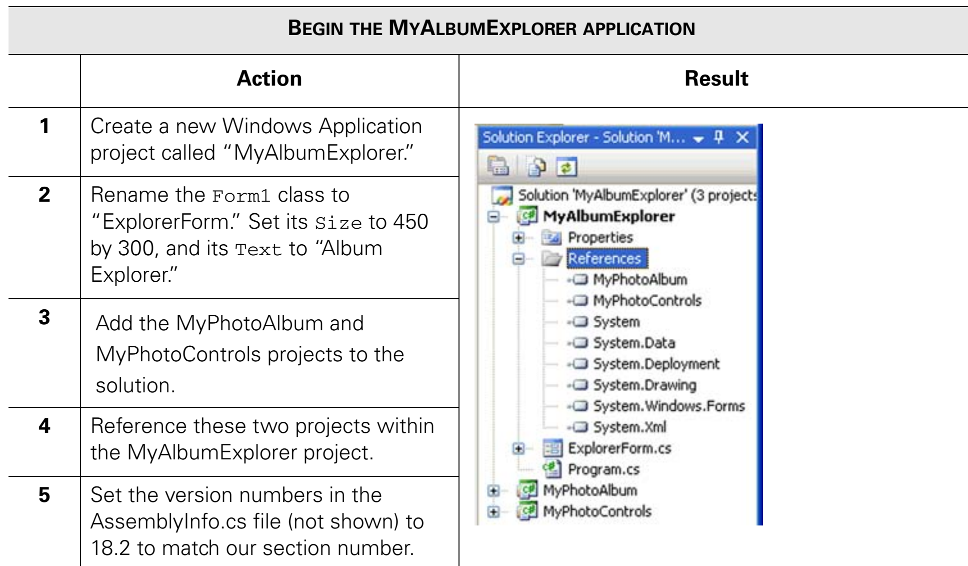
Other image properties include the `ImageIndex` property to indicate the index of the default image and the `SelectedImageIndex` property to indicate the index of the default image for a selected node. The default images can alternately be specified by image name, using the `ImageKey` and `SelectedImageKey` properties, which we demonstrate in the next section.

A second set of images can also be displayed to represent the state of each node. The images for the state icons are defined by the `StateImageList` property. When the `CheckBoxes` property is `true` and state images are defined, the first two images in the `StateImageList` collection are used to indicate the unchecked and checked state, respectively.

One other set of members we should probably highlight consists of the `After` and `Before` events. A few of these are shown in .NET Table 18.2, but be aware that before and after events occur when a user clicks on node check boxes, expands or collapses nodes, edits node labels, or selects nodes.

18.2.3 Creating an explorer interface

So let's see a tree view in action. The following steps begin a new MyAlbumExplorer project for use in this and the next chapter, as well as show the creation of an explorer-style interface.



BEGIN THE MYALBUMEXPLORER APPLICATION		
	Action	**Result**
1	Create a new Windows Application project called "MyAlbumExplorer."	
2	Rename the `Form1` class to "ExplorerForm." Set its `Size` to 450 by 300, and its `Text` to "Album Explorer."	
3	Add the MyPhotoAlbum and MyPhotoControls projects to the solution.	
4	Reference these two projects within the MyAlbumExplorer project.	
5	Set the version numbers in the AssemblyInfo.cs file (not shown) to 18.2 to match our section number.	

	Begin the MyAlbumExplorer application (continued)	
	Action	**Result**

6	In the ExplorerForm.cs source file, override the `OnLoad` method to display the version number in the title bar.	`protected override void OnLoad(EventArgs e)`
		`{`
		`  // Assign title bar`
		`  Version v = new Version(Application.`
		`                   ProductVersion);`
		`  this.Text = String.Format(`
		`      "MyAlbumExplorer {0:#}.{1:#}",`
		`      v.Major, v.Minor);`
		
		`  base.OnLoad(e);`
		`}`

7	Set the `Icon` property for the `ExplorerForm` form to the file "icons/WinXP/camera.ico" in the common image library.	camera.ico
8	Also set this icon as the Application icon in the MyAlbumExplorer project properties.	This icon displays in Windows operating system to represent the application. **Note:** We discussed icons for both forms and applications in chapter 15 in section 15.1.2.

This creates our new solution, with the version number displayed in the title bar and a new icon that represents our application. As you may recall, the common image library mentioned in step 7 is our shorthand for the graphics files provided with Visual Studio.

To turn this into an explorer interface, we need to split the display area into two sections, with a tree on the left and information about the selected tree node on the right. To do this, we need a `SplitContainer` and `TreeView` control, which we add here so they are available in the next section.

	Begin the MyAlbumExplorer application	
	Action	**Result**
9	Drag a `SplitContainer` control onto the form.	Album Explorer
10	Drag a `TreeView` control onto the left side of the split container. **Settings** **Property** — **Value** (Name) — albumTree Dock — Fill ShowNodeToolTips — True	

When you add the split container, Visual Studio displays the text Panel1 and Panel2 on each `SplitterPanel` instance to make them easier to identify. Our application does not do very much yet, but it does work. The next section discusses how to populate this tree view with our album data.

18.3 TREE NODES

Each element in a tree view is called a tree view node, or simply a node. Tree views contain nodes, which in turn may contain other nodes, which may contain still other nodes, and so forth. Each node in the tree is represented by a `TreeNode` object. Nodes in a tree view can be created at design time in Visual Studio, or dynamically within application code.

This section presents the `TreeNode` class in detail and discusses how to use this class within an application.

18.3.1 The TreeNode class

The `TreeNode` class provides members for presenting and manipulating nodes within a tree view. A summary of these members is shown in .NET Table 18.3. The table focuses on members that are somewhat unique to the `TreeNode` class. The class also supports members similar to the Windows Forms `Control` class, such as the `BackColor`, `ContextMenuStrip`, `ForeColor`, `Parent`, `Tag`, and `Text` properties. Of these, only the `Text` property is shown in the table, since it defines the string that appears as the node's label.

In addition to the `ImageIndex` and `SelectedImageIndex` properties shown in .NET Table 18.3, there are other properties that define the images displayed next to the node. The available images are defined by the `ImageList` and `StateImageList` properties for the containing `TreeView` control. The `ImageKey` and `SelectedImageKey` properties can be used to specify an image by its name, or key, rather than its index. The `StateImageIndex` or `StateImageKey` property can be used to specify the image used to indicate the state of the node. These default and selected image properties apply when the associated `TreeView` property has an assigned `ImageList`; the state image properties only apply when a `StateImageList` instance is assigned to the tree.

Also worth a mention are the node navigation properties in the `TreeNode` class. The `NextNode` and `PrevNode` properties get the next or previous sibling node, at the same level as the current node. The `NextVisibleNode` and `PrevVisibleNode` properties get the next or previous child, sibling, or parent node that is visible in the tree.

.NET Table 18.3 TreeNode class

The `TreeNode` class is a marshaled object that represents an element, or node, within a `TreeView` control. A `TreeNode` object can contain other nodes to represent a hierarchy of objects within a tree view. Each `TreeNode` instance is contained by at most one `TreeView` or `TreeNode` object. This class is part of the `System.Windows.Forms` namespace, and inherits from the `System.MarshalByRefObject` class.

Public Properties	FirstNode	Gets the first node, if any, contained by this node.
	FullPath	Gets the path to this node, starting at the root node, using the tree's `PathSeparator` setting as a delimiter.
	ImageIndex	Gets or sets the index into the tree's image list of the default image for this node.
	IsEditing	Gets whether this node is currently being edited.
	IsExpanded	Gets whether the children of this node are displayed.
	IsSelected	Gets whether this node is currently selected.
	Level	Gets the zero-based depth of this node in the tree.
	NodeFont	Gets or sets the font used to display the node's label.
	Nodes	Gets the collection of nodes contained by this node.
	SelectedImage-Index	Gets or sets the index into the tree's image list of the image to display when this node is selected. The `Index` property gets or sets the index of the default image.
	Text	Gets or sets the label text to display for this node.
	ToolTipText	Gets or sets the tooltip for this node.
	TreeView	Gets the `TreeView` control containing this node.
Public Methods	Collapse	Ensures that no children of this node are displayed.
	ExpandAll	Expands all tree nodes contained by this node.
	Toggle	Toggles the tree node between the expanded or collapsed state, based on the `IsExpanded` setting.

18.3.2 Creating tree nodes

Now that we've reviewed the members of the `TreeNode` class, let's talk about how to create these objects. When a `TreeView` control is placed on a form within Visual Studio, you can add nodes to it within the designer or dynamically. The TreeNode Editor window, shown in figure 18.6, is available from the Properties window or the context menu for the `TreeView` control.

Programmatically, of course, tree nodes are created much like any other class. The `TreeNode` constructor has a number of overloads, and allows the caller to define the label, child nodes, or associated images.

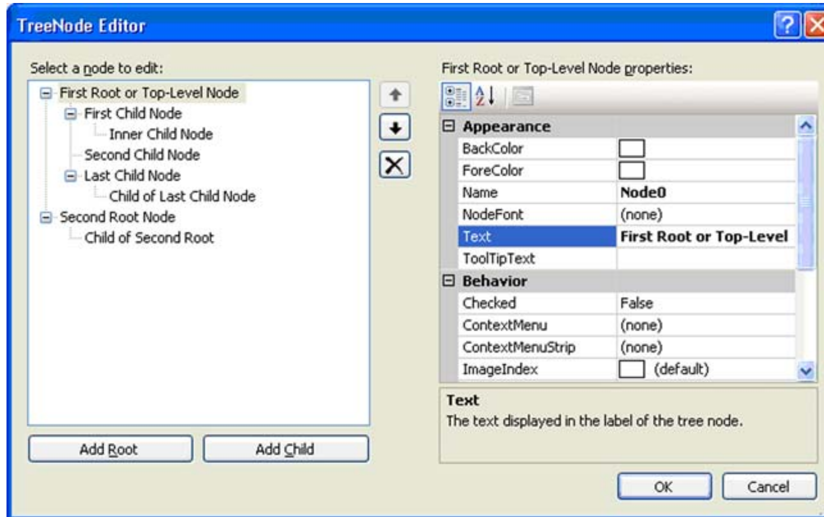


Figure 18.6 The TreeNode Editor in Visual Studio allows both root and child nodes to be created within the designer.

Listing 18.2 illustrates how our MyAlbumExplorer application can be extended to add tree nodes for each album in the default album directory. This code assumes that a `ScrollablePictureBox` control named `spbxPhoto` is docked to the right side of the split container, and that `BeforeSelect` and `AfterSelect` event handlers have been defined for the `TreeView` control. The application resulting from these changes is shown in figure 18.7.

Listing 18.2 Sample MyAlbumExplorer implementation that creates album and photograph nodes programmatically

```
. . .
using System.IO;
using Manning.MyPhotoAlbum;
using Manning.MyPhotoControls;

namespace MyAlbumExplorer
{
  public partial class ExplorerForm : Form
  {
    public ExplorerForm()
    {
      InitializeComponent();
    }

    private Photograph _priorPhoto = null;

    protected override void OnLoad(EventArgs e)
    {
      // Assign title bar
```

```
  Version v = new Version(Application.ProductVersion);
  this.Text = String.Format("MyAlbumExplorer {0:#}.{1:#}",
    v.Major, v.Minor);

  // Define nodes for the default albums
  albumTree.Nodes.Clear();
  string[] albums = Directory.GetFiles(
                        AlbumManager.DefaultPath, "*.abm");
  foreach (string file in albums)
  {
    string baseName = Path.GetFileNameWithoutExtension(file);
    try
    {
      // Define root album node
      PhotoAlbum album = AlbumStorage.ReadAlbum(file);

      string title = album.Title;
      if (String.IsNullOrEmpty(title))
        title = baseName;

      TreeNode node = albumTree.Nodes.Add(title);
      node.ToolTipText = file;

      foreach (Photograph p in album)
      {
        // Define child photograph node
        TreeNode child = node.Nodes.Add(p.Caption);
        child.ToolTipText = p.FileName;
        child.Tag = p;
      }
    }
    catch (AlbumStorageException)
    {
      // Unable to open album
      TreeNode node = albumTree.Nodes.Add(baseName);
      node.ToolTipText = "Unable to open album: " + file;
    }
  }

  base.OnLoad(e);
}

private void albumTree_BeforeSelect(object sender,
    TreeViewCancelEventArgs e)
{
  if (_priorPhoto != null)
  {
    spbxPhoto.Image = null;
    _priorPhoto.ReleaseImage();
    _priorPhoto = null;
  }
}

private void albumTree_AfterSelect(object sender,
    TreeViewEventArgs e)
```

1 Creates album node

2 Creates photograph node

3 Disposes of prior image

```
    {
      Photograph p = e.Node.Tag as Photograph;
      if (p != null)
        spbxPhoto.Image = p.Image;

      _priorPhoto = p;
    }
  }
}
```

❹ **Assigns selected image**

This code uses many of the concepts and ideas discussed in part 2 of the book. A few key points worth a mention are as follows.

❶ When the form is loaded, a root node is created for each album in the default album directory. If available, the album's title is used as the node's label; otherwise the base file name of the album is used. Notice the `catch` block that handles `AlbumStorageException` errors by creating a top-level node for the errant album. This exception occurs if the album is encrypted or if some other recognizable problem occurs.

❷ When an album loads successfully, a child node is created for each photograph in the album. The node's `Tag` property is assigned to the associated `Photograph` object.

❸ Since our application may display many images, we need to be careful not to leave too many resource-depleting bitmaps lying around. Our `BeforeSelect` event handler ensures that any photo previously displayed in the scrollable picture box is released before a new photo is selected.

❹ The `AfterSelect` event handler looks to see if the selected node is tagged with a `Photograph` object. If so, then its image is displayed; otherwise the node is ignored. Our `ScrollablePictureBox` custom control handles any required scrolling automatically.

Before and after events in the `TreeView` class receive different event arguments. All after events receive a `TreeViewEventArgs` object, as shown in .NET Table 18.4. All before events receive a `TreeViewCancelEventArgs` object. The latter class provides the same `Action` and `Node` properties as the former class, and it inherits from

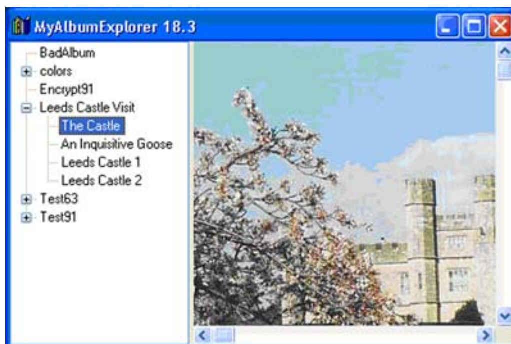


Figure 18.7
This application demonstrates the TreeView control used in conjunction with the ScrollablePictureBox custom control created in chapter 17.

the CancelEventArgs class to give the caller the option of canceling the event by assigning true to the Cancel property.

.NET Table 18.4 TreeViewEventArgs class

The TreeViewEventArgs class specifies the event data for events that occur after a tree view operation completes. This class is part of the System.Windows.Forms namespace, and inherits from the System.EventArgs class.

Public Properties	Action	Gets the TreeViewAction enumeration value representing the action that caused this event.
	Node	Gets the TreeNode object for the current operation.

The application in listing 18.2 is fine and works quite well. It can be extended to support all sorts of features and capabilities. A disadvantage of this approach is that all of the code to manipulate the TreeView and other controls is part of the ExplorerForm.cs file. As the application grows, this file gets larger and larger, and thus harder and harder to maintain.

An alternate approach is to encapsulate the tree view as custom controls. This would force us to encapsulate the logic for the customized control in a single file, providing better organization and simplifying reuse if desired. We finish this chapter by creating such a TreeView control for the purpose of displaying and manipulating album directories. This custom tree view is incorporated into the MyAlbumExplorer interface as part of our discussion.

TRY IT! Add the code in listing 18.2 to the ExplorerForm.cs file to get this application working in Visual Studio. There are many ways to improve this application. One addition you might try is to dock a ListBox control on the right side of the split container, and display the selected album's photographs in this list whenever an album node is selected. To do this, you need to assign the Visible property on both the ListBox and ScrollablePictureBox controls whenever a node is selected to ensure that only the desired control is visible.

18.4 CUSTOM TREE VIEWS

Like any control, the TreeView class can be customized in all sorts of ways to provide better, faster, and stronger functionality than what is provided in the base Windows Forms control. We discussed such customization in chapter 17, so we do not go through this again.

Sometimes it is useful to customize a control not so much for reuse or functionality purposes, but as a way to organize or otherwise encapsulate portions of a graphical interface. We introduced this idea in chapter 14, where we discussed the idea of defining UserControl classes to encapsulate a set of interfaces for an application, in

that case a user control object that held a calendar showing dates when photographs were taken.

The `UserControl` example in chapter 14 represents this alternate reason to customize a control: to encapsulate a portion of an interface. This not only prevents the associated `Form` class from holding thousands of lines of user interface code, but also forces better encapsulation of the functionality required.

In this section we see how this idea is applied to the `TreeView` class. We create an `AlbumTreeView` custom control that displays one or more albums in a tree. This class is not strictly required, but places everything we need to display albums and photographs in our MyAlbumExplorer application in a single class.

18.4.1 Creating a custom tree view

To begin this great work, let's create the `AlbumTreeView` class and indicate the namespaces required. There is nothing new here.

HIDE THE TOOLBAR IN OUR CHILD FORM

	Action	Result

1	Add a new class to the MyAlbumExplorer project called AlbumTreeView.	`using System;`
		`using System.ComponentModel;`
		`using System.IO;`
		`using System.Windows.Forms;`
		
		`using Manning.MyPhotoAlbum;`
		`using Manning.MyPhotoControls;`
		
		`namespace MyAlbumExplorer`
		`{`

2	Indicate we will use the indicated `System` and `Manning` namespaces.	

3	Make the `AlbumTreeView` class internal and inherit from the `TreeView` class.	`  internal class AlbumTreeView : TreeView`
		`  {`
		
		`  }`
		`}`

This defines our new control class, which we fill in over the course of this section. We should create a test application for this new control to verify that these and subsequent changes work correctly. This might be a form that contains only the `AlbumTreeView` class and displays the albums in the default album directory.

To save time and space, we do not create such a test application here, but feel free to do this if you wish so you can verify our implementation as we go along. The rest of us must wait until the end of the chapter to see this control in action.

With our control defined, the next step is to define exactly what type of support we need to provide in the class.

18.4.2 Creating custom tree nodes

The goal of our tree view is to display albums. Albums appear in directories, and albums contain photographs. So in our tree, we should probably have three types of nodes: album directory nodes, album nodes, and photograph nodes. Rather than trying to manage these within the tree view control itself, we create a custom class for each type of node.

A custom `TreeNode` class, much like a custom `TreeView`, is quite useful to encapsulate the behavior required for a specific node as well as custom tasks that users may wish to perform against the node.

So, without further adieu, allow me to present the `AlbumDirectoryNode` class.

CREATE AN ALBUMDIRECTORYNODE CLASS

	Action	Result

1	In the MyAlbumExplorer project, create an internal class called `AlbumDirectoryNode`.
	
	Note: This could be a public class, but internal works fine for our purposes.
		`using System;`
		`using System.IO;`
		`using System.Windows.Forms;`
		
		`namespace MyAlbumExplorer`
		`{`
		`  internal class AlbumDirectoryNode : TreeNode`
		`  {`

2	Create a private field and public property for the directory represented by this node.	`    private string _albumDir;`
		`    public string AlbumDirectory`
		`        { get { return _albumDir; } }`

3	Define a constructor that accepts the name and directory for this node.	`    public AlbumDirectoryNode(string name,`
		`        string albumDir) : base(name)`
		`    {`
		`      if (albumDir == null)`
		`        throw new ArgumentNullException("albumDir");`
		`      if (!Directory.Exists(albumDir))`
		`        throw new ArgumentException(`
		`            "albumDir is not a valid directory");`

4	Assign the directory field to the given value, and create a fake child node in this node.	`      _albumDir = albumDir;` `      this.Nodes.Add("child");`

5	Also define both image keys for the node to use the name "AlbumDir."	`      this.ImageKey = "AlbumDir";`
		`      this.SelectedImageKey = "AlbumDir";`
		`    }`

6	Define an `AlbumFiles` property that gets the collection of album files in the directory.	`    public string[] AlbumFiles`
		`    {`
		`      get { return Directory.GetFiles(`
		`              AlbumDirectory, "*.abm"); }`
		`    }`

Create an AlbumDirectoryNode class *(continued)*		
	Action	**Result**

7	Define a private field and public `AlbumNodes` property to retrieve the collection of album nodes for this directory node.	`private AlbumNode[] _albumNodes = null;`
		`public AlbumNode[] AlbumNodes`
		`{`
		`  get`
		`  {`
		`    CreateAlbumNodes();`
		`    return _albumNodes;`
		`  }`
		`}`

8	Define a `CreateAlbumNodes` method to create the album nodes, if necessary.	`public void CreateAlbumNodes()`
		`{`
		`  string[] files = AlbumFiles;`
		`  int count = files.Length;`

9	Replace the fake node with an `AlbumNode` for each album file in the directory.
	
	Note: This relies on the `AlbumNode` constructor, which we create in a moment.
		`  if (_albumNodes == null`
		`        || _albumNodes.Length != count)`
		`  {`
		`    Nodes.Clear();`
		`    _albumNodes = new AlbumNode[count];`
		`    for (int i = 0; i < count; i++)`
		`    {`
		`      // Add album node`
		`      string s = files[i];`
		`      string name`
		`        = Path.GetFileNameWithoutExtension(s);`
		`      _albumNodes[i] = new AlbumNode(name, s);`
		`    }`
		
		`    Nodes.AddRange(_albumNodes);`
		`  }`
		`}`
		`}`
		`}`

This code defines everything we need to manage an album directory node in our `TreeView` control. The constructor defines a fake child node to ensure that the `TreeView` control displays a plus (+) sign next to the node. When the user expands the node, our control will need to call the `CreateAlbumNodes` method to replace the fake node with the actual nodes for the album.

Also note that we assign an `ImageKey` value for the node. This has little meaning here, but becomes relevant later in the chapter when we incorporate our `AlbumTreeView` class into our application.

The other nodes are a bit more involved, so we create them in the next section.

18.4.3 Using interfaces with custom nodes

At times, especially in more complex applications, it is useful to share some base functionality among the various custom node classes. One approach for this is to define a base class, for example `BaseTreeNode`. All custom nodes inherit from the base class and override or otherwise inherit the base functionality as appropriate.

Another approach is to define an interface and have some or all of the custom node classes inherit this interface. We take the interface approach in our album and

photograph nodes in order to refresh each node when the associated object changes. The following steps define an `IRefreshableNode` interface for this purpose and begin our implementation of the `AlbumNode` class.

<table>
<tr><th colspan="3">CREATE AN ALBUMNODE CLASS</th></tr>
<tr><th></th><th>Action</th><th>Result</th></tr>
<tr><td>1</td><td>Add a new internal interface in the project called <code>IRefreshableNode</code>, and define a single <code>RefreshNode</code> method within this interface.</td><td>

```
namespace MyAlbumExplorer
{
  internal interface IRefreshableNode
  {
    void RefreshNode();
  }
}
```

</td></tr>
<tr><td>2</td><td>Add a new internal class called <code>AlbumNode</code> to the project, and have it inherit from the <code>TreeNode</code> class and support the <code>IDisposable</code> and <code>IRefreshableNode</code> interfaces.</td><td>

```
using System;
using System.IO;
using System.Windows.Forms;

using Manning.MyPhotoAlbum;
using Manning.MyPhotoControls;

namespace MyAlbumExplorer
{
 internal class AlbumNode
     : TreeNode, IDisposable, IRefreshableNode
 {
```

</td></tr>
<tr><td>3</td><td>Create private fields to hold the path and manager for the album, and a public <code>AlbumPath</code> property to retrieve the path.</td><td>

```
  private string _albumPath;
  private AlbumManager _manager;

  public string AlbumPath
      { get { return _albumPath; } }
```

</td></tr>
<tr><td>4</td><td>Define a constructor that accepts the name and album path for the new node.</td><td>

```
  public AlbumNode(string name, string albumPath)
    : base(name)
  {
    if (albumPath == null)
      throw new ArgumentNullException("albumPath");
    if (!File.Exists(albumPath))
      throw new ArgumentException(
          "albumPath is not a valid path");

    _manager = null;
    _albumPath = Path.GetFullPath(albumPath);
    this.Nodes.Add("child");
```

</td></tr>
<tr><td>5</td><td>Set the image key for the node to “AlbumLock” if the album is encrypted. Otherwise use “Album” as the key, with “AlbumSelect” as the selected key name.</td><td>

```
    if (AlbumStorage.IsEncrypted(albumPath))
    {
      this.ImageKey = "AlbumLock";
      this.SelectedImageKey = "AlbumLock";
    }
    else
    {
      this.ImageKey = "Album";
      this.SelectedImageKey = "AlbumSelect";
    }
  }
```

</td></tr>
</table>

<table>
<tr><th colspan="3">CREATE AN ALBUMNODE CLASS (CONTINUED)</th></tr>
<tr><th></th><th>Action</th><th>Result</th></tr>
<tr><td>6</td><td>Implement a <code>GetManager</code> method to retrieve the manager for the node. Accept an <code>interactive</code> parameter to indicate whether to ask for the album password if necessary.</td><td><pre>
public AlbumManager GetManager(bool interactive)
{
  if (_manager == null)
  {
    string path = AlbumPath;
    string pwd = null;

    try
    {
</pre></td></tr>
<tr><td>7</td><td>If the album is encrypted, allow the user to enter the password if running interactively.

If not interactive, or the user cancels the password dialog box, abort the method by returning <code>null</code>.</td><td><pre>
      if (AlbumStorage.IsEncrypted(path))
      {
        DialogResult result = DialogResult.None;
        if (interactive)
        {
          result = MessageBox.Show("The album "
            + path + " is encrypted. "
            + "Do you wish to open this album?",
            "Encrypted Album",
            MessageBoxButtons.YesNo,
            MessageBoxIcon.Question,
            MessageBoxDefaultButton.Button2);
        }

        if (result != DialogResult.Yes
          || !AlbumController.CheckAlbumPassword(
                                  path, ref pwd))
          return null; // cancelled
      }
</pre></td></tr>
<tr><td>8</td><td>If the manager is opened successfully, reset the default and selected image key settings.</td><td><pre>
      _manager = new AlbumManager(path, pwd);
      this.ImageKey = "Album";
      this.SelectedImageKey = "AlbumSelect";
    }
</pre></td></tr>
<tr><td>9</td><td>If an <code>AlbumStorage</code> exception occurs, set the image keys to “AlbumError” and the manager to <code>null</code>.</td><td><pre>
    catch (AlbumStorageException ex)
    {
      if (interactive)
        MessageBox.Show("The album could not "
          + "be opened [" + ex.Message + "]");

      this.ImageKey = "AlbumError";
      this.SelectedImageKey = "AlbumError";
      _manager = null;
    }
  }
</pre></td></tr>
<tr><td>10</td><td>When finished, return the resulting manager.</td><td><pre>
  return _manager;
}
</pre></td></tr>
<tr><td>11</td><td>Implement a <code>GetAlbum</code> method to return the <code>PhotoAlbum</code> associated with the manager, if any.</td><td><pre>
public PhotoAlbum GetAlbum(bool interactive)
{
  AlbumManager mgr = GetManager(interactive);

  if (mgr == null) return null;
  else return mgr.Album;
}
</pre></td></tr>
</table>

<table>
<tr><th colspan="3">CREATE AN ALBUMNODE CLASS (CONTINUED)</th></tr>
<tr><th></th><th>Action</th><th>Result</th></tr>
<tr><td>12</td><td>Implement a <code>RemoveChildren</code> method to clean up any open images and clear the child nodes of the album node.</td><td><pre>
public void RemoveChildren()
{
  AlbumManager mgr = GetManager(false);
  if (mgr != null)
  {
    foreach (Photograph p in mgr.Album)
      p.ReleaseImage();
  }

  Nodes.Clear();
  Nodes.Add("child");
}
</pre></td></tr>
<tr><td>13</td><td>Implement the <code>IDisposable</code> interface to dispose of any manager created for this node.</td><td><pre>
public void Dispose()
{
  if (_manager != null)
    _manager.Album.Dispose();
  _manager = null;
}
</pre></td></tr>
<tr><td>14</td><td>Implement the <code>IRefreshableNode</code> interface to refresh the <code>Text</code> property of the node, if necessary.</td><td><pre>
public void RefreshNode()
{
  AlbumManager mgr = GetManager(false);
  if (mgr != null && this.Text != mgr.ShortName)
    this.Text = mgr.ShortName;
}
</pre></td></tr>
<tr><td>15</td><td>Define a <code>RenameAlbum</code> method to rename the path for the associated album to a given name.

How-to
Rely on a <code>RenameAlbum</code> method from the <code>AlbumManager</code> class, which does not yet exist. Display a message if an <code>ArgumentException</code> is thrown to indicate that the album name already exists.</td><td><pre>
public bool RenameAlbum(string newName)
{
  try
  {
    AlbumManager mgr = GetManager(false);
    if (mgr == null)
      _albumPath = AlbumManager.RenameAlbum(
                        AlbumPath, newName);
    else
    {
      mgr.RenameAlbum(newName);
      _albumPath = mgr.FullName;
    }

    return true;
  }
  catch (ArgumentException)
  {
    MessageBox.Show("Unable to rename album. An "
        + "album with that name already exists.");
    return false;
  }
}
</pre></td></tr>
</table>

CREATE AN ALBUMNODE CLASS *(CONTINUED)*

	Action	Result
16	Define an `UpdatePath` method to modify the node when the path for the album is modified externally.	see code below

```
  public void UpdatePath(string newPath)
  {
    if (!File.Exists(newPath))
      throw new ArgumentException(
          "newPath must be valid path");

    AlbumManager mgr = GetManager(false);
    if (mgr != null)
    {
      // Just pull new info from the manager
      _albumPath = mgr.FullName;
      Text = mgr.ShortName;
    }
    else
    {
      // use given path to update node
      _albumPath = newPath;
      Text = Path.GetFileNameWithoutExtension(
                                       newPath);
    }
  }
 }
}
```

	Action	Result
17	In the MyPhotoAlbum project, add a `RenameAlbum` method to the `AlbumManager` class that invokes a corresponding static method.	see code below

```
  public void RenameAlbum(string newName)
  {
    _name = RenameAlbum(FullName, newName);
  }
```

	Action	Result
18	Implement a static `RenameAlbum` method that accepts an old and new filename for an album.	see code below

```
  public static string RenameAlbum(
      string oldPath, string newName)
  {
    string dir = Path.GetDirectoryName(oldPath);
    string ext = Path.GetExtension(oldPath);
```

	Action	Result
19	Construct the new path and use the `File.Move` method to rename the file. Throw an `ArgumentException` if the new file already exists.	see code below

```
    string newPath = dir + Path.
        DirectorySeparatorChar + newName + ext;

    if (File.Exists(newPath))
    {
      throw new ArgumentException(
          "A file with the name "
            + newPath + " already exists.");
    }

    // Presume no error is thrown here
    File.Move(oldPath, newPath);
    return newPath;
  }
```

This defines our `AlbumNode` class, along with the methods that will prove useful as we develop our application. I am cheating here a bit, since I have already figured out what we need in the future from this class. In practice, a core class is typically defined and extended to support additional functionality as required by the application.

The `PhotoNode` class to represent photograph nodes is defined in a similar manner.

<table>
<tr><th colspan="3">CREATE A PHOTONODE CLASS</th></tr>
<tr><th></th><th>Action</th><th>Result</th></tr>
<tr><td>20</td><td>In the MyAlbumExplorer project, define an internal <code>PhotoNode</code> class.</td><td><pre>using System;
using System.IO;
using System.Windows.Forms;
using Manning.MyPhotoAlbum;

namespace MyAlbumExplorer
{
 internal class PhotoNode
     : TreeNode, IRefreshableNode
 {</pre></td></tr>
<tr><td>21</td><td>Define a private field and public <code>Photograph</code> property to hold the photo associated with this node.</td><td><pre> private Photograph _photo;
  public Photograph Photograph
      { get { return _photo; } }</pre></td></tr>
<tr><td>22</td><td>Define a constructor that accepts the <code>Photograph</code> to associate with the new node. Use the caption for the node's text, and the image key "Photo."</td><td><pre> public PhotoNode(Photograph photo) : base()
  {
    if (photo == null)
      throw new ArgumentNullException("photo");

    _photo = photo;
    Text = photo.Caption;
    ImageKey = "Photo";
    SelectedImageKey = "Photo";
  }</pre></td></tr>
<tr><td>23</td><td>Implement the <code>IRefresh-ableNode</code> interface to reset the node's text to the photo-graph's caption.</td><td><pre> public void RefreshNode()
  {
    Text = Photograph.Caption;
  }
 }
}</pre></td></tr>
</table>

As you can see, our `PhotoNode` class provides external access to the `Photograph` associated with the node. This completes our three `TreeNode` classes for use within our custom tree view. With these available, we are ready to continue our `AlbumTreeView` implementation.

18.4.4 Expanding and collapsing tree nodes

Let's begin with the creation of nodes in the tree. As mentioned earlier in the chapter, the `TreeView` class defines before and after events for a number of actions, including the expansion and contraction of tree nodes. We can employ these methods to dynamically add child nodes to our tree view.

The following steps display our three types of nodes in the tree, using the `Album-DirectoryNode` class as the top-level node.

DYNAMICALLY EXPAND AND COLLAPSE NODES

	Action	Result
1	Define a new `AddAlbum-Directory` method in the `AlbumTreeView` class that adds a new directory node to the tree.	see code 1 below
2	Override the `OnBeforeExpand` method to add the appropriate child nodes before a node is expanded.	see code 2 below
3	Implement an `ExpandAlbum-Directory` method to ensure the album nodes are created before the directory node expands. **How-to** Use the `BeginUpdate` and `EndUpdate` methods to disable drawing of the tree.	see code 3 below
4	Implement an `ExpandAlbum` method to ensure the photo nodes are created before the album node expands.	see code 4 below

1.
```
public AlbumDirectoryNode AddAlbumDirectory(
    string name, string albumDir)
{
  // Create a new top-level node
  AlbumDirectoryNode node
      = new AlbumDirectoryNode(name, albumDir);
  this.Nodes.Add(node);
  return node;
}
```

2.
```
protected override void OnBeforeExpand(
    TreeViewCancelEventArgs e)
{
  base.OnBeforeExpand(e);

  if (e.Node is AlbumDirectoryNode)
    ExpandAlbumDirectory(
        e.Node as AlbumDirectoryNode);
  else if (e.Node is AlbumNode)
    ExpandAlbum(e.Node as AlbumNode);
}
```

3.
```
private void ExpandAlbumDirectory(
    AlbumDirectoryNode node)
{
  // Add a node per album
  BeginUpdate();
  try
  {
    // Make sure the album nodes exist
    node.CreateAlbumNodes();
  }
  finally {  EndUpdate();  }
}
```

4.
```
private void ExpandAlbum(AlbumNode node)
{
  AlbumManager mgr = node.GetManager(true);
  if (mgr != null)
  {
    BeginUpdate();
    try
    {
      node.Nodes.Clear();
      foreach (Photograph p in mgr.Album)
      {
        PhotoNode newNode = new PhotoNode(p);
        node.Nodes.Add(newNode);
      }
    }
    finally {  EndUpdate();  }
  }
}
```

DYNAMICALLY EXPAND AND COLLAPSE NODES *(CONTINUED)*		
	Action	**Result**

5	Override the `OnAfter-Collapse` method to clean up child nodes after a tree node has been collapsed.	`protected override void OnAfterCollapse(`
		`    TreeViewEventArgs e)`
		`{`
		`  // Leave album directory nodes intact`
		`  // Clean up album nodes`
		`  if (e.Node is AlbumNode)`
		`    CollapseAlbum(e.Node as AlbumNode);`
		
		`  base.OnAfterCollapse(e);`
		`}`

6	Implement a `CollapseAlbum` method that removes the children from a given album node.	`private void CollapseAlbum(AlbumNode node)`
		`{`
		`  node.RemoveChildren();`
		`}`

This enables our tree view to internally expand and collapse the nodes, independent of the application it appears in. Child nodes are only created as they are needed, and `PhotoNode` objects are removed whenever their parent node is collapsed.

Two other features worth supporting in our tree view are selection and editing.

18.4.5 Selecting and editing tree nodes

As we have seen in other collection controls, the `TreeView` control allows a user to select a node in the control using either the keyboard or the mouse. The `BeforeSelect` and `AfterSelect` events are provided to process this action, and the events receive the `TreeViewCancelEventArgs` and `TreeViewEventArgs` classes discussed in the prior section.

In our custom tree view, we do not need to modify the selection behavior, but we do want to enable users of our control to select specific nodes. To support this, we provide two methods to aid in such efforts.

SUPPORT NODE SELECTION		
	Action	**Result**

1	In the `AlbumTreeView` class, define a `SelectChild` method. Implement this method to select the `PhotoNode` within a given album node that is associated with a given photograph.	`public void SelectChild(`
		`  AlbumNode node, Photograph photo)`
		`{`
		`  foreach (TreeNode n in node.Nodes)`
		`  {`
		`    PhotoNode pNode = n as PhotoNode;`
		`    if (pNode != null`
		`        && pNode.Photograph == photo)`
		`    {`
		`      SelectedNode = n;`
		`      break;`
		`    }`
		`  }`
		`}`

SUPPORT NODE SELECTION *(CONTINUED)*		
	Action	**Result**

2	Define an alternate `SelectChild` method to select an `AlbumNode` within a given album directory node that has a given album path.	`public void SelectChild(`
		`  AlbumDirectoryNode node, string albumPath)`
		`{`
		`  foreach (TreeNode n in node.Nodes)`
		`  {`
		`    AlbumNode aNode = n as AlbumNode;`
		`    if (aNode != null`
		`        && String.Equals(aNode.AlbumPath,`
		`             albumPath, StringComparison.`
		`                 InvariantCultureIgnoreCase))`
		`    {`
		`      SelectedNode = n;`
		`      break;`
		`    }`
		`  }`
		`}`

Another feature of tree views is the ability to edit the node text associated with a node. An edit can be issued programmatically using the `TreeNode.BeginEdit` method. The `BeforeLabelEdit` and `AfterLabelEdit` events that occur during editing both receive the `NodeLabelEditEventArgs` class, detailed in .NET Table 18.5.

.NET Table 18.5 NodeLabelEditEventArgs class

The `NodeLabelEditEventArgs` class specifies the event data for events that occur before and after a tree node edit operation takes place. This class is part of the `System.Windows.Forms` namespace, and inherits from the `System.EventArgs` class.

Public Properties	CancelEdit	Gets or sets whether the operation should be canceled. This can be set before or after the node is edited.
	Label	Gets the new text to assign to the node's label.
	Node	Gets the `TreeNode` object being edited.

Editing is typically used to rename the file name or some property associated with a node in the tree. In our case, we rename the album filename or the photo caption text. We do not permit album directories to be edited. The following steps implement the required editing logic.

SUPPORT NODE EDITS		
	Action	**Result**

3	Override the `OnAfterLabelEdit` method in the `AlbumTreeView` class.	`protected override void OnAfterLabelEdit(`
		`    NodeLabelEditEventArgs e)`
		`{`

SUPPORT NODE EDITS *(CONTINUED)*		
	Action	**Result**

4	If the edit is blank or was aborted, cancel the operation.	`if (String.IsNullOrEmpty(e.Label))`
		`{`
		`  e.CancelEdit = true;`
		`  return;`
		`}`

5	If the node is an album node, rename the underlying album path.	`if (e.Node is AlbumNode)`
		`{`
		`  // Rename the underlying album`
		`  AlbumNode node = e.Node as AlbumNode;`
		`  e.CancelEdit = !node.RenameAlbum(e.Label);`
		`}`

6	If the node is a photo node, modify the caption of the associated `Photograph` instance	`else if (e.Node is PhotoNode)`
		`{`
		`  // Modify the photo caption`
		`  PhotoNode node = e.Node as PhotoNode;`
		`  node.Photograph.Caption = e.Label;`
		`  SaveAlbumChanges();`
		`}`
		`}`

7	Also override the `OnKeyDown` method to edit the current node when the F2 key is pressed.	`protected override void OnKeyDown(KeyEventArgs e)`
		`{`
		`  if (e.KeyCode == Keys.F2)`
		`  {`
		`    if (SelectedNode != null)`
		`      SelectedNode.BeginEdit();`
		`    e.Handled = true;`
		`  }`
		`}`

In addition to handling a node edit appropriately, this code handles the F2 key to initiate an edit of the selected node. This matches the behavior in Windows Explorer and other applications, and seems appropriate here as well. We discussed keyboard events in chapter 10.

To round out our new control, we also add a few methods that will prove useful as we integrate this control into our MyAlbumExplorer application.

DEFINE SOME SUPPORTING METHODS		
	Action	**Result**

8	Define a method in the `AlbumTreeView` class that refreshes the current node using the `IRefreshableNode` interface.	`public void RefreshNode()`
		`{`
		`  IRefreshableNode refresh`
		`      = SelectedNode as IRefreshableNode;`
		
		`  if (refresh != null)`
		`    refresh.RefreshNode();`
		`}`

DEFINE SOME SUPPORTING METHODS *(CONTINUED)*

	Action	Result

9 Define another method to save any changes associated with the current node. Within this method, determine the album to save.

```
internal void SaveAlbumChanges()
{
  // Find the album to save
  AlbumNode aNode = SelectedNode as AlbumNode;
  if (aNode == null)
  {
    PhotoNode pNode = SelectedNode as PhotoNode;
    if (pNode != null)
      aNode = pNode.Parent as AlbumNode;
  }
```

10 If an album is found, then save the changes if an `AlbumManager` is available for the album.

```
  if (aNode != null)
  {
    AlbumManager mgr = aNode.GetManager(true);
    if (mgr.Album.HasChanged)
    {
      // Save data and update node
      mgr.Save();
      aNode.RefreshNode();
```

11 Also update any photo nodes if the album node is expanded.

```
      if (aNode.IsExpanded)
      {
        // Update photographs, as necessary
        foreach (PhotoNode pNode in aNode.Nodes)
        {
          // Assumes no photos added / deleted
          pNode.RefreshNode();
        }
      }
    }
  }
}
```

12 Define a `FindAlbumNode` method to locate the album node associated with the current album directory.

```
internal AlbumNode FindAlbumNode(string path)
{
  AlbumDirectoryNode dirNode
    = SelectedNode as AlbumDirectoryNode;
  if (dirNode != null)
  {
    foreach (AlbumNode node in dirNode.AlbumNodes)
    {
      if (String.Equals(node.AlbumPath, path,
            StringComparison.
                InvariantCultureIgnoreCase))
        return node;
    }
  }

  return null;
}
```

DEFINE SOME SUPPORTING METHODS *(CONTINUED)*		
	Action	**Result**

13	Finally, define a `FindPhotoNode` method to locate the photo node associated with the current album.	`internal PhotoNode FindPhotoNode(Photograph photo)`
		`{`
		`  AlbumNode albumNode = SelectedNode as AlbumNode;`
		`  if (albumNode != null)`
		`  {`
		`    albumNode.Expand();`
		`    foreach (PhotoNode node in albumNode.Nodes)`
		`    {`
		`      if (node.Photograph == photo)`
		`        return node;`
		`    }`
		`  }`
		
		`  return null;`
		`}`

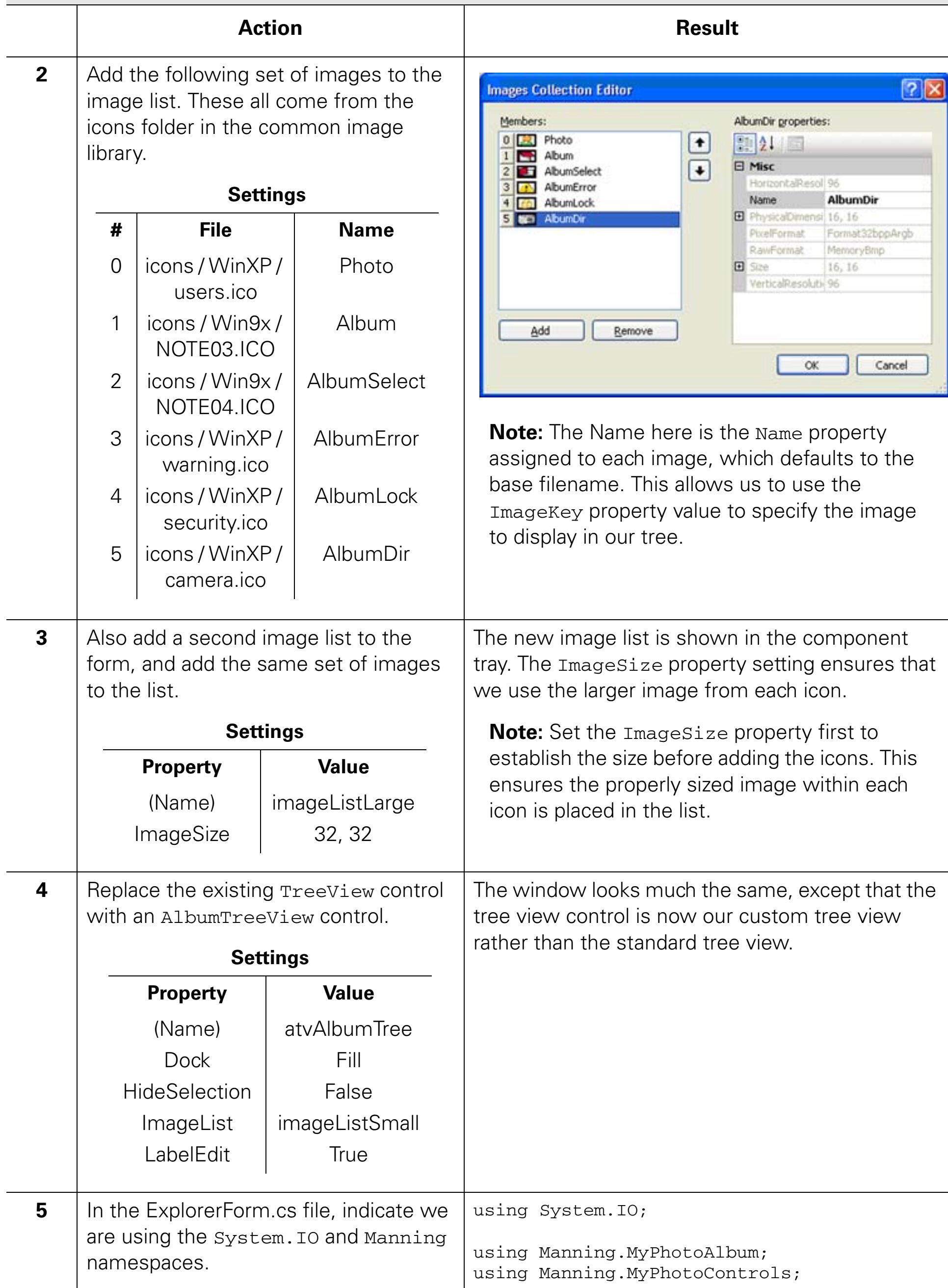
This completes our `AlbumTreeView` control. The next step is to employ it within an application. The bulk of this work is done in the next chapter. Even so, we would be remiss if we didn't show at least a little example here, so we finish out the chapter with this.

18.4.6 Integrating a custom tree view control

As you may have noticed, our tree view control never did provide a set of images to display. Instead we defined `ImageKey` property settings for each node. This allows an external application to dictate what images are displayed by defining the appropriate image names. Ideally, our `AlbumTreeView` control would provide some written documentation that explains all this. For our purposes, we simply demonstrate this in the following steps.

USE THE ALBUMTREEVIEW CONTROL IN OUR EXPLORER APPLICATION		
	Action	**Result**
1	In the ExplorerForm.cs [Design] window of the MyAlbumExplorer project, add an `ImageList` called "imageListSmall" to the designer.	The new image list is shown in the component tray.

USE THE ALBUMTREEVIEW CONTROL IN OUR EXPLORER APPLICATION *(CONTINUED)*



	Action	Result
2	Add the following set of images to the image list. These all come from the icons folder in the common image library. **Settings** **#** / **File** / **Name** 0 / icons / WinXP / users.ico / Photo 1 / icons / Win9x / NOTE03.ICO / Album 2 / icons / Win9x / NOTE04.ICO / AlbumSelect 3 / icons / WinXP / warning.ico / AlbumError 4 / icons / WinXP / security.ico / AlbumLock 5 / icons / WinXP / camera.ico / AlbumDir	(Images Collection Editor screenshot) **Note:** The Name here is the `Name` property assigned to each image, which defaults to the base filename. This allows us to use the `ImageKey` property value to specify the image to display in our tree.
3	Also add a second image list to the form, and add the same set of images to the list. **Settings** **Property** / **Value** (Name) / imageListLarge ImageSize / 32, 32	The new image list is shown in the component tray. The `ImageSize` property setting ensures that we use the larger image from each icon. **Note:** Set the `ImageSize` property first to establish the size before adding the icons. This ensures the properly sized image within each icon is placed in the list.
4	Replace the existing `TreeView` control with an `AlbumTreeView` control. **Settings** **Property** / **Value** (Name) / atvAlbumTree Dock / Fill HideSelection / False ImageList / imageListSmall LabelEdit / True	The window looks much the same, except that the tree view control is now our custom tree view rather than the standard tree view.

5	In the ExplorerForm.cs file, indicate we are using the `System.IO` and `Manning` namespaces.	`using System.IO;`
		
		`using Manning.MyPhotoAlbum;`
		`using Manning.MyPhotoControls;`

<table>
<tr><th colspan="3">USE THE ALBUMTREEVIEW CONTROL IN OUR EXPLORER APPLICATION (CONTINUED)</th></tr>
<tr><th></th><th>Action</th><th>Result</th></tr>
<tr><td>6</td><td>In the override of the <code>OnLoad</code> method, add an album directory node for the default album folder location.</td><td><pre>
protected override void OnLoad(. . .)
{
  . . .
  atvAlbumTree.Nodes.Clear();
  atvAlbumTree.AddAlbumDirectory(
      "Default Albums",
      AlbumManager.DefaultPath);

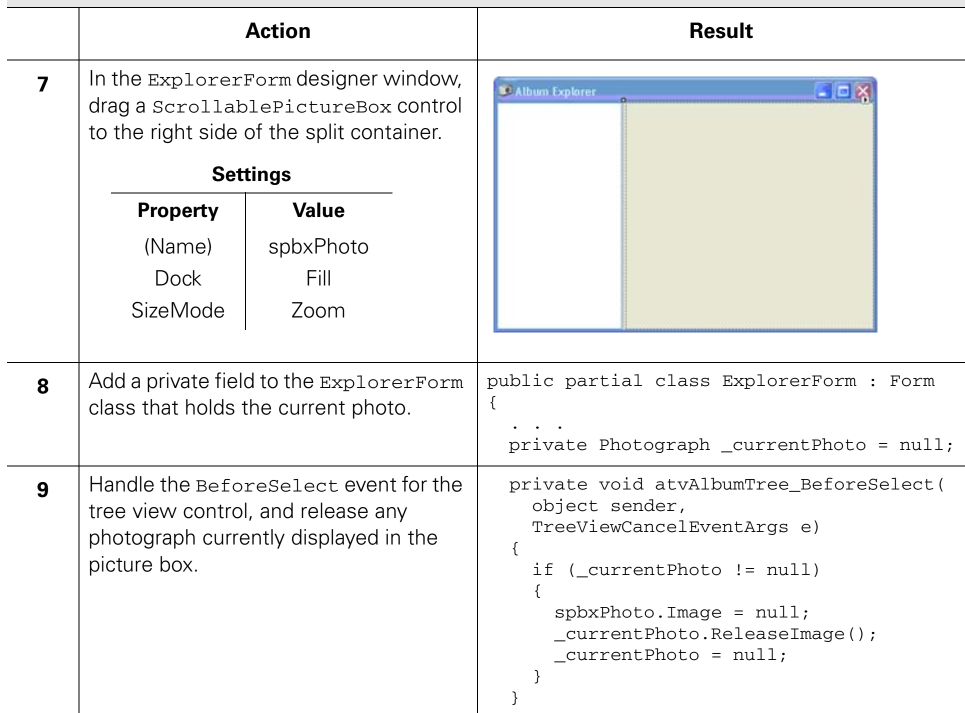
  base.OnLoad(e);
}
</pre></td></tr>
</table>

At this point, we have a fully functional `AlbumTreeView` control on our form. Once we add the top-level directory node, our custom control handles the rest by allowing album and photograph nodes to appear as required.

The images we use for our image lists are perhaps not the best images we could wish for, but they serve our purposes here. This is where a good graphics artist would be useful.

We can round out our little example by dropping a `ScrollablePictureBox` control on the right side of the form to display a selected photograph.

<table>
<tr><th colspan="3">USE THE ALBUMTREEVIEW CONTROL IN OUR EXPLORER APPLICATION</th></tr>
<tr><th></th><th>Action</th><th>Result</th></tr>
<tr><td>7</td><td>In the <code>ExplorerForm</code> designer window, drag a <code>ScrollablePictureBox</code> control to the right side of the split container.

Settings<table><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>spbxPhoto</td></tr><tr><td>Dock</td><td>Fill</td></tr><tr><td>SizeMode</td><td>Zoom</td></tr></table></td><td>
</td></tr>
<tr><td>8</td><td>Add a private field to the <code>ExplorerForm</code> class that holds the current photo.</td><td><pre>
public partial class ExplorerForm : Form
{
  . . .
  private Photograph _currentPhoto = null;
</pre></td></tr>
<tr><td>9</td><td>Handle the <code>BeforeSelect</code> event for the tree view control, and release any photograph currently displayed in the picture box.</td><td><pre>
private void atvAlbumTree_BeforeSelect(
  object sender,
  TreeViewCancelEventArgs e)
{
  if (_currentPhoto != null)
  {
    spbxPhoto.Image = null;
    _currentPhoto.ReleaseImage();
    _currentPhoto = null;
  }
}
</pre></td></tr>
</table>

USE THE ALBUMTREEVIEW CONTROL IN OUR EXPLORER APPLICATION *(CONTINUED)*		
	Action	**Result**

10	Handle the `AfterSelect` event to display the photo if the current node is a `PhotoNode` instance.	`private void atvAlbumTree_AfterSelect(`
		`  object sender, TreeViewEventArgs e)`
		`{`
		`  if (e.Node is PhotoNode)`
		`    DisplayPhoto(e.Node as PhotoNode);`
		`}`

11	Implement a `DisplayPhoto` method to display the image for a given photo node in the scrollable picture box.	`private void DisplayPhoto(`
		`  PhotoNode photoNode)`
		`{`
		`  _currentPhoto = photoNode.Photograph;`
		`  spbxPhoto.Image = _currentPhoto.Image;`
		`}`

This completes our example. The code used to display a photograph here is much like the code we presented in listing 18.2, so we won't discuss it again. Compile and run the application to see our custom tree view in action.

18.5 RECAP

In this rather long chapter, we presented the concept of interface styles and discussed the tree view control. We reviewed three common interface styles: single document interfaces (SDI), explorer interfaces, and multiple document interfaces (MDI).

This chapter focused on the explorer interface style. The `SplitContainer` class neatly divides a container such as a form or panel into two parts. We created a new MyAlbumExplorer application utilizing this control as an example of an explorer interface.

The bulk of the chapter examined tree view controls. The `TreeView` class displays a hierarchy of nodes, each represented by a `TreeNode` class instance.

We also built a custom tree view control for displaying photo albums, with a custom `TreeNode` for each of our three types of nodes. Our custom control manipulated these nodes to provide the appropriate functionality. We finished the chapter by employing this control in our MyAlbumExplorer application.

The next chapter continues this example by presenting the `ListView` control and illustrating its use within our sample explorer interface.

.NET PROGRAMMING

WINDOWS FORMS IN ACTION

Erik Brown

With the .NET Framework you can create rich graphical interfaces and compelling user experiences. Whether you design business forms, build interactive graphical interfaces, or code smart client applications, you need to master Windows Forms.

Windows Forms in Action, a revised edition of the popular *Windows Forms Programming with C#*, gets you going quickly by immersing you in a practical running example. Easy to follow instructions, numerous graphics, and unique "information maps" guide you through the entire Windows Forms namespace—all in the powerful C# language. If you're new to Windows Forms, you'll explore fundamentals like labels, menus, buttons, and panels. Seasoned developers can sink their teeth into advanced concepts such as reusable control libraries, merging tool strips, persistent application settings, explorer-style interfaces, and two-way data binding. The appendices include a handy visual index of the Windows Forms classes.

What's Inside

- Owner-drawn lists and tabs
- List, tree, and data grid views
- Data streaming and encryption
- Masked text boxes and date controls
- Standard and ClickOnce deployment
- How to spice up your applications with: custom controls, drag and drop, background processing, automatic text completion, embedded web browsing

With over 17 years of experience, **Erik Brown** is a developer, architect, consultant, and manager. He is a veteran of three successful startups and currently works as a program manager at Unisys Corporation. Erik lives in northern Virginia with his wife and two daughters.

"Each chapter is worth the price of the whole book!"

—Berndt Hamboeck
Senior Architect and Team Leader, United Nations

"Excellent for developers new to Windows Forms."

—Dave Corun
Director of Microsoft Technologies, Catalyst IT Services

"Precisely what I needed to get up to speed quickly. Engaging."

—Joe Litton
Enterprise Application Developer
MCP, SCJP, PCLP

"Right on target."

—Jack Herrington
Author and Software Engineer

Ask the Author

Ebook edition

www.manning.com/eebrown2

9 781932 394658 54995
ISBN 1-932394-65-6

MANNING $49.95 US/$64.95 Canada