



C H A P T E R 1 0

List controls

10.1 List boxes 315	10.4 Combo box edits 339
10.2 Multiselection list boxes 325	10.5 Owner-drawn lists 343
10.3 Combo boxes 333	10.6 Recap 352

This chapter continues our discussion of the Windows Forms controls available in the .NET Framework. The controls we saw in chapter 9 each presented a single item, such as a string of text or a button with associated text. In this chapter we will look at some controls useful for presenting collections of items in Windows-based applications.

While it is certainly possible to use a multiline `Textbox` control to present a scrollable list of items, this control does not allow the user to select and manipulate individual items. This is where the `Listbox` and other list controls come in. These controls present a scrollable list of objects that can be individually selected, highlighted, moved, and otherwise manipulated by your program. In this chapter we will look at the `Listbox` and `ComboBox` controls in some detail. We will discuss the following topics:

- Presenting a collection of objects using the `Listbox` class.
- Supporting single and multiple selections in a list box.
- Drawing custom list items in a list box.
- Displaying a selection using the `ComboBox` class.
- Dynamically interacting with the items in a combo box.

Note that the `ListView` and `TreeView` classes can also be used with collections of objects. These classes are covered in chapters 14 and 15.

We will take a slightly different approach to presenting the list controls here. Rather than using the `MyPhotos` application we have come to know and love, this chapter will build a new application for displaying the contents of an album, using the existing `MyPhotoAlbum.dll` library. This will demonstrate how a library can be reused to quickly build a different view of the same data. Our new application will be called `MyAlbumEditor`, and is shown in figure 10.1.

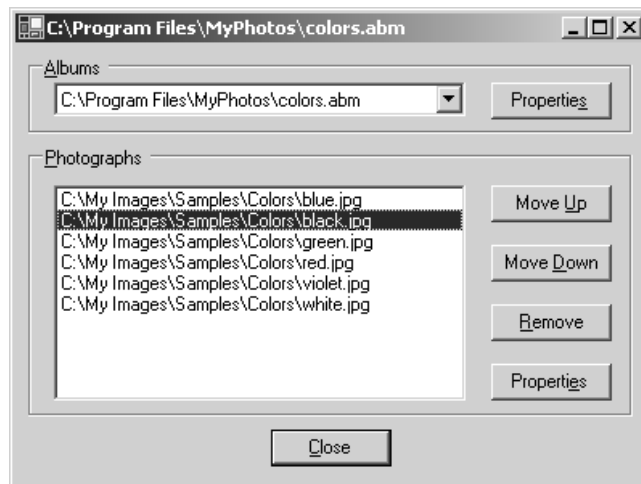


Figure 10.1
The `MyAlbumEditor` application does not include a menu or status bar.

10.1 LIST BOXES

A list box presents a collection of objects as a scrollable list. In this section we look at the `ListControl` and `ListBox` classes. We will create a list box as part of a new `MyAlbumEditor` application that displays the collection of photographs in a `PhotoAlbum` object. We will also support the ability to display our `PhotoEditDlg` dialog box for a selected photograph.

Subsequent sections in this chapter will extend the capabilities of this application with multiple selections of photographs and the use of combo boxes.

10.1.1 CREATING A LIST BOX

The `ListBox` and `ComboBox` controls both present a collection of objects. A list box displays the collection as a list, whereas a combo box, as we shall see, displays a single item, with the list accessible through an arrow button. In the window in figure 10.1, the photo album is displayed within a `ComboBox`, while the collection of photographs is displayed in a `ListBox`. Both of these controls are derived from the `ListControl` class, which defines the basic collection and display functionality required in both controls. A summary of this class appears in .NET Table 10.1.

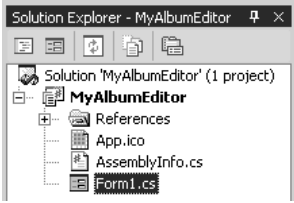
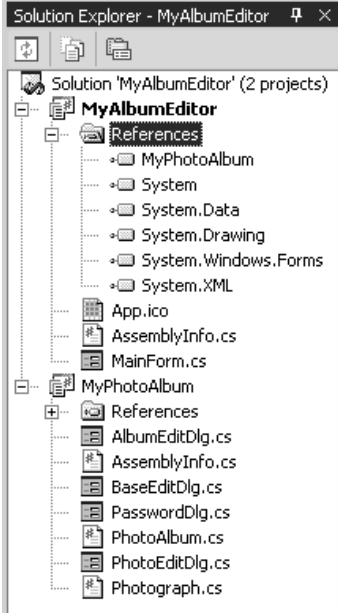
.NET Table 10.1 ListControl class

The `ListControl` class is an abstract class for presenting a collection of objects to the user. You do not normally inherit from this class; instead the derived classes `ListBox` and `ComboBox` are normally used.

This class is part of the `System.Windows.Forms` namespace, and inherits from the `Control` class. See .NET Table 4.1 on page 104 for a list of members inherited by this class.

Public Properties	<code>DataSource</code>	Gets or sets the data source for this control. When set, the individual items cannot be modified.
	<code>DisplayMember</code>	Gets or sets the property to use when displaying objects in the list control. If none is set or the setting is not a valid property, then the <code>ToString</code> property is used.
	<code>SelectedIndex</code>	Gets or sets the zero-based index of the object selected in the control.
	<code>SelectedValue</code>	Gets or sets the value of the object selected in the control.
	<code>ValueMember</code>	Gets or sets the property to use when retrieving the value of an item in the list control. By default, the object itself is retrieved.
Public Methods	<code>GetItemText</code>	Returns the text associated with a given item, based on the current <code>DisplayMember</code> property setting.
Public Events	<code>DataSourceChanged</code>	Occurs when the <code>DisplaySource</code> property changes
	<code>DisplayMemberChanged</code>	Occurs when the <code>DisplayMember</code> property changes.

Let's see how to use some of these members to display the list of photographs contained in an album. The following steps create a new `MyAlbumEditor` application. We will use this application throughout this chapter to demonstrate how various controls are used. Here, we will open an album and display its contents in a `ListBox` using some of the members inherited from `ListControl`.

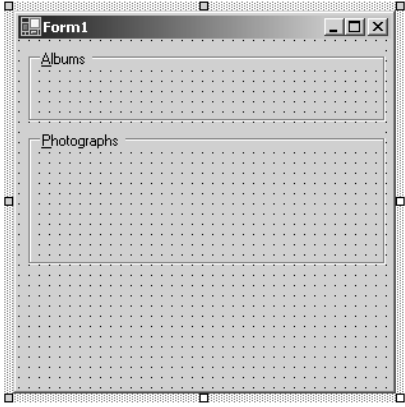
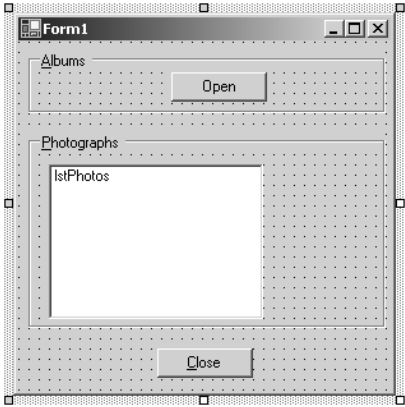
CREATE THE MYALBUMEDITOR PROJECT		
	Action	Result
1	<p>Create a new project called "MyAlbumEditor."</p> <p>How-to Use the File menu, or the keyboard shortcut Ctrl+Shift+N. Make sure you close your existing solution, if any.</p>	<p>The new project appears in the Solution Explorer window, with the default Form1 form shown in the designer window.</p> 
2	Rename the Form1.cs file to MainForm.cs.	
3	In the MainForm.cs source file, rename the C# class to MainForm.	<pre>public class MainForm: System.Windows.Forms.Form { ... }</pre>
4	<p>Add the MyPhotoAlbum project to the solution.</p> <p>How-to a. Right-click on the MyAlbumEditor solution. b. Select Existing Project... from the Add menu. c. In the Add Existing Project window, locate the MyPhotoAlbum directory. d. Select the MyPhotoAlbum.csproj file from within this directory.</p>	
5	<p>Reference the MyPhotoAlbum project within the MyAlbumEditor project.</p> <p>How-to Right-click the References item in the MyAlbumEditor project and display the Add Reference dialog.</p>	

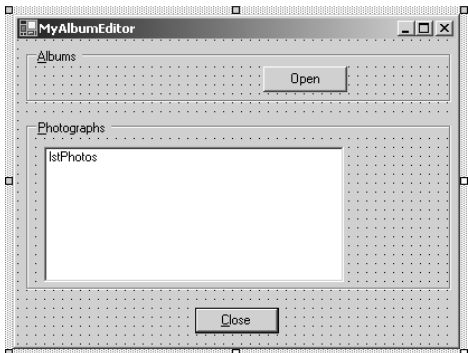
These steps should be familiar to you if you have been following along from the beginning of the book. Since we encapsulated the PhotoAlbum and Photograph classes in a separate library in chapter 5, these objects, including the dialogs created in chapter 9, are now available for use in our application. This is quite an important point, so I will say it again. The proper encapsulation of our objects in the MyPhoto-

Album library in chapters 5 and 9 makes the development of our new application that much easier, and permits us to focus our attention on the list controls.

With this in mind, let's toss up a couple of buttons and a list so we can see how the `ListBox` control works.

Set the version number of the `MyAlbumEditor` application to 10.1.

CREATE THE CONTROLS FOR OUR NEW APPLICATION																											
	Action	Result																									
6	<p>Drop two <code>GroupBox</code> controls onto the form.</p> <p>How-to As usual, drag them from the Toolbox window.</p> <table> <tr> <th colspan="3">Settings</th></tr> <tr> <th>GroupBox</th><th>Property</th><th>Value</th></tr> <tr> <td rowspan="2">First</td><td>Anchor</td><td>Top, Left, Right</td></tr> <tr> <td>Text</td><td>&Albums</td></tr> <tr> <td rowspan="2">Second</td><td>Anchor</td><td>Top, Bottom, Left, Right</td></tr> <tr> <td>Text</td><td>&Photo-graphs</td></tr> </table>	Settings			GroupBox	Property	Value	First	Anchor	Top, Left, Right	Text	&Albums	Second	Anchor	Top, Bottom, Left, Right	Text	&Photo-graphs										
Settings																											
GroupBox	Property	Value																									
First	Anchor	Top, Left, Right																									
	Text	&Albums																									
Second	Anchor	Top, Bottom, Left, Right																									
	Text	&Photo-graphs																									
7	<p>Drop a <code>Button</code> control into the Albums group box, a <code>ListBox</code> control into the Photographs group box, and a <code>Button</code> control at the base of the form.</p> <table> <tr> <th colspan="3">Settings</th></tr> <tr> <th>Control</th><th>Property</th><th>Value</th></tr> <tr> <td rowspan="3">Open Button</td><td>(Name)</td><td>btnOpen</td></tr> <tr> <td>Anchor</td><td>Top, Right</td></tr> <tr> <td>Text</td><td>&Open</td></tr> <tr> <td rowspan="2">ListBox</td><td>(Name)</td><td>lstPhotos</td></tr> <tr> <td>Anchor</td><td>Top, Bottom, Left, Right</td></tr> <tr> <td rowspan="3">Close Button</td><td>(Name)</td><td>btnClose</td></tr> <tr> <td>Anchor</td><td>Bottom</td></tr> <tr> <td>Text</td><td>&Close</td></tr> </table>	Settings			Control	Property	Value	Open Button	(Name)	btnOpen	Anchor	Top, Right	Text	&Open	ListBox	(Name)	lstPhotos	Anchor	Top, Bottom, Left, Right	Close Button	(Name)	btnClose	Anchor	Bottom	Text	&Close	 <p>Note: A couple points to note here. First, the <code>Anchor</code> settings define the resize behavior of the controls within their container. Note that the <code>Button</code> and <code>ListBox</code> here are anchored within their respective group boxes, and not to the <code>Form</code> itself.</p> <p>Second, since our application will not have a menu bar, we use the standard <code>Close</code> button as the mechanism for exiting the application.</p>
Settings																											
Control	Property	Value																									
Open Button	(Name)	btnOpen																									
	Anchor	Top, Right																									
	Text	&Open																									
ListBox	(Name)	lstPhotos																									
	Anchor	Top, Bottom, Left, Right																									
Close Button	(Name)	btnClose																									
	Anchor	Bottom																									
	Text	&Close																									

CREATE THE CONTROLS FOR OUR NEW APPLICATION (continued)												
	Action	Result										
8	<p>Set the properties for the MainForm form.</p> <table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>AcceptButton</td><td>btnClose</td></tr><tr><td>Size</td><td>400, 300</td></tr><tr><td>Text</td><td>MyAlbumEditor</td></tr></table> <p>Note: When you enter the new Size setting, note how the controls automatically resize within the form based on the assigned Anchor settings.</p>	Settings		Property	Value	AcceptButton	btnClose	Size	400, 300	Text	MyAlbumEditor	
Settings												
Property	Value											
AcceptButton	btnClose											
Size	400, 300											
Text	MyAlbumEditor											

Our form is now ready. You can compile and run if you like. Before we talk about this in any detail, we will add some code to make our new `ListBox` display the photographs in an album.

Some of the new code added by the following steps mimics code we provided for our `MyPhotos` application. This is to be expected, since both interfaces operate on photo album collections.

DISPLAY THE CONTENTS OF AN ALBUM IN THE LISTBOX CONTROL		
	Action	Result
9	In the <code>MainForm.cs</code> file, indicate we are using the <code>Manning.MyPhotoAlbum</code> namespace.	<pre>... using Manning.MyPhotoAlbum;</pre>
10	Add some member variables to track the current album and whether it has changed.	<pre>private PhotoAlbum _album; private bool _bAlbumChanged = false;</pre>
11	Override the <code>OnLoad</code> method to initialize the album. Note: The <code>OnLoad</code> method is called a single time after the form has been created and before the form is initially displayed. This method is a good place to perform one-time initialization for a form.	<pre>protected override void OnLoad (EventArgs e) { // Initialize the album _album = new PhotoAlbum(); base.OnLoad(e); }</pre>
12	Add a click handler for the Close button to exit the application.	<pre>private void btnClose_Click (object sender, System.EventArgs e) { Close(); }</pre>

DISPLAY THE CONTENTS OF AN ALBUM IN THE LISTBOX CONTROL <i>(continued)</i>		
	Action	Result
13	<p>Add a <code>closeAlbum</code> method to close a previously opened album.</p> <p>How-to Display a dialog to ask if the user wants to save any changes they have made.</p>	<pre>private void CloseAlbum() { if (_bAlbumChanged) { _bAlbumChanged = false; DialogResult result = MessageBox.Show("Do you want " + "to save your changes to " + _album.FileName + "?", "Save Changes?", MessageBoxButtons.YesNo, MessageBoxIcon.Question); if (result == DialogResult.Yes) { _album.Save(); } } _album.Clear(); }</pre>
14	<p>Override the <code>onClosing</code> method to ensure the album is closed on exit.</p>	<pre>protected override void OnClosing (CancelEventArgs e) { CloseAlbum(); }</pre>
15	<p>Add a Click handler for the Open button to open an album and assign it to the <code>ListBox</code>.</p> <p>How-to</p> <ol style="list-style-type: none"> Close any previously open album. Use the <code>OpenFileDialog</code> class to allow the user to select an album. Use the <code>PhotoAlbum.Open</code> method to open the file. Assign the album's file name to the title bar of the form. Use a separate method for updating the contents of the list box. 	<pre>private void btnOpen_Click (object sender, System.EventArgs e) { CloseAlbum(); using (OpenFileDialog dlg = new OpenFileDialog()) { dlg.Title = "Open Album"; dlg.Filter = "abm files (*.abm)" + " *.abm All Files (*.*) *.*"; dlg.InitialDirectory = PhotoAlbum.DefaultDir; try { if (dlg.ShowDialog() == DialogResult.OK) { _album.Open(dlg.FileName); this.Text = _album.FileName; UpdateList(); } } catch (Exception) { MessageBox.Show("Unable to open " + "album\n" + dlg.FileName, "Open Album Error", MessageBoxButtons.OK, MessageBoxIcon.Error); } } }</pre>

DISPLAY THE CONTENTS OF AN ALBUM IN THE LISTBOX CONTROL <i>(continued)</i>		
	Action	Result
16	Implement a protected <code>UpdateList</code> method to initialize the <code>ListBox</code> control.	<pre>protected void UpdateList() { lstPhotos.DataSource = _album; }</pre>

That's it! No need to add individual photographs one by one or perform other complicated steps to fill in the list box. Much of the code is similar to code we saw in previous chapters. The one exception, the `UpdateList` method, simply assigns the `DataSource` property of the `ListBox` control to the current photo album.

```
protected void UpdateList()
{
    lstPhotos.DataSource = _album;
}
```

The `DataSource` property is part of the *data binding* support in Windows Forms. Data binding refers to the idea of assigning one or more values from some source of data to the settings for one or more controls. A data source is basically any array of objects, and in particular any class that supports the `IList` interface.¹ Since the `PhotoAlbum` class is based on `IList`, each item in the list, in this case each `Photograph`, is displayed by the control. By default, the `ToString` property for each contained item is used as the display string. If you recall, we implemented this method for the `Photograph` class in chapter 5 to return the file name associated with the photo.

Compile and run your code to display your own album. An example of the output is shown in figure 10.2. In the figure, an album called `colors.abm` is displayed, with each photograph in the album named after a well-known color. Note how the `GroupBox` controls display their keyboard access keys, namely `Alt+A` and `Alt+P`. When activated, the focus is set to the first control in the group box, based on the assigned tab order.

¹ We will discuss data binding more generally in chapter 17.

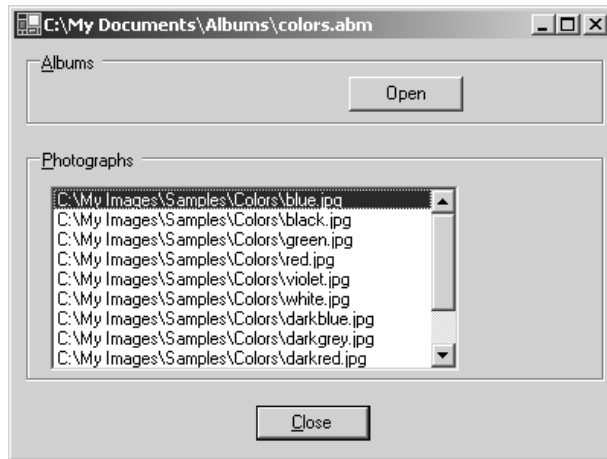


Figure 10.2
By default, the `ListBox` control displays a scroll bar when the number of items to display exceeds the size of the box.

You will also note that there is a lot of blank space in our application. Not to worry. These spaces will fill up as we progress through the chapter.

TRY IT! The `DisplayMember` property for the `ListBox` class indicates the name of the property to use for display purposes. In our program, since this property is not set, the default `ToString` property inherited from the `Object` class is used. Modify this property in the `UpdateList` method to a property specific to the `Photograph` class, such as “`FileName`” or “`Caption`.” Run the program again to see how this affects the displayed photographs.

The related property `ValueMember` specifies the value returned by members such as the `SelectedValue` property. By default, this property will return the object instance itself.

10.1.2 HANDLING SELECTED ITEMS

As you might expect, the `ListBox` class supports much more than the ability to display a collection of objects. Particulars of this class are summarized in .NET Table 10.2. In the `MyAlbumEditor` application, the list box is a single-selection, single-column list corresponding to the contents of the current album. There are a number of different features we will demonstrate in our application. For starters, let’s display the dialogs we created in chapter 9.

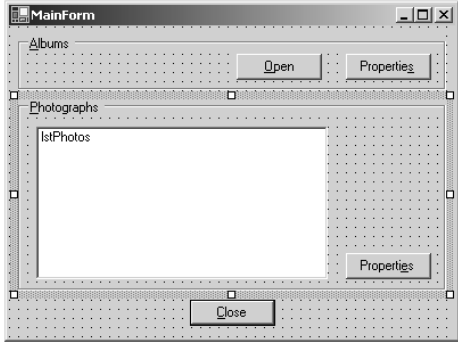
The album dialog can be displayed using a normal button. For the `PhotoEditDlg` dialog, we would like to display the properties of the photograph that are currently selected in the list box. As you may recall, this dialog displays the photograph at the current position within the album, which seemed quite reasonable for our `MyPhotos` application. To make this work here, we will need to modify the current position to correspond to the selected item.

.NET Table 10.2 ListBox class

The `ListBox` class represents a list control that displays a collection as a scrollable window. A list box can support single or multiple selection of its items, and each item can display as a simple text string or a custom graphic. This class is part of the `System.Windows.Forms` namespace, and inherits from the `ListControl` class. See .NET Table 10.1 on page 316 for a list of members inherited by this class.

Public Static Fields	<code>DefaultItemHeight</code>	The default item height for an owner-drawn <code>ListBox</code> object.
	<code>NoMatches</code>	The value returned by <code>ListBox</code> methods when no matches are found during a search.
Public Properties	<code>DrawMode</code>	Gets or sets how this list box should be drawn.
	<code>ItemHeight</code>	Gets or sets the height of an item in the list box.
	<code>Items</code>	Gets the collection of items to display.
	<code>MultiColumn</code>	Gets or sets whether this list box should support multiple columns. Default is <code>false</code> .
	<code>SelectedIndices</code>	Gets a collection of zero-based indices for the items selected in the list box.
	<code>SelectedItem</code>	Gets or sets the currently selected object.
	<code>SelectedItems</code>	Gets a collection of all items selected in the list.
	<code>SelectionMode</code>	Gets or sets how items are selected in the list box.
	<code>Sorted</code>	Gets or sets whether the displayed list should be automatically sorted.
Public Methods	<code>TopIndex</code>	Gets the index of the first visible item in the list.
	<code>BeginUpdate</code>	Prevents the control from painting its contents while items are added to the list box.
	<code>ClearSelected</code>	Deselects all selected items in the control.
	<code>FindString</code>	Returns the index of the first item with a display value beginning with a given string.
	<code>GetSelected</code>	Indicates whether a specified item is selected.
	<code>IndexFromPoint</code>	Returns the index of the item located at the specified coordinates.
Public Events	<code>SetSelected</code>	Selects or deselects a given item.
	<code>DrawItem</code>	Occurs when an item in an owner-drawn list box requires painting.
	<code>MeasureItem</code>	Occurs when the size of an item in an owner-drawn list box is required.
	<code>SelectedIndexChanged</code>	Occurs whenever a new item is selected in the list box, for both single and multiple selection boxes.

The following steps detail the changes required to display our two dialogs.

DISPLAY THE PROPERTY DIALOGS																						
	Action	Result																				
1	<p>In the MainForm.cs [Design] window, add two buttons to the form as shown in the graphic.</p> <table> <tr> <th colspan="3">Settings</th></tr> <tr> <th>Button</th><th>Property</th><th>Value</th></tr> <tr> <td rowspan="3">album</td><td>(Name)</td><td>btnAlbumProp</td></tr> <tr> <td>Anchor</td><td>Top, Right</td></tr> <tr> <td>Text</td><td>Propertie&s</td></tr> <tr> <td rowspan="3">photo</td><td>(Name)</td><td>btnPhotoProp</td></tr> <tr> <td>Anchor</td><td>Top, Right</td></tr> <tr> <td>Text</td><td>Properti&es</td></tr> </table>	Settings			Button	Property	Value	album	(Name)	btnAlbumProp	Anchor	Top, Right	Text	Propertie&s	photo	(Name)	btnPhotoProp	Anchor	Top, Right	Text	Properti&es	
Settings																						
Button	Property	Value																				
album	(Name)	btnAlbumProp																				
	Anchor	Top, Right																				
	Text	Propertie&s																				
photo	(Name)	btnPhotoProp																				
	Anchor	Top, Right																				
	Text	Properti&es																				
2	<p>Add a Click event handler for album's Properties button.</p> <p>How-to</p> <ol style="list-style-type: none"> Within this handler, display an Album Properties dialog box for the current album. If the user modifies the properties, mark the album as changed and update the list. 	<pre>private void btnAlbumProp_Click (object sender, System.EventArgs e) { using (AlbumEditDlg dlg = new AlbumEditDlg(_album)) { if (dlg.ShowDialog() == DialogResult.OK) { _bAlbumChanged = true; UpdateList(); } } }</pre>																				
3	<p>Add a Click event handler for the photograph's Properties button to display the PhotoEditDlg form.</p> <p>How-to</p> <ol style="list-style-type: none"> Within the handler, if the album is empty then simply return. Set the current position in the album to the selected photograph. Display a Photo Properties dialog box for the photograph at the current position. If the user modifies the properties, mark the album as changed and update the list. 	<pre>private void btnPhotoProp_Click (object sender, System.EventArgs e) { if (_album.Count == 0) return; if (lstPhotos.SelectedIndex >= 0) { _album.CurrentPosition = lstPhotos.SelectedIndex; } using (PhotoEditDlg dlg = new PhotoEditDlg(_album)) { if (dlg.ShowDialog() == DialogResult.OK) { _bAlbumChanged = true; UpdateList(); } } }</pre>																				

DISPLAY THE PROPERTY DIALOGS <i>(continued)</i>		
	Action	Result
4	<p>Also display the photograph's properties when the user double-clicks on the list.</p> <p>How-to Handle the <code>DoubleClick</code> event for the <code>ListBox</code> control.</p>	<pre>private void lstPhotos_DoubleClick (object sender, System.EventArgs e) { btnPhotoProp.PerformClick(); }</pre>

In the code to display the Photograph Properties dialog, note how the `SelectedIndex` property is used. If no items are selected, then `SelectedIndex` will contain the value `-1`, and the current position in the album is not modified. When a photograph is actually selected, the current position is updated to the selected index. This assignment relies on the fact that the order of photographs in the `ListBox` control matches the order of photographs in the album itself.

```
if (lstPhotos.SelectedIndex >= 0)
    _album.CurrentPosition = lstPhotos.SelectedIndex;
```

For both dialogs, a `C#` using block ensures that any resources used by the dialog are cleaned up when we are finished. We also call `UpdateList` to update our application with any relevant changes made. In fact, neither property dialog permits any changes that we would display at this time. Even so, updating the list is a good idea in case we add such a change in the future.

Compile and run your application to ensure that the dialog boxes display correctly. Note how easily we reused these dialogs in our new application. Make some changes and then reopen an album to verify that everything works as you expect.

One minor issue with our application occurs when the album is empty. When a user clicks the photo's Properties button, nothing happens. This is not the best user interface design, and we will address this fact in the next section.

So far our application only allows a single item to be selected at a time. List boxes can also permit multiple items to be selected simultaneously—a topic we will examine next.

10.2 MULTISELECTION LIST BOXES

So far we have permitted only a single item at a time to be selected from our list. In this section we enable multiple item selection, and add some buttons to perform various actions based on the selected items. Specifically, we will add Move Up and Move Down buttons to alter the position of the selected photographs, and a Remove button to delete the selected photographs from the album.

10.2.1 Enabling multiple selection

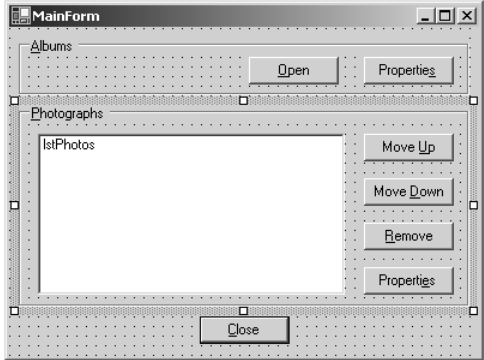
Enabling the `ListBox` to allow multiple selections simply requires setting the right property value, namely the `SelectionMode` property, to the value `MultiSimple` or `MultiExtended`. We discuss this property in detail later in the section.

Whenever you enable new features in a control, in this case enabling multiple selection in our list box, it is a good idea to review the existing functionality of the form to accommodate the new feature. In our case, what does the Properties button in the Photographs group box do when more than a single item is selected? While we could display the properties of the first selected item, this seems rather arbitrary. A more logical solution might be to disable the button when multiple items are selected. This is, in fact, what we will do here.

Since the Properties button will be disabled, we should probably have some other buttons that make sense when multiple items are selected. We will add three buttons. The first two will move the selected items up or down in the list as well as within the corresponding `PhotoAlbum` object. The third will remove the selected items from the list and the album.

The steps required are shown in the following table:

Set the version number of the `MyAlbumEditor` application to 10.2.

ENABLE MULTIPLE SELECTIONS IN THE LIST BOX																															
	Action		Result																												
1	In the MainForm.cs [Design] window, modify the SelectionMode property for the list box to be MultiExtended.		This permits multiple items to be selected similarly to how files can be selected in Windows Explorer.																												
2	Add three new buttons within the Photographs group box as shown in the graphic.																														
	<table><tr><th colspan="3">Settings</th></tr><tr><th>Button</th><th>Property</th><th>Value</th></tr><tr><td rowspan="3">Move Up</td><td>(Name)</td><td>btnMoveUp</td></tr><tr><td>Anchor</td><td>Top, Right</td></tr><tr><td>Text</td><td>Move &Up</td></tr><tr><td rowspan="3">Move Down</td><td>(Name)</td><td>btnMoveDown</td></tr><tr><td>Anchor</td><td>Top, Right</td></tr><tr><td>Text</td><td>Move &Down</td></tr><tr><td rowspan="3">Remove</td><td>(Name)</td><td>btnRemove</td></tr><tr><td>Anchor</td><td>Top, Right</td></tr><tr><td>Text</td><td>&Remove</td></tr></table>		Settings			Button	Property	Value	Move Up	(Name)	btnMoveUp	Anchor	Top, Right	Text	Move &Up	Move Down	(Name)	btnMoveDown	Anchor	Top, Right	Text	Move &Down	Remove	(Name)	btnRemove	Anchor	Top, Right	Text	&Remove		
Settings																															
Button	Property	Value																													
Move Up	(Name)	btnMoveUp																													
	Anchor	Top, Right																													
	Text	Move &Up																													
Move Down	(Name)	btnMoveDown																													
	Anchor	Top, Right																													
	Text	Move &Down																													
Remove	(Name)	btnRemove																													
	Anchor	Top, Right																													
	Text	&Remove																													

ENABLE MULTIPLE SELECTIONS IN THE LIST BOX <i>(continued)</i>		
	Action	Result
3	<p>Set the <code>Enabled</code> property for the four buttons in the Photographs group box to <code>false</code>.</p> <p>How-to</p> <ol style="list-style-type: none"> Click the first button. Hold down the <code>Ctrl</code> key and click the other buttons so that all four buttons are highlighted. Display the Properties window. Set the <code>Enabled</code> item to <code>False</code>. <p>Note: This technique can be used to set a common property for any set of controls on a form to the same value.</p>	<p>The code in the <code>InitializeComponent</code> method for all four buttons is modified so that their <code>Enabled</code> properties are set to <code>false</code>.</p> <pre> btnMoveUp.Enabled = false; . . . btnMoveDown.Enabled = false; . . . </pre>
4	<p>Rewrite the <code>UpdateList</code> method to add each item to the list manually.</p> <p>Note: The <code>BeginUpdate</code> method prevents the list box from drawing the control while new items are added. This improves performance and prevents the screen from flickering.</p>	<p>This allows us to manipulate and modify the individual items in the list, which is prohibited when filling the list with the <code>DisplaySource</code> property.</p> <pre> private void UpdateList() { lstPhotos.BeginUpdate(); lstPhotos.Items.Clear(); foreach (Photograph photo in _album) { lstPhotos.Items.Add(photo); } lstPhotos.EndUpdate(); } </pre>
5	<p>Handle the <code>SelectedIndexChanged</code> event for the <code>ListBox</code> control.</p> <p>How-to</p> <p>This is the default event for all list controls, so simply double-click on the control.</p>	<pre> private void lstPhotos_SelectedIndexChanged (object sender, System.EventArgs e) { int numSelected = lstPhotos.SelectedIndices.Count; } </pre>
6	<p>Implement this handler to enable or disable the buttons in the Photographs group box based on the number of items selected in the list box.</p> <p>Note: The Move Up button should be disabled if the first item is selected. The Move Down button should be disabled if the last item is selected. The <code>GetSelected</code> method is used to determine if a given index is currently selected.</p>	<pre> bool someSelected = (numSelected > 0); btnMoveUp.Enabled = (someSelected && !lstPhotos.GetSelected(0)); btnMoveDown.Enabled = (someSelected && (!lstPhotos.GetSelected(lstPhotos.Items.Count - 1))); btnRemove.Enabled = someSelected; btnPhotoProp.Enabled = (numSelected == 1); } </pre>

You can compile and run this code if you like. Our new buttons do not do anything, but you can watch them become enabled and disabled as you select items in a newly opened album.

We assigned the `MultiExtended` selection mode setting to the `ListBox.SelectionMode` property, which permits selecting a range of items using the mouse or keyboard. This is one of four possible values for the `SelectionMode` enumeration, as described in .NET Table 10.3.

TRY IT! Change the list box selection mode to `MultiSimple` and run your program to see how the selection behavior differs between this and the `MultiExtended` mode.

Our next task will be to provide an implementation for these buttons. We will pick up this topic in the next section.

10.2.2 HANDLING THE MOVE UP AND MOVE DOWN BUTTONS

Now that our list box allows multiple selections, we need to implement our three buttons that handle these selections from the list. This will permit us to discuss some collection and list box methods that are often used when processing multiple selections in a list.

We will look at the Move Up and Move Down buttons first. There are two problems we need to solve. The first is that our `PhotoAlbum` class does not currently provide an easy way to perform these actions. We will fix this by adding two methods to our album class for this purpose.

.NET Table 10.3 SelectionMode enumeration

The `SelectionMode` enumeration specifies the selection behavior of a list box control, such as the `ListBox` and `CheckedListBox` classes. This enumeration is part of the `System.Windows.Forms` namespace.

Enumeration Values	None	Items cannot be selected.
	One	A single item can be selected using a mouse click or the space bar key.
	MultiSimple	Multiple items can be selected. Items are selected or deselected using a mouse click or the space bar.
	MultiExtended	Multiple items can be selected. This extends simple selection to permit a range of items to be selected using a drag of the mouse or the Shift, Ctrl, and arrow keys.

The second problem is that if we move an item, then the index value of that item changes. For example, if we want to move items 3 and 4 down, then item 3 should move to position 4, and item 4 to position 5. As illustrated in figure 10.3, if we first

move item 3 down, it becomes item 4. If you then move item 4 down, you would effectively move the original item 3 into position 5.

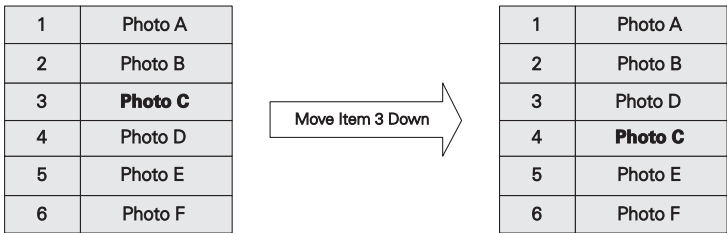


Figure 10.3 When the third item in the list is moved down, the original fourth item moves into position 3.

The trick here, as you may realize, is to move item 4 first, and then move item 3. In general terms, to move multiple items down, we must move the items starting from the bottom. Conversely, to move multiple items up, we must start at the top.

We will begin with the new methods required in the `PhotoAlbum` class.

Set the version number of the `MyPhotoAlbum` library to 10.2.

IMPLEMENT MOVE METHODS IN PHOTOALBUM CLASS		
	Action	Result
1	<p>In the <code>PhotoAlbum.cs</code> window, add a <code>MoveBefore</code> method to move a photograph at a specified index to the previous position.</p> <p>How-to</p> <ul style="list-style-type: none">a. Ensure the given index is valid.b. Remove the <code>Photograph</code> at this index from the list.c. Insert the removed photograph at the new position.	<pre>public void MoveBefore(int i) { if (i > 0 && i < this.Count) { Photograph photo = this[i]; this.RemoveAt(i); this.Insert(i-1, photo); } }</pre>
2	<p>Add a <code>MoveAfter</code> method to move a photograph at a specified index to the subsequent position.</p>	<pre>public void MoveAfter(int i) { if (i >= 0 && i < this.Count-1) { Photograph photo = this[i]; this.RemoveAt(i); this.Insert(i+1, photo); } }</pre>

With these methods in place, we are ready to implement `Click` event handlers for our `Move Up` and `Move Down` buttons. These handlers are shown in the following steps:

HANDLE THE MOVE BUTTONS		
	Action	Result
3	Implement a Click event handler for the Move Up button. Note: We could have used a foreach loop over the indices array here. This was written as a for loop to be consistent with the implementation of the Move Down handler.	<pre> private void btnMoveUp_Click (object sender, System.EventArgs e) { ListBox.SelectedIndexCollection indices = lstPhotos.SelectedIndices; int[] newSelects = new int[indices.Count]; // Move the selected items up for (int i = 0; i < indices.Count; i++) { int index = indices[i]; _album.MoveBefore(index); newSelects[i] = index - 1; } _bAlbumChanged = true; UpdateList(); // Reset the selections. lstPhotos.ClearSelected(); foreach (int x in newSelects) { lstPhotos.SetSelected(x, true); } } </pre>
4	Implement the Click handler for the Move Down button.	<pre> private void btnMoveDown_Click (object sender, System.EventArgs e) { ListBox.SelectedIndexCollection indices = lstPhotos.SelectedIndices; int[] newSelects = new int[indices.Count]; // Move the selected items down for (int i = indices.Count - 1; i >= 0; i--) { int index = indices[i]; _album.MoveAfter(index); newSelects[i] = index + 1; } _bAlbumChanged = true; UpdateList(); // Reset the selections. lstPhotos.ClearSelected(); foreach (int x in newSelects) { lstPhotos.SetSelected(x, true); } } </pre>

Both of these methods employ a number of members of the `ListBox` class. Let's examine the Move Down button handler in detail as a way to discuss these changes.

```

private void btnMoveDown_Click(object sender, System.EventArgs e)
{
    ListBox.SelectedIndexCollection indices = lstPhotos.SelectedIndexes;
    int[] newSelects = new int[indices.Count];

    // Move the selected items down
    for (int i = indices.Count - 1; i >= 0; i--)
    {
        int index = indices[i];
        _album.MoveAfter(index);
        newSelects[i] = index + 1;
    }

    _bAlbumChanged = true;
    UpdateList();

    // Reset the selections.
    lstPhotos.ClearSelected();
    foreach (int x in newSelects)
    {
        lstPhotos.SetSelected(x, true);
    }
}

```

Retrieve the selected items ①

Move selected items down ②

Update the list box ③

Reselect the items ④

The following points are highlighted in the code:

- ① A local `indices` variable is created to hold the index values of the selected items. The `SelectedIndices` property returns a `ListBox.SelectedIndexCollection` instance containing an array of the selected index values. The related `SelectedItems` property returns the actual objects selected. Note that an array of integers is also created to hold the new index positions of the objects after they have been moved.
- ② Starting from the bottom of the list, each selected item is moved down in the album. Note that the `MoveDown` button is disabled if the last item is selected, so we know for certain that `index + 1` will not produce an index which is out of range.
- ③ Once all the changes have been made to our album, we update the list box with the new entries. Note that the `UpdateList` method has a side effect of clearing the current selections from the list.
- ④ Once the list has been updated, the items need to be reselected. The `newSelects` array was created for this purpose. The `ClearSelected` method is used to remove any default selections added by the `UpdateList` method, and the `SetSelected` method is used to select each entry in the array.

You can run the application here if you like to see how these buttons work. The next section discusses the `Remove` button implementation.

10.2.3 HANDLING THE REMOVE BUTTON

The `Remove` button is a bit like the `Move Down` button. We have to be careful that the removal of one item does not cause us to remove incorrect entries on subsequent

items. We will again loop through the list of selected items starting from the end to avoid this problem.

Also note that by removing the selected photographs, we are making an irreversible change to the photo album. As a result, this is a good place to employ the `MessageBox` class to ensure that the user really wants to remove the photos.

HANDLE THE REMOVE BUTTON		
	Action	Result
1	Add a Click handler to the Remove button.	<pre>private void btnRemove_Click (object sender, System.EventArgs e) {</pre>
2	Implement this handler to confirm with the user that they really want to remove the selected photos. How-to Use the <code>MessageBox</code> class with the <code>Question</code> icon.	<pre> string msg; int n = lstPhotos.SelectedItems.Count; if (n == 1) msg = "Do you really want to " + "remove the selected photo?"; else msg = String.Format("Do you really want to " + "remove the {0} selected photos?", n); DialogResult result = MessageBox.Show(msg, "Remove Photos?", MessageBoxButtons.YesNo, MessageBoxIcon.Question);</pre>
3	If the user says Yes, then remove the selected items. How-to Use the <code>SelectedIndices</code> property.	<pre> if (result == DialogResult.Yes) { ListBox.SelectedIndexCollection indices = lstPhotos.SelectedIndices; for (int i = indices.Count - 1; i >= 0; i--) { _album.RemoveAt(indices[i]); } _bAlbumChanged = true; UpdateList(); } }</pre>

This code uses the `SelectedItems` property to retrieve the collection of selected objects. This property is used to determine how many items are selected so that our message to the user can include this information.

```
int n = lstPhotos.SelectedItems.Count;
```

To perform the deletion, we use the `SelectedIndices` property to retrieve the index numbers of each selected object. Since our list is based on the `PhotoAlbum` class, we know that the index in the list box corresponds to the index in the album. Removing a selection is a simple matter of removing the object at the given index from the album.

```
ListBox.SelectedIndexCollection indices = lstPhotos.SelectedIndices;
for (int i = indices.Count - 1; i >= 0; i--)
{
    _album.RemoveAt(indices[i]);
}
```

Compile and run the application to see the Remove button and the rest of the interface in action. Note that you can remove photographs and move them around and still decide not to save these changes when the album is closed.

If you look at our application so far, there is still some space available in the Albums group box. This space is intended for a `ComboBox` control holding the list of available albums. Now that we have seen different ways to use the `ListBox` control, it's time to take a look at the other .NET list control: the `ComboBox` class.

10.3 COMBO BOXES

A list box is quite useful for presenting a list of strings, such as the photographs in an album. There are times when only one item will ever be selected, or when the extra space necessary to display a list box is problematic or unnecessary. The `ComboBox` class is a type of `ListControl` object that displays a single item in a text box and permits selection from an associated list box. Since a user can enter new values into the text box control directly, a `ComboBox` allows additional items to be added much more simply than a `ListBox` control.

Features specific to the `ComboBox` class are shown in .NET Table 10.4. As you can see, a number of members are reminiscent of members from both the `ListBox` class and the `TextBox` class. The `TextBox` area of the control is sometimes called the editable portion of the control, even though it is not always editable, and the `ListBox` portion may be called the dropdown portion, since the list drops down below the text box portion for some display styles.

10.3.1 CREATING A COMBO BOX

In our `MyAlbumEditor` application, we will add a `ComboBox` control to permit quick and easy access to the list of albums stored in the default album directory. The entries for this control will be taken from the album file names discovered in this directory, and the user will not be able to add new entries by hand. Figure 10.4 shows how our application will look after this change, with the `ComboBox` dropdown list displayed.

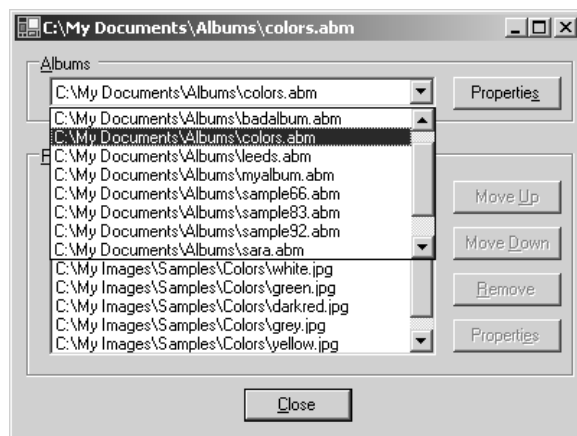


Figure 10.4
The dropdown list for a `ComboBox` is hidden until the user clicks on the small down arrow to reduce the amount of space required for the control on the

.NET Table 10.4 ComboBox class

The `ComboBox` class is a `ListControl` object that combines a `TextBox` control with a `ListBox` object. A user can select an item from the list or enter an item manually. A `ComboBox` can be displayed with or without the list box portion shown and with or without the text box portion editable, depending on the setting of the `DropDownStyle` property. When the list box portion is hidden, a down arrow is provided to display the list of available items. This class is part of the `System.Windows.Forms` namespace, and inherits from the `ListControl` class. See .NET Table 10.1 on page 316 for a list of members inherited by this class.

Public Properties	<code>DrawMode</code>	Gets or sets how elements in the list are drawn in a window.
	<code>DropDownStyle</code>	Gets or sets the style used to display the edit and list box controls in the combo box.
	<code>DropDownWidth</code>	Gets or sets the width of the list box portion of the control.
	<code>DroppedDown</code>	Gets or sets whether the combo box is currently displaying its list box portion.
	<code>Items</code>	Gets or sets the collection of items contained by this combo box.
	<code>MaxDropDownItems</code>	Gets or sets the maximum number of items permitted in the list box portion of the control.
	<code>MaxLength</code>	Gets or sets the maximum number of characters permitted in the text box portion of the control.
	<code>SelectedItem</code>	Gets or sets the currently selected item in the control.
	<code>SelectedText</code>	Gets or sets any text that is selected in the text box portion of the control.
	<code>Sorted</code>	Gets or sets whether the items in the control are sorted alphabetically.
Public Methods	<code>BeginUpdate</code>	Prevents the control from painting its contents while items are added to the list box.
	<code>SelectAll</code>	Selects all text in the text box portion of the control.
Public Events	<code>DrawItem</code>	Occurs when an owner-drawn combo box requires repainting.
	<code>DropDown</code>	Occurs just before the dropdown portion of a combo box is displayed.
	<code>SelectionChangeCommitted</code>	Occurs when the selected item in the control has changed and that change is confirmed.

The steps required to create the combo box for our application are as follows:

Set the version number of the MyAlbumEditor application to 10.3.

REPLACE OPEN BUTTON WITH A COMBOBOX CONTROL														
	Action	Result												
1	Delete the Open button in the MainForm.cs [Design] window.	The button and all related code added by Visual Studio are removed from the MainForm.cs source file. Any nonempty event handlers, in this case btnOpen_Click, remain in the file and must be removed manually.												
2	<p>Drag a ComboBox control into the left side of the Albums group box as shown in the graphic.</p> <table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>cmbxAlbums</td></tr><tr><td>Anchor</td><td>Top, Left, Right</td></tr><tr><td>DropDownStyle</td><td>DropDownList</td></tr><tr><td>Sorted</td><td>True</td></tr></table>	Settings		Property	Value	(Name)	cmbxAlbums	Anchor	Top, Left, Right	DropDownStyle	DropDownList	Sorted	True	
Settings														
Property	Value													
(Name)	cmbxAlbums													
Anchor	Top, Left, Right													
DropDownStyle	DropDownList													
Sorted	True													
3	<p>Replace the btnOpen_Click method in the MainForm.cs source file with an OpenAlbum method to open a given album file.</p> <p>Note: Most of the existing code for the btnOpen_Click method is removed. Any exception that occurs here will be the responsibility of the caller.</p>	<pre>private void OpenAlbum(string fileName) { CloseAlbum(); // Open the given album file _album.Open(fileName); this.Text = _album.FileName; UpdateList(); }</pre>												
4	Set the Enabled property for the Properties button in the Albums group box to false.	Note: We will enable this button when a valid album is selected in the combo box control.												
5	<p>Initialize the contents of the combo box in the OnLoad method.</p> <p>How-to</p> <p>Use the static GetFiles method from the Directory class to retrieve the set of album files in the default album directory.</p>	<pre>protected override void OnLoad(EventArgs e) { // Initialize the album _album = new PhotoAlbum(); // Initialize the combo box cmbxAlbums.DataSource = Directory.GetFiles(PhotoAlbum.DefaultDir, "*.abm"); base.OnLoad(e); }</pre>												
6	At the top of the file, indicate that we are using objects in the System.IO namespace.	<pre>... using System.IO;</pre>												

As we saw for our `ListBox` control, the `DataSource` property provides a quick and easy way to assign a collection of objects to the `cmbxAlbums` control. In this case, the `Directory.GetFiles` method returns an array of strings containing the set of file names in the given directory that match the given search string.

Our `ComboBox` is created with the `DropDownStyle` property set to `DropDownList`. This setting is taken from the `ComboBoxStyle` enumeration, and indicates that the list box associated with the combo box should not be displayed by default, and that the user cannot manually enter new values into the control. A complete list of values provided by the `ComboBoxStyle` enumeration is shown in .NET Table 10.5.

.NET Table 10.5 `ComboBoxStyle` enumeration

The `ComboBoxStyle` enumeration specifies the display behavior of a combo box control. This enumeration is part of the `System.Windows.Forms` namespace.

Enumeration Values	<code>DropDown</code>	The text portion of the control is editable. The list portion is only displayed when the user clicks an arrow button on the control. This is the default.
	<code>DropDownList</code>	The text portion of the control is not editable. The list portion is only displayed when the user clicks an arrow button on the control.
	<code>Simple</code>	The text portion of the control is editable, and the list portion of the control is always visible.

Feel free to compile and run your program if you like. The combo box will display the available albums, without the ability to actually open an album. Opening an album requires that we handle the `SelectedItemChanged` event for our combo box, which is the topic of the next section.

10.3.2 HANDLING THE SELECTED ITEM

Our `ComboBox` currently displays a selected album, but it doesn't actually open it. The previous section replaced the `Click` handler for the now-deleted `Open` button with an `OpenAlbum` method, so all we need to do here is recognize when a new album is selected and open the corresponding album.

The one issue we must deal with is the case where an invalid album exists. While we initialized our control to contain only album files ending with ".abm," it is still possible that one of these album files contains an invalid version number or other problem that prevents the album from loading. The following steps handle this case by disabling the `Properties` button and `ListBox` control when such a problem occurs. An appropriate error message is also displayed in the title bar.

OPEN THE ALBUM SELECTED IN THE COMBO BOX		
	Action	Result
1	Add a <code>SelectedItemChanged</code> handler to the combo box control.	<pre>private void cmbxAlbums_SelectedIndexChanged(object sender, System.EventArgs e) {</pre>
2	In the implementation of this handler, make sure the selected item is a new album. Note: If the selected album has not actually changed, there is no need to reload it.	<pre> string albumPath = cmbxAlbums.SelectedItem.ToString(); if (albumPath == _album.FileName) return;</pre>
3	Try to open the album.	<pre> try { CloseAlbum(); OpenAlbum(albumPath);</pre>
4	If the album is opened successfully, enable the album Properties button, and set the background color of the list box to normal window color.	<pre> btnAlbumProp.Enabled = true; lstPhotos.BackColor = SystemColors.Window; }</pre>
5	When an error occurs, display a message in the title bar to reflect this fact.	<pre> catch (Exception) { // Unable to open album this.Text = "Unable to open selected album";</pre>
6	Also clear the list box, set its background color to match the surrounding controls, and disable the album Properties button on the form.	<pre> lstPhotos.Items.Clear(); lstPhotos.BackColor = SystemColors.Control; btnAlbumProp.Enabled = false; } }</pre>

This code provides both text and visual cues on whether the selected album was successfully opened. Note how the `SelectedItem` property is used to retrieve the current selection. Even though we know this is a `string`, the framework provides us an object instance, so `ToString` must be called to extract the actual text.

```
string albumPath = cmbxAlbums.SelectedItem.ToString();
```

When the selected album opens successfully, the `ListBox` background is painted the normal window color as defined by the system and the Properties button in the Albums group box is enabled. Figure 10.1 at the beginning of this chapter shows the interface with a successfully opened album. When the album fails to open, the exception is caught and the title bar on the form is set to indicate this fact. In addition, the `ListBox` background is painted the default background color for controls and the Button control is disabled.


```

catch (Exception)
{
    // Unable to open album
    this.Text = "Unable to open selected album";
    lstPhotos.Items.Clear();
    lstPhotos.BackColor = SystemColors.Control;
    btnAlbumProp.Enabled = false;
}

```

An example of this situation appears in figure 10.5. The specified album, `badalbum.abm`, could not be opened, and between the title bar and the window this fact should be fairly clear.

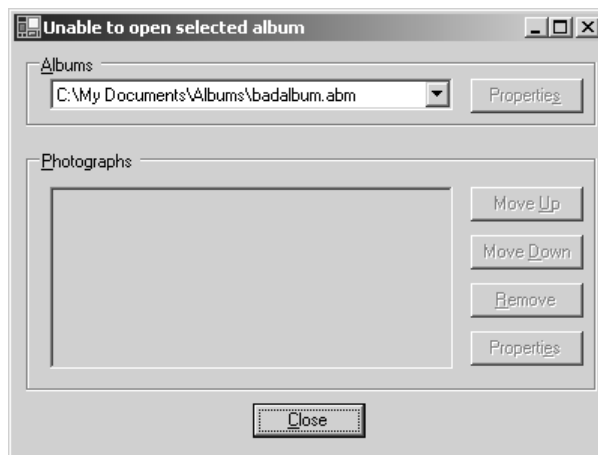


Figure 10.5
When the selected album cannot be loaded, only the Close button remains active.

TRY IT! The ComboBox in our application does not allow the user to manually enter a new album. This could be a problem if the user has created some albums in other directories. To fix this, add a `ContextMenu` object to the form and associate it with the Albums group box. Add a single menu item called “Add Album...” to this menu and create a `Click` event handler to allow the user to select additional album files to add to the combo box via the `OpenFileDialog` class.

Note that you have to modify the `ComboBox` to add the albums from the default directory manually within the `OnLoad` method. At present, since the `DataSource` property is assigned, the `Items` collection cannot be modified directly. Use `BeginUpdate` and `EndUpdate` to add a set of albums via the `Add` method in the `Items` collection, both in the `OnLoad` method and in the new `Click` event handler.

The next section provides an example of how to handle manual edits within a combo box.

10.4 COMBO BOX EDITS

The ComboBox created in the previous section used a fixed set of list entries taken from a directory on the disk. This permitted us to use the `DataSource` property for the list of items, and the `DropDownList` style to prevent the user from editing the text entry.

In this section we will create another ComboBox that permits manual updates to its contents by the user. Such a control is very useful when there are likely to be only a few possible entries, and you want the user to create additional entries as necessary. It so happens that we have just this situation for the `Photographer` property of our `Photograph` class.

Within a given album, there are likely to be only a handful of photographers for the images in that album. A combo box control is a good choice to permit the user to select the appropriate entry from the drop-down list. When a new photographer is required, the user can enter the new name in the text box.

Figure 10.6 shows how this combo box will look. You may notice that this list only displays four photographers, whereas our previous album combo box displayed eight album files at a time. A ComboBox control displays eight items by default. We will shorten the size here so that the list does not take up too much of the dialog window.

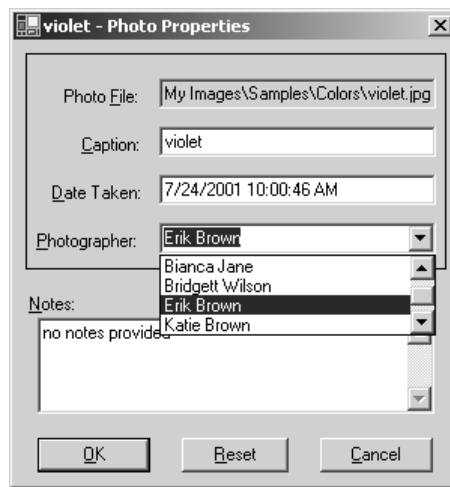


Figure 10.6
Note how the dropdown for the ComboBox extends outside of the Panel control. This is permitted even though the control is contained by the panel.

We will add this control to the `MyAlbumEditor` application in two parts. First we will create and initialize the contents of the control, and then we will support the addition of new photographers by hand.

10.4.1 REPLACING THE PHOTOGRAPHER CONTROL

The creation of our combo box within the `PhotoEditDlg` form is much like the one we created for the `MyAlbumEditor` application, with the exception of a few settings. The steps required to create this control are shown in the following table:

Set the version number of the `MyPhotoAlbum` library to 10.4.

ADD THE PHOTOGRAPHER COMBO BOX

	Action	Result												
1	In the PhotoEditDlg.cs [Design] window, delete the TextBox control associated with the Photographer label.	The control is removed from the form, and the code generated by Visual Studio is removed as well. The subsequent steps modify the manually entered code associated with this control.												
2	Place a ComboBox control on the form where the text box used to be. <div><table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>cmbxPhotographer</td></tr><tr><td>MaxDropDown</td><td>4</td></tr><tr><td>Sorted</td><td>True</td></tr><tr><td>Text</td><td>photographer</td></tr></table></div>	Settings		Property	Value	(Name)	cmbxPhotographer	MaxDropDown	4	Sorted	True	Text	photographer	The MaxDropDown property here specifies that the list portion of the combo box displays at most four items at a time, with any remaining items accessible via the scroll bar.
Settings														
Property	Value													
(Name)	cmbxPhotographer													
MaxDropDown	4													
Sorted	True													
Text	photographer													
3	Modify the ResetSettings method to initialize the items in the new combo box if necessary	<pre>protected override void ResetSettings() { // Initialize the ComboBox settings if (cmbxPhotographer.Items.Count == 0) {</pre>												
4	First add the “unknown” photographer to ensure that the list is never empty.	<pre>// Create the list of photographers cmbxPhotographer.BeginUpdate(); cmbxPhotographer.Items.Clear(); cmbxPhotographer.Items. Add("unknown");</pre>												
5	Then add to the ComboBox control any other photographers found in the album. How-to Use the Items.Contains method to check that a photographer is not already in the list. Note: This code is not terribly efficient, since it rescans the entire list each time the method is called. A better solution might be to modify the PhotoAlbum class to maintain the list of photographers assigned to Photograph objects in the album.	<pre>foreach (Photograph ph in _album) { if (ph.Photographer != null && !cmbxPhotographer.Items. Contains(ph.Photographer)) { cmbxPhotographer.Items. Add(ph.Photographer); } } cmbxPhotographer.EndUpdate(); }</pre>												

ADD THE PHOTOGRAPHER COMBO BOX <i>(continued)</i>		
	Action	Result
6	Select the photographer of the current photo in the combo box.	<pre> Photograph p = _album.CurrentPhoto; if (p != null) { txtPhotoFile.Text = p.FileName; txtCaption.Text = p.Caption; txtDate.Text = p.DateTaken.ToString(); cmbxPhotographer.SelectedItem = p.Photographer; txtNotes.Text = p.Notes; } </pre>
7	Update the SaveSettings method to save the photographer entered into the combo box. Note: We will stop ignoring the txtDate setting in the next chapter.	<pre> protected override bool SaveSettings() { Photograph p = _album.CurrentPhoto; if (p != null) { p.Caption = txtCaption.Text; // Ignore txtDate setting for now p.Photographer = cmbxPhotographer.Text; p.Notes = txtNotes.Text; } return true; } </pre>

Note how this code uses both the `SelectedItem` and `Text` properties for the `ComboBox` control. The `SelectedItem` property retrieves the object corresponding to the item selected in the list box, while the `Text` property retrieves the string entered into the text box. Typically these two values correspond to each other, but this is not always true, especially when the user manipulates the text value directly, as we shall see next.

10.4.2 UPDATING THE COMBO BOX DYNAMICALLY

With our control on the form, we now need to handle manual entries in the text box. This is normally handled via events associated with the `ComboBox` control. The `Validated` event, discussed in chapter 9, can be used to verify that a user-provided entry is part of the list and also add it to the list if necessary. The `TextChanged` event can be used to process the text while the user is typing.

We will handle both of these events in our code. First, let's add a `Validated` event handler, and then add code to auto-complete the entry as the user types.

VALIDATE THE PHOTOGRAPHER ENTRY		
	Action	Result
1	Add a Validated event handler for the cmbxPhotographer control.	<pre>private void cmbxPhotographer_Validated (object sender, System.EventArgs e) {</pre>
2	To implement this handler, get the text currently entered in the control.	<pre> string pg = cmbxPhotographer.Text;</pre>
3	If the cmbxPhotographer control does not contain this text, then add the new string to the combo box.	<pre> if (!cmbxPhotographer.Items.Contains(pg)) { _album.CurrentPhoto.Photographer = pg; cmbxPhotographer.Items.Add(pg); }</pre>
4	Set the selected item to the new text.	<pre> cmbxPhotographer.SelectedItem = pg; }</pre>

Our ComboBox is now updated whenever the user enters a new photographer, and the new entry will be available to other photographs in the same album.

Another change that might be nice is if the dialog automatically completed a partially entered photographer that is already on the list. For example, if the photographer “Erik Brown” is already present, and the user types in “Er,” it would be nice to complete the entry on the user’s behalf.

Of course, if the user is typing “Erin Smith,” then we would not want to prevent the user from doing so. This can be done by causing the control to select the auto-filled portion of the name as the user types. You will be able to experiment with this behavior yourself after following the steps in the subsequent table.

AUTO-COMPLETE THE TEXT ENTRY AS THE USER TYPES		
	Action	Result
5	Add a TextChanged event handler for the cmbxPhotographer control.	<pre>private void cmbxPhotographer_TextChanged (object sender, System.EventArgs e) {</pre>
6	Search for the current text in the list portion of the combo box.	<pre> string text = cmbxPhotographer.Text; int index = cmbxPhotographer.FindString(text);</pre>
7	If found, then adjust the text in the control to include the remaining portion of the matching entry.	<pre> if (index >= 0) { // Found a match string newText = cmbxPhotographer. Items[index].ToString(); cmbxPhotographer.Text = newText; cmbxPhotographer.SelectionStart = text.Length; cmbxPhotographer.SelectionLength = newText.Length - text.Length; } }</pre>

This code uses the `FindString` method to locate a match for the entered text. This method returns the index of the first object in the list with a display string beginning with the specified text. If no match is found, then a `-1` is returned.

```
int index = cmbxPhotographer.FindString(text);
```

When a match is found, the text associated with this match is extracted from the list and assigned to the text box portion of the control.

```
if (index >= 0)
{
    // Found a match
    string newText = cmbxPhotographer.Items[index].ToString();
    cmbxPhotographer.Text = newText;
}
```

The additional text inserted into the text box is selected using the `SelectionStart` and `SelectionLength` properties. The `SelectionStart` property sets the cursor location, and the `SelectionLength` property sets the amount of text to select.

```
cmbxPhotographer.SelectionStart = text.Length;
cmbxPhotographer.SelectionLength = newText.Length - text.Length;
}
```

TRY IT! The list portion of the control can be forced to appear as the user types with the `DroppedDown` property. Set this property to `true` in the `TextChanged` handler to display the list box when a match is found.

You may have realized that this handler introduces a slight problem with the use of the backspace key. When text is selected and the user presses the backspace key, the selected text is deleted rather than the previously typed character as a user would normally expect. Fix this behavior by handling the `KeyPress` event, discussed in chapters 9 and 12, to force the control to delete the last character typed rather than the selected text.

Before leaving our discussion of `ListControl` objects, it is worth noting that the controls we have discussed so far all contain textual strings. The .NET Framework automatically handles the drawing of these text strings within the list window. It is possible to perform custom drawing of the list elements, in a manner not too different than the one we used for our owner-drawn status bar panel in chapter 4.

As a final example in this chapter, let's take a look at how this is done.

10.5 **OWNER-DRAWN LISTS**

Typically, your `ListBox` and `ComboBox` controls will each display a list of strings. You assign objects to the list, and the `ToString` method is used to retrieve the string to display in the list. The string value of a specific property can be displayed in place of the `ToString` method by setting the `DisplayMember` property for the list. The .NET Framework retrieves and draws these strings on the form, and life is good.

There are times when you do not want to display a string, or when you would like to control exactly how the string looks. For these situations you must draw the list manually. This is referred to as an *owner-drawn list*, and the framework provides specific events and other mechanisms for drawing the list items in this manner.

In this section we modify our main `ListBox` control for the application to optionally include a small representation of the image associated with each photograph. Such an image is sometimes called a *thumbnail*, since it is a “thumbnail-sized” image. An example of our list box displaying these thumbnails is shown in figure 10.7. As you can see, the list includes a thumbnail image as well as the caption string from the photograph.



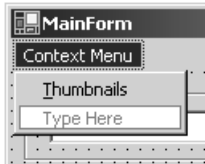
Figure 10.7
The `ListBox` here shows both the image and the caption for each photograph. Note how none of the items are selected in this list.

We will permit the user to switch between the thumbnail and pure text display using a context menu associated with the list box. This menu will be somewhat hidden, since users will not know it exists until they right-click on the list control. A hidden menu is not necessarily a good design idea, but it will suffice for our purposes. We will begin our example by adding this new menu.

10.5.1 ADDING A CONTEXT MENU

Since we would like to dynamically switch between an owner-drawn and a framework-drawn control, we need a way for the user to select the desired drawing method. We will use a menu for this purpose, and include a check mark next to the menu when the thumbnail images are shown. Context menus were discussed in chapter 3, so the following steps should be somewhat familiar.

Set the version number of the MyAlbumEditor application to 10.5.

ADD A CONTEXT MENU										
	Action	Result								
1	Add a ContextMenu control named ctxtPhotoList to the form in the MainForm.cs [Design] window.									
2	<div>Add a single menu item to this context menu.</div> <table><tr><th colspan="2">Settings</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>(Name)</td><td>menuThumbs</td></tr><tr><td>Text</td><td>&Thumbnail</td></tr></table>	Settings		Property	Value	(Name)	menuThumbs	Text	&Thumbnail	
Settings										
Property	Value									
(Name)	menuThumbs									
Text	&Thumbnail									
3	Set the ContextMenu property for the ListBox control to this new menu.									
4	Add a Click handler for the new menu item to reverse the Checked state of this menu.	<pre>private void menuThumbs_Click (object sender, System.EventArgs e) { menuThumbs.Checked = ! menuThumbs.Checked; }</pre>								
5	When checking the menu, set the DrawMode for the Photographs list to be owner-drawn.	<pre>if (menuThumbs.Checked) { lstPhotos.DrawMode = DrawMode.OwnerDrawVariable; }</pre>								
6	When unchecking the menu, set the DrawMode to its default setting. Also reset the default item height.	<pre>else { lstPhotos.DrawMode = DrawMode.Normal; lstPhotos.ItemHeight = lstPhotos.Font.Height + 2; } }</pre>								

The Click handler for our new menu simply toggles its Checked flag and sets the drawing mode based on the new value. The DrawMode property is used for both the ListBox and ComboBox controls to indicate how each item in the list will be drawn. The possible values for this property are shown in .NET Table 10.6. Since the size of our photographs in an album may vary, we allow the size of each element in the list to vary as well. As a result, we use the DrawMode.OwnerDrawVariable setting in our code.

The ItemHeight property contains the default height for each item in the list. When the DrawMode property is set to Normal, we set this property to the height of the current font plus 2 pixels. For our owner-drawn list, the item height depends on the size of the photograph we wish to draw. This requires that we assign the item height dynamically, and this is our next topic.

.NET Table 10.6 DrawMode enumeration

The `DrawMode` enumeration specifies the drawing behavior for the elements of a control. This enumeration is part of the `System.Windows.Forms` namespace. Controls that use this enumeration include the `ListBox`, `CheckedListBox`, and `ComboBox` classes, although the `CheckedListBox` class only supports the `Normal` setting.

Enumeration Values	<code>Normal</code>	All elements in the control are drawn by the .NET Framework and are the same size.
	<code>OwnerDrawFixed</code>	Elements in the control are drawn manually and are the same size.
	<code>OwnerDrawVariable</code>	Elements in the control are drawn manually and may vary in size.

10.5.2 SETTING THE ITEM HEIGHT

Since a `ListBox` normally holds text in a specific font, the default height of each list box item is just large enough to accommodate this font. In our case, we want to draw an image in each item, so the height of the default font is likely a bit on the small side. We can assign a more appropriate item height by handling the `MeasureItem` event. This event occurs whenever the framework requires the size of an owner-drawn item.

Note that this event does not occur with the setting `DrawMode.OwnerDrawFixed`, since the items are by definition all the same size. For this setting, the `ItemHeight` property should be assigned to the common height of the items. Since we are using the `DrawMode.OwnerDrawVariable` setting, this event will occur each time a list item must be custom drawn.

.NET Table 10.7 MeasureItemEventArgs class

The `MeasureItemEventArgs` class provides the event data necessary to determine the size of an owner-drawn item. This class is part of the `System.Windows.Forms` namespace, and inherits from the `System.EventArgs` class.

Public Properties	<code>Graphics</code>	Gets the graphics object to use when calculating measurements.
	<code>Index</code>	Gets the index of the item to measure.
	<code>ItemHeight</code>	Gets or sets the height of the specified item.
	<code>ItemWidth</code>	Gets or sets the width of the specified item.

A `MeasureItem` event handler receives a `MeasureItemEventArgs` class instance to permit an application to set the width and height of a given item. Specifics of this class are shown in .NET Table 10.7. In our case, we are drawing an image followed by a string. We will fit the image into a 45×45 pixel box, and use the `Caption` property as the string portion.

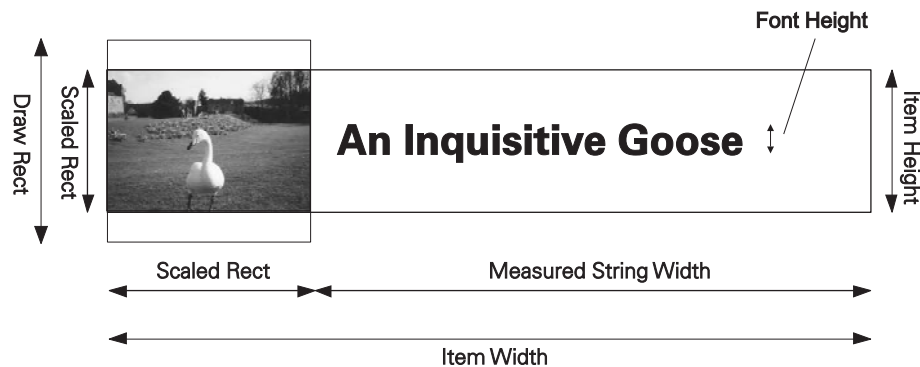


Figure 10.8 This figure shows the various measurements used to calculate a list item's width and height.

The following steps implement the code required for the `MeasureItem` event. Figure 10.8 illustrates the various measurements used to determine the width and height of the item.

CALCULATE THE LIST ITEM SIZE DYNAMICALLY		
	Action	Result
1	In the MainForm.cs window, add a static Rectangle to the MainForm class to hold the drawing rectangle for the image.	<pre>private static Rectangle _drawRect = new Rectangle(0,0,45,45);</pre>
2	Add a <code>MeasureItem</code> event handler for the <code>lstPhotos</code> list box.	<pre>private void lstPhotos_MeasureItem (object sender, Windows.Forms.MeasureItemEventArgs e) {</pre>
3	Calculate the size of the image when scaled into the drawing rectangle.	<pre>Photograph p = _album[e.Index]; Rectangle scaledRect = p.ScaleToFit(_drawRect);</pre>
4	Calculate the item's height.	<pre>e.ItemHeight = Math.Max(scaledRect.Height, lstPhotos.Font.Height) + 2;</pre>
5	Calculate the item's width.	<pre>e.ItemWidth = scaledRect.Width + 2 + (int) e.Graphics.MeasureString(p.Caption, lstPhotos.Font).Width; }</pre>

For the item's height, this code uses the larger of the scaled item's height and the `ListBox` control's font height, plus 2 pixels as padding between subsequent items in the list.

```
e.ItemHeight = Math.Max(scaledRect.Height, lstPhotos.Font.Height) + 2;
```

For the item's width, the width of the scaled image plus the width of the drawn string is used, plus 2 pixels as padding between the image and the text. To do this, the

Graphics.MeasureString method is used to calculate the size of the string when drawn with the Font object used by the ListBox control.

```
e.ItemWidth = scaledRect.Width + 2
+ e.Graphics.MeasureString(p.Caption, lstPhotos.Font);
```

Our final task is to draw the actual items using the DrawItem event.

10.5.3 DRAWING THE LIST ITEMS

As you may recall, the DrawItem event and related DrawItemEventArgs class were discussed in chapter 4. See .NET Table 4.4 on page 119 for an overview of the DrawItemEventArgs class.

Before we look at how to draw the list items in our application, let's make a small change to the Photograph class to improve the performance of our drawing. Since we may have to draw an item multiple times, it would be nice to avoid drawing the thumbnail from the entire image each time. To avoid this, let's create a Thumbnail property in our Photograph class to obtain a more appropriately sized image.

Set the version number of the MyPhotoAlbum library to 10.5.

STORE A THUMBNAIL IMAGE IN THE PHOTOGRAPH OBJECT		
	Action	Result
1	In the Photograph.cs file, create an internal _thumbnail field to store the new thumbnail image.	<pre>... private Bitmap _thumbnail = null;</pre>
2	Update the Dispose method to properly dispose of the new object.	<pre>public void Dispose() { if (_bitmap != null && _bitmap != InvalidPhotoImage) _bitmap.Dispose(); if (_thumbnail != null) _thumbnail.Dispose(); _bitmap = null; _thumbnail = null; }</pre>
3	Add a static constant to store the default width and height for a thumbnail.	<pre>private const int ThumbSize = 90;</pre>

STORE A THUMBNAIL IMAGE IN THE PHOTOGRAPH OBJECT		
	Action	Result
4	<p>Add a property to retrieve the thumbnail.</p> <p>Note: While we draw our list items into a 45-pixel box, we draw our thumbnail into a 90-pixel box. Aside from the fact that we might want to use the Thumbnail property in other code, it is beneficial, when downsizing an image, to have an original image with a higher resolution than the final size.</p>	<pre> public Bitmap Thumbnail { get { if (_thumbnail == null) { // Create the "thumbnail" bitmap Rectangle sr = this.ScaleToFit(new Rectangle(0,0, ThumbSize,ThumbSize)); Bitmap bm = new Bitmap(sr.Width, sr.Height); Graphics g = Graphics.FromImage(bm); GraphicsUnit u = g.PageUnit; g.DrawImage(this.Image, bm.GetBounds(ref u)); _thumbnail = bm; } return _thumbnail; } } </pre>

This ensures that we will not have to load up and scale the full-size image every time we draw an item. With this property in place, we have everything we need to draw our list items.

HANDLE THE DRAWITEM EVENT TO DRAW A LIST ITEM		
	Action	Result
5	Add a static Brush field to the MainForm.cs file.	<pre> private static SolidBrush _textBrush = new SolidBrush(SystemColors.WindowText); </pre> <p>Note: This will improve the performance of our handler by eliminating the need to recreate a brush each time an item is drawn.</p>
6	Add a DrawItem event handler for the ListBox control.	<pre> private void lstPhotos_DrawItem (object sender, System.Windows.Forms.DrawItemEventArgs e) { </pre>
7	To implement this method, get the Graphics and Photograph objects required for this handler.	<pre> Graphics g = e.Graphics; Photograph p = _album[e.Index]; </pre>

HANDLE THE DRAWITEM EVENT TO DRAW A LIST ITEM <i>(continued)</i>		
	Action	Result
8	<p>Calculate the Rectangle that will contain the thumbnail image.</p> <p>How-to</p> <ol style="list-style-type: none"> Use <code>e.Bounds</code> to obtain the bounding rectangle for item. Adjust this rectangle based on the size of the scaled image. 	<pre>Rectangle scaledRect = p.ScaleToFit(_drawRect); Rectangle imageRect = e.Bounds; imageRect.Y += 1; imageRect.Height = scaledRect.Height; imageRect.X += 2; imageRect.Width = scaledRect.Width;</pre>
9	<p>Draw the thumbnail image into this rectangle.</p> <p>How-to</p> <ol style="list-style-type: none"> Use <code>DrawImage</code> to paint the thumbnail into the rectangle. Use <code>DrawRectangle</code> to paint a black border around the image. 	<pre>g.DrawImage(p.Thumbnail, imageRect); g.DrawRectangle(Pens.Black, imageRect);</pre>
10	<p>Calculate the Rectangle that will contain the caption for the image.</p> <p>How-to</p> <p>Use the bounding rectangle without the image area and centered vertically for the current font.</p>	<pre>Rectangle textRect = new Rectangle(imageRect.Right + 2, imageRect.Y + ((imageRect.Height - e.Font.Height) / 2), e.Bounds.Width - imageRect.Width - 4, e.Font.Height);</pre>
11	<p>If the current item is selected, make sure the text will appear selected as well.</p> <p>How-to</p> <ol style="list-style-type: none"> Use the <code>State</code> property to determine if this item is selected. Use the system <code>Highlight</code> color for the background. Use the <code>HighlightText</code> color for the actual text. 	<pre>if ((e.State & DrawItemState.Selected) == DrawItemState.Selected) { _textBrush.Color = SystemColors.Highlight; g.FillRectangle(_textBrush, textRect); _textBrush.Color = SystemColors.HighlightText; }</pre> <p>Note: The <code>State</code> property used here defines the state settings for the current item. This contains an or'd set of values taken from the <code>DrawItemState</code> enumeration. The code here is preferred over the use of a method such as <code>ListBox.GetSelected</code> since these and other methods may not reflect recent user changes until after the <code>DrawItem</code> event is processed.</p>

HANDLE THE DRAWITEM EVENT TO DRAW A LIST ITEM <i>(continued)</i>		
	Action	Result
12	<p>If the current item is not selected, make sure the text will appear normally.</p> <p>How-to</p> <p>a. Use the system window color for the background.</p> <p>b. Use the WindowText color for the actual text.</p>	<pre> else { _textBrush.Color = SystemColors.Window; g.FillRectangle(_textBrush, textRect); _textBrush.Color = SystemColors.WindowText; } </pre>
13	<p>Draw the caption string in the text rectangle using the default font.</p>	<pre> g.DrawString(p.Caption, e.Font, _textBrush, textRect); } </pre>

Well done! You’ve just created your first owner-drawn list box. This code provides a number of features that should be useful in your own applications. It includes how to draw the image as well as the string for the item, and how to handle selected and deselected text. Compile and run the application. Click the Thumbnail context menu and watch the list display thumbnails. Click it again and the list reverts to normal strings.

TRY IT! Our list box currently displays the file name for each photograph when DrawMode is Normal, and the caption string when DrawMode is Owner-DrawVariable. It would be nice if the user could select which string to display in either mode.

Try implementing this change by adding additional entries to the ListBox control’s context menu. Add a parent menu called “Display As,” and a submenu to allow the user to select between “File Name,” “Caption,” and “Photographer.” Based on their selection, set the DisplayMember property for the list to the appropriate property string.

In normal draw mode, the framework picks up the DisplayMember property automatically. For the DrawItem event, you will need to retrieve the appropriate string based on the DisplayMember value. You can use string comparisons to do this, or use the System.Reflection namespace classes and types. This namespace is not discussed in detail in this book, but the following code excerpt can be used at the end of your DrawItem event handler to dynamically determine the value associated with the property corresponding to a given string.

```

PropertyInfo pi = typeof(Photograph).
    GetProperty(lstPhotos.DisplayMember);
object propValue = pi.GetValue(p, null);
g.DrawString(propValue.ToString(), e.Font,
    _textBrush, textRect);

```

This completes our discussion of list controls. The next section provides a quick recap of the chapter just in case you have already forgotten.

10.6 RECAP

This chapter discussed the basic list classes in the .NET Framework, namely the `ListBox` and `ComboBox` controls. We created a new application for this purpose, the `MyAlbumEditor` application, and built this application from the ground up using our existing `MyPhotoAlbum` library.

We began with a discussion of the common base class for list controls, namely the `ListControl` class, followed by a discussion of both single and multiple selection in the `ListBox` class. We saw how to enable and disable controls on the form based on the number of items selected, and how to handle double clicks for quick access to a common operation.

For the `ComboBox` class, we created a noneditable `ComboBox` to hold the list of available album files. Modifying the selected value automatically closed the previous album and opened the newly selected one. We then looked at an editable `ComboBox` for our photographer setting in the `PhotoEditDlg` dialog box. We discussed how to dynamically add new items to the list, and how to automatically select an existing item as the user is typing.

We ended with a discussion of owner-drawn list items by providing the option of displaying image thumbnails in our list box. We saw how to draw both images and text, including selected text.

There are additional controls than those discussed in chapters 9 and 10, of course. We will see some of these in the next chapter, and others as we continue our progression through the book. In chapter 11 we continue with our new `MyAlbumEditor` application, and look at `Tab` pages as a way to organize large amounts of information on a single form.