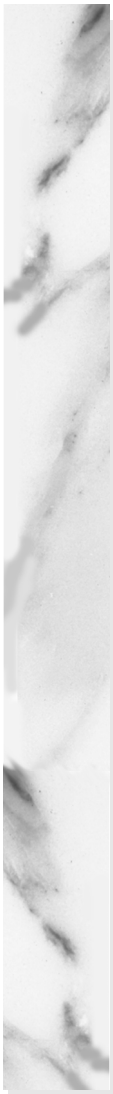
Windows Forms Controls



Topics in This Chapter

- *Introduction:* A class hierarchy diagram offers a natural way to group Windows Forms controls by their functionality.
- *Button Controls:* The `Button`, `CheckBox`, and `RadioButton` controls are designed to permit users to make one or more selections on a form.
- *PictureBox and TextBoxt Controls:* The `PictureBox` control is used to display and scale images; the `TextBox` control can be used to easily display and edit single or multiple lines of text.
- *List Controls:* The `ListBox`, `ComboBox`, and `CheckListBox` offer different interfaces for displaying and manipulating data in a list format.
- *ListView and TreeView Controls:* The `ListView` offers multiple views for displaying data items and their associated icons. The `TreeView` presents hierarchical information in an easy-to-navigate tree structure.
- *Timer and Progress Bar Controls:* A timer can be used to control when an event is invoked, a `ProgressBar` to visually monitor the progress of an operation.
- *Building a User Control:* When no control meets an application's needs, a custom one can be crafted by combining multiple controls or adding features to an existing one.
- *Moving Data Between Controls:* Drag and drop provides an easy way for users to copy or move an item from one control to another. .NET offers a variety of classes and events required to implement this feature.
- *Using Resources:* Resources required by a program, such as title, descriptive labels, and images, can be embedded within an application's assembly or stored in a *satellite* assembly. This is particularly useful for developing international applications.

Chapter 7

The previous chapter introduced the `Control` class and the methods, properties, and events it defines for all controls. This chapter moves beyond that to examine the specific features of individual controls. It begins with a survey of the more important .NET controls, before taking an in-depth look at how to implement controls such as the `TextBox`, `ListBox`, `TreeView`, and `ListView`. Also included is a discussion of the .NET drag-and-drop features that are used to move or copy data from one control to another.

Windows Forms (WinForms) are not restricted to using the standard built-in controls. Custom GUI controls can be created by extending an existing control, building a totally new control, or fashioning a user control from a set of related widgets. Examples illustrate how to extend a control and construct a user control. The chapter concludes with a look at resource files and how they are used to create GUI applications that support users from multiple countries and cultures.

7.1 A Survey of .NET Windows Forms Controls

The `System.Windows.Forms` namespace contains a large family of controls that add both form and function to a Windows-based user interface. Each control inherits a common set of members from the `Control` class. To these, it adds the methods, properties, and events that give the control its own distinctive behavior and appearance.

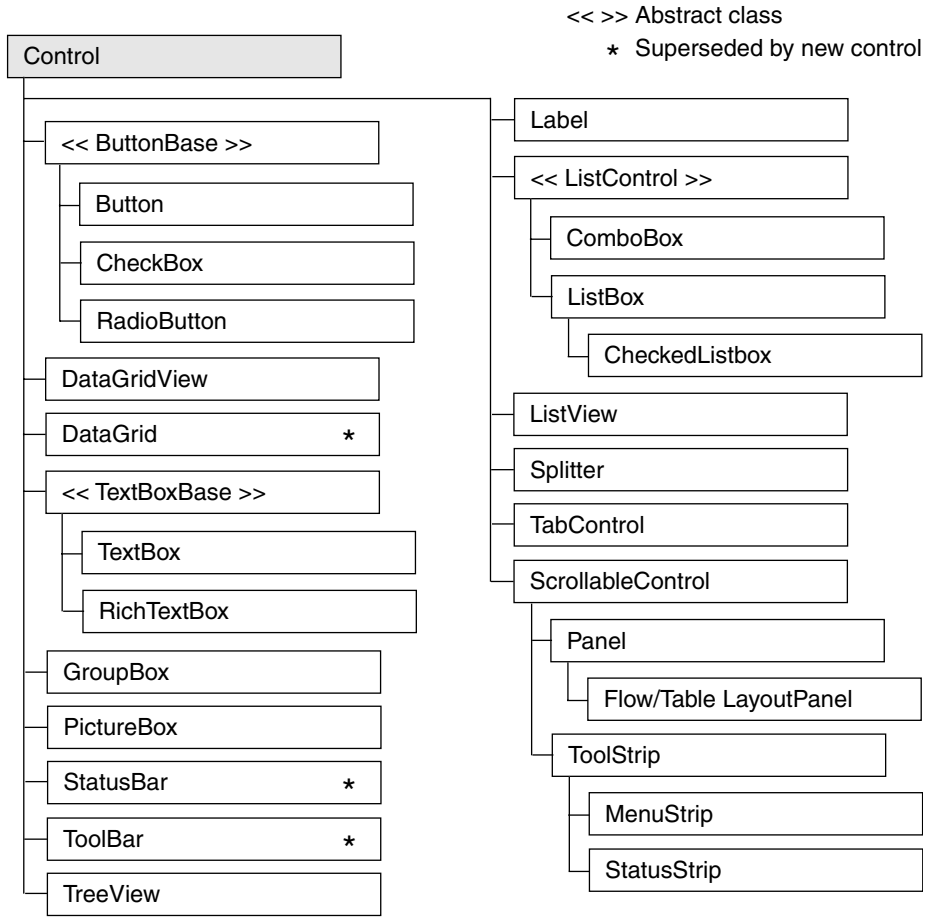


Figure 7-1 Windows Forms control hierarchy

Figure 7-1 shows the inheritance hierarchy of the Windows Forms controls. The controls marked by an asterisk (`*`) exist primarily to provide backward compatibility between .NET 2.0 and .NET 1.x. Specifically, the `DataGrid` has been superseded by the `DataGridView`, the `StatusBar` by the `StatusStrip`, and the `ToolBar` by the `ToolStrip`. Table 7-1 provides a summary of the more frequently used controls in this hierarchy.

Table 7-1 Selected Windows Forms Controls

Control	Use	Description
`Button`	Fires an event when a mouse click occurs or the Enter or Esc key is pressed.	Represents a button on a form. Its `text` property determines the caption displayed on the button's surface.
`CheckBox`	Permits a user to select one or more options.	Consists of a check box with text or an image beside it. The check box can also be represented as a button by setting: `checkBox1.Appearance = Appearance.Button`
`CheckedListBox`	Displays list of items.	`ListBox` with checkbox preceding each item in list.
`ComboBox`	Provides `TextBox` and `ListBox` functionality.	Hybrid control that consists of a text-box and a drop-down list. It combines properties from both the `TextBox` and the `ListBox`.
`DataGridView` `GridView`	Manipulates data in a grid format.	The `DataGridView` is the foremost control to represent relational data. It supports binding to a database. The `DataGridView` was introduced in .NET 2.0 and supersedes the `DataGrid`.
`GroupBox`	Groups controls.	Use primarily to group radio buttons; it places a border around the controls it contains.
`ImageList`	Manages a collection of images.	Container control that holds a collection of images used by other controls such as the `ToolStrip`, `ListView`, and `TreeView`.
`Label`	Adds descriptive information to a form.	Text that describes the contents of a control or instructions for using a control or form.
`ListBox`	Displays a list of items—one or more of which may be selected.	May contain simple text or objects. Its methods, properties, and events allow items to be selected, modified, added, and sorted.

Table 7-1 Selected Windows Forms Controls *(continued)*

Control	Use	Description
`ListView`	Displays items and subitems.	May take a grid format where each row represents a different item and subitems. It also permits items to be displayed as icons.
`MenuStrip`	Adds a menu to a form.	Provides a menu and submenu system for a form. It supersedes the `MainMenu` control.

`Panel`
`FlowPanelLayout`
`TablePanelLayout`
	Groups controls.	A visible or invisible container that groups controls. Can be made scrollable.
		`FlowPanelLayout` automatically aligns controls vertically or horizontally.
		`TablePanelLayout` aligns controls in a grid.

`PictureBox`	Contains a graphic.	Used to hold images in a variety of standard formats. Properties enable images to be positioned and sized within control's borders.
`ProgressBar`	Depicts an application's progress.	Displays the familiar progress bar that gives a user feedback regarding the progress of some event such as file copying.
`RadioButton`	Permits user to make one choice among a group of options.	Represents a Windows radio button.
`StatusStrip`	Provides a set of panels that indicate program status.	Provides a status bar that is used to provide contextual status information about current form activities.
`TextBox`	Accepts user input.	Can be designed to accept single- or multi-line input. Properties allow it to mask input for passwords, scroll, set letter casing automatically, and limit contents to read-only.
`TreeView`	Displays data as nodes in a tree.	Features include the ability to collapse or expand, add, remove, and copy nodes in a tree.

This chapter lacks the space to provide a detailed look at each control. Instead, it takes a selective approach that attempts to provide a flavor of the controls and features that most benefit the GUI developer. Notable omissions are the `DataGridView` control, which is included in the discussion of data binding in Chapter 12, "Data Binding with Windows Forms Controls," and the menu controls that were discussed in Chapter 6, "Building Windows Forms Applications."

7.2 Button Classes, Group Box, Panel, and Label

The Button Class

A button is the most popular way to enable a user to initiate some program action. Typically, the button responds to a mouse click or keystroke by firing a `Click` event that is handled by an event handler method that implements the desired response.

constructor: `public Button()`

The constructor creates a button instance with no label. The button's `Text` property sets its caption and can be used to define an access key (see *Handling Button Events* section); its `Image` property is used to place an image on the button's background.

Setting a Button's Appearance

Button styles in .NET are limited to placing text and an image on a button, making it flat or three-dimensional, and setting the background/foreground color to any available color. The following properties are used to define the appearance of buttons, check boxes, and radio buttons:

`FlatStyle` This can take four values: `FlatStyle.Flat`, `FlatStyle.Popup`, `FlatStyle.Standard`, and `FlatStyle.System`. `Standard` is the usual three-dimensional button. `Flat` creates a flat button. `Popup` creates a flat button that becomes three-dimensional on a mouse-over. `System` results in a button drawn to suit the style of the operating system.

`Image` Specifies the image to be placed on the button. The `Image.FromFile` method is used to create the image object from a specified file:

```
button1.Image = Image.FromFile("c:\\book.gif");
```

`ImageAlign`	Specifies the position of the image on the button. It is set to a value of the `ContentAlignment` enum:
	`button1.ImageAlign = ContentAlignment.MiddleRight;`
`TextAlign`	Specifies the position of text on the image using the `ContentAlignment` value.

Handling Button Events

A button's `Click` event can be triggered in several ways: by a mouse click of the button, by pressing the Enter key or space bar, or by pressing the Alt key in combination with an access key. An access key is created by placing an `&` in front of one of the characters in the control's `Text` property value.

The following code segment declares a button, sets its access key to C, and registers an event handler to be called when the `Click` event is triggered:

```
Button btnClose = new Button();
btnClose.Text= "&Close";  // Pushing ALT + C triggers event
btnClose.Click += new EventHandler(btnClose_Clicked);
// Handle Mouse Click, ENTER key, or Space Bar
private void btnClose_Clicked(object sender, System.EventArgs e)
{ this.Close();  }
```

Note that a button's `Click` event can also occur in cases when the button does not have focus. The `AcceptButton` and `CancelButton` form properties can specify a button whose `Click` event is triggered by pushing the Enter or Esc keys, respectively.

Core Suggestion

Set a form's `CancelButton` property to a button whose `Click` event handler closes the form. This provides a convenient way for users to close a window by pushing the Esc key.

The CheckBox Class

The `CheckBox` control allows a user to select a combination of options on a form—in contrast to the `RadioButton`, which allows only one selection from a group.

`constructor:` `public CheckBox()`

The constructor creates an unchecked check box with no label. The `Text` and `Image` properties allow the placement of an optional text description or image beside the box.

Setting a CheckBox's Appearance

Check boxes can be displayed in two styles: as a traditional check box followed by text (or an image) or as a toggle button that is raised when unchecked and flat when checked. The appearance is selected by setting the `Appearance` property to `Appearance.Normal` or `Appearance.Button`. The following code creates the two check boxes shown in Figure 7-2.

```
// Create traditional check box
this.checkBox1 = new CheckBox();
this.checkBox1.Location =
        new System.Drawing.Point(10,120);
this.checkBox1.Text = "La Traviata";
this.checkBox1.Checked = true;
// Create Button style check box
this.checkBox2 = new CheckBox();
this.checkBox2.Location =
        new System.Drawing.Point(10,150);
this.checkBox2.Text = "Parsifal";
this.checkBox2.Appearance = Appearance.Button;
this.checkBox2.Checked = true;
this.checkBox2.TextAlign = ContentAlignment.MiddleCenter;
```

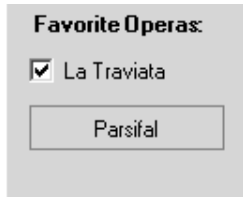


Figure 7-2 `CheckBox` styles

The RadioButton Class

The `RadioButton` is a selection control that functions the same as a check box except that only one radio button within a group can be selected. A group consists of multiple controls located within the same immediate container.

`constructor:` `public RadioButton()`

The constructor creates an unchecked `RadioButton` with no associated text. The `Text` and `Image` properties allow the placement of an optional text description or image beside the box. A radio button's appearance is defined by the same properties used with the check box and button: `Appearance` and `FlatStyle`.

Placing Radio Buttons in a Group

Radio buttons are placed in groups that allow only one item in the group to be selected. For example, a 10-question multiple choice form would require 10 groups of radio buttons. Aside from the functional need, groups also provide an opportunity to create an aesthetically appealing layout.

The frequently used `GroupBox` and `Panel` container controls support background images and styles that can enhance a form's appearance. Figure 7-3 shows the striking effect (even more so in color) that can be achieved by placing radio buttons on top of a `GroupBox` that has a background image.

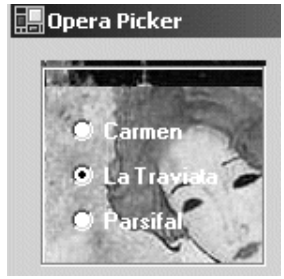


Figure 7-3 Radio buttons in a `GroupBox` that has a background image

Listing 7-1 presents a sample of the code that is used to place the radio buttons on the `GroupBox` control and make them transparent so as to reveal the background image.

Listing 7-1 Placing Radio Buttons in a `GroupBox`

```
using System.Drawing;
using System.Windows.Forms;
public class OperaForm : Form
{
   private RadioButton radioButton1;
   private RadioButton radioButton2;
   private RadioButton radioButton3;
   private GroupBox groupBox1;
   public OperaForm()
   {
      this.groupBox1 = new GroupBox();
      this.radioButton3 = new RadioButton();
      this.radioButton2 = new RadioButton();
```

Listing 7-1 Placing Radio Buttons in a `GroupBox` *(continued)*

```
        this.radioButton1 = new RadioButton();
        // All three radio buttons are created like this
        // For brevity only code for one button is included
        this.radioButton3.BackColor = Color.Transparent;
        this.radioButton3.Font = new Font("Microsoft Sans Serif",
                                          8.25F, FontStyle.Bold);
        this.radioButton3.ForeColor =
             SystemColors.ActiveCaptionText;
        this.radioButton3.Location = new Point(16, 80);
        this.radioButton3.Name = "radioButton3";
        this.radioButton3.Text = "Parsifal";
        // Group Box
        this.groupBox1 = new GroupBox();
        this.groupBox1.BackgroundImage =
             Image.FromFile("C:\\opera.jpg");
        this.groupBox1.Size = new Size(120, 112);
        // Add radio buttons to groupbox
        groupBox1.Add( new Control[]{radioButton1,radiobutton2,
                                     radioButton3});
    }
}
```

Note that the `BackColor` property of the radio button is set to `Color.Transparent`. This allows the background image of `groupBox1` to be displayed. By default, `BackColor` is an *ambient* property, which means that it takes the color of its parent control. If no color is assigned to the radio button, it takes the `BackColor` of `groupBox1` and hides the image.

The GroupBox Class

A `GroupBox` is a container control that places a border around its collection of controls. As demonstrated in the preceding example, it is often used to group radio buttons; but it is also a convenient way to organize and manage any related controls on a form. For example, setting the `Enabled` property of a group box to `false` disables all controls in the group box.

`constructor:` `public GroupBox()`

The constructor creates an untitled `GroupBox` having a default width of 200 pixels and a default height of 100 pixels.

The Panel Class

The `Panel` control is a container used to group a collection of controls. It's closely related to the `GroupBox` control, but as a descendent of the `ScrollableControl` class, it adds a scrolling capability.

constructor: `public Panel()`

Its single constructor creates a borderless container area that has scrolling disabled. By default, a `Panel` takes the background color of its container, which makes it invisible on a form.

Because the `GroupBox` and `Panel` serve the same purpose, the programmer is often faced with the choice of which to use. Here are the factors to consider in selecting one:

- A `GroupBox` may have a visible caption, whereas the `Panel` does not.
- A `GroupBox` always displays a border; a `Panel`'s border is determined by its `BorderStyle` property. It may be set to `BorderStyle.None`, `BorderStyle.Single`, or `BorderStyle.Fixed3D`.
- A `GroupBox` does not support scrolling; a `Panel` enables automatic scrolling when its `AutoScroll` property is set to `true`.

A `Panel` offers no features to assist in positioning or aligning the controls it contains. For this reason, it is best used when the control layout is known at design time. But this is not always possible. Many applications populate a form with controls based on criteria known only at runtime. To support the dynamic creation of controls, .NET offers two layout containers that inherit from `Panel` and automatically position controls within the container: the `FlowLayoutPanel` and the `TableLayoutPanel`.

The FlowLayoutPanel Control

Figure 7-4 shows the layout of controls using a `FlowLayoutPanel`.

Figure 7-4 `FlowLayoutPanel`

This "no-frills" control has a single parameterless constructor and two properties worth noting: a `FlowDirection` property that specifies the direction in which controls

are to be added to the container, and a `WrapControls` property that indicates whether child controls are rendered on another row or truncated.

The following code creates a `FlowLayoutPanel` and adds controls to its collection:

```
FlowLayoutPanel flp = new FlowLayoutPanel();
flp.FlowDirection = FlowDirection.LefttoRight;
// Controls are automatically positioned left to right
flp.Controls.Add(Button1);
flp.Controls.Add(Button2);
flp.Controls.Add(TextBox1);
flp.Controls.Add(Button3);
this.Controls.Add(flp);     // Add container to form
```

The `FlowDirection` enumerator members are `BottomUp`, `LeftToRight`, `RighttoLeft`, and `TopDown`. `LefttoRight` is the default.

TableLayoutPanel Control

Figure 7-5 shows the grid layout that results from using a `TableLayoutPanel` container.

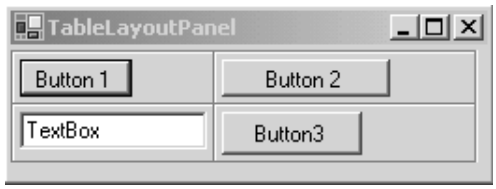


Figure 7-5 `TableLayoutPanel` organizes controls in a grid

This code segment creates a `TableLayoutPanel` and adds the same four controls used in the previous example. Container properties are set to define a layout grid that has two rows, two columns, and uses an `Inset` border style around each cell. Controls are always added to the container moving left-to-right, top-to-bottom.

```
TableLayoutPanel tlp = new TableLayoutPanel();
// Causes the inset around each cell
tlp.CellBorderStyle = TableLayoutPanelCellBorderStyle.Inset;
tlp.ColumnCount = 2;      // Grid has two columns
tlp.RowCount    = 2;      // Grid has two rows
// If grid is full add extra cells by adding column
tlp.GrowStyle = TableLayoutPanelGrowStyle.AddColumns;
// Padding (pixels)within each cell (left, top, right, bottom)
```

```
tlp.Padding = new Padding(1,1,4,5);
tlp.Controls.Add(Button1);
tlp.Controls.Add(Button2);
// Other controls added here
```

The `GrowStyle` property is worth noting. It specifies how controls are added to the container when all of its rows and columns are filled. In this example, `AddColumns` specifies that a column be added to accommodate new controls. The other options are `AddRows` and `None`; the latter causes an exception to be thrown if an attempt is made to add a control when the panel is filled.

The Label Class

The `Label` class is used to add descriptive information to a form.

constructor: `public Label()`

The constructor creates an instance of a label having no caption. Use the `Text` property to assign a value to the label. The `Image`, `BorderStyle`, and `TextAlign` properties can be used to define and embellish the label's appearance.

Figure 7-6 Label containing an image and text

The following code creates the label shown in Figure 7-6:

```
Label imgLabel = new Label();
imgLabel.BackColor= Color.White;
Image img = Image.FromFile("c:\\rembrandt.jpg");
imgLabel.Image= img;
imgLabel.ImageAlign= ContentAlignment.TopCenter;
imgLabel.Text="Rembrandt";
imgLabel.TextAlign= ContentAlignment.BottomCenter;
imgLabel.BorderStyle= BorderStyle.Fixed3D;
imgLabel.Size = new Size(img.Width+10, img.Height+25);
```

One of its less familiar properties is `UseMnemonic`. By setting it to `true` and placing a mnemonic (& followed by a character) in the label's text, you can create an access key. For example, if a label has a value of `&Sum`, pressing Alt-S shifts the focus to the control (based on tab order) following the label.

7.3 PictureBox and TextBox Controls

The PictureBox Class

The `PictureBox` control is used to display images having a bitmap, icon, metafile, JPEG, GIF, or PNG format. It is a dynamic control that allows images to be selected at design time or runtime, and permits them to be resized and repositioned within the control.

constructor: `public PictureBox()`

The constructor creates an empty (`Image = null`) picture box that has its `SizeMode` property set so that any images are displayed in the upper-left corner of the box.

The two properties to be familiar with are `Image` and `SizeMode`. `Image`, of course, specifies the graphic to be displayed in the `PictureBox`. `SizeMode` specifies how the image is rendered within the `PictureBox`. It can be assigned one of four values from the `PictureBoxSizeMode` enumeration:

1. `AutoSize`. `PictureBox` is sized to equal the image.
2. `CenterImage`. Image is centered in box and clipped if necessary.
3. `Normal`. Image is place in upper-left corner and clipped if necessary.
4. `StretchImage`. Image is stretched or reduced to fit in box.

Figure 7-7 illustrates some of the features of the `PictureBox` control. It consists of a form with three small picture boxes to hold thumbnail images and a larger picture box to display a full-sized image. The large image is displayed when the user double-clicks on a thumbnail image.

The code, given in Listing 7-2, is straightforward. The event handler `ShowPic` responds to each `DoubleClick` event by setting the `Image` property of the large `PictureBox` (`bigPicture`) to the image contained in the thumbnail. Note that the original images are the size of `bigPicture` and are automatically reduced (by setting `SizeMode`) to fit within the thumbnail picture boxes.

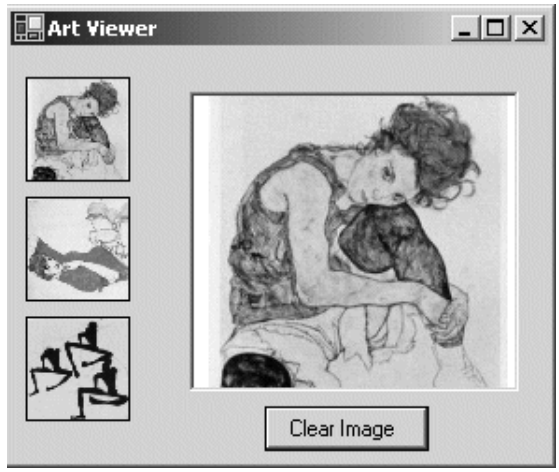


Figure 7-7 Thumbnail images in small picture boxes are displayed at full size in a larger viewing window

Listing 7-2 Working with Picture Boxes

```
using System;
using System.Drawing;
using System.Windows.Forms;
public class ArtForm : Form
{
   private PictureBox bigPicture;
   private PictureBox tn1;
   private PictureBox tn2;
   private PictureBox tn3;
   private Button btnClear;
   public ArtForm()
   {
      bigPicture = new PictureBox();
      tn1 = new PictureBox();
      tn2 = new PictureBox();
      tn3 = new PictureBox();
      btnClear = new Button();
      bigPicture.Location = new Point(90, 30);
      bigPicture.Name = "bigPicture";
      bigPicture.Size = new Size(160, 160);
      this.Controls.Add(bigPicture);
```

Listing 7-2 Working with Picture Boxes *(continued)*

```
        // Define picturebox to hold first thumbnail image
        tn1.BorderStyle = BorderStyle.FixedSingle;
        tn1.Cursor = Cursors.Hand;
        tn1.Image = Image.FromFile("C:\\schiele1.jpg");
        tn1.Location = new Point(8, 16);
        tn1.Name = "tn1";
        tn1.Size = new Size(56, 56);
        tn1.SizeMode = PictureBoxSizeMode.StretchImage;
        this.Controls.Add(tn1);
        // Code for other thumbnails would go here
        // Button to clear picture box
        btnClear.Location = new Point(136, 192);
        btnClear.Name = "btnClear";
        btnClear.Size = new Size(88, 24);
        btnClear.Text = "Clear Image";
        this.Controls.Add(btnClear);
        btnClear.Click += new EventHandler(this.btnClear_Click);
        // Set up event handlers for double click events
        tn1.DoubleClick += new EventHandler(ShowPic);
        tn2.DoubleClick += new EventHandler(ShowPic);
        tn3.DoubleClick += new EventHandler(ShowPic);
    }
    static void Main()
    {
        Application.Run(new ArtForm());
    }
    private void btnClear_Click(object sender, EventArgs e)
    {
        bigPicture.Image = null;   // Clear image
    }
    private void ShowPic (object sender, EventArgs e)
    {
        // Sender is thumbnail image that is double clicked
        bigPicture.Image = ((PictureBox) sender).Image;
    }
}
```

The TextBox Class

The familiar `TextBox` is an easy-to-use control that has several properties that affect its appearance, but few that control its content. This leaves the developer with the task of setting up event handlers and data verification routines to control what is entered in the box.

```
constructor: public TextBox()
```

The constructor creates a `TextBox` that accepts one line of text and uses the color and font assigned to its container. From such humble origins, the control is easily transformed into a multi-line text handling box that accepts a specific number of characters and formats them to the left, right, or center. Figure 7-8 illustrates some of the properties used to do this.

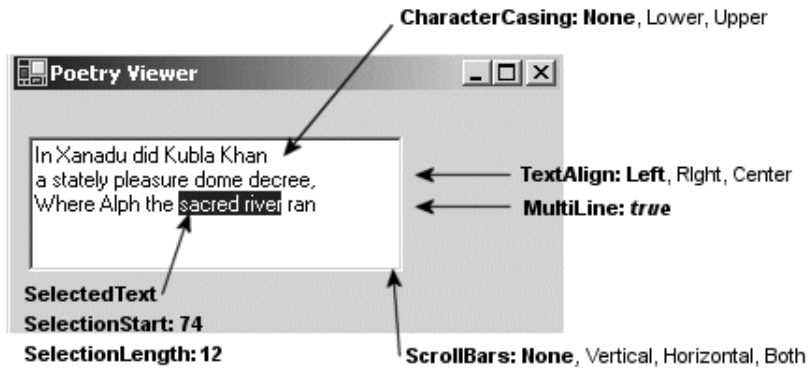


Figure 7-8 `TextBox` properties

The text is placed in the box using the `Text` property and `AppendText` method:

```
txtPoetry.Text =
   "In Xanadu did Kubla Khan\r\na stately pleasure dome decree,";
txtPoetry.AppendText("\r\nWhere Alph the sacred river ran");
```

A couple of other `TextBox` properties to note are `ReadOnly`, which prevents text from being modified, and `PasswordChar`, which is set to a character used to mask characters entered—usually a password.

TextBoxes and Carriage Returns

When storing data from a `TextBox` into a database, you want to make sure there are no special characters embedded in the text, such as a carriage return. If you look at the `TextBox` properties, you'll find `AcceptsReturn`, which looks like a simple solution. Setting it to `false` should cause a `TextBox` to ignore the user pressing an Enter key. However, the name of this property is somewhat misleading. It only works when the form's `AcceptButton` property is set to a button on the form. Recall that this property causes the associated button's `Click` handler to be executed when the Enter key is pressed. If `AcceptButton` is not set (and the `MultiLine` property of the text box is set to `true`), the `TextBox` receives a newline (`\r\n`) when the Enter key is pushed.

This leaves the developer with the task of handling unwanted carriage returns. Two approaches are available: capture the keystrokes as they are entered or extract the characters before storing the text. The first approach uses a keyboard event handler, which you should be familiar with from the previous chapter.

```
// Set up event handler in constructor for TextBox txtPoetry
txtPoetry.KeyPress += new KeyPressEventHandler(onKeyPress);

private void onKeyPress( object sender, KeyPressEventArgs e)
{
   if(e.KeyChar == (char)13) e.Handled = true;
}
```

Setting `Handled` to `true` prevents the carriage return/linefeed from being added to the text box. This works fine for keyboard entry but has no effect on a cut-and-paste operation. To cover this occurrence, you can use the keyboard handling events described in Chapter 6 to prevent pasting, or you can perform a final verification step that replaces any returns with a blank or any character of your choice.

```
txtPoetry.Text = txtPoetry.Text.Replace(Environment.NewLine," ");
```

Core Note

Two common approaches for entering a carriage return/linefeed programmatically into a `TextBox` are

```
txtPoetry.Text = "Line 1\r\nLine 2";
txtPoetry.Text = "Line 1"+Environment.NewLine+"Line 2";
```

7.4 ListBox, CheckedListBox, and ComboBox Classes

The ListBox Class

The `ListBox` control is used to provide a list of items from which the user may select one or more items. This list is typically text but can also include images and objects. Other features of the `ListBox` include methods to perform text-based searches, sorting, multi-column display, horizontal and vertical scroll bars, and an easy way to override the default appearance and create owner-drawn `ListBox` items.

```
constructor: public ListBox()
```

The constructor creates an empty `ListBox`. The code to populate a `ListBox` is typically placed in the containing form's constructor or `Form.Load` event handler. If the `ListBox.Sorted` property is set to `true`, `ListBox` items are sorted alphabetically in ascending order. Also, vertical scroll bars are added automatically if the control is not long enough to display all items.

Adding Items to a ListBox

A `ListBox` has an `Items` collection that contains all elements of the list. Elements can be added by binding the `ListBox` to a data source (described in Chapter 11, "ADO.NET") or manually by using the `Add` method. If the `Sorted` property is `false`, the items are listed in the order they are entered. There is also an `Insert` method that places an item at a specified location.

```
lstArtists.Items.Add("Monet");
lstArtists.Items.Add("Rembrandt");
lstArtists.Items.Add("Manet");
lstArtists.Items.Insert(0, "Botticelli"); //Place at top
```

Core Note

To prevent a `ListBox` from repainting itself each time an item is added, execute the `ListBox.BeginUpdate` method prior to adding and `ListBox.EndUpdate` after the last item is added.

List boxes may also contain objects. Because an object may have many members, this raises the question of what is displayed in the `TextBox` list. Because by default a `ListBox` displays the results of an item's `ToString` method, it is necessary to override this `System.Object` method to return the string you want displayed. The following class is used to create `ListBox` items:

```
// Instances of this class will be placed in a ListBox
public class Artist
{
   public string BDate, DDate, Country;
   private string firstname;
   private string lastname;
   public Artist(string birth, string death, string fname,
                 string lname, string ctry)
   {
      BDate = birth;
```

```
      DDate = death;
      Country = ctry;
      firstname = fname;
      lastname = lname;
   }
   public override string ToString() {
      return (lastname+" , "+firstname);
   }
   public string GetLName    {
      get{ return lastname;}
   }
   public string GetFName    {
      get{ return firstname;}
   }
}
```

`ToString` has been overridden to return the artist's last and first names, which are displayed in the `ListBox`. The `ListBox` (Figure 7-9) is populated using these statements:

```
lstArtists.Items.Add
   (new Artist("1832", "1883", "Edouard", "Manet","Fr" ));
lstArtists.Items.Add
   (new Artist("1840", "1926", "Claude", "Monet","Fr"));
lstArtists.Items.Add
   (new Artist("1606", "1669", "Von Rijn", "Rembrandt","Ne"));
lstArtists.Items.Add
   (new Artist("1445", "1510", "Sandre", "Botticelli","It"));
```

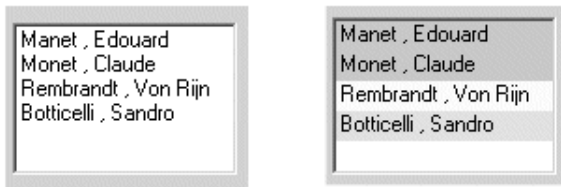


Figure 7-9 `ListBox` items: (A) Default and (B) Custom drawn

Selecting and Searching for Items in a ListBox

The `SelectionMode` property determines the number of items a `ListBox` allows to be selected at one time. It takes four values from the `SelectionMode` enumeration: `None`, `Single`, `MultiSingle`, and `MultiExtended`. `MultiSingle` allows selection by clicking an item or pressing the space bar; `MultiExtended` permits the use of the Shift and Ctrl keys.

The `SelectedIndexChanged` event provides an easy way to detect when an item in a `ListBox` is selected. It is fired when the user clicks on an item or uses the arrow keys to traverse a list. A common use is to display further information about the selection in other controls on the form. Here is code that displays an artist's dates of birth and death when the artist's name is selected from the `ListBox` in Figure 7-9:

```
// Set up event handler in constructor
lstArtists.SelectedIndexChanged += new EventHandler(ShowArtist);
//
private void ShowArtist(object sender, EventArgs e)
{
   // Cast to artist object in order to access properties
   Artist myArtist = lstArtists.SelectedItem as Artist;
   if (myArtist != null) {
      txtBirth.Text = myArtist.Dob; // Place dates in text boxes
      txtDeath.Text = myArtist.Dod;
   }
}
```

The `SelectedItem` property returns the item selected in the `ListBox`. This object is assigned to `myArtist` using the `as` operator, which ensures the object is an `Artist` type. The `SelectedIndex` property can also be used to reference the selected item:

```
myArtist = lstArtists.Items[lstArtists.SelectedIndex] as Artist;
```

Working with a multi-selection `ListBox` requires a different approach. You typically do not want to respond to a selection event until all items have been selected. One approach is to have the user click a button to signal that all choices have been made and the next action is required. All selections are exposed as part of the `SelectedItems` collection, so it is an easy matter to enumerate the items:

```
foreach (Artist a in lstArtists.SelectedItems)
   MessageBox.Show(a.GetLName);
```

The `SetSelected` method provides a way to programatically select an item or items in a `ListBox`. It highlights the item(s) and fires the `SelectedIndexChanged` event. In this example, `SetSelected` is used to highlight all artists who were born in France:

```
lstArtists.ClearSelected();   // Clear selected items
for (int ndx =0; ndx < lstArtists.Items.Count-1; ndx ++)
{
   Artist a = lstArtists.Items[ndx] as Artist;
   if (a.country == "Fr") lstArtists.SetSelected(ndx,true);
}
```

Customizing the Appearance of a ListBox

The `ListBox`, along with the `ComboBox`, `MenuItem`, and `TabControl` controls, is an *owner-drawn* control. This means that by setting a control property, you can have it fire an event when the control's contents need to be drawn. A custom event handler takes care of the actual drawing.

To enable owner drawing of the `ListBox`, the `DrawMode` property must be set to one of two `DrawMode` enumeration values: `OwnerDrawFixed` or `OwnerDrawVariable`. The former draws each item a fixed size; the latter permits variable-sized items. Both of these cause the `DrawItem` event to be fired and rely on its event handler to perform the drawing.

Using the `ListBox` from the previous example, we can use the constructor to set `DrawMode` and register an event handler for the `DrawItem` event:

```
lstArtists.DrawMode = DrawMode.OwnerDrawFixed;
lstArtists.ItemHeight = 16;  // Height (pixels) of item
lstArtists.DrawItem += new DrawItemEventHandler(DrawList);
```

The `DrawItemEventHandler` delegate has two parameters: the familiar `sender` object and the `DrawItemEventArgs` object. The latter is of more interest. It contains properties related to the control's appearance and state as well as a couple of useful drawing methods. Table 7-2 summarizes these.

Table 7-2 `DrawItemEventArgs` Properties

Member	Description
`BackColor`	Background color assigned to the control.
`Bounds`	Defines the coordinates of the item to be drawn as a `Rectangle` object.
`Font`	Returns the font assigned to the item being drawn.
`ForeColor`	Foreground color of the control. This is the color of the text displayed.
`Graphics`	Represents the surface (as a `Graphics` object) on which the drawing occurs.

Table 7-2 `DrawItemEventArgs` Properties *(continued)*

Member	Description
`Index`	The index in the control where the item is being drawn.
`State`	The state of the item being drawn. This value is a `DrawItemState` enumeration. For a `ListBox`, its value is `Selected (1)` or `None(0)`.
`DrawBackground()`	Draws the default background.
`DrawFocusRectangle()`	Draws the focus rectangle around the item if it has focus.

`Index` is used to locate the item. `Font`, `BackColor`, and `ForeColor` return the current preferences for each. `Bounds` defines the rectangular area circumscribing the item and is used to indicate where drawing should occur. `State` is useful for making drawing decisions based on whether the item is selected. This is particularly useful when the `ListBox` supports multiple selections. We looked at the `Graphics` object briefly in the last chapter when demonstrating how to draw on a form. Here, it is used to draw in the `Bounds` area. Finally, the two methods, `DrawBackground` and `DrawFocusRectangle`, are used as their name implies.

The event handler to draw items in the `ListBox` is shown in Listing 7-3. Its behavior is determined by the operation being performed: If an item has been selected, a black border is drawn in the background to highlight the selection; if an item is added, the background is filled with a color corresponding to the artist's country, and the first and last names of the artist are displayed.

The routine does require knowledge of some GDI+ concepts (see Chapter 8, ".NET Graphics Using GDI+"). However, the purpose of the methods should be clear from their name and context: `FillRectangle` fills a rectangular area defined by the `Rectangle` object, and `DrawString` draws text to the `Graphics` object using a font color defined by the `Brush` object. Figure 7-9(B) shows the output.

Listing 7-3 Event Handler to Draw Items in a `ListBox`

```
private void DrawList(object sender, DrawItemEventArgs e)
{
   // Draw ListBox Items
   string ctry;
   Rectangle rect = e.Bounds;
   Artist a = lstArtists.Items[e.Index] as Artist;
   string artistName = a.ToString();
   if ( (e.State & DrawItemState.Selected) ==
                   DrawItemState.Selected )
   {
```

Listing 7-3 Event Handler to Draw Items in a `ListBox` *(continued)*

```
        // Draw Black border around the selected item
        e.Graphics.DrawRectangle(Pens.Black,rect);
    } else {
        ctry = a.Country;
        Brush b;   // Object used to define backcolor
        // Each country will have a different backcolor
        b = Brushes.LightYellow;      // Netherlands
        if (ctry == "Fr")  b = Brushes.LightGreen;
        if (ctry == "It")  b = Brushes.Yellow;
        e.Graphics.FillRectangle(b,rect);}
        e.Graphics.DrawString(artistName,e.Font,
                              Brushes.Black,rect);
    }
}
```

Other List Controls: the ComboBox and the CheckedListBox

The `ComboBox` control is a hybrid control combining a `ListBox` with a `TextBox` (see Figure 7-10). Like the `ListBox`, it derives from the `ListControl` and thus possesses most of the same properties.

Figure 7-10 `ComboBox` and `CheckedListBox` controls are variations on `ListBox`

Visually, the `ComboBox` control consists of a text box whose contents are available through its `Text` property and a drop-down list from which a selected item is available through the `SelectedItem` property. When an item is selected, its textual representation is displayed in the text box window. A `ComboBox` can be useful in constructing questionnaires where the user selects an item from the drop-down list or, optionally, types in his own answer. Its construction is similar to the `ListBox`:

```
ComboBox cbArtists = new ComboBox();
cbArtists.Size = new System.Drawing.Size(120, 21);
cbArtists.MaxDropDownItems= 4;  // Max number of items to display
cbArtists.DropDownWidth = 140;  // Width of drop-down portion
cbArtists.Items.Add(new Artist("1832", "1883",
                                "Edouard", "Manet","Fr" ));
// Add other items here...
```

The `CheckedListBox` is a variation on the `ListBox` control that adds a check box to each item in the list. The default behavior of the control is to select an item on the first click, and check or uncheck it on the second click. To toggle the check on and off with a single click, set the `CheckOnClick` property to `true`.

Although it does not support multiple selections, the `CheckedListBox` does allow multiple items to be checked and includes them in a `CheckedItems` collection. The code here loops through a collection of `Artist` objects that have been checked on the control:

```
// List all items with checked box.
foreach (Artist a in clBox.CheckedItems)
   MessageBox.Show(a.ToString()); //   -> Monet, Claude
```

You can also iterate through the collection and explicitly determine the checked state:

```
For (int i=0; I< clBox.Items.Count; i++)
{
   if(clBox.GetItemCheckState(i) == CheckState.Checked)
   { Do something }  else {do something if not checked }
}
```

7.5 The ListView and TreeView Classes

The ListView Class

`ListView` is another control that displays lists of information. It represents data relationally as items and subitems. The data can be represented in a variety of formats that include a multi-column grid and large or small icons to represent item data. Also, images and check boxes can adorn the control.

Figure 7-11 illustrates the basic properties and methods used to lay out a Details view of the control—a format obviously tailored to displaying database tables. The first column contains text for an *item*—as well as a picture—the remaining columns contain *subitems* for the parent item.

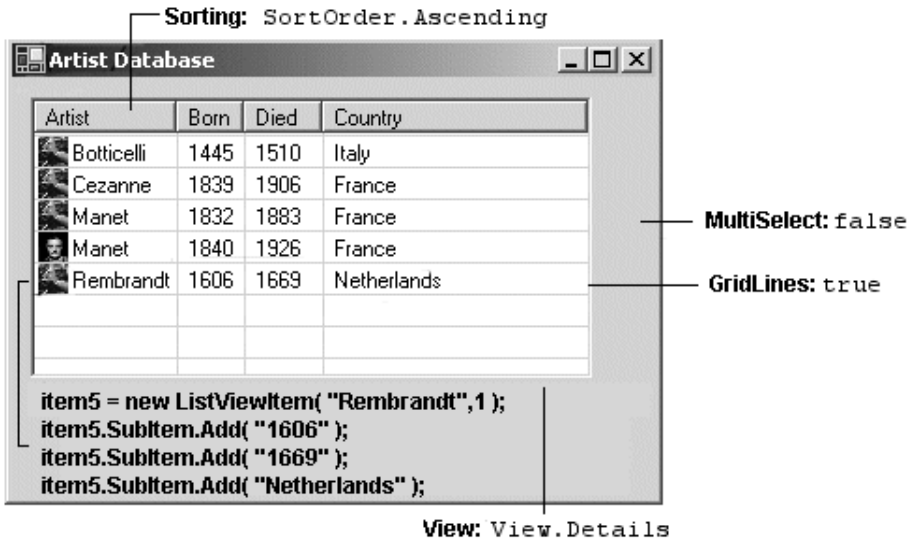


Figure 7-11 `ListView` control

Let's look at how this style of the `ListView` is constructed.

Creating a ListView Object

The `ListView` is created with a parameterless constructor:

```
ListView listView1 = new ListView();
```

Define Appearance of ListView Object

```
// Set the view to show details
listView1.View = View.Details;
```

The `View` property specifies one of five layouts for the control:

- `Details`. An icon and item's text are displayed in column one. Sub-items are displayed in the remaining columns.
- `LargeIcon`. A large icon is shown for each item with a label below the icon.
- `List`. Each item is displayed as a small icon with a label to its right. The icons are arranged in columns across the control.
- `SmallIcon`. Each item appears in a single column as a small icon with a label to its right.
- `*Tile`. Each item appears as a full-size icon with the label and sub-item details to the right of it. Only available for Windows XP and 2003.

Core Note

The `ListView.View` property can be changed at runtime to switch among the possible views. In fact, you may recognize that the view options correspond exactly to the View menu options available in Windows Explorer.

After the `Details` view is selected, other properties that define the control's appearance and behavior are set:

```
// Allow the user to rearrange columns
listView1.AllowColumnReorder = true;
// Select the entire row when selection is made
listView1.FullRowSelect = true;
// Display grid lines
listView1.GridLines = true;
// Sort the items in the list in ascending order
listView1.Sorting = SortOrder.Ascending;
```

These properties automatically sort the items, permit the user to drag columns around to rearrange their order, and cause a whole row to be highlighted when the user selects an item.

Set Column Headers

In a `Details` view, data is not displayed until at least one column is added to the control. Add columns using the `Columns.Add` method. Its simplest form is

```
ListView.Columns.Add(caption, width, textAlign)
```

`Caption` is the text to be displayed. `Width` specifies the column's width in pixels. It is set to –1 to size automatically to the largest item in the column, or –2 to size to the width of the header.

```
// Create column headers for the items and subitems
   listView1.Columns.Add("Artist", -2, HorizontalAlignment.Left);
   listView1.Columns.Add("Born", -2, HorizontalAlignment.Left);
   listView1.Columns.Add("Died", -2, HorizontalAlignment.Left);
   listView1.Columns.Add("Country", -2, HorizontalAlignment.Left);
```

The `Add` method creates and adds a `ColumnHeader` type to the `ListView`'s `Columns` collection. The method also has an overload that adds a `ColumnHeader` object directly:

```
ColumnHeader cHeader:
cHeader.Text = "Artist";
cHeader.Width = -2;
cHeader.TextAlign = HorizontalAlignment.Left;
ListView.Columns.Add(ColumnHeader cHeader);
```

Create ListView Items

Several overloaded forms of the `ListView` constructor are available. They can be used to create a single item or a single item and its subitems. There are also options to specify the icon associated with the item and set the foreground and background colors.

Constructors:

```
public ListViewItem(string text);
public ListViewItem(string[] items );
public ListViewItem(string text,int imageIndex );
public ListViewItem(string[] items,int imageIndex );
public ListViewItem(string[] items,int imageIndex,
         Color foreColor,Color backColor,Font font);
```

The following code demonstrates how different overloads can be used to create the items and subitems shown earlier in Figure 7-8:

```
// Create item and three subitems
ListViewItem item1 = new ListViewItem("Manet",2);
item1.SubItems.Add("1832");
item1.SubItems.Add("1883");
item1.SubItems.Add("France");
// Create item and subitems using a constructor only
ListViewItem item2 = new ListViewItem
       (new string[] {"Monet","1840","1926","France"}, 3);
// Create item and subitems with blue background color
ListViewItem item3 = new ListViewItem
       (new string[] {"Cezanne","1839","1906","France"}, 1,
       Color.Empty, Color.LightBlue, null);
```

To display the items, add them to the `Items` collection of the `ListView` control:

```
// Add the items to the ListView
   listView1.Items.AddRange(
      new ListViewItem[]{item1,item2,item3,item4,item5});
```

Specifying Icons

Two collections of images can be associated with a `ListView` control as `ImageList` properties: `LargeImageList`, which contains images used in the `LargeIcon` view;

and SmallImageList, which contains images used in all other views. Think of these as zero-based arrays of images that are associated with a ListViewItem by the imageIndex parameter in the ListViewItem constructor. Even though they are referred to as icons, the images may be of any standard graphics format.

The following code creates two ImageList objects, adds images to them, and assigns them to the LargeImageList and SmallImageList properties:

```
// Create two ImageList objects
   ImageList imageListSmall = new ImageList();
   ImageList imageListLarge = new ImageList();
   imageListLarge.ImageSize = new Size(50,50); // Set image size
   // Initialize the ImageList objects
   // Can use same images in both collections since they're resized
   imageListSmall.Images.Add(Bitmap.FromFile("C:\\botti.gif"));
   imageListSmall.Images.Add(Bitmap.FromFile("C:\\cezanne.gif"));
   imageListLarge.Images.Add(Bitmap.FromFile("C:\\botti.gif"));
   imageListLarge.Images.Add(Bitmap.FromFile("C:\\cezanne.gif"));
   // Add other images here
   // Assign the ImageList objects to the ListView.
   listView1.LargeImageList = imageListLarge;
   listView1.SmallImageList = imageListSmall;
   ListViewItem lvItem1 = new ListViewItem("Cezanne",1);
```

An index of 1 selects the cezanne.gif images as the large and small icons. Specifying an index not in the ImageList results in the icon at index 0 being displayed. If neither ImageList is defined, no icon is displayed. Figure 7-12 shows the ListView from Figure 7-11 with its view set to View.LargeIcon:

```
listView1.View = View.LargeIcon;
```

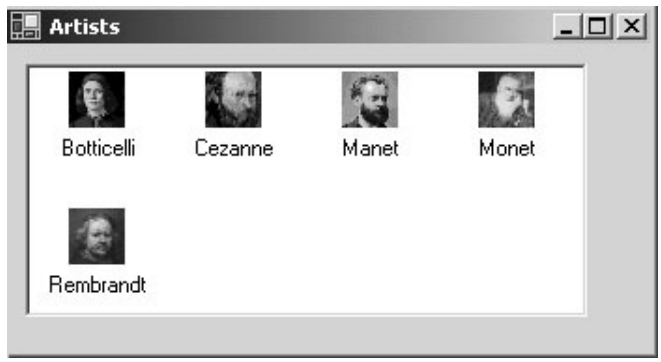


Figure 7-12 LargeIcon view

Working with the ListView Control

Common tasks associated with the `ListView` control include iterating over the contents of the control, iterating over selected items only, detecting the item that has focus, and—when in `Details` view—sorting the items by any column. Following are some code segments to perform these tasks.

Iterating over All Items or Selected Items

You can use `foreach` to create nested loops that select an item and then iterate through the collection of subitems for the item in the outside loop:

```
foreach (ListViewItem lvi in listView1.Items)
{
   string row = "";
   foreach(ListViewItem.ListViewSubItem sub in lvi.SubItems)
   {
      row += " " + sub.Text;
   }
   MessageBox.Show(row); // List concatenated subitems
}
```

There are a couple of things to be aware of when working with these collections. First, the first subitem (index 0) element actually contains the text for the item—not a subitem. Second, the ordering of subitems is not affected by rearranging columns in the `ListView` control. This changes the appearance but does not affect the underlying ordering of subitems.

The same logic is used to list only selected items (`MultiSelect = true` permits multiple items to be selected). The only difference is that the iteration occurs over the `ListView.SelectedItems` collection:

```
foreach (ListViewItem lvisel in listView1.SelectedItems)
```

Detecting the Currently Selected Item

In addition to the basic control events such as `Click` and `DoubleClick`, the `ListView` control adds a `SelectedIndexChanged` event to indicate when focus is shifted from one item to another. The following code implements an event handler that uses the `FocusedItem` property to identify the current item:

```
// Set this in the constructor
listView1.SelectedIndexChanged +=
      new EventHandler(lv_IndexChanged);
// Handle SelectedIndexChanged Event
private void lv_IndexChanged(object sender, System.EventArgs e)
```

```
{
   string ItemText = listView1.FocusedItem.Text;
}
```

Note that this code can also be used with the `Click` events because they also use the `EventHandler` delegate. The `MouseDown` and `MouseUp` events can also be used to detect the current item. Here is a sample `MouseDown` event handler:

```
private void listView1_MouseDown(object sender, MouseEventArgs e)
{
   ListViewItem selection = listView1.GetItemAt(e.X, e.Y);
   if (selection != null)
   {
      MessageBox.Show("Item Selected: "+selection.Text);
   }
}
```

The `ListView.GetItemAt` method returns an item at the coordinates where the mouse button is pressed. If the mouse is not over an item, `null` is returned.

Sorting Items on a ListView Control

Sorting items in a `ListView` control by column values is a surprisingly simple feature to implement. The secret to its simplicity is the `ListViewItemSorter` property that specifies the object to sort the items anytime the `ListView.Sort` method is called. Implementation requires three steps:

1. Set up a delegate to connect a `ColumnClick` event with an event handler.
2. Create an event handler method that sets the `ListViewItemSorter` property to an instance of the class that performs the sorting comparison.
3. Create a class to compare column values. It must inherit the `IComparer` interface and implement the `IComparer.Compare` method.

The following code implements the logic: When a column is clicked, the event handler creates an instance of the `ListViewItemComparer` class by passing it the column that was clicked. This object is assigned to the `ListViewItemSorter` property, which causes sorting to occur. Sorting with the `IComparer` interface is discussed in Chapter 4, "Working with Objects in C#").

```
// Connect the ColumnClick event to its event handler
listView1.ColumnClick +=new ColumnClickEventHandler(ColumnClick);
// ColumnClick event handler
private void ColumnClick(object o, ColumnClickEventArgs e)
```

```
{
   // Setting this property immediately sorts the
   // ListView using the ListViewItemComparer object
   this.listView1.ListViewItemSorter =
       new ListViewItemComparer(e.Column);
}
// Class to implement the sorting of items by columns
class ListViewItemComparer : IComparer
{
   private int col;
   public ListViewItemComparer()
   {
      col = 0;   // Use as default column
   }
   public ListViewItemComparer(int column)
      {
      col = column;
   }
   // Implement IComparer.Compare method
   public int Compare(object x, object y)
   {
      string xText = ((ListViewItem)x).SubItems[col].Text;
      string yText = ((ListViewItem)y).SubItems[col].Text;
      return String.Compare(xText, yText);
   }
}
```

The TreeView Class

As the name implies, the `TreeView` control provides a tree-like view of hierarchical data as its user interface. Underneath, its programming model is based on the familiar tree structure consisting of parent nodes and child nodes. Each node is implemented as a `TreeNode` object that can in turn have its own `Nodes` collection. Figure 7-13 shows a `TreeView` control that is used in conjunction with a `ListView` to display `enum` members of a selected assembly. (We'll look at the application that creates it shortly.)

The TreeNode Class

Each item in a tree is represented by an instance of the `TreeNode` class. Data is associated with each node using the `TreeNode`'s `Text`, `Tag`, or `ImageIndex` properties. The `Text` property holds the node's label that is displayed in the `TreeView` control. `Tag` is an object type, which means that any type of data can be associated with the node by assigning a custom class object to it. `ImageIndex` is an index to an `ImageList` associated with the containing `TreeView` control. It specifies the image to be displayed next to the node.

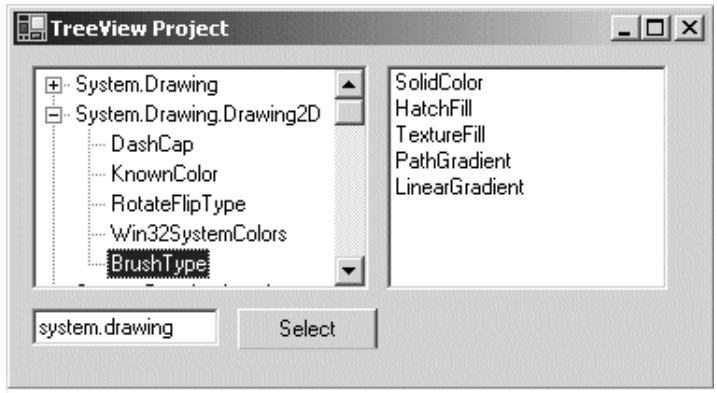


Figure 7-13 Using `TreeView` control (left) and `ListView` (right) to list enum values

In addition to these basic properties, the `TreeNode` class provides numerous other members that are used to add and remove nodes, modify a node's appearance, and navigate the collection of nodes in a node tree (see Table 7-3).

Table 7-3 Selected Members of the `TreeNode` Class

Use	Member	Description
Appearance	`BackColor`, `ForeColor`	Sets the background color and text color of the node.
	`Expand()`, `Collapse()`	Expands the node to display child nodes or collapses the tree so no child nodes are shown.
Navigation	`FirstNode`, `LastNode`, `NextNode`, `PrevNode`	Returns the first or last node in the collection. Returns the next or previous node (sibling) relative to the current node.
	`Index`	The index of the current node in the collection.
	`Parent`	Returns the current node's parent.
Node Manipulation	`Nodes.Add()`, `Nodes.Remove()`, `Nodes.Insert()`, `Nodes.Clear()`	Adds or removes a node to a `Nodes` collection. `Insert` adds a node at an indexed location, and `Clear` removes all tree nodes from the collection.
	`Clone()`	Copies a tree node and entire subtree.

Let's look at how `TreeView` and `TreeNode` members are used to perform fundamental `TreeView` operations.

Adding and Removing Nodes

The following code creates the tree in Figure 7-14 using a combination of `Add`, `Insert`, and `Clone` methods. The methods are performed on a preexisting `treeView1` control.

```
TreeNode tNode;
// Add parent node to treeView1 control
tNode = treeView1.Nodes.Add("A");
// Add child node: two overloads available
tNode.Nodes.Add(new TreeNode("C"));
tNode.Nodes.Add("D"));
// Insert node after C
tNode.Nodes.Insert(1,new TreeNode("E"));
// Add parent node to treeView1 control
tNode = treeView1.Nodes.Add("B");
```

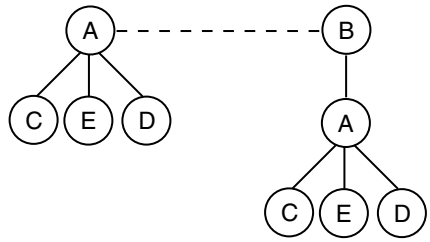


Figure 7-14 `TreeView` node representation

At this point, we still need to add a copy of node A and its subtree to the parent node B. This is done by cloning the A subtree and adding it to node B. Node A is referenced as `treeView1.Nodes[0]` because it is the first node in the control's collection. Note that the `Add` method appends nodes to a collection, and they can be referenced by their zero-based position within the collection:

```
// Clone first parent node and add to node B
TreeNode clNode = (TreeNode) treeView1.Nodes[0].Clone();
tNode.Nodes.Add(clNode);
// Add and remove node for demonstration purposes
tNode.Nodes.Add("G");
tNode.Nodes.Remove(tNode.LastNode);
```

Iterating Through the Nodes in a TreeView

As with any collection, the `foreach` statement provides the easiest way to loop through the collection's members. The following statements display all the top-level nodes in a control:

```
foreach (TreeNode tn in treeView1.Nodes)
{
   MessageBox.Show(tn.Text);
   // If (tn.IsVisible) true if node is visible
   // If (tn.IsSelected) true if node is currently selected
}
```

An alternate approach is to move through the collection using the `TreeNode.NextNode` property:

```
tNode = treeView1.Nodes[0];
while (tNode != null) {
   MessageBox.Show(tNode.Text);
   tNode = tNode.NextNode;
}
```

Detecting a Selected Node

When a node is selected, the `TreeView` control fires an `AfterSelect` event that passes a `TreeViewEventArgs` parameter to the event handling code. This parameter identifies the action causing the selection and the node selected. The `TreeView` example that follows illustrates how to handle this event.

You can also handle the `MouseDown` event and detect the node using the `GetNodeAt` method that returns the node—if any—at the current mouse coordinates.

```
private void treeView1_MouseDown(object sender, MouseEventArgs e)
{
   TreeNode tn = treeView1.GetNodeAt(e.X, e.Y);
   // You might want to remove the node:  tn.Remove()
}
```

A TreeView Example That Uses Reflection

This example demonstrates how to create a simple object browser (refer to Figure 7-13) that uses a `TreeView` to display enumeration types for a specified assembly. When a node on the tree is clicked, the members for the selected enumeration are displayed in a `ListView` control.

Information about an assembly is stored in its metadata, and .NET provides classes in the `System.Reflection` namespace for exposing this metadata. The code in Listing 7-4 iterates across the types in an assembly to build the `TreeView`. The

parent nodes consist of unique namespace names, and the child nodes are the types contained in the namespaces. To include only `enum` types, a check is made to ensure that the type inherits from `System.Enum`.

Listing 7-4 Using a `TreeView` and Reflection to List Enums in an Assembly

```
using System.Reflection;
//
private void GetEnums()
{
   TreeNode tNode=null;
   Assembly refAssembly ;
   Hashtable ht= new Hashtable();  // Keep track of namespaces
   string assem = AssemName.Text;  // Textbox with assembly name
   tvEnum.Nodes.Clear();           // Remove all nodes from tree
   // Load assembly to be probed
   refAssembly = Assembly.Load(assem);
   foreach (Type t in refAssembly.GetTypes())
   {
      // Get only types that inherit from System.Enum
      if(t.BaseType!=null && t.BaseType.FullName=="System.Enum")
      {
         string myEnum  = t.FullName;
         string  nSpace =
                   myEnum.Substring(0,myEnum.LastIndexOf("."));
         myEnum= myEnum.Substring(myEnum.LastIndexOf(".")+1) ;
         // Determine if namespace in hashtable
         if( ht.Contains(nSpace))
         {
            // Find parent node representing this namespace
            foreach (TreeNode tp in tvEnum.Nodes)
            {
              if(tp.Text == myEnum) { tNode=tp; break;}
            }
         }
         else
         {
            // Add parent node to display namespace
            tNode = tvEnum.Nodes.Add(nSpace);
            ht.Add(nSpace,nSpace);
         }
```

Listing 7-4 Using a `TreeView` and Reflection to List Enums in an Assembly *(continued)*

```
            // Add Child - name of enumeration
            TreeNode cNode = new TreeNode();
            cNode.Text= myEnum;
            cNode.Tag = t;     // Contains specific enumeration
            tNode.Nodes.Add(cNode);
        }
    }
}
```

Notice how reflection is used. The static `Assembly.Load` method is used to create an `Assembly` type. The `Assembly.GetTypes` is then used to return a `Type` array containing all types in the designated assembly.

```
refAssembly = Assembly.Load(assem);
foreach (Type t in refAssembly.GetTypes())
```

The `Type.FullName` property returns the name of the type, which includes the namespace. This is used to extract the `enum` name and the namespace name. The `Type` is stored in the `Tag` field of the child nodes and is used later to retrieve the members of the enum.

After the `TreeView` is built, the final task is to display the field members of an enumeration when its node is clicked. This requires registering an event handler to be notified when an `AfterSelect` event occurs:

```
tvEnum.AfterSelect += new
   TreeViewEventHandler(tvEnum_AfterSelect);
```

The event handler identifies the selected node from the `TreeViewEventArgs.Node` property. It casts the node's `Tag` field to a `Type` class (an `enumerator` in this case) and uses the `GetMembers` method to retrieve the type's members as `MemberInfo` types. The name of each field member—exposed by the `MemberInfo.Name` property—is displayed in the `ListView`:

```
// ListView lView;
// lView.View = View.List;
private void tvEnum_AfterSelect(Object sender,
                                TreeViewEventArgs e)
{
   TreeNode tn = e.Node;   // Node selected
   ListViewItem lvItem;
```

```
   if(tn.Parent !=null)    // Exclude parent nodes
   {
      lView.Items.Clear(); // Clear ListView before adding items
      Type cNode = (Type) tn.Tag;
      // Use Reflection to iterate members in a Type
      foreach (MemberInfo mi in cNode.GetMembers())
      {
         if(mi.MemberType==MemberTypes.Field &&
                           mi.Name != "value__" )  // skip this
         {
            lView.Items.Add(mi.Name);
         }
      }
   }
}
```

7.6 The ProgressBar, Timer, and StatusStrip Classes

The `ProgressBar` and `Timer` are lightweight controls that have complementary roles in an application: The `Timer` initiates action and the `ProgressBar` reflects the status of an operation or action. In fact, the `Timer` is not a control, but a component that inherits from the `ComponentModel.Component` class. It is used most often in processes to regulate some background activity. This may be a periodic update to a log file or a scheduled backup of data. A `ProgressBar`, on the other hand, provides visual feedback regarding the progress of an operation—such as file copying or steps in an installation.

The third class discussed in this section is the `StatusStrip`, which is often used in conjunction with a timer and `ProgressBar`. It's rendered on a form as a strip divided into one or more sections or panes that provide status information. Each section is implemented as a control that is added to the `StatusStrip` container. For a control to be included in the `StatusStrip`, it must inherit from the `ToolStripItem` class.

Building a StatusStrip

Let's now build a form that includes a multi-pane `StatusStrip`. As shown in Figure 7-15, the strip consists of a label, progress bar, and panel controls. The label (`ToolStripLabel`) provides textual information describing the overall status of the application. The progress bar is implemented as a `ToolStripProgressBar` object. It is functionally equivalent to a `ProgressBar`, but inherits from `ToolStripItem`. A

`StatusStripPanel` shows the elapsed time since the form was launched. An event handler that is triggered by a timer updates both the progress bar and clock panel every five seconds.

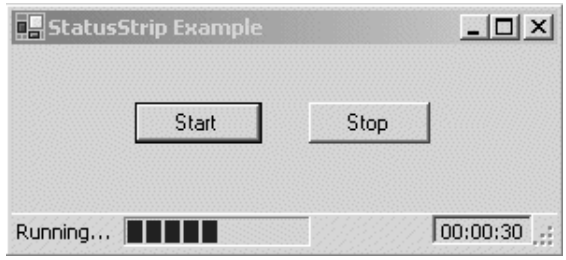


Figure 7-15 `StatusStrip` with `Label`, `ProgressBar`, and `Panel`

Listing 7-5 contains the code to create the `StatusStrip`. The left and right ends of the progress bar are set to represent the values 0 and 120, respectively. The bar is set to increase in a step size of 10 units each time the `PerformStep` method is executed. It recycles every minute.

The `Timer` controls when the bar is incremented and when the elapsed time is updated. Its `Interval` property is set to a value that controls how frequently its `Tick` event is fired. In this example, the event is fired every 5 seconds, which results in the progress bar being incremented by 10 units and the elapsed time by 5 seconds.

Listing 7-5 `StatusStrip` That Uses a `ProgressBar` and `Timer`

```
// These variables have class scope
Timer currTimer;
StatusStrip statusStrip1;
StatusStripPanel panel1;
ToolStripProgressBar pb;
DateTime startDate = DateTime.Now;

private void BuildStrip()
{
   currTimer = new Timer();
   currTimer.Enabled = true;
   currTimer.Interval = 5000; // Fire tick event every 5 seconds
   currTimer.Tick += new EventHandler(timer_Tick);
   // Panel to contain elapsed time
```

Listing 7-5 `StatusStrip` That Uses a `ProgressBar` and `Timer` *(continued)*

```
    panel1 = new StatusStripPanel();
    panel1.BorderStyle = Border3DStyle.Sunken;
    panel1.Text = "00:00:00";
    panel1.Padding = new Padding(2);
    panel1.Name = "clock";
    panel1.Alignment = ToolStripItemAlignment.Tail; //Right align
    // Label to display application status
    ToolStripLabel ts = new ToolStripLabel();
    ts.Text = "Running...";
    // ProgressBar to show time elapsing
    pb = new ToolStripProgressBar();
    pb.Step    = 10;      // Size of each step or increment
    pb.Minimum = 0;
    pb.Maximum = 120;     // Allow 12 steps
    // Status strip to contain components
    statusStrip1 = new StatusStrip();
    statusStrip1.Height = 20;
    statusStrip1.AutoSize = true;
    // Add components to strip
    statusStrip1.Items.AddRange(new ToolStripItem[] {
       ts, pb, panel1 } );
    this.Controls.Add(statusStrip1);
}
private void timer_Tick(object sender, EventArgs e)
{
    // Get difference between current datetime
    // and form startup time
    TimeSpan ts = DateTime.Now.Subtract(startDate);
    string elapsed = ts.Hours.ToString("00") + ":" +
          ts.Minutes.ToString("00") +
          ":" + ts.Seconds.ToString("00");
    ((StatusStripPanel)statusStrip1.Items[
          "clock"]).Text= elapsed;
    // Advance progress bar
    if (pb.Value == pb.Maximum) pb.Value = 0;
    pb.PerformStep();   // Increment progress bar
}
```

The `StatusStripPanel` that displays the elapsed time has several properties that control its appearance and location. In addition to those shown here, it has an `Image` property that allows it to display an image. The `StatusStripPanel` class

inherits from the `ToolStripLabel` class that is used in the first pane. Both can be used to display text, but the panel includes a `BorderStyle` property that `ToolStripLabel` lacks.

7.7 Building Custom Controls

At some point, you will face a programming task for which a standard WinForms control does not provide the functionality you need. For example, you may want to extend a `TextBox` control so that its background color changes according to its content, group a frequently used set of radio buttons into a single control, or create a new control that shows a digital clock face with the date underneath. These needs correspond to the three principal types of custom controls:

1. A control that derives from an existing control and extends its functionality.
2. A control that can serve as container to allow multiple controls to interact. This type of control is referred to as a *user control*. It derives directly from `System.Windows.Forms.UserControl` rather than `Control`, as do standard controls.
3. A control that derives directly from the `Control` class. This type of control is built "from scratch," and it is the developer's responsibility to draw its GUI interface and implement the methods and properties that allow it to be manipulated by code.

Let's now look at how to extend an existing control and create a user control.

Extending a Control

The easiest way to create a custom control is to extend an existing one. To demonstrate this, let's derive a `TextBox` that accepts only digits. The code is quite simple. Create a new class `NumericTextBox` with `TextBox` as its base class. The only code required is an event handler to process the `KeyPress` event and accept only a digit.

```
class NumericTextBox: TextBox
{
   public NumericTextBox()
   {
      this.KeyPress += new KeyPressEventHandler(TextBoxKeyPress);
   }
   protected void TextBoxKeyPress(object sender,
                                  KeyPressEventArgs e)
```

```
    {
        if (! char.IsDigit(e.KeyChar)) e.Handled = true;
    }
}
```

After the extended control is compiled into a DLL file, it can be added to any form.

Building a Custom UserControl

Think of a user control as a subform. Like a form, it provides a container surface on which related widgets are placed. When compiled, the entire set of controls is treated as a single user control. Of course, users still can interact directly with any of the member controls. Programmatic and design-time access to control members is available through methods and properties defined on the user control.

The easiest way to design a control is with an IDE such as Visual Studio.NET (VS.NET), which makes it easy to position and size controls. The usual way to create a user control in VS.NET is to open a project as a *Windows Control Library* type. This immediately brings up a control designer window. The design window can also be accessed in a Windows Application by selecting Project – Add User Control from the top menu bar or right-clicking on the Solution Explorer and selecting Add – Add User Control. Although VS.NET can speed up the process of creating a control, it does not generate any proprietary code that cannot be duplicated using a text editor.

A UserControl Example

As an example, let's create a control that can be used to create a questionnaire. The control consists of a label whose value represents the question, and three radio buttons contained on a panel control that represent the user's choice of answers. The control exposes three properties: one that assigns the question to the label, one to set the background color of the panel control, and another that identifies the radio button associated with the user's answer.

Figure 7-16 shows the layout of the user control and the names assigned to each contained control.

Here is how the members are represented as fields within the `UserControl1` class:

```
public class UserControl1 : System.Windows.Forms.UserControl
{
   private Panel panel1;
   private RadioButton radAgree;
   private RadioButton radDisagree;
   private RadioButton radUn;
   private Label qLabel;
```

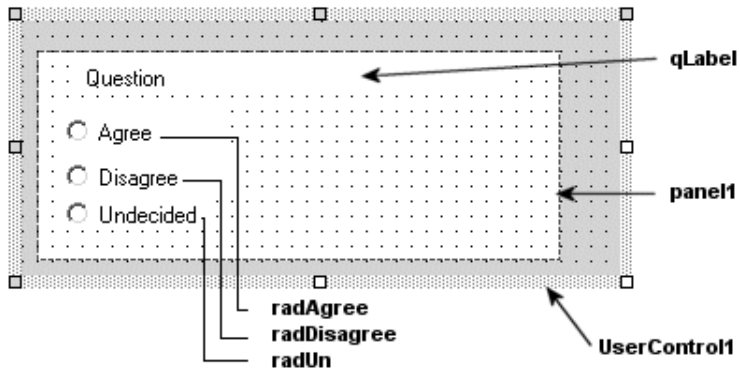


Figure 7-16 Layout of a custom user control

Listing 7-6 contains the code for three properties: `SetQ` that sets the label's text property to the question, `PanelColor` that sets the color of the panel, and `Choice`, which returns the answer selected by the user as a `Choices` enum type.

Listing 7-6 Implementing Properties for a Custom User Control

```
public enum Choices
{
   Agree      = 1,
   DisAgree   = 2,
   Undecided     = 3,
}
public string SetQ
{
   set {qLabel.Text = value;}
   get {return(qLabel.Text);}
}
public Color PanelColor
{
   set {panel1.BackColor= value;}
   get {return(panel1.BackColor);}
}
public Choices Choice
{
   get
   {
```

Listing 7-6 Implementing Properties for a Custom User Control *(continued)*

```
        Choices usel;
        usel = Choices.Undecided;
        if (radDisagree.Checked) usel= Choices.DisAgree;
        if (radAgree.Checked) usel = Choices.Agree;
        return(usel);}
    }
}
```

Using the Custom User Control

If the user control is developed as part of a VS.NET Windows Application project, it is automatically added to the tool box under the Windows Forms tab. Simply select it and drop it onto the form. Otherwise, you have to right-click on a tool box tab, select Customize ToolBox, browse for the control, and add it to the tool box.

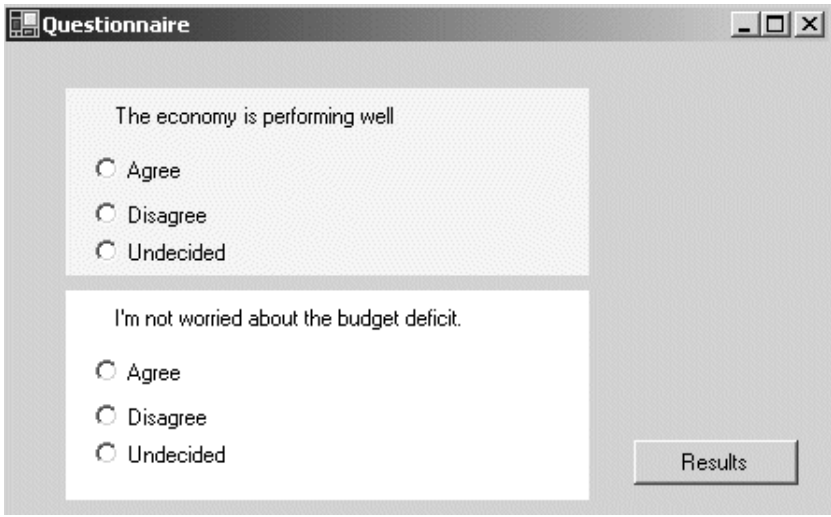


Figure 7-17 Custom user controls on a form

Figure 7-17 provides an example of using this new control. In this example, we place two control instances on the form and name them `Q1` and `Q2`:

```
private usercontrol.UserControl1 Q1;
private usercontrol.UserControl1 Q2;
```

The properties can be set in the constructor or at runtime in the `Form.Load` event handler. If using VS.NET, the properties can be set at design time using the Property Browser.

```
Q1.SetQ = "The economy is performing well";
Q2.SetQ = "I'm not worried about the budget deficit.";
Q1.PanelColor = Color.Beige;
```

The final step in the application is to do something with the results after the questionnaire has been completed. The following code iterates through the controls on the form when the button is clicked. When a `UserControl1` type is encountered, its `Choice` property is used to return the user's selection.

```
private void button1_Click(object sender, System.EventArgs e)
{
   foreach (Control ct in this.Controls)
   {
      if (ct is usercontrol.UserControl1)
      {
         UserControl1 uc = (UserControl1)ct;
         // Display control name and user's answer
         MessageBox.Show(ct.Name+" "+
              uc.Choice.ToString());
      }
   }
 }
```

Working with the User Control at Design Time

If you are developing an application with VS.NET that uses this custom control, you will find that the Property Browser lists all of the read/write properties. By default, they are placed in a Misc category and have no description associated with them. To add a professional touch to your control, you should create a category for the control's events and properties and add a textual description for each category member.

The categories and descriptions available in the Property Browser come from metadata based on attributes attached to a type's members. Here is an example of attributes added to the `PanelColor` property:

```
[Browsable(true),
Category("QControl"),
Description("Color of panel behind question block")]
public Color PanelColor
{
   set {panel1.BackColor = value;}
   get {return (panel1.BackColor);}
}
```

The `Browsable` attribute indicates whether the property is to be displayed in the browser. The default is `true`. The other two attributes specify the category under which the property is displayed and the text that appears below the Property Browser when the property is selected.

Always keep in mind that the motive for creating custom user controls is reusability. There is no point in spending time creating elaborate controls that are used only once. As this example illustrates, they are most effective when they solve a problem that occurs repeatedly.

7.8 Using Drag and Drop with Controls

The ability to drag data from one control and drop it onto another has long been a familiar feature of GUI programming. .NET supports this feature with several classes and enumerations that enable a control to be the target and/or source of the drag-and-drop operation.

Overview of Drag and Drop

The operation requires a source control that contains the data to be moved or copied, and a target control that receives the dragged data. The source initiates the action in response to an event—usually a `MouseDown` event. The source control's event handler begins the actual operation by invoking its `DoDragDrop` method. This method has two parameters: the data being dragged and a `DragDropEffects` enum type parameter that specifies the effects or actions the source control supports (see Table 7-4).

Table 7-4 `DragDropEffects` Enumeration

Member	Description
`All`	The data is moved to the target control, and scrolling occurs in the target control to display the newly positioned data.
`Copy`	Data is copied from target to source.
`Link`	Data from the source is linked to the target.
`Move`	The data is moved from the source to the target control.
`None`	The target control refuses to accept data.
`Scroll`	Scrolling occurs or will occur on the target control.

As the mouse moves across the form, the `DoDragDrop` method determines the control under the current cursor location. If this control has its `AllowDrop` property set to `true`, it is a valid drop target and its `DragEnter` event is raised. The `DragEnter` event handler has two tasks: to verify that the data being dragged is an acceptable type and to ensure the requested action (`Effect`) is acceptable. When the actual drop occurs, the destination control raises a `DragDrop` event. This event handler is responsible for placing the data in the target control (see Figure 7-18).

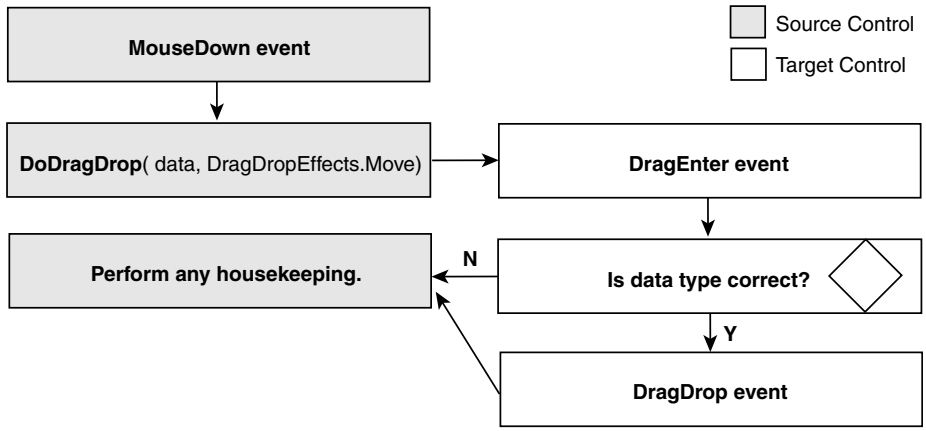


Figure 7-18 Sequence of events in drag-and-drop operation

After the `DragDrop` event handler finishes, the source control performs any cleanup operations. For example, if the operation involves moving data—as opposed to copying—the data must be removed from the source control.

To demonstrate these ideas, let's create an application that assigns players to a team from a roster of available players (see Figure 7-19). Team A is created by dragging names from the Available Players to the Team A list. Both lists are implemented with list boxes, and the Available Players list is set for single selection.

A name is selected by pressing the right mouse button and dragging the name to the target list. To add some interest, holding the Ctrl key copies a name rather than moving it.

After the form and controls are created, the first step is to set up the source control (`lstPlayers`) to respond to the `MouseDown` event and the target control (`lstTeamA`) to handle the `DragEnter` and `DragDrop` events:

```
lstPlayers.MouseDown +=
      new MouseEventHandler(Players_MouseDown);
lstTeamA.DragEnter   += new DragEventHandler(TeamA_DragEnter);
lstTeamA.DragDrop    += new DragEventHandler(TeamA_Drop);
```

The next step is to code the event handlers on the source and target control(s) that implement the drag-and-drop operation.

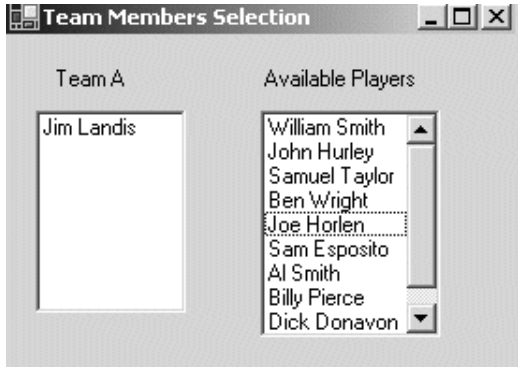


Figure 7-19 Drag-and-drop example

Source Control Responsibilities

The `MouseDown` event handler for the source `ListBox` first checks to ensure that an item has been selected. It then calls `DoDragDrop`, passing it the value of the selected item as well as the acceptable effects: `Move` and `Copy`. The `DragDropEffects` enumeration has a `FlagsAttribute` attribute, which means that any bitwise combination of its values can be passed. The value returned from this method is the effect that is actually used by the target. The event handler uses this information to perform any operations required to implement the effect. In this example, a move operation means that the dragged value must be removed from the source control.

Listing 7-7 Initiating a Drag-and-Drop Operation from the Source Control

```
private void Players_MouseDown(object sender, MouseEventArgs e)
{
   if ( lstPlayers.SelectedIndex >=0)
   {
      string players;
      int ndx = lstPlayers.SelectedIndex;
      DragDropEffects effect;
      players = lstPlayers.Items[ndx].ToString();
      if(players != "")
      {
```

Listing 7-7 Initiating a Drag-and-Drop Operation from the Source Control *(continued)*

```
            // Permit target to move or copy data
            effect = lstPlayers.DoDragDrop(players,
                  DragDropEffects.Move | DragDropEffects.Copy);
            // Remove item from ListBox since move occurred
            if (effect == DragDropEffects.Move)
            lstPlayers.Items.RemoveAt(ndx);
         }
      }
}
```

Target Control Responsibilities

The destination control must implement the event handlers for the `DragEnter` and `DragDrop` events. Both of these events receive a `DragEventArgs` type parameter (see Table 7-5) that contains the information required to process the drag-and-drop event.

Table 7-5 `DragEventArgs` Properties

Member	Description
`AllowedEffect`	The effects that are supported by the source control. Example to determine if Move is supported: `if ((e.AllowedEffect & DragDropEffects.Move) == DragDropEffects.Move)`
`Data`	Returns the `IDataObject` that contains data associated with this operation. This object implements methods that return information about the data. These include `GetData`, which fetches the data, and `GetDataPresent`, which checks the data type.
`Effect`	Gets or sets the target drop effect.
`KeyState`	Returns the state of the Alt key, Ctrl key, Shift key, and mouse buttons as an integer: 1—Left mouse button 8—Ctrl key 2—Right mouse button 16—Middle mouse button 4—Shift key 32—Alt key
`X, Y`	x and y coordinates of the mouse pointer.

The `Data`, `Effect`, and `KeyState` members are used as follows:

- `Data.GetDataPresent` is used by the `DragEnter` event handler to ensure that the data is a type the target control can process.
- The `DragDrop` event handler uses `Data.GetData` to access the data being dragged to it. The parameter to this method is usually a static field of the `DataFormats` class that specifies the format of the returned data.
- The `DragEnter` event handler uses `KeyState` to determine the status of the mouse and keys in order to determine the effect it will use to process the data. Recall that in this example, pressing the Ctrl key signals that data is to copied rather than moved.
- `Effect` is set by the `DragEnter` event handler to notify the source as to how—or if—it processed the data. A setting of `DragDropEffects.None` prevents the `DragDrop` event from firing.

Listing 7-8 shows the code for the two event handlers.

Listing 7-8 Handling the `DragEnter` and `DragDrop` Events

```
[FlagsAttribute]
enum KeyPushed
{
   // Corresponds to DragEventArgs.KeyState values
   LeftMouse   = 1,
   RightMouse  = 2,
   ShiftKey    = 4,
   CtrlKey     = 8,
   MiddleMouse = 16,
   AltKey      = 32,
}
private void TeamA_DragEnter(object sender, DragEventArgs e)
{
   KeyPushed kp = (KeyPushed) e.KeyState;
   // Make sure data type is string
   if (e.Data.GetDataPresent(typeof(string)))
   {
      // Only accept drag with left mouse key
      if ( (kp & KeyPushed.LeftMouse) == KeyPushed.LeftMouse)
      {
         if ((kp & KeyPushed.CtrlKey) == KeyPushed.CtrlKey)
         {
            e.Effect = DragDropEffects.Copy;    // Copy
```

Listing 7-8 Handling the `DragEnter` and `DragDrop` Events *(continued)*

```
            }
            else
            {
               e.Effect = DragDropEffects.Move;  // Move
            }
         }
         else  // Is not left mouse key
         {
            e.Effect = DragDropEffects.None;
         }
      } else    // Is not a string
      {
         e.Effect = DragDropEffects.None;
      }
   }
   // Handle DragDrop event
   private void TeamA_Drop(object sender, DragEventArgs e)
   {
      // Add dropped data to TextBox
      lstTeamA.Items.Add(
         (string) e.Data.GetData(DataFormats.Text));
   }
```

An `enum` is created with the `FlagsAttributes` attribute to make checking the `KeyState` value easier and more readable. The logical "anding" of `KeyState` with the value of the `CtrlKey` (8) returns a value equal to the value of the `CtrlKey` if the Ctrl key is pressed.

A control can serve as source and target in the same application. You could make this example more flexible by having the list boxes assume both roles. This would allow you to return a player from `lstTeamA` back to the `lstPlayers ListBox`. All that is required is to add the appropriate event handlers.

Core Note

Drag and drop is not just for text. The `DataFormats` class predefines the formats that can be accepted as `static` fields. These include `Bitmap`, `PenData`, `WaveAudio`, and numerous others.

7.9 Using Resources

Figure 7-7, shown earlier in the chapter, illustrates the use of `PictureBox` controls to enlarge and display a selected thumbnail image. Each thumbnail image is loaded into the application from a local file:

```
tn1 = new PictureBox();
tn1.Image = Image.FromFile("c:\\schiele1.jpg");
```

This code works fine as long as the file `schiele1.jpg` exists in the root directory of the user's computer. However, relying on the directory path to locate this file has two obvious disadvantages: The file could be deleted or renamed by the user, and it's an external resource that has to be handled separately from the code during installation. Both problems can be solved by embedding the image in the assembly rather than treating it as an external resource.

Consider a GUI application that is to be used in multiple countries with different languages. The challenge is to adapt the screens to each country. At a minimum, this requires including text in the native language, and may also require changing images and the location of controls on the form. The ideal solution separates the logic of the program from the user interface. Such a solution treats the GUI for each country as an interchangeable resource that is loaded based on the culture settings (the country and language) of the computer.

The common denominator in these two examples is the need to bind an external *resource* to an application. .NET provides special resource files that can be used to hold just about any nonexecutable data such as strings, images, and persisted data. These resource files can be included in an assembly—obviating the need for external files—or compiled into *satellite assemblies* that can be accessed on demand by an application's main assembly.

Let's now look at the basics of working with resource files and how to embed them in assemblies; then, we will look at the role of satellite assemblies in *localized* applications.

Working with Resource Files

Resource files come in three formats: `*.txt` files in name/value format, `*.resx` files in an XML format, and `*.resources` files in a binary format. Why three? The text format provides an easy way to add string resources, the XML version supports both strings and other objects such as images, and the binary version is the binary equivalent of the XML file. It is the only format that can be embedded in an assembly—the other formats must be converted into a `.resources` file before they can be linked to an assembly. Figure 7-20 illustrates the approaches that can be used to create a `.resources` file.

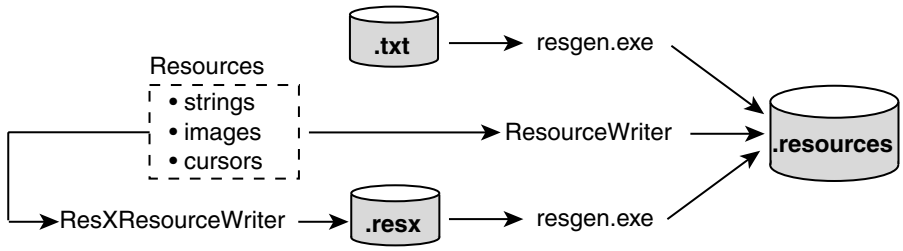


Figure 7-20 A `.resources` file can be created from a text file, resources, or a `.resx` file

The `System.Resources` namespace contains the types required to manipulate resource files. It includes classes to read from and write to both resource file formats, as well as load resources from an assembly into a program.

Creating Resource Strings from a Text File

Resource files containing string values are useful when it is necessary for a single application to present an interface that must be customized for the environment in which it runs. A resource file eliminates the need to code multiple versions of an application; instead, a developer creates a single application and multiple resource files that contain the interface captions, text, messages, and titles. For example, an English version of an application would have the English resource file embedded in its assembly; a German version would embed the German resource file. Creating resource strings and accessing them in an application requires four steps:

1. Create a text file with the name/value strings to be used in the application. The file takes this format:

   ```
   ;German version   (this is a comment)
   Language=German
   Select=Wählen Sie aus
   Page=Seite
   Previous=Vorherig
   Next=Nächst
   ```

2. Convert the text file to a `.resources` file using the Resource File Generator utility `resgen.exe`:

   ```
   > resgen german.txt  german.resources
   ```

   Note that the text editor used to create the text file should save it using UTF-8 encoding, which `resgen` expects by default.

3. Use the `System.Resources.ResourceManager` class to read the strings from the resource file. As shown here, the `ResourceManager` class accepts two arguments: the name of the resource file and the assembly containing it. The `Assembly` class is part of the `System.Reflection` namespace and is used in this case to return the current assembly. After the resource manager is created, its `GetString` method is used by the application to retrieve strings from the resource file by their string name:

```
// new ResourceManager(resource file, assembly)
ResourceManager rm = new ResourceManager(
      "german",Assembly.GetExecutingAssembly());
nxtButton.Text= rm.GetString("Next");
```

4. For this preceding code to work, of course, the resource file must be part of the application's assembly. It's bound to the assembly during compilation:

```
csc /t:exe /resource:german.resources myApp.cs
```

Using the ResourceWriter Class to Create a .resources File

The preceding solution works well for adding strings to a resource file. However, a resource file can also contain other objects such as images and cursor shapes. To place these in a `.resources` file, .NET offers the `System.Resources.ResourceWriter` class. The following code, which would be placed in a utility or helper file, shows how to create a `ResourceWriter` object and use its `AddResource` method to store a string and image in a resource file:

```
IResourceWriter writer = new ResourceWriter(
        "myResources.resources");  // .Resources output file
Image img = Image.FromFile(@"c:\schiele1.jpg");
rw.AddResource("Page","Seite");    // Add string
rw.AddResource("artistwife",img);  // Add image
rw.Close();                        // Flush resources to the file
```

Using the ResourceManager Class to Access Resources

As we did with string resources, we use the `ResourceManager` class to access object resources from within the application. To illustrate, let's return to the code presented at the beginning of this section:

```
tn1.Image = Image.FromFile("C:\\schiele1.jpg");
```

The `ResourceManager` allows us to replace the reference to an external file, with a reference to this same image that is now part of the assembly. The `GetString` method from the earlier example is replaced by the `GetObject` method:

```
ResourceManager rm = new
      ResourceManager("myresources",
                      Assembly.GetExecutingAssembly());
// Extract image from resources in assembly
tn1.Image = (Bitmap) rm.GetObject("artistwife");
```

Using the ResXResourceWriter Class to Create a .resx File

The `ResXResourceWriter` class is similar to the `ResourceWriter` class except that it is used to add resources to a `.resx` file, which represents resources in an intermediate XML format. This format is useful when creating utility programs to read, manage, and edit resources—a difficult task to perform with the binary `.resources` file.

```
ResXResourceWriter rwx = new
      ResXResourceWriter(@"c:\myresources.resx");
Image img = Image.FromFile(@"c:\schiele1.jpg");
rwx.AddResource("artistwife",img);  // Add image
rwx.Generate();   // Flush all added resources to the file
```

The resultant file contains XML header information followed by name/value tags for each resource entry. The actual data—an image in this case—is stored between the value tags. Here is a section of the file `myresources.resx` when viewed in a text editor:

```
<data name="face" type="System.Drawing.Bitmap, System.Drawing,
   Version=1.0.3300.0,Culture=neutral,
   PublicKeyToken=b03f5f7f11d50a3a" mimetype="application/x-
   microsoft.net.object.bytearray.base64">
<value>       ----   Actual Image bytes go here ----
</value>
```

Note that although this example stores only one image in the file, a `.resx` file can contain multiple resource types.

Using the ResXResourceReader Class to Read a .resx file

The `ResXResourceReader` class provides an `IDictionaryEnumerator` (see Chapter 4) that is used to iterate through the tag(s) in a `.resx` file. This code segment lists the contents of a resource file:

```
ResXResourceReader rrx = new
      ResXResourceReader("c:\\myresources.resx");
// Enumerate the collection of tags
foreach (DictionaryEntry de in rrx)
{
   MessageBox.Show("Name: "+de.Key.ToString()+"\nValue: " +
                   de.Value.ToString());
   // Output --> Name:  artistwife
   //        --> Value: System.Drawing.Bitmap
}
rrx.Close();
```

Converting a .resx File to a .resources File

The `.resx` file is converted to a `.resources` file using `resgen.exe`:

```
resgen myresources.resx  myresources.resources
```

If the second parameter is not included, the output file will have the same base name as the source file. Also, note that this utility can be used to create a `.resources` file from a `.resx` file. The syntax is the same as in the preceding example—just reverse the parameters.

VS.NET and Resources

Visual Studio.NET automatically creates a `.resx` file for each form in a project and updates them as more resources are added to the project. You can see the resource file(s) by selecting the Show All Files icon in the Solution Explorer.

When a build occurs, `.resources` files are created from the `.resx` files. In the code itself, a `ResourceManager` object is created to provide runtime access to the resources:

```
ResourceManager resources = new ResourceManager(typeof(Form1));
```

Using Resource Files to Create Localized Forms

In .NET vernacular, a *localized* application is one that provides multi-language support. This typically means providing user interfaces that display text and images customized for individual countries or cultures. The .NET resource files are designed to support such applications.

In a nutshell, resource files can be set up for each culture being supported. For example, one file may have all the control labels and text on its interface in German; another may have the same controls with French text. When the application runs, it

looks at the culture settings of the computer it is running on and pulls in the appropriate resources. This little bit of magic is accomplished by associating resource files with the `CultureInfo` class that designates a language, or language and culture. The resource files are packaged as *satellite assemblies*, which are resource files stored as DLLs.

Resource Localization Using Visual Studio.NET

To make a form localized, you must set its `Localizable` property to `true`. This has the effect of turning each control on a form into a resource that has its properties stored in the form's `.resx` file. This sets the stage for creating separate `.resx` files for each culture a form supports.

Recall from Chapter 5, "C# Text Manipulation and File I/O," that a culture is specified by a two-character language code followed by an optional two-character country code. For example, the code for English in the United States is `en-US`. The terms *neutral culture* and *specific culture* are terms to describe a culture. A specific culture has both the language and country specified; a neutral culture has only the language. Consult the MSDN documentation on the `CultureInfo` class for a complete list of culture names.

To associate other cultures with a form, set the form's `Language` property to another locale from the drop-down list in the Properties window. This causes a `.resx` file to be created for the new culture. You can now customize the form for this culture by changing text, resizing controls, or moving controls around. This new property information is stored in the `.resx` file for this culture only—leaving the `.resx` files for other cultures unaffected.

The resource files are stored in folders, as shown in Figure 7-21. When the project is built, a *satellite assembly* is created to contain the resources for each culture, as shown in Figure 7-22. This DLL file has the same name in each folder.

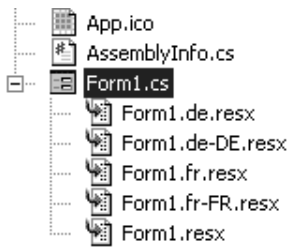


Figure 7-21 VS.NET resource files for multiple cultures

Figure 7-22 Satellite assembly

Determining Localization Resources at Runtime

By default, an application's thread has its `CurrentThread.CurrentUICulture` property set to the culture setting of the machine it is running on. Instances of the `ResourceManager`, in turn, use this value to determine which resources to load. They do this by searching for the satellite assembly in the folder associated with the culture—a reason why the naming and location of resource folders and files is important. If no culture-specific resources are found, the resources in the main assembly are used.

Core Note

The easiest way to test an application with other culture settings is to set the `CurrentUICulture` to the desired culture. The following statement, for example, is placed before `InitializeComponent()` in VS.NET to set the specific culture to German:

```
System.Threading.Thread.CurrentThread.CurrentUICulture =
   new System.Globalization.CultureInfo("de-DE");
```

Creating a Satellite Assembly Without VS.NET

One of the advantages of using satellite assemblies is that they can be added to an application, or modified, without recompiling the application. The only requirements are that a folder be set up along the proper path, and that the folder and satellite assembly have the proper name.

Suppose you have a `.resx` file that has been converted by your translator to French Canadian. You can manually create and add a satellite assembly to the application in three steps:

1. Convert the `.resx` file to a `.resources` file:

   ```
   filmography.Form1.fr-CA.resources
   ```

2. Convert the .resources file to a satellite assembly using the Assembly Linker (`Al.exe`):

   ```
   Al.exe
         /t:lib
         /embed:filmography.Form1.fr-CA.resources
         /culture:fr-CA
         /out:filmography.resources.dll
   ```

3. Create the `fr-CA` folder beneath `Release` folder and copy the new assembly file into it.

Placing the satellite assembly in the proper folder makes it immediately available to the executable and does not require compiling the application.

7.10 Summary

There are more than 50 GUI controls available in the .NET Framework Class Library. This chapter has taken a selective look at some of the more important ones. They all derive from the `System.Windows.Forms.Control` class that provides the inherited properties and methods that all the controls have in common.

Although each control is functionally unique, it is possible to create a taxonomy of controls based on similar characteristics and behavior. The button types, which are used to intitiate an action or make a selection, include the simple `Button`, `CheckBox`, and `RadioButton`. These are often grouped using a `GroupBox` or `Panel` control. The `TextBox` can be used to hold a single line of text or an entire document. Numerous methods are available to search the box and identify selected text within it. The `PictureBox` is available to hold images and has a `SizeMode` property that is used to position and size an image within the box.

Several controls are available for presenting lists of data. The `ListBox` and `ComboBox` display data in a simple text format. However, the underlying data may be a class object with multiple properties. The `TreeView` and `ListView` are useful for displaying data with a hierarchical relationship. The `ListView` can display data in multiple views that include a grid layout and icon representation of data. The `TreeView` presents a tree metaphor to the developer, with data represented as parent and child nodes.

Most of the controls support the drag-and-drop operation that makes it easy to move or copy data from one control to another. The source control initiates the action by calling a `DoDragDrop` method that passes the data and permissible effects to the target control.

For applications that require nonstandard controls, .NET lets you create custom controls. They may be created from scratch, derived from an existing control, or created as a combination of controls in a user control container.

7.11 Test Your Understanding

1. Why is a container control such as a `GroupBox` used with radio buttons?

2. What is the `SizeMode` property set to in order to automatically resize and fill an image in a `PictureBox`?

3. Suppose you place objects in a `ListBox` that have these properties:

   ```
   string Vendor, string ProductID, int Quantity
   ```

   How do you have the `ListBox` display the `ProductID` and `Quantity`?

4. What event is fired when an item in a `ListBox` is selected? What `ListBox` properties are used to identify the selected item?

5. What property and value are set on a `ListView` to display its full contents in a grid layout?

6. Which `TreeNode` property can be used to store object data in a `TreeView` node?

7. Which two events must the destination control in a drag-and-drop operation support?

8. The Property Browser in VS.NET uses metadata to categorize a control's properties and events and assign default values. How do you generate this information for the properties in a custom control?

9. What class is used to read text from a text resource file embedded in an assembly? What method is used to read values from the file?