2

Windows Forms Controls: RadioButton, CheckBox, ListBox, CheckedListBox, and ComboBox

In Depth

In the previous chapter, you learned about some of the commonly used controls, such as `Button`, `Label`, `TextBox`, `RichTextBox`, and `MaskedTextBox`. This chapter describes some of the standard Windows Forms controls, RadioButton, CheckBox, ListBox, CheckedListBox, and ComboBox. These controls form the basic layout of an application using which users interact with the application. You would call these controls as selection controls since all of them are primarily used for selection purpose. The RadioButton control is used in a situation where you want to select only one option from the group of options; whereas, the CheckBox control allows you to select multiple options from the group. The ListBox control lists all the items and allows you to select the items from the list. The CheckedListBox control is similar to the ListBox control; in addition, it displays check boxes beside every item in the list. Instead of selecting the item, you can select the check box of that particular item you want to select the CheckedListBox control. The ComboBox control is similar to the ListBox control; the difference is that the ComboBox control displays the items as the drop-down list.

The basic idea of this chapter is to help you decide which control to use for which purpose and provide you some idea about the most commonly used properties, methods, and events of the control. This chapter describes the general purpose of using the controls. The chapter starts with the discussion on the controls- `RadioButton`, `CheckBox`, `ListBox`, `CheckedListBox`, and `ComboBox`- and also lists their properties, methods, and events. Next, you learn about the practical implementation of these controls in the Immediate Solution of the chapter.

Let's start our discussion with the RadioButton control.

The RadioButton Control

A `RadioButton` control, also known as an option button, is used to select one option from a set of options. Radio buttons and check boxes may appear similar to you except for one main difference. A check box can work independently; whereas, radio buttons always work in groups. This means that whenever you select one radio button from a group of radio buttons, the other radio buttons in the group automatically get deselected. In contrast, you can select any number of check boxes from a list of check boxes. In other words, radio buttons are used for single-select choices, while check boxes are used for multiple-select options. Figure 2.1 shows radio buttons as they appear at design time:

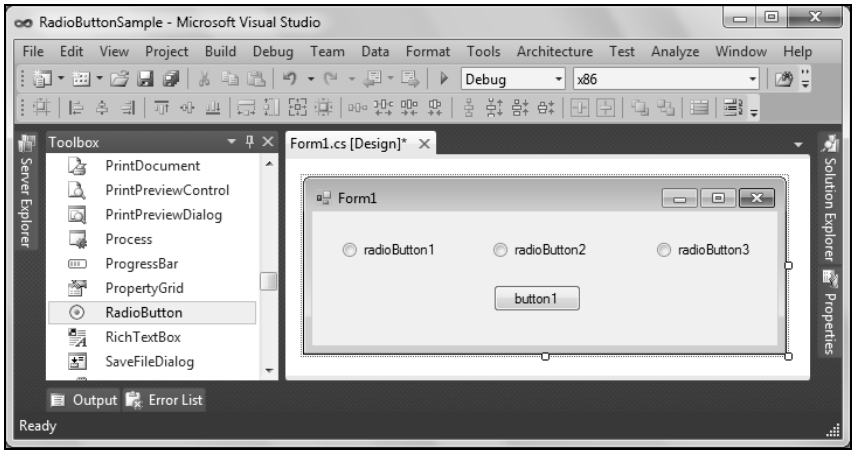

Figure 2.1: Showing the Radio Buttons

You can display text, images, or both simultaneously on the RadioButton controls. You can use the `Checked` property of the control to get or set the state of a radio button. The appearance of the radio button may be altered to appear as a toggle-style button or as a standard radio button by setting the `Appearance` property of the RadioButton control. In addition, if you want to place more than one group of radio buttons on the same form, you must place the radio buttons in different container controls, such as `GroupBox` or `Panel` control.

Following is the class hierarchy of the `RadioButton` class:

```
System.Object
 System.MarshalByRefObject
        System.ComponentModel.Component
               System.Windows.Forms.Control
                      System.Windows.Forms.ButtonBase
                             System.Windows.Forms.RadioButton
```

Table 2.1 lists the important properties of the `RadioButton` class:

Table 2.1: Showing the Noteworthy Properties of the RadioButton Class

Property	Description
`Appearance`	Retrieves or sets the value for this property that determines the appearance of the radio button.
`AutoCheck`	Retrieves or sets a value indicating whether or not the `Checked` value and the appearance of the control should automatically change when the radio button is clicked.
`CheckAlign`	Retrieves or sets the location of the check box portion of the radio button. A radio button consists of a check box portion and a text. The check box portion determines whether the radio button is checked or unchecked and the text is used to set a caption on the radio button. When you set the `CheckAlign` property of the RadioButton control, the location of the check box portion changes to the value assigned and the text remains as it is.
`Checked`	Retrieves or sets a value indicating whether the radio button is clicked.
`Enabled`	Retrieves or sets a value indicating whether or not the user can select the radio button.
`Text`	Retrieves or sets the caption for a radio button.
`TextAlign`	Retrieves or sets the alignment of the text in a radio button.
`Visible`	Retrieves or sets the value indicating whether the radio button is displayed or not.

Table 2.2 lists the important methods of the `RadioButton` class:

Table 2.2: Showing the Noteworthy Methods of the RadioButton Class

Method	Description
`PerformClick()`	Generates a `Click` event when a user clicks a radio button
`Show()`	Displays the radio button to the user

Table 2.3 lists the notable events of the `RadioButton` class:

Table 2.3: Showing the Noteworthy Events of the RadioButton Class

Event	Description
`AppearanceChanged`	Occurs when the value of the `Appearance` property changes
`CheckedChanged`	Occurs when the value of the `Checked` property changes
`Click`	Occurs when the user clicks the associated radio button

The CheckBox Control

A `CheckBox` control is represented as a box shaped control that enables the user to select or deselect an option. `This` control accepts either `true` or `false` as a value. You click a check box to select it and click it again to

deselect it. When you select the check box, it holds the value `true`; and it holds the value `false` when you deselect it.

Although a check box is usually selected or cleared, this control also has a third indeterminate state. When the check box is in the indeterminate state, it is grayed-out. Applications use this state only when partial or unknown selection needs to be represented. Consider an example, where a single check box is used to change the font style of the text box to bold, italics, or a combination of both. Now, if the text is in either bold or italics, you can easily determine using the check box. However, when the text is both in italics and bold, it becomes difficult to specify the current state of the check box. This state is known as the indeterminate state. To enable indeterminate state, set the `ThreeState` property of the check box to `true`. Figure 2.2 shows check boxes as they appear at design time:

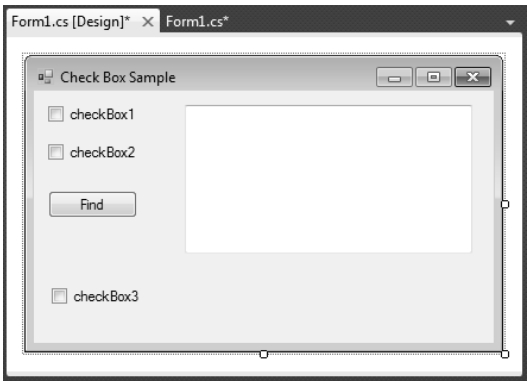

Figure 2.2: Displays Check Boxes at Design Time

The check boxes are based on the `CheckBox` class. Following is the class hierarchy of the `CheckBox` class:

```
System.Object
   System.MarshalByRefObject
         System.ComponentModel.Component
                  System.Windows.Forms.Control
                           System.Windows.Forms.ButtonBase
                                    System.Windows.Forms.CheckBox
```

Table 2.4 lists the notable properties of the `CheckBox` class:

Table 2.4: Showing the Noteworthy Properties of the CheckBox Class

Property	Description
`Appearance`	Retrieves or sets the appearance of a check box
`AutoCheck`	Specifies whether or not the `Checked` or `CheckState` values and the appearance of the check box should be automatically changed when the check box is clicked
`CheckAlign`	Retrieves or sets the horizontal and vertical alignment of a check box in a control
`Checked`	Retrieves or sets a value indicating whether or not the check box is in the checked state
`CheckState`	Retrieves or sets the state of a check box
`Text`	Retrieves or sets the caption for the check box
`TextAlign`	Retrieves or sets the alignment of the caption on the check box
`ThreeState`	Specifies whether or not the check box should allow three check states, rather than two

Table 2.5 lists the notable events of the `CheckBox` class:

Table 2.5: Showing the Noteworthy Events of the CheckBox Class

Event	Description
`AppearanceChanged`	Occurs when the `Appearance` property changes
`CheckedChanged`	Occurs when the value of the `Checked` property changes
`CheckStateChanged`	Occurs when the value of the `CheckState` property changes
`Click`	Occurs when the user selects the check box

A `CheckBox` control can display image, corresponding text associated with the check box, or both simultaneously. Use the `ImageAlign` and `TextAlign` properties to determine the location where the caption and picture should appear in the `CheckBox` control.

You can also use the `ImageList` and `ImageIndex` properties of the check box to display an image on the check box. Let's now explore the `ListBox` control.

The ListBox Control

A `ListBox` control displays a list of items from which the user can select one or more item. Figure 2.3 shows list boxes as they appear at design time:

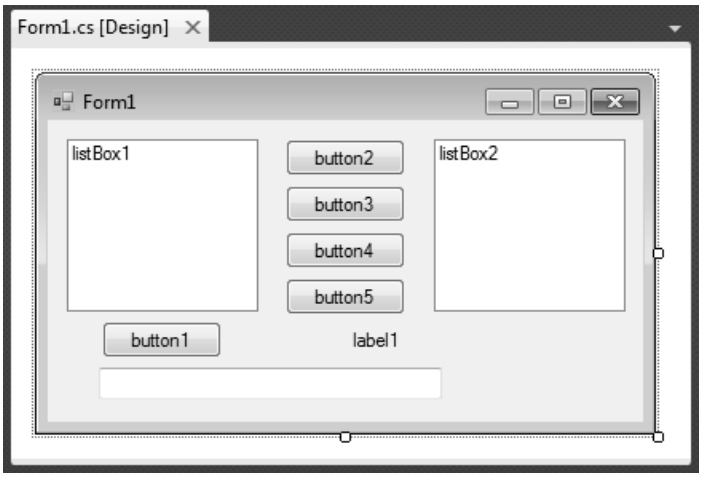

Figure 2.3: Showing the Design View of the ListBox Control

If the items exceed a specified limit, a scroll bar automatically appears to let the user to scroll through the list. You can also scroll list boxes horizontally by setting the `MultiColumn` property to `true`. In addition, when the `ScrollAlwaysVisible` property is set to `true`, a scroll bar always appears.

Following is the class hierarchy for the `ListBox` class:

```
System.Object
  System.MarshalByRefObject
        System.ComponentModel.Component
               System.Windows.Forms.Control
                      System.Windows.Forms.ListControl
                             System.Windows.Forms.ListBox
```

As you can see in the class hierarchy, the list boxes are derived from the `Control` class, but not directly—instead, they are derived directly from the `ListControl` class.

Table 2.6 lists the notable properties of the `ListBox` class:

Table 2.6: Showing the Noteworthy Properties of the ListBox Class	
Property	**Description**
BorderStyle	Retrieves or sets the type of border that is drawn around the list box.
ColumnWidth	Retrieves or sets the width of columns in a multicolumn list box.
HorizontalExtent	Retrieves or sets the horizontal scrolling area of a list box.
HorizontalScrollbar	Retrieves or sets the value indicating whether or not a horizontal scroll bar should be displayed in a list box.
ItemHeight	Retrieves or sets the height of an item in a list box.
Items	Returns a list of items of the list box.
MultiColumn	Retrieves or sets a value indicating whether or not the list box supports multiple columns.
ScrollAlwaysVisible	Retrieves or sets a value indicating whether or not a vertical scroll bar should always be displayed.
SelectedIndex	Retrieves or sets the index of the currently selected item of the list box.
SelectedIndices	Retrieves and displays a collection that contains the indices of all the selected items in the list box.
SelectedItem	Retrieves or sets the selected item in the list box.
SelectedItems	Retrieves and displays a collection containing the selected items of the list box.
SelectionMode	Retrieves or sets the method in which items are selected. This property enables you to determine how many items a user can select at a time and how the user can make multiple selections.
Sorted	Retrieves or sets a value indicating whether or not the items in the list box are sorted. If you set this property to `true`, the items are sorted alphabetically.
Text	Retrieves the text of the selected item in the list box.
TopIndex	Retrieves or sets the index of the first item of the list box.

Table 2.7 lists the noteworthy methods of the `ListBox` class:

Table 2.7: Showing the Noteworthy Methods of the ListBox Class	
Method	**Description**
ClearSelected()	Deselects all the items in a list box
FindString()	Finds the first item in the list box that begins with the indicated string
FindStringExact()	Finds the first item in the list box that exactly matches the indicated string
GetSelected()	Returns `true` if the indicated item is selected
SetSelected()	Selects or deselects the indicated item in a list box

Table 2.8 lists the notable events of the `ListBox` class:

Table 2.8: Showing the Noteworthy Events of the ListBox Class	
Event	**Description**
Click	Occurs when the list box is clicked
SelectedIndexChanged	Occurs when the `SelectedIndex` property has changed

The items in list boxes are stored in the `Items` collection. The `Items.Count` property holds the number of items in the list (the value of the `Items.Count` property is always one more than the largest possible `SelectedIndex` value because `selected index` is zero-based, i.e., the `selected index` always starts with zero index).

To add or delete items in a `ListBox` control, you can use the `Items.Add`, `Items.Insert`, `Items.Clear`, `Items.Remove`, or `Items.RemoveAt()` method. You can also add a number of items to a list box simultaneously with the `AddRange()` method. In addition, you can add and remove items to the list by using the `Items` property at design time.

You can also support multiple selections in list boxes. The `SelectionMode` property determines the number of items that can be selected at a time and can hold one of the following values:

- `MultiExtended`—Specifies that multiple items can be selected and the user can use the SHIFT, CTRL, and arrow keys to make selections
- `MultiSimple`—Specifies that multiple items can be selected
- `None`—Specifies that no items can be selected
- `One`—Specifies that only one item can be selected

When you support multiple selections, you can use the `Items` property to access the items in the list box, the `SelectedItems` property to access the selected items, and the `SelectedIndices` property to access the selected indices.

The CheckedListBox Control

Similar to the `ListBox` control, the `CheckedListBox` control also displays a list of items. However, a CheckedListBox control also displays a check box to the left of individual items. This allows you to scroll through the list of items and select them by checking the respective check boxes. You can see the `CheckedListBox` control, as shown in Figure 2.4:

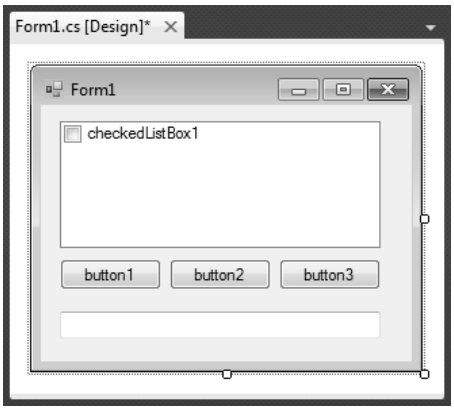

Figure 2.4: The CheckedListBox Control at Design Time

Following is the inheritance hierarchy of the `CheckedListBox` class:

```
System.Object
  System.MarshalByRefObject
        System.ComponentModel.Component
              System.Windows.Forms.Control
                    System.Windows.Forms.ListControl
                          System.Windows.Forms.ListBox
                                System.Windows.Forms.CheckedListBox
```

Similar to other controls, the `CheckedListBox` control has several properties, methods, and events. These properties, methods, and events can be accessed and customized (optionally) as per the requirements of your application.

Table 2.9 lists the noteworthy properties of the `CheckedListBox` class:

Table 2.9: Showing the Noteworthy Properties of the CheckedListBox Class	
Property	**Description**
`CheckedItems`	Holds the collection of checked items in this checked list box
`CheckOnClick`	Retrieves or sets a value indicating if the check box should be toggled when the corresponding item is selected
`ItemHeight`	Returns the height of an item
`Items`	Returns a collection of items in a checked list box
`MultiColumn`	Retrieves or sets a value indicating if the checked list box allows multiple columns
`SelectedIndex`	Retrieves or sets the index of the selected item in a checked list box
`SelectedIndices`	Retrieves a collection that contains zero-based indices of all currently selected items in the checked list box
`SelectedItem`	Retrieves or sets the selected item in the checked list box
`SelectedItems`	Retrieves a collection containing the selected items in the checked list box
`SelectionMode`	Retrieves or sets the current selection mode
`Sorted`	Retrieves or sets a value indicating if the items in the checked list box should be sorted alphabetically
`Text`	Retrieves the text of the selected item in a checked list box

Table 2.10 lists the noteworthy methods of the `CheckedListBox` class:

Table 2.10: Showing the Noteworthy Methods of the CheckedListBox Class	
Method	**Description**
`FindString()`	Finds the first item, which begins with the indicated string, in the checked list box
`FindStringExact()`	Finds the first item in the checked list box that exactly matches the indicated string
`GetItemChecked()`	Retrieves a value indicating if the specified item is checked or not
`GetItemCheckState()`	Retrieves the check state of the current item
`GetSelected()`	Retrieves a value indicating if the specified item is selected
`SetItemChecked()`	Checks the item at the specified index
`SetItemCheckState()`	Retrieves the specified check state of the item
`SetSelected()`	Selects or clears the selection for the specified item in a checked list box

Table 2.11 lists the noteworthy events of the `CheckedListBox` class:

Table 2.11: Showing the Noteworthy Events of the CheckedListBox Class	
Event	**Description**
`Click`	Occurs when the user selects the checked list box
`ItemCheck`	Occurs when the checked state of an item changes
`SelectedIndexChanged`	Occurs when the `SelectedIndex` property has changed

Similar to standard list boxes, you can access the items in a checked list box using the `Items` property. To check an item, the user has to double-click a check box by default, unless you set the `CheckOnClick` property to `true`, in which case the user just needs to perform a single click on the check box to select it.

You can get the checked items in a checked list box with the `CheckedItems` and `CheckedIndices` properties. You can also use the `GetItemChecked()` method to verify whether or not an item is checked. In addition, you can use the `ItemCheck` event to handle check events, and the `SetItemChecked()` method to check or uncheck items.

Checked list boxes can also support three states by using the `CheckState` enumeration—`Checked`, `Indeterminate`, and `Unchecked`. You must set the state of Indeterminate in the code because the user interface does not provide a way of doing so. To use three-state check boxes, you can use the `GetItemCheckState` and `SetItemCheckState()` methods instead of the `GetItemChecked` and `SetItemChecked()` methods.

The ComboBox Control

The `ComboBox` control is used to display data in a drop-down combo box. The combo box is made up of two parts. The top part is a text box that allows the user to type in all or part of a list item. The other part is a list box that displays a list of items from which the user can select one or more items. In other words, a combo box allows the user to either select an item from the list or enter the data through the keyboard. Figure 2.5 shows the `ComboBox` control as it appears at design time:

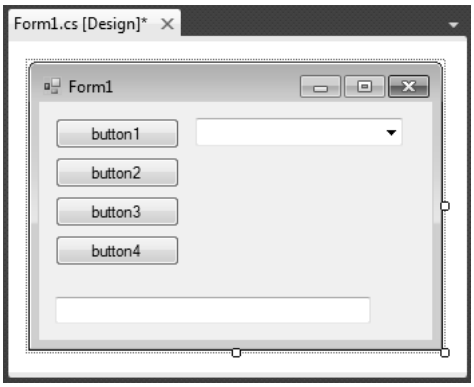

Figure 2.5: Showing the Design View of the ComboBox Control

Following is the inheritance hierarchy of the `ComboBox` class:

```
System.Object
  System.MarshalByRefObject
        System.ComponentModel.Component
              System.Windows.Forms.Control
                    System.Windows.Forms.ListControl
                          System.Windows.Forms.ComboBox
```

As you can see in the inheritance hierarchy, the `ComboBox` class is derived from the `ListControl` class, similar to the `ListBox` class.

Table 2.12 lists the notable properties of the `ComboBox` class:

Table 2.12: Showing the Noteworthy Properties of the ComboBox Class

Property	Description
`AutoCompleteCustomSource`	Retrieves or sets a custom `StringCollection` class to use when the `AutoCompleteSource` property is set to `CustomSource`. This property helps in incorporating auto-complete feature in combo box.
`AutoCompleteMode`	Retrieves or sets an option that controls the automatic completion for the combo box.

Table 2.12: Showing the Noteworthy Properties of the ComboBox Class	
Property	**Description**
AutoCompleteSource	Retrieves or sets a value indicating the source of complete strings used for automatic completion.
DataSource	Retrieves or sets the data source for the combo box.
DropDownHeight	Retrieves or sets the height in pixels for the drop-down portion of the combo box.
DropDownStyle	Retrieves or sets the style of the combo box.
DropDownWidth	Retrieves or sets the width of the drop-down part of the combo box.
DroppedDown	Retrieves or sets a value indicating whether the combo box is displaying its drop-down part.
FlatStyle	Retrieves or sets the appearance of the combo box.
Focused	Retrieves a value specifying if the combo box has the focus.
ItemHeight	Retrieves or sets the height of an item in a combo box.
Items	Retrieves a collection of the items in the combo box.
MaxDropDownItems	Retrieves or sets the maximum number of items to be shown in the drop-down part of a combo box.
MaxLength	Retrieves or sets the maximum number of characters a user can enter in the text box of the combo box.
SelectedIndex	Retrieves or sets the index of the currently selected item.
SelectedItem	Retrieves or sets currently selected item in the combo box.
SelectedText	Retrieves or sets the selected text in the text box part of a combo box.
SelectionLength	Retrieves or sets the number of characters selected in the text box part of the combo box.
SelectionStart	Retrieves or sets the starting index of selected text in the combo box.
Sorted	Retrieves or sets a value indicating if the items in the combo box are sorted.
Text	Retrieves or sets the text associated with the combo box.

Table 2.13 lists the noteworthy methods of the ComboBox class:

Table 2.13: Showing the Noteworthy Methods of the ComboBox Class	
Method	**Description**
FindString()	Finds the first item in the combo box that begins with the specified string
FindStringExact()	Finds the item that matches the specified string exactly
Select()	Selects a range of text
SelectAll()	Selects all the text in the text box of the combo box

Table 2.14 lists the noteworthy events of the ComboBox class:

Table 2.14: Showing the Noteworthy Events of the ComboBox Class	
Event	**Description**
DropDown	Occurs when the drop-down portion of a combo box is shown
DropDownClosed	Occurs when the drop-down portion of the combo box is no longer visible

Table 2.14: Showing the Noteworthy Events of the ComboBox Class	
Event	**Description**
`DropDownStyleChanged`	Occurs when the `DropDownStyle` property has changed
`SelectedIndexChanged`	Occurs when the `SelectedIndex` property has changed
`SelectionChangeCommitted`	Occurs when the selected item has changed and that change also appears in the combo box
`TextUpdate`	Occurs when the combo box has formatted the text, but prior to the appearance of the text

You can use the `Text` property of a combo box to set and access the text in the text box of the combo box. You can use the `SelectedIndex` property to get the selected list item. You can change the selected item by changing the `SelectedIndex` value in the code, which displays the corresponding item in the text box portion of the combo box. As with list boxes, if no items are selected, the `SelectedIndex` value is set to `-1`. If the first item in the list is selected, the `SelectedIndex` value is set to `0`.

In addition, you can also use the `SelectedItem` property, which is similar to the `SelectedIndex` property, but returns the item selected (often a string value). The `Items.Count` property reflects the number of items in the list (the value of the `Items.Count` property is always one more than the largest possible `SelectedIndex` value because `SelectedIndex` is zero-based).

To add or delete items in a `ComboBox` control, use the `Items.Add`, `Items.Insert`, `Items.Clear`, `Items.AddRange`, `Items.Remove`, or `Items.RemoveAt()` method. Alternatively, you can add and remove items from the list by using the `Items` property at design time.

By default, a combo box displays a text box with a hidden drop-down list. The `DropDownStyle` property determines the style of the combo box to be displayed. You can set this property to display the list box of the combo box in the following combo box styles:

- `DropDown`—Specifies that the text portion is not editable and you must use an arrow to see the drop-down list box. It is the default drop-down style.
- `DropDownList`—Specifies that the text portion is editable and the user can use the arrow key to view the list.
- `Simple`—Specifies that the list is always displayed.

TIP

To display a list that the user cannot edit, you should use a list box instead of a combo box.

You can add and remove items in combo boxes in the same ways as you can with list boxes. You can also use the `BeginUpdate()` and `EndUpdate()` methods to add a large number of items to the combo box without redrawing the control each time an item is added to the list. The `FindString()` and `FindStringExact()` methods let you search for an item in the list that contains a particular search string. Similar to list boxes, you use the `SelectedIndex` property to get or set the index of the currently selected item, and the `SelectedItem` property to get or set the selected item. You can also use the `Text` property to specify the string displayed in the text box part of the combo box.

Immediate Solutions

Using the RadioButton Control

As mentioned in the *In Depth* section, a RadioButton control allows a user to select one item from an exclusive group of items. Let us see some of its properties in action by creating an application, named `RadioButtonSample` (available in the CD-ROM). This example contains a sample user interface form with three radio buttons; Left, Center, and Right, and one button control. When you select any of these radio buttons, the button control on the form aligns itself as per the selected radio button option. If you click the button, a message box displays a message indicating the selected radio button and at the same time, all the radio buttons are transformed to toggle buttons.

To create this application, you need to perform the following broad-level steps:

1. Setting the text for the `RadioButton` controls
2. Retrieving and setting the check state of the `RadioButton` controls
3. Creating toggle buttons

Let's explore these in detail next.

Setting the Text for the RadioButton Controls

You can set the text for the `RadioButton` controls in two ways – at design time and at run time. While in the design mode, you can set the `Text` property for a radio button by first selecting the radio button and setting its `Text` property in the Properties window. This example; however, explains how to set the text of the radio buttons at run time. To do so, perform the following steps in the `RadioButtonSample` application:

1. Set the `Text` property of the `Form1` form to Radio Button Sample.
2. Drag and drop a `Button` control and three `RadioButton` controls to the form and change the Text property of the Button control to Click Me.
3. Add the code, shown in Listing 2.1, on the Load event of the form to set the `Text` property of the radio buttons at run time:

Listing 2.1: Showing the Code for the Load Event of the Form

```
private void Form1_Load(object sender, EventArgs e)
{
   radioButton1.Text = "Left";
   radioButton2.Text = "Center";
   radioButton3.Text = "Right";
}
```

4. Change the Bold property of all the radio buttons to true and change the Text property of the button to Click Me.
5. Press the `F5` key on the keyboard to execute the application and the output of the application appears, as shown in Figure 2.6:

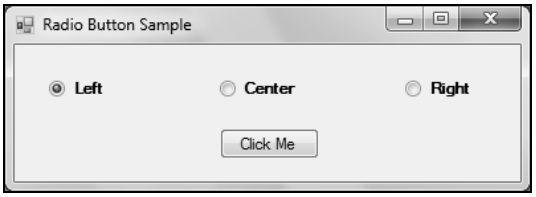

Figure 2.6: Displays the Text Set for the Radio Buttons

Retrieving and Setting the Check State of the RadioButton Controls

You can check whether or not a radio button is selected by using the `Checked` property. This property of the radio button has only two settings, `true` if the button is selected and `false` if not selected.

Let's use the same application, `RadioButtonSample`, to demonstrate whether or not a radio button is selected. To do so, perform the following steps:

1. Double-click the Click Me button in the design mode of the Form1.cs file to generate its Click event and add the code, as shown in Listing 2.2:

Listing 2.2: Showing the Code for Retrieving the Check State of Radio Buttons

```
private void button1_Click(object sender, EventArgs e)
{
   if (radioButton1.Checked == true) {
         MessageBox.Show("The Radio Button Left is selected.");
   }
   if (radioButton2.Checked == true) {
         MessageBox.Show("The Radio Button Center is selected.");
   }
   if (radioButton3.Checked == true)
   {
         MessageBox.Show("The Radio Button Right is selected.");
   }
}
```

2. Press the `F5` key on the keyboard to execute the application. When the form appears, observe that by default the first radio button is selected.
3. Click the Click Me button, a message box appears showing that the Left radio button is selected, as shown in Figure 2.7:

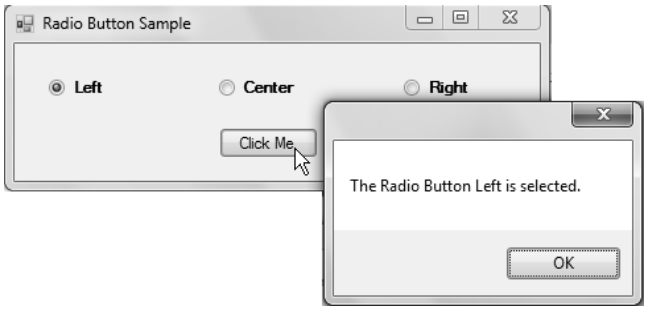

Figure 2.7: Showing a Message Box on Selection of Left Radio Button

Similarly, you can also determine whether or not, a radio button should be selected, by performing the following steps:

4. Add the highlighted code, shown in Listing 2.3, in the `Load` event of the form:

Listing 2.3: Showing the Code for Setting the Check State of Radio Buttons

```
private void Form1_Load(object sender, EventArgs e)
{
   radioButton1.Text = "Left";
   radioButton2.Text = "Center";
   radioButton3.Text = "Right";
   radioButton2.Checked = true;
}
```

5. Press the `F5` key on the keyboard to execute the application and the output appears, as shown in Figure 2.8:

Figure 2.8: Displays a Radio Button Selected

Creating Toggle Buttons

You can turn check boxes or radio buttons into toggle buttons if you set their `Appearance` property to `Button` (by default, the value of the `Appearance` property is `Normal`). Toggle buttons resemble standard buttons but act similar to the check boxes or radio buttons. Let's perform the following steps to transform radio buttons into toggle buttons:

1. Add the highlighted code, shown in Listing 2.4, in the `Click` event of the button (i.e., to transform the radio buttons to toggle buttons):

Listing 2.4: Showing the Code for Creating the Toggle Buttons

```
private void button1_Click(object sender, EventArgs e)
{
  if (radioButton1.Checked == true)
  {
        MessageBox.Show("The Radio Button Left is selected.");
  }
  if (radioButton2.Checked == true)
  {
        MessageBox.Show("The Radio Button Center is selected.");
  }
  if (radioButton3.Checked == true)
  {
        MessageBox.Show("The Radio Button Right is selected.");
  }

  radioButton1.Appearance = Appearance.Button;
  radioButton2.Appearance = Appearance.Button;
  radioButton3.Appearance = Appearance.Button;
}
```

2. Add the code, shown in Listing 2.5, to the `CheckedChanged` event of the radio buttons to align the button left, center, or right, as per the radio button selected:

Listing 2.5: Showing the Code for Aligning the Button on the Selection of a Radio Button

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
  button1.Location = new System.Drawing.Point(50, 100);
}
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
  button1.Location = new System.Drawing.Point(160, 100);
}
private void radioButton3_CheckedChanged(object sender, EventArgs e)
{
  button1.Location = new System.Drawing.Point(250, 100);
}
```

3. Press the `F5` key on the keyboard to execute the application, when the form appears, select the Right radio button. Observe that the button aligns to the right of the form, as shown in Figure 2.9:

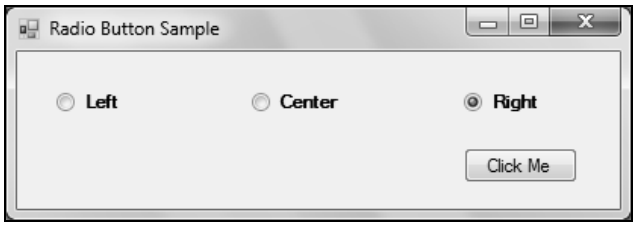

Figure 2.9: Changing the Alignment of the Button to the Right

4. Click the Click Me button, all the radio buttons are transformed to the toggle buttons, as shown in Figure 2.10:

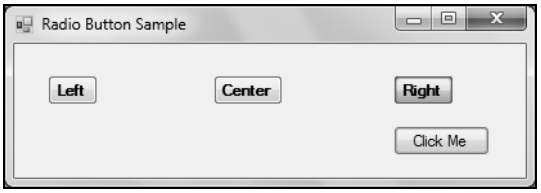

Figure 2.10: Radio Buttons Transformed to Toggle Buttons

Till now, you have learned some of the most common tasks that we normally perform with radio buttons. Let's now shift our focus to the `CheckBox` control.

Using the CheckBox Control

You are already familiar with the properties of the `CheckBox` control. To see how these properties work, let us create an application named `CheckBoxSample` (available in the CD-ROM). This application displays the total number of words and characters entered in a text box. To do so, this application uses a `TextBox` control, a `Button` control, a `Label` control, and three `CheckBox` controls. The functions of each of these `CheckBox` controls are as follows:

- The first one enables the counting of the number of characters entered in the textbox
- The second one enables the words count
- The last one is set to an indeterminate state

The last check box determines the state on the basis of the selections made in the first two check boxes. If the first and the second check boxes are not selected, the third one also remains clear; if the first two check boxes are selected, the third one also gets selected. However, if one of these is checked and the other is clear, the third check box shows an indeterminate state. Finally, a label displays the total number of characters and words entered in the text box.

To create this application, perform the following broad-level steps:

1. Set the text for the `CheckBox` controls
2. Retrieve and set the check state of the `CheckBox` controls
3. Create three-state `CheckBox` controls

Let's explore these in detail next.

Setting the Text for the CheckBox Controls

Similar to any other Windows Forms controls, you can set the text for the check box using the `Text` property. To do so, perform the following steps in the `CheckBoxSample` application:

1. Set the Text property of the Form1 form to `Check Box Sample`.
2. Drag and drop a `TextBox` control to the form and set its `Multiline` property to `true`.
3. Drag and drop a `Button` control to the form and set its Text property as Find.
4. Add three `CheckBox` controls to the form and add the code on the `Load` event of the form to set the `Text` property of the CheckBox controls, as shown in Listing 2.6:

Listing 2.6: Showing the Code for Setting the Text Property of the CheckBox Control

```
private void Form1_Load(object sender, EventArgs e)
{
  checkBox1.Text = "Chars count";
  checkBox2.Text = "Words count";
  checkBox3.Text = "Check box displaying indeterminate state";
}
```

5. Add a `Label` control and change the Text property of the Label control to blank.
6. Disable the `CheckBox3` control by setting its `Enabled` property to `false`.

7. Press the F5 key on the keyboard to execute the application. The output of the application appears, as shown in Figure 2.11:

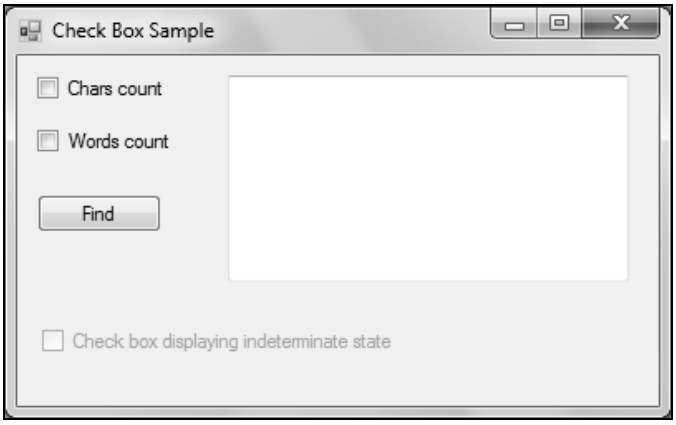

Figure 2.11: Displays the Text Set for the Check Boxes

Retrieving and Setting the Check State of the CheckBox Controls

You can see if a check box is checked by examining its Checked property. This property can be set to either true or false. Similarly, the same property is used to set the state of the check box to true or false.

Let us use the same application, CheckBoxSample, to show if a check box is selected or not. To do so, perform the following steps:

1. Double-click the Find button in the design mode of the Form1.cs file and add the code, given in Listing 2.7, in the Code Editor:

Listing 2.7: Showing the Code for Retrieving the Check State of CheckBox Controls

```
private void button1_Click(object sender, EventArgs e)
{
  if (checkBox1.Checked == true && checkBox2.Checked == true)
  {
        checkBox3.Checked = true;
  }
  if (checkBox1.Checked == false && checkBox2.Checked == false)
  {
        checkBox3.Checked = false;
  }
}
```

2. Add the highlighted code to set the check state of the check box in the Load event of the form, as shown in Listing 2.8:

Listing 2.8: Showing the Code for Setting the Check State of the Check Box

```
private void Form1_Load(object sender, EventArgs e)
{
  checkBox1.Text = "Chars count";
  checkBox2.Text = "Words count";
  checkBox3.Text = "Check box displaying indeterminate state";

  checkBox1.Checked = true;
  textBox1.Text = "Welcome to Visual C# 2010";
  label1.Text = "Total characters : " + textBox1.TextLength;
  checkBox3.CheckState = CheckState.Indeterminate;
}
```

3. Press the F5 key on the keyboard to execute the application and the output is displayed, as shown in Figure 2.12:

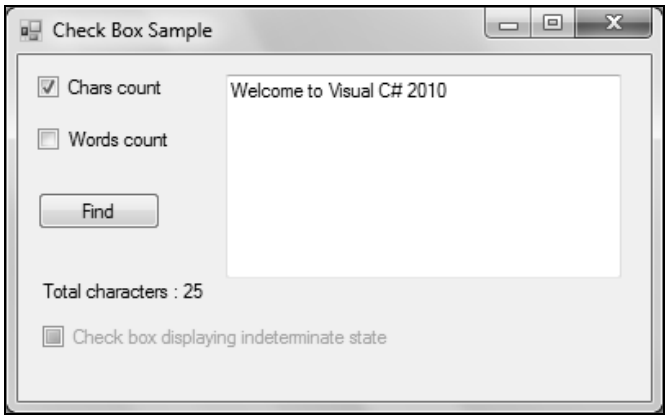

Figure 2.12: Showing the Selected Check Box

Creating Three-State CheckBox Controls

As stated earlier, a check box typically displays two states – Checked and Unchecked. However, to display the indeterminate state, the `ThreeState` property of the CheckBox control should be set to `true` and the `CheckState` property should be set to `CheckState.Indeterminate` at design time or run time.

Let us perform the following steps to create three-state `CheckBox` controls:

1. Add the highlighted code, shown in Listing 2.9, to set the indeterminate state of `CheckBox3` at the `Click` event of the button:

Listing 2.9: Showing the Code for Setting the Indeterminate State for `CheckBox3`

```
private void button1_Click(object sender, EventArgs e)
{
  if (checkBox1.Checked == true && checkBox2.Checked == true)
  {
        checkBox3.Checked = true;
  }
  if (checkBox1.Checked == false && checkBox2.Checked == false)
  {
        checkBox3.Checked = false;
  }
  if ((checkBox1.Checked == false && checkBox2.Checked == true) || (checkBox1.Checked ==
        true && checkBox2.Checked == false))
  {
        checkBox3.CheckState = CheckState.Indeterminate;
  }
}
```

2. Add the highlighted code, shown in Listing 2.10, on the `Click` event of the button for displaying the total number of characters and words entered in the text box:

Listing 2.10: Showing the Code to Display the Total Number of Characters and Words

```
private void button1_Click(object sender, EventArgs e)
{
  int wrdcount = 0;
  int chrcount = 0;
  label1.Text = "";
  checkBox3.Checked = false;
  if (checkBox1.Checked == true)
  {
        chrcount = textBox1.Text.Length;
        label1.Text = "Total characters : " + chrcount;
  }

  if (checkBox2.Checked == true)
```

```
        {
                if (textBox1.Text.Trim().Length == 0)
                {
                        label1.Text = label1.Text + " " + "Total words : 0";
                }
                else
                {
                        string[] str = textBox1.Text.Split(' ');
                        wrdcount = str.Length;
                        label1.Text = label1.Text + " " + "Total words : " + wrdcount;
                }
        }

        if (checkBox1.Checked == true && checkBox2.Checked == true)
        {
                checkBox3.Checked = true;
        }
        if (checkBox1.Checked == false && checkBox2.Checked == false)
        {
                checkBox3.Checked = false;
        }
        if ((checkBox1.Checked == false && checkBox2.Checked == true) || (checkBox1.Checked ==
                true && checkBox2.Checked == false))
        {
                checkBox3.CheckState = CheckState.Indeterminate;
        }
    }
```

In Listing 2.10, you can see that the total number of characters are calculated using the `Length` property of the text box. However, for counting the total number of words, we first apply a condition to verify whether the user has entered any text in the text box. If no text has been entered, the total characters and total words are set to zero. If the text box has some text, the function counts the words separated by a space using the `Split()` method, and stores the resulting words in a `String` array. Later, the length of the array is stored in the `wrdcount` variable, which contains the number of words entered in the text box.

3. Press the `F5` key on the keyboard to run the application; you can see the result as shown in Figure 2.13:

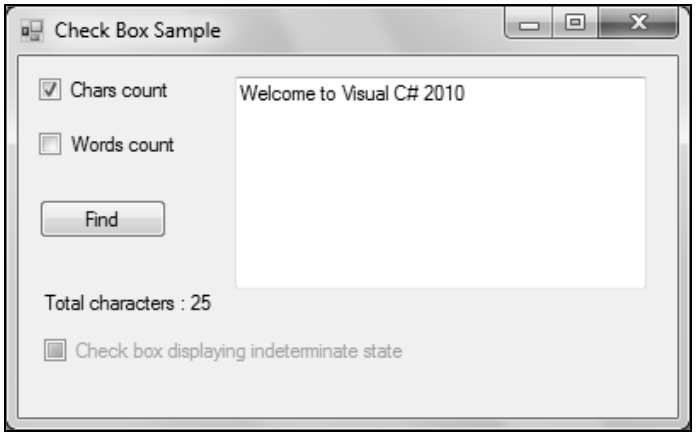

Figure 2.13: Displaying the Indeterminate State of the Check Box

As you can see in Figure 2.13, the Chars count check box is selected and displays the total number of characters entered in the text box. In addition, also notice that the last check box displays an indeterminate state. Now, select the Words count check box and click the Find button, as shown in Figure 2.14:

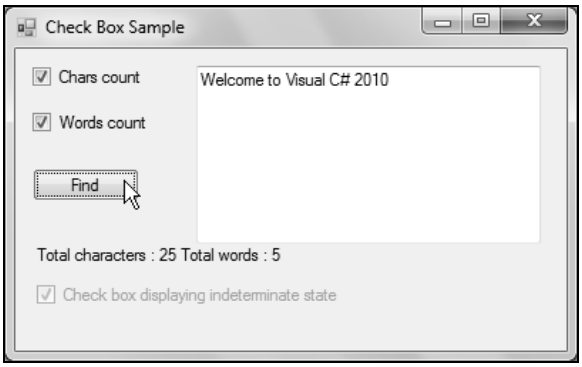

Figure 2.14: Displaying the Total Characters and Words in the Text Box

Now, you might be quite clear with the similarities and differences between the `RadioButton` and `CheckBox` controls. Let us now see how to use the `ListBox` control.

Using the ListBox Control

Using a list box, you can perform a number of actions, such as adding items to the list box, retrieving the number of items, sorting the items, and working with a selected item. Let us see each of these in detail by first creating an application called `ListBoxSample` (available in the CD-ROM). This application contains two `ListBox` controls, five `Button` controls, a `TextBox` control, and a blank `Label` control on the Form1 form. Now, set the Text property of the Button controls used in the application, as shown in Table 2.15:

Table 2.15: Showing the Text Property of the Button Controls

Text Property	Description
Fill list box	Fills the first list box with some items
Get indexes	Retrieves the indexes of all the items in the first list box and displays those indexes in the second list box
Sort items	Sorts the items in the first list box
Remove items	Removes a particular item from the first list box
Clear list box	Clears the first list box

Finally, a text box displays the selected item in a list box and a label displays the total number of items in the list box.

To create this application, perform the following broad-level steps:

1. Add items to the `ListBox` controls
2. Retrieve the index of items of the `ListBox` controls
3. Retrieve the number of items in the `ListBox` controls
4. Sort items in the `ListBox` controls
5. Retrieve the selected item of the `ListBox` controls
6. Create `ListBox` controls with multiple columns
7. Create multiselect list boxes
8. Handle the `SelectedIndexChanged` event
9. Remove items from the `ListBox` controls

Let's understand these in detail next.

Adding Items to the ListBox Controls

You can add items to a list box at either design time or run time. The first item in the list box has index 0, the next index 1, and so on. At design time, you can use the `Items` property, which stores array of items in the list box, and at run time, you can use both the `Items` property and the `Add()` method.

How do you keep track of the total number of items in a list box? You can use the `Items.Count` property, i.e., if you loop over the items in the control, use the `Items.Count` property as the maximum value to loop to. You can access items individually in a list box by index using the `Items` property, as shown the following code snippet:

```
strText = listBox1.Items[5].ToString ();
```

At design time, you can add items directly to your list box by typing them into the `Items` property in the Properties window. Selecting the `Items` property displays the String Collection Editor dialog box, and you can type item after item into the list box, as shown in Figure 2.15:

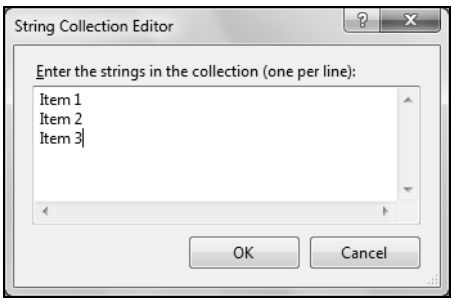

Figure 2.15: Adding Items to a List Box

At run time, you can use the `Add()` or `Insert()` method of the `Items` property to add items to the list box.

Perform the following steps to add items to a list box:

1. Double-click the Fill list box button in the design mode of the Form1.cs file and add the code shown in Listing 2.11:

Listing 2.11: Showing the Code to Add the Items to the List Box

```
private void button1_Click(object sender, EventArgs e)
{
  int intLoopIndex = 0;
  for (intLoopIndex = 1; intLoopIndex <= 20; intLoopIndex++)
  {
        listBox1.Items.Add("Item " + intLoopIndex.ToString());
  }
}
```

2. Press the `F5` key on the keyboard to execute the code and the output appears, as shown in Figure 2.16:

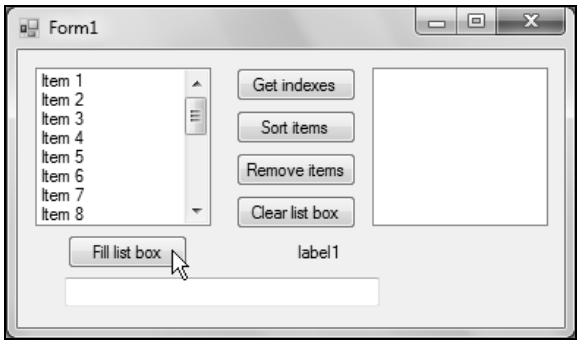

Figure 2.16: Adding Items in a List Box

Note that when you add items in a list box, they are stored by index. If you are required to add an item at a specified index, suppose at the middle of an already existing items list, you need to have two pieces of information – the item to be added and its index number. In such a case, you use the `Insert()` method for adding items at a specified index, as shown in the following code snippet:

```
listBox1.Items.Insert (3, "Item 3") ;
```

In this way, using the `Insert()` method, you can insert an item anywhere in between the list box.

Retrieving the Index of Items of the ListBox Controls

You can retrieve the index of items added in a list box, especially when you require inserting an item in between the list box at a specified index.

Now, let's learn how to retrieve the indexes of the items in a list box. To do so, double-click the Get indexes button in the design mode of the Form1.cs file and add the code, shown in Listing 2.12, on the Click event of the Get indexes button:

Listing 2.12: Showing the Code to Retrieve the Indexes of all the Items in a List Box

```
private void button2_Click(object sender, EventArgs e)
{
  int intLoopIndex = 0;
  for (intLoopIndex = 0; intLoopIndex < listBox1.Items.Count; intLoopIndex++)
  {
        listBox2.Items.Add("Index " + intLoopIndex.ToString());
  }
}
```

In Listing 2.12, notice that loop variable `intLoopIndex` has been initialized a value of 0. This is because an index of an item always starts with 0. In addition, also observe that the loop terminates at a number less than the total items in the list box.

Press the `F5` key on the keyboard to execute the application, when the form appears, click the Fill list box button and then click the Get indexes button. You can see the output, as shown in Figure 2.17:

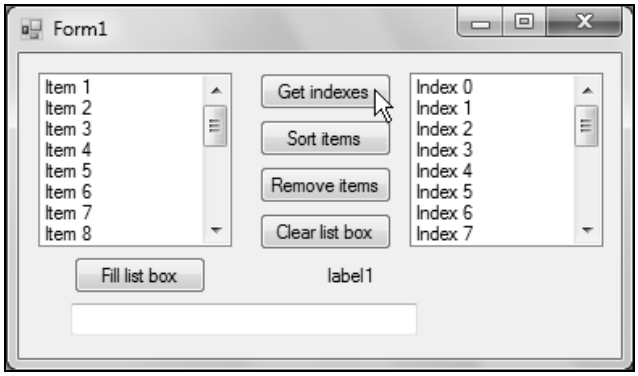

Figure 2.17: Retrieving all Item Indexes

Now using the indexes, you can very easily insert an item in between the list box.

Retrieving the Number of Items in the ListBox Controls

The `Count()` method is used to retrieve the total number of items in the list box. This is required in situations where you need to iterate through the items till the last item in the list.

Let's learn about retrieving the total number of items in a list box. To do so, double-click the Get indexes button in the design mode of the Form1.cs file and add the highlighted code at the `Click` event of the Get indexes button, as shown in Listing 2.13:

Listing 2.13: Showing the Code to Count the Total Number of Items of List Box

```
private void button2_Click(object sender, EventArgs e)
{
```

```
        int intLoopIndex = 0;
        for (intLoopIndex = 0; intLoopIndex < listBox1.Items.Count; intLoopIndex++)
        {
              listBox2.Items.Add("Index " + intLoopIndex.ToString());
        }
        label1.Text = "Total number of items : " + listBox1.Items.Count;
    }
```

Press the F5 key on the keyboard to execute the code. When the form appears, click the Fill list box button and then click the Get indexes button. You can see the result, as shown in Figure 2.18:

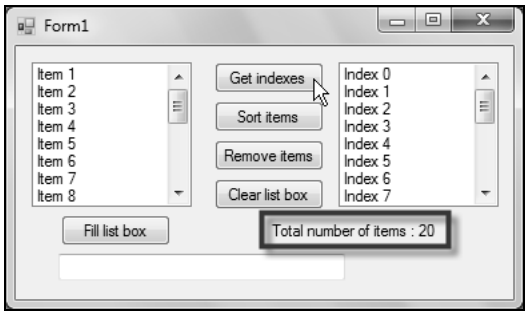

Figure 2.18: Showing the Total Number of Items

Let us now see how sorting of items in a list box is performed.

Sorting Items in the ListBox Controls

You can sort the items in a list box alphabetically by setting its Sorted property to true (it is false by default) at design time or run time.

To sort items in a list box, double-click the Sort items button in the design mode of the Form1.cs file and add the code on the Click event of the button, as shown in Listing 2.14:

Listing 2.14: Showing the Code for the Sorting the Items

```
    private void button3_Click(object sender, EventArgs e)
    {
      listBox1.Sorted = true;
    }
```

Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill list box button, Get indexes button, and Sort items button. You can see the result, as shown in Figure 2.19:

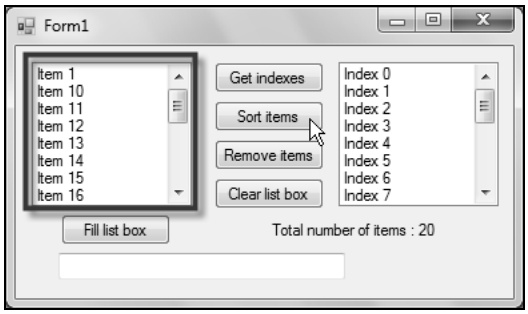

Figure 2.19: Displaying Sorted List

NOTE

You should know; however, that sorting a list box can change the indices of the items in that list box (unless they were already in alphabetical order). After the sorting is completed, the first item in the newly sorted list has index 0, the next index 1, and so on.

Retrieving the Selected Item of the ListBox Controls

If a list box supports only single selection, you can use the `SelectedIndex` property to get the index of the selected item, and the `SelectedItem` property to get the selected item in a list box.

Listing 2.15 shows how we can display a selected item with its index at the `Click` event of the list box:

Listing 2.15: Showing the Code to Display the Selected Items

```
private void listBox1_Click(object sender, EventArgs e)
{
   textBox1.Text = "The selected item : " + listBox1.SelectedItem.ToString() + " has an
   index : " + listBox1.SelectedIndex;
}
```

Press the F5 key on the keyboard to run the application. When the form appears, click the Fill list box button, Get indexes button, and finally select an item from the first list box. You can see the result, as shown in Figure 2.20:

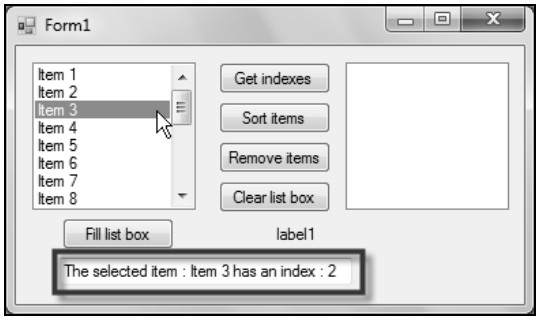

Figure 2.20: Displaying the Selected Item and its Index

NOTE

ListBox controls support multiple selections if you set their SelectionMode property to either MultiSimple or MultiExtended. Use the SelectedItems and SelectedIndices properties of the list box to make them multi-select. You can refer to the Creating Multiselect List Boxes section of this chapter for more information about selecting the multiple items in the ListBox control.

Creating ListBox Controls with Multiple Columns

You can display the items in the list box in multiple columns by setting its `MultiColumn` property to `true`. Now, on the Load event of the Form1 form, add the code, as shown in the following code snippet:

```
listBox1.MultiColumn = true;
```

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Fill list box button and the Get indexes button. The output is displayed, as shown in Figure 2.21:

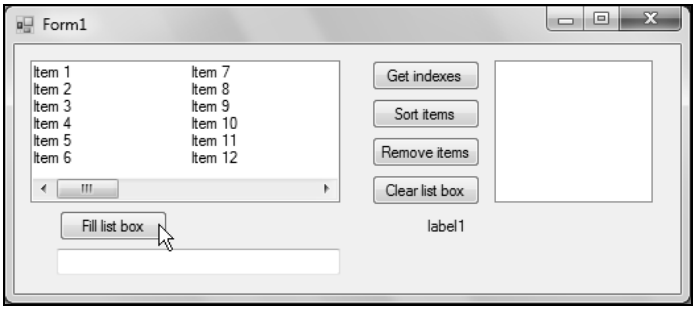

Figure 2.21: Displaying Multi-column List Box

Creating Multiselect List Boxes

You need to set the `SelectionMode` property of the list box to `MultiExtended` to make the list box a multi-select list box. Following are the possible values to be set in the `SelectionMode` property:

- `MultiExtended`—Enables selection of multiple items, and the user can use the `SHIFT`, `CTRL`, and arrow keys to make selections
- `MultiSimple`—Enables selection of multiple items
- `None`—Prevents you to select items
- `One`—Enables selection of only one item

To enable multiple selection, add the following code snippet on the Load event of the Form1 form:

```
private void Form1_Load(object sender, EventArgs e)
{
   listBox1.MultiColumn = true;
   listBox1.SelectionMode = SelectionMode.MultiExtended;
}
```

To indicate how multiple selections look, we also use the `SetSelected()` method of the list box, which you can use to set some items in the list box selected. The code for selecting the items of a list box at runtime is highlighted in Listing 2.16:

Listing 2.16: Showing the Code for Selecting Multiple Items in the List Box

```
private void button1_Click(object sender, EventArgs e)
{
   int intLoopIndex = 0;
   for (intLoopIndex = 1; intLoopIndex <= 20; intLoopIndex++)
   {
         listBox1.Items.Add("Item " + intLoopIndex.ToString());
   }

   listBox1.SetSelected(1, true);
   listBox1.SetSelected(3, true);
}
```

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Fill list box button, the Get indexes button, and finally click the items in the first list box. You can see the output, as shown in Figure 2.22:

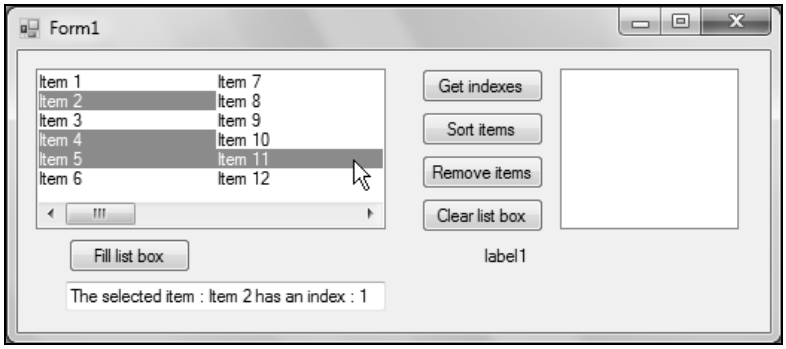

Figure 2.22: Showing the Multiselect, Multicolumn List Box

As you can see in Figure 2.22, the list box supports multiple columns and selections.

Handling the SelectedIndexChanged Event

You can use the `SelectedIndexChanged` event, which is the default event for list boxes, to handle the case where the selected item changes in a list box. Add the code, shown in Listing 2.17, on the SelectedIndexChanged event of the ListBox control to see how to handle the `SelectedIndexChanged` event:

Listing 2.17: Handling the `SelectedIndexChanged` Event

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
            if (listBox1.SelectedItems.Count > 1)
            {
                textBox1.Text = "";
```

```
                    textBox1.Text = "Selected items: ";
                    foreach (var Item in listBox1.SelectedItems)
                    {
                        textBox1.Text += Item.ToString() + " ";
                    }
                }
    }
```

Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill list box button, the Get indexes button, and finally click some items in the first list box. You can see that the selected items appear in the text box, as shown in Figure 2.23:

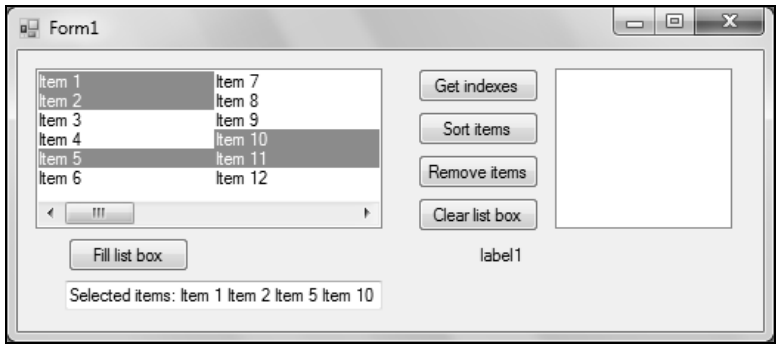

Figure 2.23: Handling List Box Events

In Figure 2.23, all the selected items in the list box are displayed in the text box.

Removing Items from the ListBox Controls

You can remove one item at a time from a list box using either the Remove() or RemoveAt() method. To remove items from the list box, perform the following steps:

1. Double-click the Remove items button in the design mode of the Form1.cs file and add the following code snippet on the Click event of the button:

```
listBox1.Items.Remove("Item 6");
listBox1.Items.RemoveAt(0);
```

In the preceding code snippet, the Remove() method takes the item to be removed as a parameter; whereas, the RemoveAt() method takes the index of the item as a parameter.

2. Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill list box button and then click the Remove items button. The result is, as shown in Figure 2.24:

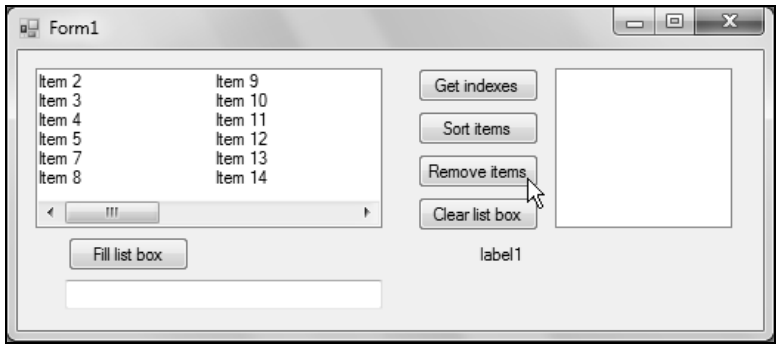

Figure 2.24: Removing Selected Items from the List Box

In Figure 2.24, the Remove items button has removed items Item 1 and Item 6.

Similarly, you can use the Clear() method of the Items collection to clear a list box.

3. Double-click the Clear list box button in the design mode of the Form1.cs file and add the following code snippet on the Click event of the button control:

```
listBox1.Items.Clear();
textBox1.Text = "";
```

4. Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill list box button and then the Clear list box button to clear all the items from the list box, as shown in Figure 2.25:

Figure 2.25: Showing an Empty List Box

What you have seen are some of the most common properties, methods, and events associated with a list box. Now, let us discuss the CheckedListBox control.

Using the CheckedListBox Control

As discussed in the *In Depth* section of this chapter, checked list boxes support check boxes for each item in a list. In a checked list box, you can add items, set and retrieve the check state of the items, and handle its ItemCheck event, which you will see in the coming topics.

Let us explore each of these in detail by first creating an application, named CheckedListBoxSample (available in the CD-ROM). In this application, add a CheckedListBox control, three Button controls, and a TextBox control. Change the Text property of the Button controls to Fill items, Uncheck items, and Find checked, respectively.

To create this application, perform the following broad-level steps:

1. Add items to the CheckedListBox controls
2. Set the check state of items in the CheckedListBox controls
3. Retrieve the check state of items in the CheckedListBox controls
4. Handle the ItemCheck event of the CheckedListBox controls

Let's discuss each of these in detail next.

Adding Items to the CheckedListBox Controls

You can add items to the CheckedListBox control similar to the ListBox control. However, you can also pass another argument to the Items.Add() method call at the time of adding the items in the checked list box. With this argument, you specify whether the added item is selected by default or not. Now, double-click the Fill item button in the design mode of the Form1.cs file and add the code, shown in Listing 2.18, on the Click event of the button:

Listing 2.18: Showing the Code for Adding Items to the Checked List Box

```
private void button1_Click(object sender, EventArgs e)
{
  checkedListBox1.Items.Clear();
  checkedListBox1.Items.Add("Mango", true);
  checkedListBox1.Items.Add("Grapes", true);
  checkedListBox1.Items.Add("Orange", false);
  checkedListBox1.Items.Add("Apple", false);
```

```
      checkedListBox1.Items.Add("Banana", true);
      checkedListBox1.Items.Add("Guava", false);
    }
```

Press the F5 key on the keyboard to execute the application. You can see the result of this code by clicking the Fill items button, as shown in Figure 2.26:

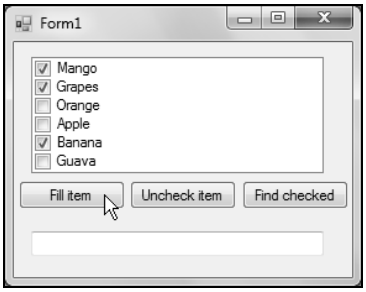

Figure 2.26: Showing a Checked List Box

Let us now see how to set the check state of items in a checked list box.

Setting the Check State of Items in the CheckedListBox Controls

You can use the SetItemChecked() method of the checked list box to select or clear items in a checked list box by passing a value of true or false, respectively. In this example, we are clearing the selection of some of the selected items in the checked list box. To do so, double-click the Uncheck item button in the design mode of the Form1.cs file and add the following code snippet on the Click event of the button for clearing the selecting of items:

```
    checkedListBox1.SetItemChecked(1, false);
    checkedListBox1.SetItemChecked(4, false);
```

Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill Item button and then the Uncheck item button. Clicking the Uncheck item button clear the items, as shown in Figure 2.27:

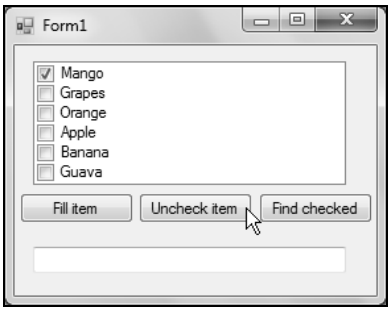

Figure 2.27: Clearing the Selection of Items in Checked List Box

Retrieving the Check State of Items in the CheckedListBox Controls

You can determine whether an item displays a checkmark in a checked list box using the GetItemChecked() method, which returns true if an item is selected. For example, in the CheckedListBoxSample example, you can loop over all items in the checked list box and display the selected items in a text box. To do so, double-click the Find checked button in the design mode of the Form1.cs file and add the code on its Click event, as shown in Listing 2.19:

Listing 2.19: Showing the Code for Retrieving the Check State of Items

```
    private void button3_Click(object sender, EventArgs e)
    {
      int intLoopIndex = 0;
      string strText = null;
      strText = "Checked Items: ";
```

```
    for (intLoopIndex = 0; intLoopIndex <= (checkedListBox1.Items.Count - 1);
     intLoopIndex++)
    {
          if (checkedListBox1.GetItemChecked(intLoopIndex) == true)
          {
                strText += checkedListBox1.Items[intLoopIndex].ToString() + " ";
          }
    }
    textBox1.Text = strText;
}
```

Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill Item button and then the Find checked button. You can see the results in Figure 2.28, where the text box displays the checked items:

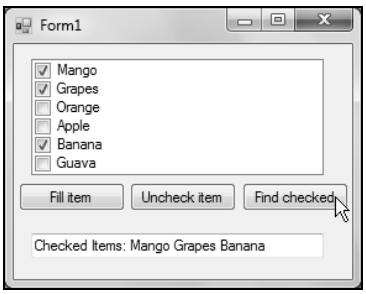

Figure 2.28: Showing the Selected Items in a Text Box

There is an alternate way to find the selected items in checked list box by using the CheckedItems property, which holds all the selected items. Listing 2.20 shows the code to find the selected item:

Listing 2.20: Showing the Code for Retrieving Checked Items using the CheckedItems Property

```
private void button3_Click(object sender, EventArgs e)
{
   string strText = null;
   strText = "Checked Items: ";
   foreach (string strData in checkedListBox1.CheckedItems) {
          strText += strData + " ";
   }
   textBox1.Text = strText;
}
```

Besides CheckedItems, the CheckedIndices property also returns a collection holding the indices of the checked items in the checked list box.

Handling the ItemCheck Event of the CheckedListBox Controls

The ItemCheck event occurs when the checkmark in front of an item in the checked list box changes. The ItemCheckEventArgs object passed to the associated event handler has an Index member that gives you the index of the item whose check mark has changed, and a NewValue member that gives you the new setting for the item (which is a member of the CheckState enumeration: Checked, Indeterminate, or Unchecked—note that the item does not yet have this setting when you are handling this event; it is assigned to the item after the event handler terminates).

Let's learn how to check if an item has been selected or cleared by adding the code, shown in Listing 2.21, on the ItemCheck event of the CheckedListBox control:

Listing 2.21: Showing the Code for the ItemCheck Event of the CheckedListBox Control

```
private void checkedListBox1_ItemCheck(object sender, ItemCheckEventArgs e)
{
   switch (e.NewValue) {
          case CheckState.Checked:
                textBox1.Text = "Item " + (e.Index + 1) + " is selected";
                break;
          case CheckState.Unchecked:
```

```
                    textBox1.Text = "Item " + (e.Index + 1) + " is not selected";
                    break;
            }
        }
```

Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill Item button and then select an item from the checked list box. You can see the result in Figure 2.29, where we have just checked Apple in the checked list box, and the same have been shown in the text box:

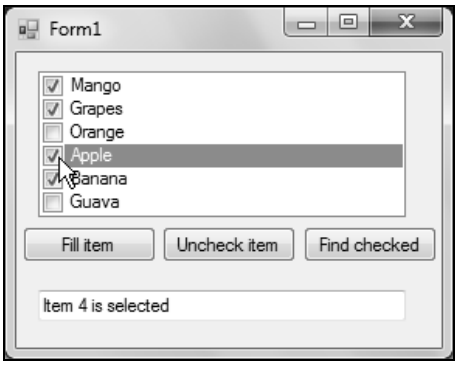

Figure 2.29: Showing the Selected Item in the Text Box

NOTE

We have set the CheckOnClick property for the CheckedListBoxSample example in this chapter to true (the default setting for this property is false), so all it takes is one click to generate an ItemCheck event. By default, however, a double-click would be needed.

Using the ComboBox Control

As discussed in the *In Depth* section of this chapter, combo boxes combine a text box and a list box (which is why they are called combo boxes). Similar to a list box, you can add items, work with the selected item, retrieve the total number of items, and sort items in a combo box. To do so, create an application named ComboBoxSample (available in CD-ROM). In this application, you will find a combo box and four buttons. Set the following text of these buttons used in the application, as follows:

- **Fill combo box**—Fills the combo box with items
- **Get selected**—Gets the items selected in the combo box
- **Sort items**—Sorts the items in the combo box
- **Storing objects**—Stores objects in the combo box

Finally, a text box is used to display the selected items in the combo box.

The following broad-level steps need to be performed to create this application:

1. Adding items in the ComboBox controls
2. Setting the drop-down style of the ComboBox controls
3. Setting the selected item of the ComboBox controls
4. Retrieving the number of items in the ComboBox controls
5. Sorting the items in a ComboBox control
6. Storing objects in the ComboBox controls
7. Using the AddRange() method
8. Handling the TextChanged and SelectedIndexChanged events

Let's discuss each of these in detail next.

Adding Items in the ComboBox Controls

As already learned, a combo box is a combination of a text box and a list box, so at design time you can change the text in the text box part by changing the `Text` property of the combo box. You can change the items in the list box part with the `Items` property (this item opens the String Collection Editor dialog box discussed for list boxes when you click it in the Properties window) at design time.

As with list boxes, you can also use the `Items.Insert()`, `Items.Add()`, and `Items.AddRange()` methods to add items to the list part of a combo box. Now, double-click the Fill combo box button in the design mode of the Form1.cs file and add the code, shown in Listing 2.22, on the Click event of the button for adding items in the ComboBox control:

Listing 2.22: Showing the Code for Adding Items in a Combo Box

```
private void button1_Click(object sender, EventArgs e)
{
  comboBox1.Items.Clear();
  int intLoopIndex = 0;
  for (intLoopIndex = 0; intLoopIndex <= 20; intLoopIndex++)
  {
        comboBox1.Items.Add("Item " + intLoopIndex.ToString());
  }
  comboBox1.Text = "Select one...";
}
```

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Fill combo box button. You can see the result in Figure 2.30:

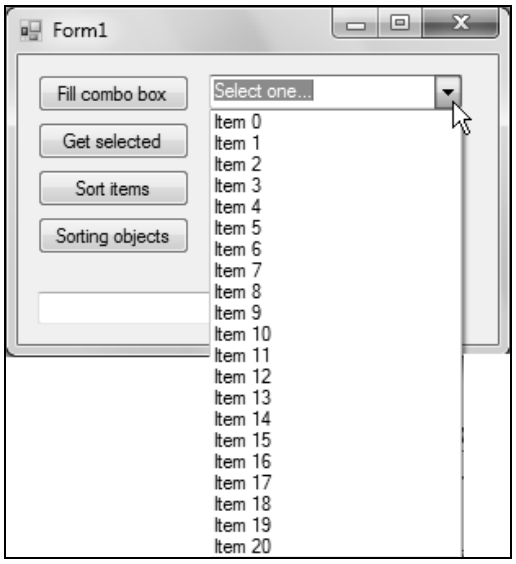

Figure 2.30: Populating Items in a Combo Box

Let us now see how to specify different styles to the combo box.

Setting the Drop-down Style of the ComboBox Controls

You might think there is only one kind of combo box, but there are really three types. You can select the type you want with the `DropDownStyle` property of the combo box.

The default type of combo box is made up of a text box and a drop-down list. However, you can also have combo boxes where the list does not drop down (the list is always open, and you have to make sure to provide space for it when you add the combo box to your form), and combo boxes where the user can only select from

the list. Following are the settings for the DropDownStyle property of the combo box (these are members of the ComboBoxStyle enumeration):

- DropDown (default value)—Includes a drop-down list and a text box. The user can select from the list or type in the text box.
- Simple—Includes a text box and a list, which does not drop down. The user can select from the list or type in the text box. The size of a simple combo box includes both the edit and list portions. By default, a simple combo box is sized so that none of the list is displayed. Increase the Height property to display more of the list.
- DropDownList—Includes only drop-down list and the user can only select the existing value from the drop-down list. This is a good one to keep in mind when you want to restrict the input from the user.

You can see a Simple dropdown style combo box in Figure 2.31, created by setting the DropDownStyle property to Simple in the Properties window of the combo box:

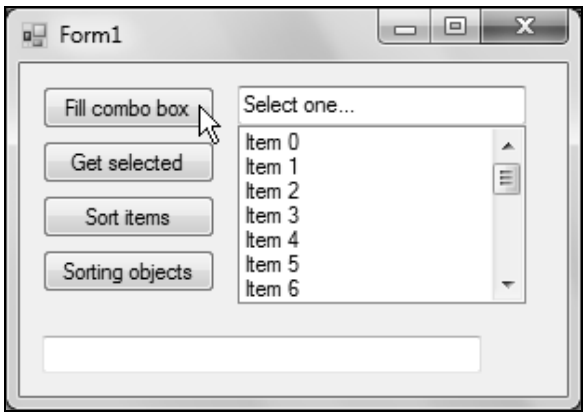

Figure 2.31: Displaying a Simple Style Combo Box

The next topic deals with setting the selected items in a combo box.

Setting the Selected Item of the ComboBox Controls

When you make a selection in a combo box, the new selection appears in the text box of the combo box. Therefore, it is easy to get the text of the current selection—you just use the Text property of the combo box.

You can also use the SelectedIndex and SelectedItem properties to get the index of the selected item and its value, respectively. In the ComboBoxSample example, we display the information in about the currently selected item in a combo box, when the user clicks the Get selected button. Now, add the code, shown in Listing 2.23, on the Click event of the Get selected button:

Listing 2.23: Showing the Code to Display Selected Item and Selected Index of Combo Box

```
private void button2_Click(object sender, EventArgs e)
{
  if (comboBox1.SelectedIndex > -1)
  {
        int selectedIndex = 0;
        selectedIndex = comboBox1.SelectedIndex;
        object selectedItem = null;
        selectedItem = comboBox1.SelectedItem;
        textBox1.Text = "Selected item text: " + selectedItem.ToString() + " Selected
        index: " + selectedIndex.ToString();
  }
}
```

Press the F5 key on the keyboard to execute the application. When the form appears, click the Fill combo box button and then the Get selected button. The result is displayed, as shown in Figure 2.32:

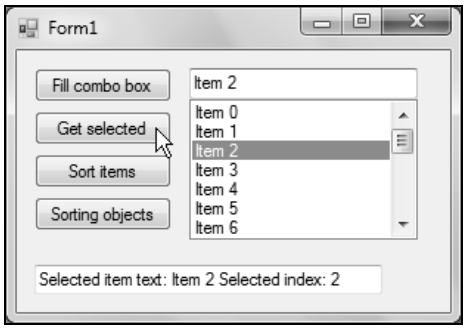

Figure 2.32: Displaying the Selected Item and Selected Index of the Combo Box in a Text Box

TIP

If you want to restrict the users to insert the item in the combo box or to modify the existing item of the combo box, set the DropDownStyle property of the combo box to DropDownList. In this style of combo boxes, the user cannot type into the text part of the control.

Retrieving the Number of Items in the ComboBox Controls

A combo box can contain various items, and counting all the items may consume more time and efforts. This problem can be solved with the help of the `Item.Count` property of the combo box. The `Item.Count` property counts all the items listed in the combo box and gives you the result. Now, add the highlighted code on the Click event of the Fill combo box button, as shown in Listing 2.24:

Listing 2.24: Displays the Total Number of Items in a Combo Box

```
private void button1_Click(object sender, EventArgs e)
{
   comboBox1.Items.Clear();
   int intLoopIndex = 0;
   for (intLoopIndex = 0; intLoopIndex <= 20; intLoopIndex++) {
         comboBox1.Items.Add("Item " + intLoopIndex.ToString());
   }
   comboBox1.Text = "Select one...";
   MessageBox.Show("The combo box contains " + comboBox1.Items.Count + " items.");
}
```

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Fill combo box button. The result is displayed, as shown in Figure 2.33:

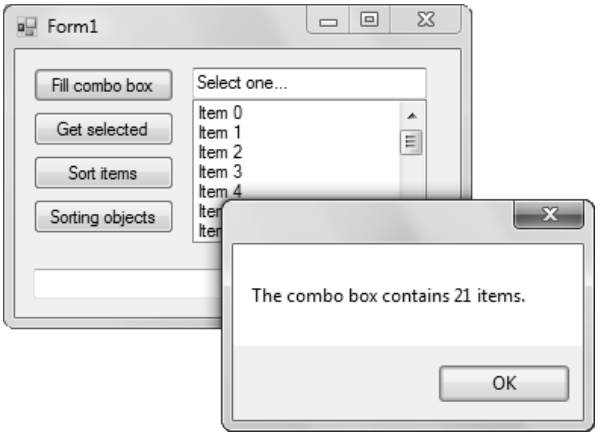

Figure 2.33: Displaying the Total Number of Items in the Combo Box

The following topic deals with the sorting of the items in a combo box.

Sorting the Items in a ComboBox Control

The use of sorting can easily be understood by considering a scenario. Suppose, you have been newly commissioned to write the guidebook to the zoo with Visual C# and everything looks great—except for one thing. The program features a combo box with a list of animals that the user can select to learn more about, and it would be great if you could make that list appear in an alphabetical order—but the zoo keeps adding and trading animals all the time. Still, it is no problem, because you can leave the work up to the combo box itself if you set its `Sorted` property to `true` (the default is `false`).

For example, we set the `Sorted` property to `true` for a combo box, `ComboBox1`. Now, it does not matter in what order you add items to that combo box. To do so, double-click the Sort items button in the design mode of the Form1.cs file and add the following code snippet on its Click event:

```
private void button3_Click(object sender, EventArgs e)
{
 comboBox1.Sorted = true;
}
```

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Fill combo box button and then the Sort items button. The combo box appears sorted, as shown in Figure 2.34:

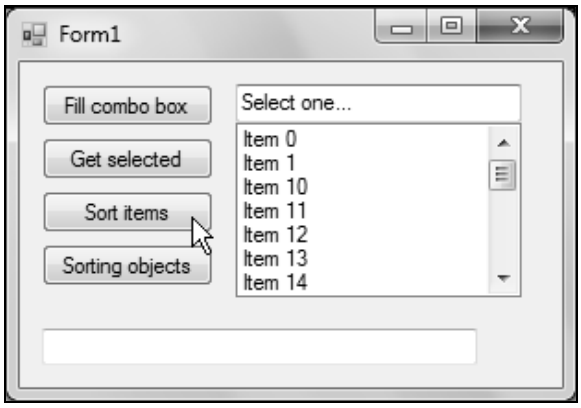

Figure 2.34: Sorting the Items in a Combo Box

TIP

You should know; however, that sorting a combo box can change the indices of the items in that combo box (unless they were already in alphabetical order). After the sorting is finished, the first item in the newly sorted combo list has index 0, the next index 1, and so on.

Storing Objects in the ComboBox Controls

Besides storing the data of primitive data types, you can also store the objects of a class in a combo box control. Let's create a new class, `DataItem`, in the `ComboBoxSample` application and store its objects in the `ComboBox` control. Listing 2.25 shows the code of the `DataItem` class:

Listing 2.25: Showing the Code of the DataItem Class

```
public class DataItem
{
 private float Data;
 private string Name;
 public DataItem(string NameArgument, float Value)
 {
        Name = NameArgument;
        Data = Value;
 }
 public override string ToString()
```

```
    {
            return System.Convert.ToString(Name);
    }
    public float GetData()
    {
            return Data;
    }
   }
```

In Listing 2.25, the `ToString()` method of the `DataItem` class is overridden because this method is called when the combo box needs to display the name of each item. In addition, a `GetData()` method is also added to get the internal, private data from the objects.

When the form loads, we can create 21 objects of the `DataItem` class—for example item 5 is named Item 5 and store the internal value `5`—and place them in the combo box with the `Items.Add()` method. To do so, double-click the Store objects button in the design mode of the Form1.cs file and add the code, shown in Listing 2.26, on its Click event:

Listing 2.26: Showing the Code to Store Objects using the `Add()` Method

```
private void button4_Click(object sender, EventArgs e)
{
 comboBox1.Items.Clear();
 DataItem[] Objects = new DataItem[21];
 int intLoopIndex = 0;
 for (intLoopIndex = 0; intLoopIndex <= 20; intLoopIndex++)
 {
        Objects[intLoopIndex] = new DataItem("Item " + intLoopIndex,
        System.Convert.ToSingle(intLoopIndex));
        comboBox1.Items.Add(Objects[intLoopIndex]);
 }
 comboBox1.Text = "Select one...";
}
```

You can use another method to add objects to the combo box that is the `AddRange()` method. The following topic deals with the same.

*Using the **AddRange**() Method*

Since we have stored the `DataItem` objects in an array (named `Objects`), there is another way of adding these items to a combo box or list box—you can use the `AddRange()` method to add all the objects in the `Objects` array to the combo box simultaneously, as shown in Listing 2.27:

Listing 2.27: Showing the Code to Store Objects using the `AddRange()` Method

```
private void button4_Click(object sender, EventArgs e)
{
 comboBox1.Items.Clear();
 DataItem[] Objects = new DataItem[21];
 int intLoopIndex = 0;
 for (intLoopIndex = 0; intLoopIndex <= 20; intLoopIndex++)
 {
        Objects[intLoopIndex] = new DataItem("Item " + intLoopIndex,
        System.Convert.ToSingle(intLoopIndex));
 }
 comboBox1.Items.AddRange(Objects);
 comboBox1.Text = "Select one...";
}
```

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Storing objects button. When you execute the code either using the `Add()` method or the `AddRange()` method, the output appears as shown in Figure 2.35:

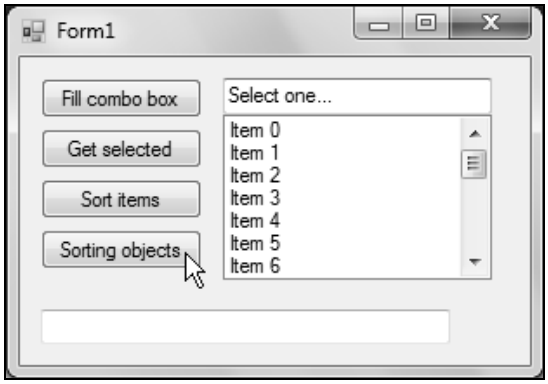

Figure 2.35: Storing Objects in Combo Box

Let us learn some of the frequently used events of the combo box, such as `TextChanged` and `SelectedIndexChanged` events.

*Handling the **TextChanged** and **SelectedIndexChanged** Events*

Combo boxes are combinations of text boxes and list boxes, and the combination means that there are two sets of input events:

- `TextChanged` event—Occurs when the user types into the text box
- `SelectedIndexChanged` event—Occurs when the user uses the list box part of the combo box

NOTE

Unlike standard list boxes, you cannot make multiple selections in a list box of the combo box.

The **TextChanged** Event

When the user changes the text in a combo box, a `TextChanged` event occurs in the same way as it occurs when the user types in a standard text box. Write the code on the TextChanged event of the ComboBox control, as shown in the following code snippet:

```
private void comboBox1_TextChanged(object sender, EventArgs e)
{
 textBox1.Text = comboBox1.Text;
}
```

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Fill combo box button and type a new item in the text box of the combo box, as shown in Figure 2.36:

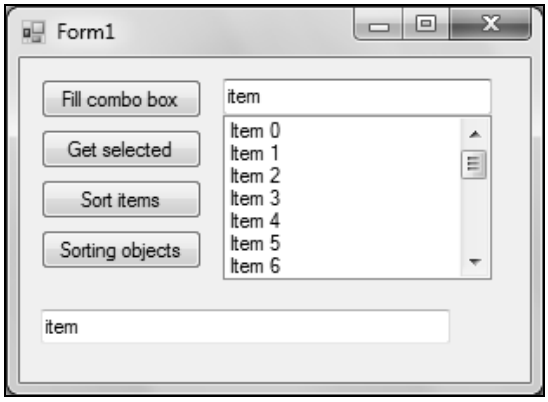

Figure 2.36: Displaying the Text in the Text Box

In Figure 2.36, whatever you type in the combo box is automatically displayed in the text box simultaneously.

The **SelectedIndexChanged** Event

When the selection changes in a combo box, a `SelectionChanged` event occurs, and you can use the `SelectedIndex` and `SelectedItem` properties to get the index of the currently selected item and the item itself. Generate the SelectedIndexChanged event of the combo box and add the code, as shown in Listing 2.28:

Listing 2.28: Showing the Code for the `SelectedIndexChanged` Event of the Combo Box

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
 if (comboBox1.SelectedIndex > -1)
 {
        if (comboBox1.SelectedItem.GetType() == typeof(DataItem))
        {
                textBox1.Text = "The data for the item you selected is: " +
                ((DataItem)comboBox1.SelectedItem).GetData();
        }
 }
}
```

In Listing 2.28, when the user selects an item in the combo box, we can use the `SelectedItem` property to get the selected object, and the `GetData()` method of the selected object to get its stored data (note that we must cast the item to an object of the `DataItem` class first), which we display in a text box.

Press the `F5` key on the keyboard to execute the application. When the form appears, click the Storing objects button and then select an item from the combo box, as shown in Figure 2.37:

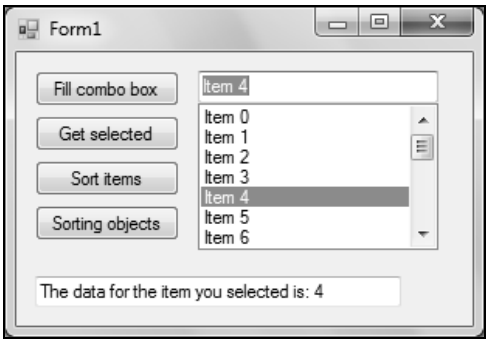

Figure 2.37: Displaying the Value in the TextBox on Selecting the Combo Box Items

Now, let's summarize the main topics discussed in this chapter.

Summary

In this chapter, you have learned about some standard Windows Forms controls, such as RadioButton, CheckBox, ListBox, CheckedListBox, and ComboBox. You have also learned about their properties, methods, and events. Finally, you have learned to implement and use these form controls in various applications.

In the next chapter, we continue our discussion on Windows Forms controls and learn about another set of standard controls, such as ListView, TreeView, ImageList, PictureBox, Panel, GroupBox, and TabControl.