UNIT-II

SYLLABUS

Basic Windows Controls: Textbox Control- ListBox, CheckedListBox-Scrollbar and TrackBar Controls-More Windows Control-The common Dialog Controls-The Rich TextBox Control - Handling Strings, characters and Dates. The TreeView and ListView Controls: Examining the Advanced Controls-The TreeView Control-The ListView Control

Basic Windows Controls

The TextBox Control

- The TextBox control is the primary mechanism for displaying and entering text and is one of the most common elements of the Windows user interface.
- The TextBox control is a small text editor that provides all the basic text-editing facilities: inserting and selecting text, scrolling if the text doesn't fit in the control's area, and even exchanging text with other applications through the Clipboard.

Basic Properties

MultiLine

This property determines whether the TextBox control will hold a single line or multiple lines of text. By default, the control holds a single line of text. To change this behavior, set the MultiLine property to True.

ScrollBars

This property controls the attachment of scroll bars to the TextBox control if the text exceeds the control's dimensions.

Single-line text boxes can't have a scroll bar attached, even if the text exceeds the width of the control. Multiline text boxes can have a horizontal or a vertical scroll bar, or both.

If you attach a horizontal scroll bar to the TextBox control, the text won't wrap automatically as the user types. To start a new line, the user must press Enter.

WordWrap

This property determines whether the text is wrapped automatically when it reaches the right edge of the control. The default value of this property is True. If the control has a horizontal scroll bar, however, you can enter very long lines of text. The contents of the control will scroll to the left, so that the insertion point is always visible as you type. You can turn off the horizontal scroll bar and still enter long lines of text; just use the left/right arrows to bring any part of the text into view.

AcceptsReturn, AcceptsTab

These two properties specify how the TextBox control reacts to the Return (Enter) and Tab keys. The Enter key activates the default button on the form, if there is one. The default button is usually an OK button that can be activated with the Enter key, even if it doesn't have the focus. In a multiline TextBox control, however, we want to be able to use the Enter key to change lines.

The AcceptsTab property determines how the control reacts to the Tab key. Normally, the Tab key takes you to the next control in the tab order. In a TextBox control, however, you may wish for the Tab key to insert a Tab character in the text of the control instead; to do this, set this property to True. The default value of the AcceptsTab property is False, so that users can move to the next control with the Tab key.

MaxLength

This property determines the number of characters the TextBox control will accept. Its default value is 32,767, which was the maximum number of characters the VB6 version of the control could hold. Set this property to zero, so that the text can have any length, up to the control's capacity limit 2 GB, or 2,147,483,647 characters to be exact.

Text-Manipulation Properties

Most of the properties for manipulating text in a TextBox control are available at runtime only.

Text

The most important property of the TextBox control is the Text property, which holds the control's text. This property is also available at design time so that you can assign some initial text to the control.

There are two methods of setting the Text property at design time.

For single-line TextBox controls, set the Text property to a short string, as usual. For multiline TextBox controls, open the Lines property and enter the text on the String Collection Editor window, which will appear.

At runtime, use the Text property to extract the text entered by the user or to replace the existing text by assigning a new value to the property. The Text property is a string and can be used as argument with the usual string-manipulation functions of Visual Basic. It also supports all the members of the String class.

The following expression returns the number of characters in the *TextBox1* control:

```
Dim strLen As Integer
strLen = TextBox1.Text.Length
```

VB6 programmers are accustomed to calling the Len() function, which does the same:

```
strLen = Len(TextBox1.Text)
```

To clear the control, you can set its Text property to a blank string:

```
TextBox1.Text = ""
```

or call the control's Clear method:

```
TextBox1.Clear
```

The IndexOf method of the String class will locate a string within the control's text. The following statement returns the location of the first occurrence of the string "Visual" in the text:

```
Dim location As Integer
location = TextBox1.Text.IndexOf("Visual")
                    Or
location = Instr(TextBox1.Text, "Visual")
```

The InStr() function allows to specify whether the search will be case-sensitive or not, while the IndexOf method doesn't.

To store the control's contents in a file, use a statement such as

```
StrWriter.Write(TextBox1.Text)
```

To retrieve the contents of the file, use a statement such as

```
TextBox1.Text = StrReader.ReadToEnd
```

where *StrReader* and *StrWriter* are two properly declared StreamReader and StreamWriter object variables.

Locating a String in a TextBox

```
Dim startIndex = -1
startIndex = TextBox1.Text.IndexOf("basic", startIndex + 1)
While startIndex > 0
Console.WriteLine("String found at " & startIndex)
startIndex = TextBox1.Text.IndexOf("basic", startIndex + 1)
End While
```

The following statement appends a string to the existing text on the control:

```
TextBox1.Text = TextBox1.Text & newString
```

To append a string to a TextBox control, use the following statement:

```
TextBox1.AppendText(newString)
```

ReadOnly, Locked

If you want to display text on a TextBox control but prevent users from editing it, you can set the ReadOnly property to True.

To prevent the editing of the TextBox control with VB6, you had to set the Locked property to True.

Lines

Lines is a string array where each element holds a line of text. The first line of the text is stored in the element Lines(0), the second line of text is stored in the element Lines(1), and so on. You can iterate through the text lines with a loop like the following:

```
Dim iLine As Integer
For iLine = 0 To TextBox1.Lines.GetUpperBound(0)– 1
{ process string TextBox1.Lines(iLine) }
Next
```

PasswordChar

Available at design time, this property turns the characters typed into any character you specify.

Text-Selection Properties

The TextBox control provides three properties for manipulating the text selected by the user: SelectedText, SelectionStart, and SelectionLength.

SelectedText

This property returns the selected text, enabling you to manipulate the current selection from within your code. For example, you can replace the selection by assigning a new value to the SelectedText property. To convert the selected text to uppercase, use the ToUpper method of the String class.

```
TextBox1.SelectedText = TextBox1.SelectedText.ToUpper
```

or use the UCase() function of VB6:

```
TextBox1.SelectedText = UCase(TextBox1.SelectedText)
```

To delete the current selection, assign an empty string to the SelectedText property:

```
TextBox1.SelectedText = “”
```

SelectionStart, SelectionLength

The SelectionStart property returns or sets the position of the first character of the selected text in the control’s text.

The SelectionLength property returns or sets the length of the selected text.

```
Dim seekString As String
Dim textStart As Integer
seekString = “Visual”
textStart = TextBox1.Text.IndexOf(seekString)
If textStart > 0 Then
TextBox1.SelectionStart = selStart – 1
TextBox1.SelectionLength = seekString.Length
End If
TextBox1.ScrollToCaret()
```

HideSelection

The selected text on the TextBox will not remain highlighted when the user moves to another control or form. To change this default behavior, use the HideSelection property. It keeps text highlighted in a TextBox control while another form or a dialog box has the focus.

Text-Selection Methods

In addition to properties, the TextBox control exposes two methods for selecting text. You can select some text with the Select method, whose syntax is shown next:

```
TextBox1.Select(start, length)
```

The Select method is new to VB.NET and is equivalent to setting the SelectionStart and SelectionLength properties. To select the characters 100 through 105 on the control, call the Select method, passing the values 99 and 6 as arguments:

TextBox1.Select(99, 6)

If the TextBox control contains the string “ABCDEFGHI,” then the following statement will select the range “DEFG”:

TextBox1.Select(3, 4)

The following two statements select the text on a TextBox control with the SelectionStart and SelectionLength properties:

TextBox1.SelectionStart = selStart – 1
TextBox1.SelectionLength = word.Length

These two lines can be replaced with a single call to the Select method:

TextBox1.Select(selStart – 1, word.Length)

where *word* is a string variable holding the selection.

A variation of the Select method is the SelectAll method, which selects all the text on the control.

Undoing Edits

An interesting feature of the TextBox control is that it can automatically undo the most recent edit operation. To undo an operation from within your code, you must first examine the value of the CanUndo property. If it’s True, it means that the control can undo the operation; then you can call the Undo method to undo the most recent edit.

The ListBox, CheckedListBox, and ComboBox Controls

The ListBox, CheckedListBox, and ComboBox controls present lists of choices, from which the user can select one or more.

The ListBox control occupies a user-specified amount of space on the form and is populated with a list of items. If the list of items is longer than can fit on the control, a vertical scroll bar appears automatically. The items must be inserted in the ListBox control through the code or via the Properties window. To add items at design time, locate the Items property in the control's Properties window and click the button with the ellipsis. A new window will pop up, the String Collection Editor window, where you can add the items you want to display on the list.

The ComboBox control also contains multiple items but typically occupies less space on the screen. The real advantage to the ComboBox control, however, is that the user can enter new information in the ComboBox, rather than being forced to select from the items listed.

Basic Properties

The ListBox and ComboBox controls provide a few common properties that determine the basic functionality of the control and are set at design time.

IntegralHeight

This property is a Boolean value (True/False) that indicates whether the control's height will be adjusted to avoid the partial display of the last item. When set to True, the control's actual height may be slightly different than the size you've specified, so that only an integer number of rows are displayed.

Items

The Items property is a collection that holds the items on the control. At design time, you can populate this list through the String Collection Editor window. At runtime you can access and manipulate the items through the methods and properties of the Items collection.

MultiColumn

A ListBox control can display its items in multiple columns, if you set its MultiColumn property to True.

SelectionMode

This property determines how the user can select the list's items and must be set at design time. The SelectionMode property's values determine whether the user can select multiple items.

Table The SelectionMode Enumeration

Value	**Description**
None	No selection at all is allowed.
One (Default)	Only a single item can be selected.
MultiSimple	Simple multiple selection: A mouse click (or pressing the spacebar) selects or deselects an item in the list. You must click all the items you want to select.
MultiExtended	Extended multiple selection: Press Shift and click the mouse (or press one of the arrow keys) to expand the selection. This will highlight all the items between the previously selected item and the current selection. Press Ctrl and click the mouse to select or deselect single items in the list.

Sorted

If you want the items to be always sorted, set the control's Sorted property to True. This property can be set at design time as well as runtime.

The following items would be sorted as shown:
"AA"
"Aa"
"aA"

Uppercase characters appear before the equivalent lowercase characters, but both upper- and lowercase characters appear together.

Text
The Text property returns the selected text on the control. To access the actual object, use the SelectedItem property.

The Items Collection
To manipulate a ListBox control from within your application, you should be able to:

_ Add items to the list
_ Remove items from the list
_ Access individual items in the list
The items in the list are represented by the Items collection. Each member of the Items collection is an object.

If you add a Color and a Rectangle object to the Items collection with the following statements:

```
ListBox1.Items.Add(Color.Yellow)
ListBox1.Items.Add(New Rectangle(0, 0, 100, 100))

Console.WriteLine(ListBox1.Items.Item(0).G)
  255
Console.WriteLine(ListBox1.Items.Item(1).Width)
  100
```

Add
To add items to the list, use the Items.Add or Items.Insert method. The syntax of the Add method is

```
   ListBox1.Items.Add(item)
```

The *item* parameter is the object to be added to the list. The Add method appends new items to the end of the list, unless the Sorted property has been set to True.
The following loop adds the elements of the array *words* to a ListBox control, one at a time:

```
   Dim words(100) As String
  { statements to populate array }

Dim i As Integer
For i = 0 To 99
ListBox1.Items.Add(words(i))
Next
```

Clear

The Clear method removes all the items from the control. Its syntax is quite simple:

List1.Items.Clear

Count

This is the number of items in the list. Its syntax is

ListBox1.Items.Count

CopyTo

The CopyTo method of the Items collection retrieves all the items from a ListBox control and stores them to the array passed to the method as argument.

The syntax of the CopyTo method is

ListBox1.CopyTo(destination, index)

where *destination* is the name of the array that will accept the items and *index* is the index of an element in the array here the first item will be stored.

Insert

To insert an item at a specific location, use the Insert method, whose syntax is:

ListBox1.Items.Insert(index, item)

where *item* is the object to be added and *index* is the location of the new item.

Remove

To remove an item from the list, you must first find its position (index) in the list, and call the Remove method passing the position as argument:

ListBox1.Items.Remove(index)

The *index* parameter is the order of the item to be removed,

You can also specify the item to be removed by reference. To remove a specific item from the list, use the following syntax:

ListBox1.Items.Remove(item)

Contains

Use the Contains method to avoid the insertion of identical objects to the ListBox control. The following statements add a string to the Items collection, only if the string isn't already part of the collection:

```
Dim itm As String = "Remote Computing"
If Not ListBox1.Items.Contains(itm) Then
ListBox1.Items.Add(itm)
End If
```

Selecting Items

The ListBox control allows the user to select either one or multiple items, depending on the setting of the SelectionMode property. In a single-selection ListBox control, you can retrieve the selected item with the SelectedItem property and its index with the SelectedIndex property. SelectedItem returns the selected item, which could be an object. The text that was clicked by the user to select the item is reported by the Text property. If the control allows the selection of multiple items, they're reported with the SelectedItems property. This property is a collection of Item objects and exposes the same members as the Items collection. The SelectedItems.Count property reports the number of selected items.

To iterate through all the selected items in a multiselection ListBox control, use a loop like the following:

```
Dim itm As Object
For Each itm In ListBox1.SelectedItems
Console.WriteLine(itm)
Next
```

Searching

The single most important enhancement to the ListBox control is that it can now locate any item in the list with the FindString and FindStringExact methods. Both methods accept a string as argument (the item to be located) and a second, optional argument, the index at which the search will begin. The FindString method locates a string that partially matches the one you're searching for; Find- StringExact finds an exact match. If you're searching for "Man" and the control contains a name like "Mansfield," FindString will match the item, but FindStringExact will not.

The syntax of both methods is the same:

itemIndex = ListBox1.FindString(searchStr As String)

where *searchStr* is the string you're searching for. An alternative form of both methods allows you to specify the order of the item at which the search will begin:

itemIndex = ListBox1.FindString(searchStr As String, startIndex As Integer)

The *startIndex* argument allows you specify the beginning of the search, but you can't specify where the search will end.

The ComboBox Control

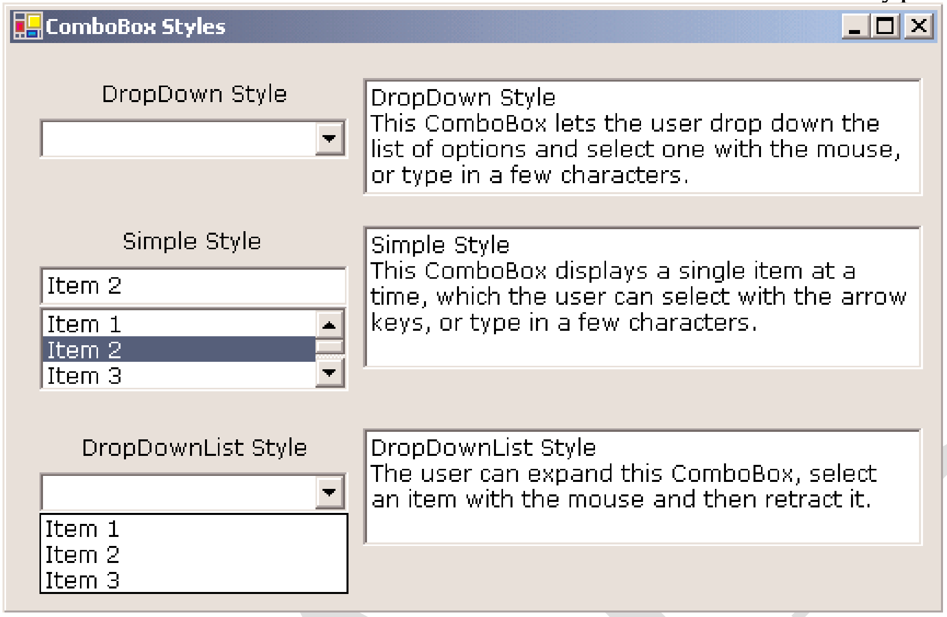
The ComboBox control is similar to the ListBox control in the sense that it contains multiple items of which the user may select one, but it typically occupies less space on-screen. The Text property of the ComboBox is read-only at runtime, and you can locate an item by assigning a value to the control's Text property.

Three types of ComboBox controls are available in Visual Basic.NET. The value of the control's Style property, whose values are as follows

Styles of the ComboBox Control

Value	**Effect**
DropDown (Default)	The control is made up of a drop-down list and a text box. The user can select an item from the list or type a new one in the text box.
DropDownList	This style is a drop-down list, from which the user can select one of its items but can't enter a new one.

Simple	The control includes a text box and a list that doesn't drop down. The user can select from the list or type in the text box.



DropDown Style
This ComboBox lets the user drop down the list of options and select one with the mouse, or type in a few characters.

Simple Style
This ComboBox displays a single item at a time, which the user can select with the arrow keys, or type in a few characters.

DropDownList Style
The user can expand this ComboBox, select an item with the mouse and then retract it.

Adding a New Item to the ComboBox Control at Runtime

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) Handles Button1.Click
        Dim itm As String
        itm = InputBox("Enter new item", "New Item")
        If itm <> "" Then AddElement(itm)
  End Sub
```

The AddElement() subroutine, which accepts a string as argument and adds it to the control, is as follows.

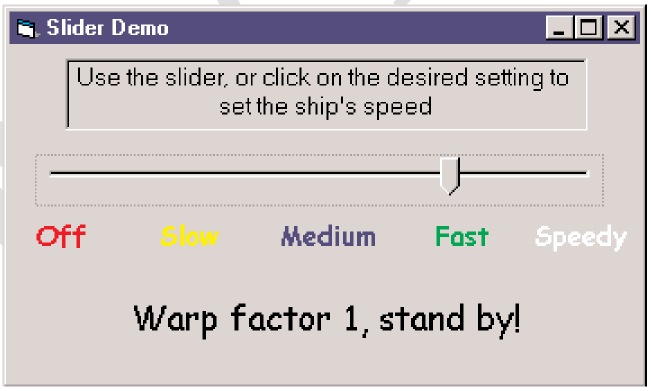
The AddElement() Subroutine

```
    Sub AddElement(ByVal newItem As String)
        Dim idx As Integer
        If Not ComboBox1.Items.Contains(newItem) Then
      idx = ComboBox1.Items.Add(newItem)
      Else
   idx = ComboBox1.Items.IndexOf(newItem)
   End If
    ComboBox1.SelectedIndex = idx
 End Sub
```

The ScrollBar and TrackBar Controls

The ScrollBar and TrackBar controls let the user specify a magnitude by scrolling a selector between its minimum and maximum values. In some situations, the user doesn't know in advance the exact value of the quantity to specify (in which case, a text box would suffice), so your application must provide a more flexible mechanism for specifying a value, along with some type of visual feedback.

The TrackBar control is the old Slider control. The TrackBar control is similar to the ScrollBar control, but it doesn't cover a continuous rangeof values. The TrackBar control has a fixed number of tick marks, which the developer can label (e.g., Off, Slow, and Speedy, as shown in the following figure.

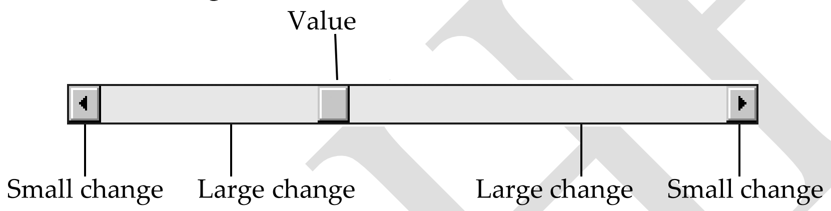


Use the slider, or click on the desired setting to set the ship's speed

Warp factor 1, stand by!

The ScrollBar Control

The ScrollBar control is a long stripe with an indicator that lets the user select a value between the two ends of the control, and it can be positioned either vertically or horizontally. Use the Orientation property to make the control vertical or horizontal. The left (or bottom) end of the control corresponds to its minimum value; the other end is the control's maximum value. The current value of the control is determined by the position of the indicator, which can be scrolled between the minimum and maximum values. The basic properties of the ScrollBar control, therefore, are properly named Minimum, Maximum, and Value (see Figure 6.13).



Minimum The control's minimum value. The default value is 0, but because this is an Integer value you can set it to negatives values as well.

Maximum The control's maximum value. The default value is 100, but you can set it to any value you can represent with the Integer data type.

Value The control's current value, specified by the indicator's position.

The ScrollBar Control's Events
The user can change the ScrollBar control's value in three ways:
By clicking the two arrows at its ends. The value of the control changes by the amount specified with the SmallChange property.
By clicking the area between the indicator and the arrows. The value of the control changes by the amount specified with the LargeChange property.
By dragging the indicator with the mouse.
You can monitor the changes on the ScrollBar's value from within your code with two events:

ValueChanged and Scroll. Both events are fired every time the indicator's position is changed. If you change the control's value from within your code, then only the ValueChanged event will be fired.

The Scroll event can be fired in response to many different actions, such as the scrolling of the indicator with the mouse or a click on one of the two buttons at the ends of the scrollbars.

The Actions That Can Cause the Scroll Event

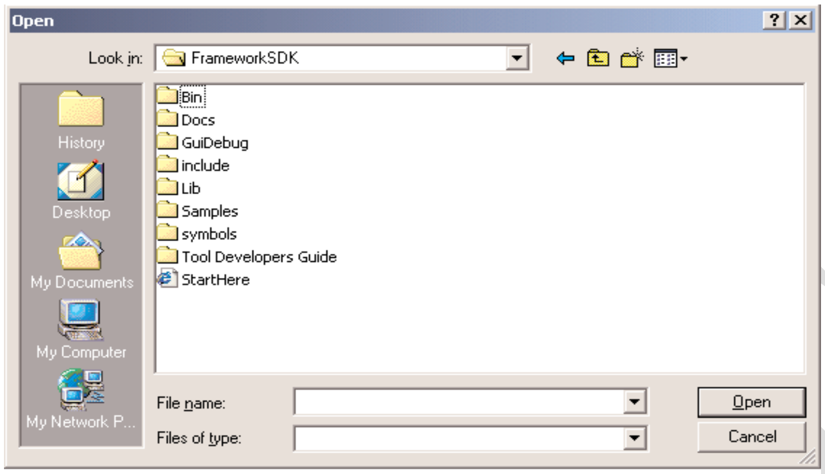
Member	**Description**
EndScroll	The user has stopped scrolling the control.
First	The control was scrolled to the Minimum position.
LargeDecrement	The control was scrolled by a large decrement (user clicked the bar between the button and the left arrow).
LargeIncrement	The control was scrolled by a large increment (user clicked the bar between the button and the right arrow).
Last	The control was scrolled to the Maximum position.
SmallDecrement	The control was scrolled by a small decrement (user clicked the left arrow).
SmallIncrement	The control was scrolled by a small increment (user clicked the right arrow).
ThumbPosition	The button was moved.
ThumbTrack	The button is being moved.

The TrackBar Control

The TrackBar control is similar to the ScrollBar control, but it lacks the granularity of ScrollBar.

Similar to the ScrollBar control, SmallChange and LargeChange properties are available. Small- Change is the smallest increment by which the Slider value can change. The user can only change the slider by the SmallChange value by sliding the indicator. To change the Slider's value by LargeChange, the user can click on either side of the indicator.

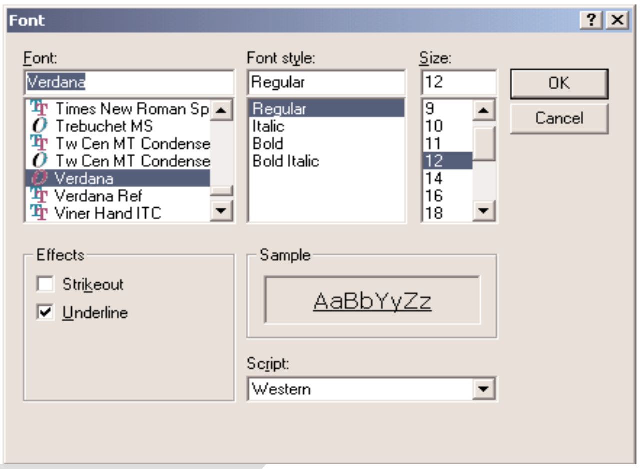
The Common Dialog Controls



Look in: FrameworkSDK

Bin
Docs
GuiDebug
include
Lib
Samples
symbols
Tool Developers Guide
StartHere

File name:
Files of type:



Font: Verdana
Font style: Regular
Size: 12
Strikeout [ ]
Underline [✓]
Script: Western

If you ever want to display an Open or Font dialog box, don't design it—it already exists. To use it, just place the appropriate common dialog control on your form and activate it from within your code by calling the ShowDialog method.

The following common dialog controls are available on the Toolbox.

OpenFileDialog Lets users select a file to open. It also allows the selection of multiple files, for applications that must process many files at once.

SaveFileDialog Lets users select or specify a filename in which the current document will be saved.

ColorDialog Lets users select a color from a list of predefined colors, or specify custom colors.

FontDialog Lets users select a typeface and style to be applied to the current text selection.

PrintDialog Lets users select and set up a printer (the page's orientation, the document pages to be printed, and so on). There are two more common dialog controls, the PrintPreviewDialog and the PageSetupDialog controls.

Using the Common Dialog Controls

To display a common dialog box from within your code, you simply call the control's ShowDialog method, which is common for all controls.

Here is the sequence of statements used to invoke the Open common dialog and retrieve the selected filename:

```
If OpenFileDialog1.ShowDialog = DialogResult.OK Then
   fileName = OpenFileDialog1.FileName
End If
```

The variable *fileName* is the full pathname of the file selected by the user. You can also set the FileName property to a filename, which will be displayed when the Open dialog box is first opened. This allows the user to click the Open button to open the preselected file or choose another file.

```
OpenFileDialog1.FileName = "C:\Documents\Doc1.doc"
   If OpenFileDialog1.ShowDialog = DialogResult.OK Then
   fileName = OpenFileDialog1.FileName
End If
```

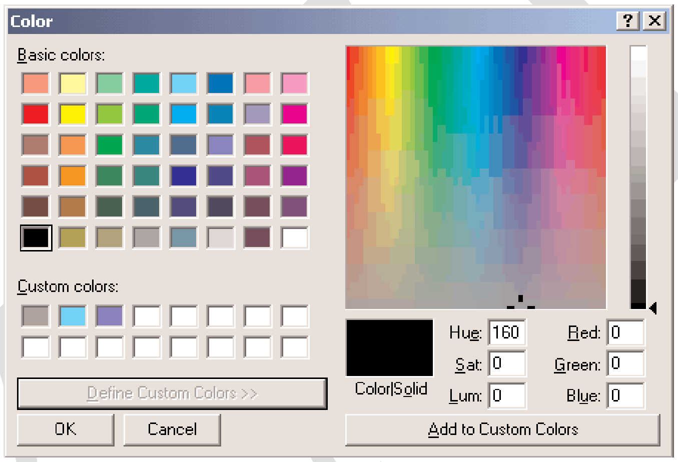
Similarly, you can invoke the Color dialog box and read the value of the selected color with the following statements:

```
If ColorDialog1.ShowDialog = DialogResult.OK Then
     selColor = ColorDialog1.Color
```

```
End If
```

The Color Dialog Box

The Color dialog box, shown in Figure 7.2, is one of the simplest dialog boxes. It has a single property, Color, which returns the color selected by the user or sets the initially selected color when the user opens the dialog box. Before opening the Color common dialog with the ShowDialog method, you can set various properties, which are described next.



Hue: 160
Sat: 0
Lum: 0
Red: 0
Green: 0
Blue: 0

The following statements set the selected color of the Color common dialog control, display the control, and then use the color selected on the control to fill the form. First, place a ColorDialog control on the form, and then insert the following statements in a button's Click event handler:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                ByVal e As System.EventArgs) Handles Button1.Click
    ColorDialog1.Color
    ColorDialog1.AllowFullOpen = True
        If ColorDialog1.ShowDialog = DialogResult.OK Then
```

```
            Me.BackColor = ColorDialog1.Color
        End If
    End Sub
```

AllowFullOpen

Set this property to True if you want users to be able to open up the dialog box and define their own custom colors.

AnyColor

This property is a Boolean value that determines whether the dialog box displays all available colors in the set of basic colors.

Color

This property is a Color value, and you can set it to any valid color. If ColorDialog1.Color = Color.Azure

```
            If ColorDialog1.ShowDialog = DialogResult.OK Then
                Me.BackColor = ColorDialog1.Color
    End If
```

CustomColors

This property indicates the set of custom colors that will be shown in the common dialog. To display three custom colors in the lower section of the Color dialog box, use a statement like the following:

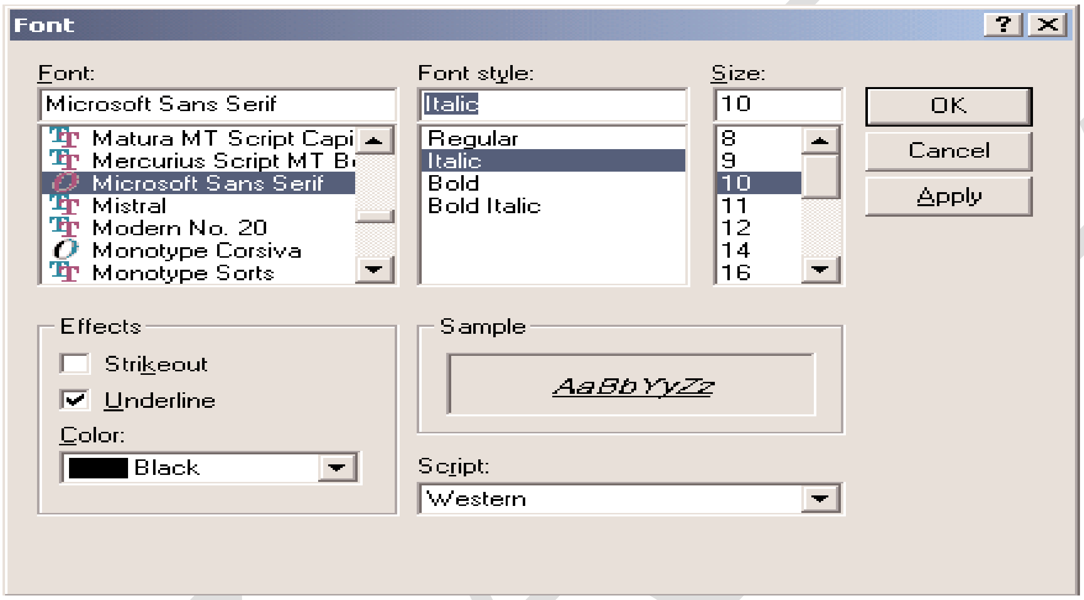
```
        Dim colors() As Integer = {222663, 35453, 7888}
        ColorDialog1.CustomColors = colors

    Dim colors() As Color = {Color.Azure, Color.Navy, Color.Teal}
```

SolidColorOnly

Indicates whether the dialog box will restrict users to selecting solid colors only. This setting should be used with systems that can display only 256 colors.

The Font Dialog Box

The user can also select the font's color and even apply the current dialog-box settings to the selected text on a control of the form without closing the dialog box, by clicking the Apply button on the Font dialog box.



```
Private Sub Button2_Click(ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) Handles Button2.Click
        FontDialog1.Font = TextBox1.Font
        If FontDialog1.ShowDialog = DialogResult.OK Then
    TextBox1.Font = FontDialog1.Font
  End If
End Sub
```

AllowScriptChange

This property is a Boolean value that indicates whether the Script combo box will be displayed on the Font common dialog.

AllowSimulations

This property is a Boolean value that indicates whether the dialog box allows the display and selection of simulated fonts.

AllowVectorFonts

This property is a Boolean value that indicates whether the dialog box allows the display and selection of vector fonts.

AllowVerticalFonts

This property is a Boolean value that indicates whether the dialog box allows the display and selection of both vertical and horizontal fonts. Its default value is False, which displays only horizontal fonts.

Color

This property sets or returns the selected font color. The user will see the option to select a color for the selected font only if you set the ShowColor property to True.

FixedPitchOnly

This property is a Boolean value that indicates whether the dialog box allows only the selection of fixed-pitch fonts.

Font

This property is a Font object. The following statements show how to preselect the font of the *TextBox1* control on the Font dialog box and how to change the same control's font to the one selected by the user on the dialog box:

```
    FontDialog1.Font = TextBox1.Font
    If FontDialog1.ShowDialog = DialogResult.OK Then
        TextBox1.Font = FontDialog1.Font
End If
```

You can create a new Font object and assign it to the control's Font property. The following statements do that:

```
Dim newFont As Font
newFont = New Font("Verdana", 12, FontStyle.Underline)
FontDialog1.Font = newFont
FontDialog1.ShowDialog()
```

FontMustExist
This property is a Boolean value that indicates whether the dialog box forces the selection of an existing font.

MaxSize, MinSize
These two properties are integers that determine the minimum and maximum point size the user can select.

ScriptsOnly
This property indicates whether the dialog box allows selection of fonts for Symbol character sets, in addition to the American National Standards Institute (ANSI) character set. Its default value is True.

ShowApply
This property is a Boolean value that indicates whether the dialog box provides an Apply button.

The following statements display the Font dialog box with the Apply button:

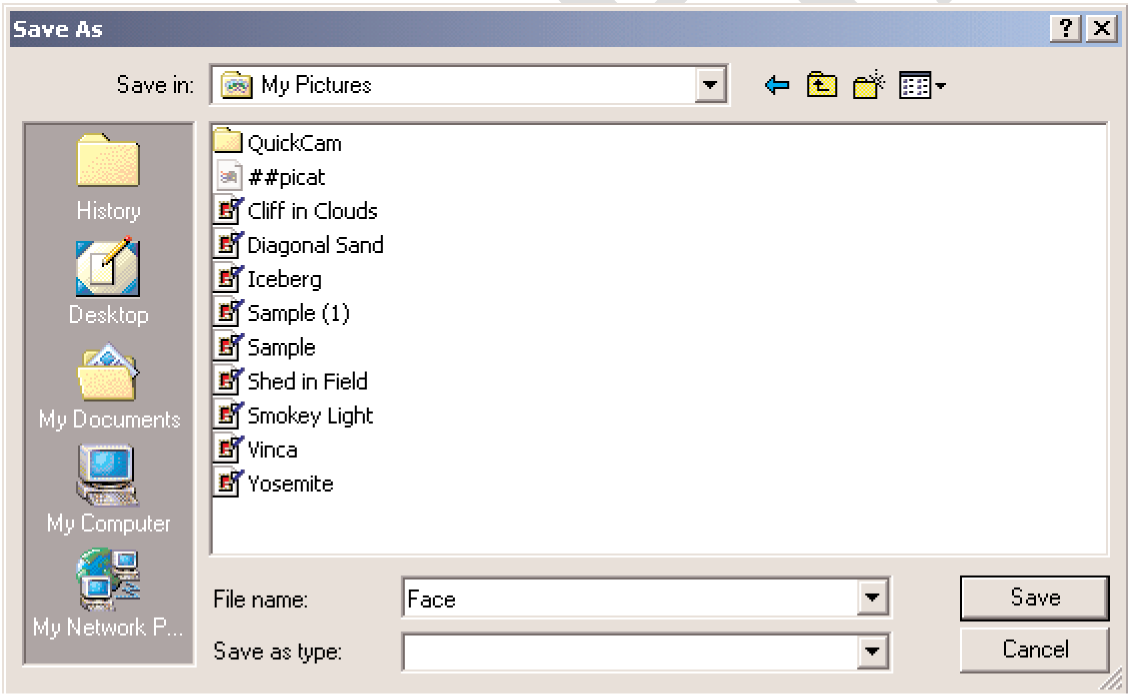
```
    Private Sub Button2_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) Handles Button2.Click
            FontDialog1.Font = TextBox1.Font
            FontDialog1.ShowApply = True
            If FontDialog1.ShowDialog = DialogResult.OK Then
      TextBox1.Font = FontDialog1.Font
    End If
  End Sub

     Private Sub FontDialog1_Apply(ByVal sender As Object, _
                      ByVal e As System.EventArgs) Handles FontDialog1.Apply
        TextBox1.Font = FontDialog1.Font
      End Sub
```

ShowColor
This property is a Boolean value that indicates whether the dialog box allows the user to select a color for the font.

ShowEffects
This property is a Boolean value that indicates whether the dialog box contains controls to allow the user to specify special text effects, such as strikethrough and underline.

The Open and Save As Dialog Boxes

Open and Save As are the two most widely used common dialog boxes, and they're implemented by the OpenFileDialog and SaveFileDialog controls.



The extension of the default file type for the application is described by the DefaultExtension property, and the list of the file types displayed in the Save As Type box is described by the Filter property. Both the DefaultExtension and the Filter properties are available in the control's Properties window at design time. At runtime, you must set them manually from within your code. To prompt the user for the file to be opened, use the following statements. This dialog box displays the files with the extension .BIN only.

```
Private Sub bttnSave_Click(ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) Handles bttnSave.Click
          OpenFileDialog1.DefaultExt = ".BIN"
          OpenFileDialog1.AddExtension = True
          OpenFileDialog1.Filter = "Binary Files|*.bin"
          If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
     Console.WriteLine(OpenFileDialog1.FileName)
   End If
 End Sub
```

The following sections describe the properties of the OpenFileDialog and SaveFileDialog controls.

AddExtension

This property is a Boolean value that determines whether the dialog box automatically adds an extension to a filename, if the user omits it. The extension added automatically is the one specified by the DefaultExtension property, which must be set before you call the ShowDialog method.

CheckFileExists

This property is a Boolean value that indicates whether the dialog box displays a warning if the user enters the name of a file that does not exist.

CheckPathExists

This property is a Boolean value that indicates whether the dialog box displays a warning if the user specifies a path that does not exist, as part of the user-supplied filename.

DefaultExtension

This property sets the default extension of the dialog box. Use this property to specify a default filename extension, such as TXT or DOC.

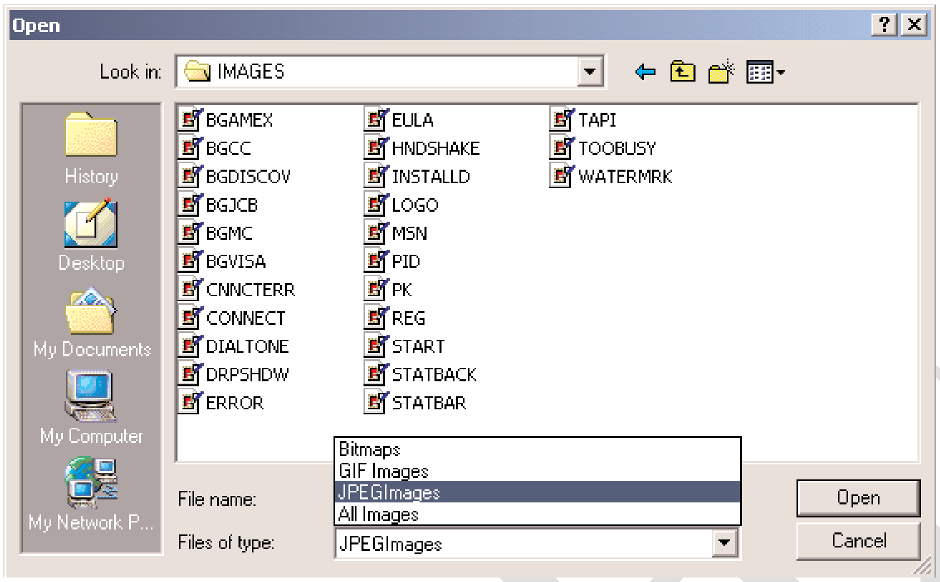
DereferenceLinks
This property indicates whether the dialog box returns the location of the file referenced by the shortcut or the location of the shortcut itself.

FileName
This property is the path of the file selected by the user on the control. If you set this property to a filename before opening the dialog box, this value will be the proposed filename.

Filter
This property is used to specify the type(s) of files displayed on the dialog box. To display text files only, set the Filter property to “Text files|*.txt”.

```
OpenFileDialog1.Filter = “Bitmaps|*.BMP|GIF Images|*.GIF|JPEG” & _
                         “Images|*.JPG|All Images|*.BMP;*.GIF;*.JPG”
```

The Open dialog box has four options, which determine what appears in the Save As Type box



FilterIndex

It is used to determine which filter will be displayed as the default when the common dialog is opened.

InitialDirectory

This property sets the initial directory (folder) in which files are displayed the first time the Open and Save dialog boxes are opened.

```
OpenFileDialog1.InitialDirectory = Application.ExecutablePath
```

The expression Application.ExecutablePath returns the path in which the application's executable file resides.

RestoreDirectory

The RestoreDirectory property is a Boolean value that indicates whether the dialog box restores the current directory before closing.

ValidateNames

This property is a Boolean value that indicates whether the dialog box accepts only valid Win32 filenames. Its default value is True, and you shouldn't change it.

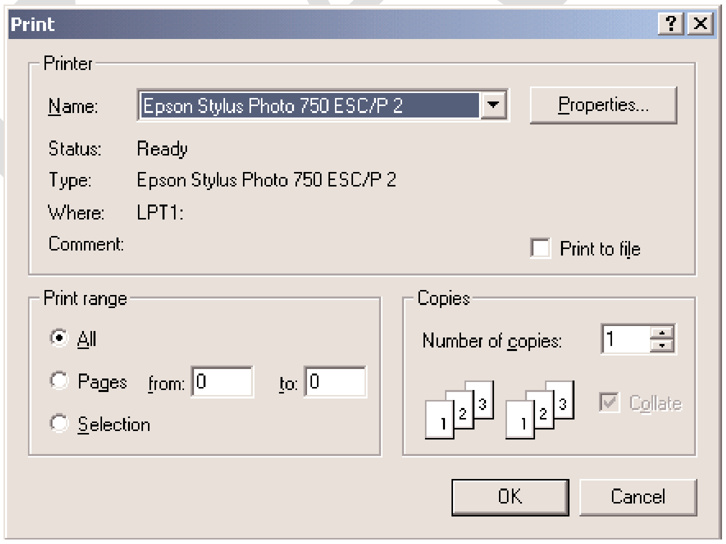
FileNames
If the Open dialog box allows the selection of multiple files, the FileNames property contains the pathnames of all selected files.

MultiSelect
This property is a Boolean value that indicates whether the user can select multiple files on the dialog box.

ReadOnlyChecked
This property is a Boolean value that indicates whether the Read-Only check box is initially selected when the dialog box first pops up.

ShowReadOnly
This property is a Boolean value that indicates whether the Read-Only check box is available.

The Print Dialog Box
The Print dialog box enables users to select a printer, set certain properties of the printout, and set up a specific printer.



Print

Printer
Name: Epson Stylus Photo 750 ESC/P 2
Status: Ready
Type: Epson Stylus Photo 750 ESC/P 2
Where: LPT1:
Comment:
Print to file [ ]

Print range
[✓ All]
Pages from: 0 to: 0
Selection

Copies
Number of copies: 1
Collate [✓]

AllowPrintToFile This property is a Boolean value that controls whether the user will be given the option to print to a file.

AllowSelection This property is a Boolean value that determines whether the user is allowed to print the current selection of the document.

AllowSomePages This property is a Boolean value that determines whether the Pages option on the dialog will be enabled.

The following statements create a new PrinterSettings object, pass it to the Print dialog box, and then display the dialog box.

```
 PrintDialog1.AllowSomePages = Tru
 PrintDialog1.AllowSelection = True
 PrintDialog1.PrinterSettings = _
              New System.Drawing.Printing.PrinterSettings()
   PrintDialog1.ShowDialog()
Console.WriteLine(“FROM PAGE: “ &
PrintDialog1.PrinterSettings.FromPage)
Console.WriteLine(“TO PAGE: “ & PrintDialog1.PrinterSettings.ToPage)
Console.WriteLine(“# OF COPIES: “ & PrintDialog1.PrinterSettings.Copies)
Console.WriteLine(“PRINTER NAME:” & PrintDialog1.PrinterSettings.PrinterName)
Console.WriteLine(“PRINT RANGE: “ & PrintDialog1.PrinterSettings.PrintRange)
Console.WriteLine(“LANDSCAPE: “ &
PrintDialog1.PrinterSettings.LandscapeAngle)
```

The output produced by the previous statements on my system looked like this:

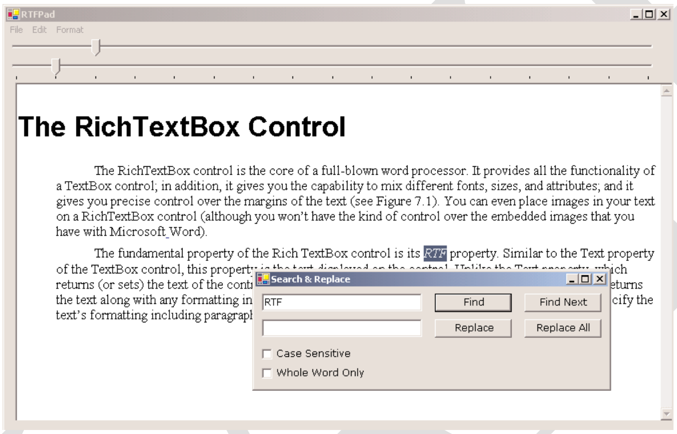
```
FROM PAGE: 3
TO PAGE: 4
# OF COPIES: 1
PRINTER NAME:Epson Stylus Photo 750 ESC/P 2
PRINT RANGE: 2
```

LANDSCAPE: 270

To set the orientation of the printout, you must click the Properties button on the Print dialogbox.

The RichTextBox Control

It provides all the functionality of a TextBox control; in addition, it gives you the capability to mix different fonts, sizes, and attributes; and it gives you precise control over the margins of the text.



The RichTextBox Control

The RichTextBox control is the core of a full-blown word processor. It provides all the functionality of a TextBox control; in addition, it gives you the capability to mix different fonts, sizes, and attributes; and it gives you precise control over the margins of the text (see Figure 7.1). You can even place images in your text on a RichTextBox control (although you won't have the kind of control over the embedded images that you have with Microsoft Word).

The fundamental property of the Rich TextBox control is its *RTF* property. Similar to the Text property of the TextBox control, this property ich returns (or sets) the text of the cont eturns the text along with any formatting in cify the text's formatting including paragrap

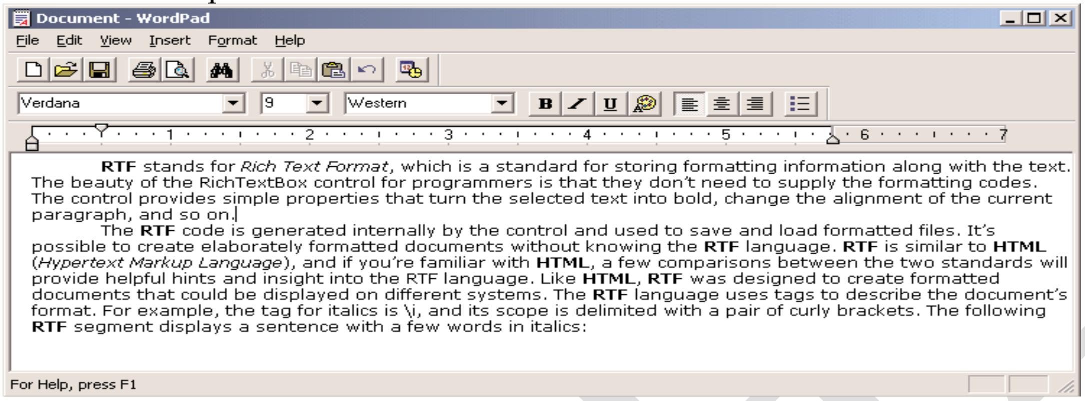
RTF

The fundamental property of the RichTextBox control is its *RTF* property. RTF stands for *Rich Text Format*, which is a standard for storing formatting information along with the text.

The RTF Language

RTF is a language that uses simple commands to specify the formatting of a document. These commands, or *tags*, are ASCII strings, such as \par and \b. RTF documents don't contain special characters and can be easily exchanged among different operating systems and computers, as long as there is an RTF-capable application to read the document.

Open the WordPad application (choose Start ➢ Programs ➢ Accessories ➢ WordPad) and enter a few lines of text. Select a few words or sentences and format them in different ways with any of WordPad's formatting commands. Then save the document in RTF format: Choose File ➢ Save As, select Rich Text Format, and then save the file as Document.rtf. If you open this file with a text editor such as Notepad, you'll see the actual RTF code that produced the document.



RTF stands for *Rich Text Format*, which is a standard for storing formatting information along with the text. The beauty of the RichTextBox control for programmers is that they don't need to supply the formatting codes. The control provides simple properties that turn the selected text into bold, change the alignment of the current paragraph, and so on.

The **RTF** code is generated internally by the control and used to save and load formatted files. It's possible to create elaborately formatted documents without knowing the **RTF** language. **RTF** is similar to **HTML** (*Hypertext Markup Language*), and if you're familiar with **HTML**, a few comparisons between the two standards will provide helpful hints and insight into the RTF language. Like **HTML**, **RTF** was designed to create formatted documents that could be displayed on different systems. The **RTF** language uses tags to describe the document's format. For example, the tag for italics is \i, and its scope is delimited with a pair of curly brackets. The following **RTF** segment displays a sentence with a few words in italics:

```
{\rtf1\ansi\ansicpg1252\deff0\deflang1033
{\fonttbl{\f0\fnil\fcharset0 Verdana;}{\f1\fswiss\fcharset0 Arial;}}
\viewkind4\uc1\pard\nowidctlpar\fi720\b\f0\fs18 RTF \b0 stands for \i Rich Text
Format\i0 , which is a standard for storing formatting information along with the
text. The beauty of the RichTextBox control for programmers is that they don\rquote
t need to supply the formatting codes. The control provides simple properties that
turn the selected text into bold, change the alignment of the current paragraph,
and so on.
```

All formatting tags are prefixed with the backslash (\) symbol. To display the \ symbol itself, insert an additional slash. Paragraphs are marked with the \par tag, and the entire document is enclosed in a pair of curly brackets. The \li and \ri tags followed by a numeric value specify the amount of the left and right indentation.

RTF is similar to HTML. RTF was designed to create formatted documents that could be displayed on different systems. The RTF language uses tags to describe the document's format. For example, the tag for italics is \i, and its scope is delimited with a pair of curly brackets. The following RTF segment displays a sentence with a few words in italics:

```
{{\b RTF} (which stands for Rich Text Format) is a {\i document formatting
language} that uses simple commands to specify the formatting of the document.}
```

The following is the equivalent HTML code:

<b>RTF</b> (which stands for Rich Text Format) is a <i>document formatting language</i> that uses simple commands to specify the formatting of the document.

The RTF Code

If you click the Show RTF button, you'll see the actual RTF code that produced the formatted document. This is all the information the RichTextBox control requires to render the document. The control is responsible for generating the RTF code and for rendering the document. You simply manipulate a few properties (the recurring theme in Visual Basic programming), and the control does the rest.

The RichTextBox's Properties

Property	What It Manipulates
SelectedText	The selected text
SelectedRTF	The RTF code of the selected text
SelectionStart	The position of the selected text's first character
SelectionLength	The length of the selected text
SelectionFont	The font of the selected text
SelectionColor	The color of the selected text
SelectionIndent, SelectionRightIndent, SelectionHangingIndent	The indentation of the selected text
RightMargin	The distance of the text's right margin from the left edge of the control, which is in effect the length of each text line
SelectionBullet	Whether the selected text is bulleted
BulletIndent	The amount of bullet indent for the selected text

SelectedText

The SelectedText property represents the selected text. To assign the selected text to a variable, use the following statement:

```
STextt=RichTextbox1.SelectedText
RichTextbox1.SelectedText=UCase(RichTextbox1.SelectedText)
```

SelectionStart, SelectionLength

SelectionStart and SelectionLength, report the position of the first selected character in the text and the length of the selection, respectively.

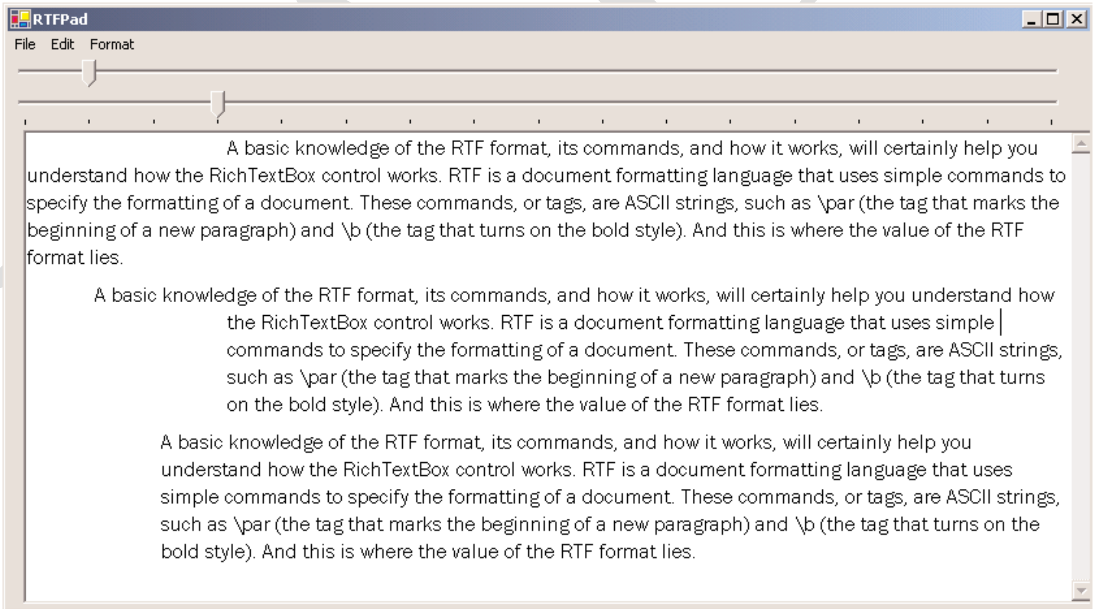
```
RichTextBox1.SelectionStart = 0
RichTextBox1.SelectionLength = Len(RichTextBox1.Text)
```

SelectionAlignment

Use this property to read or change the alignment of one or more paragraphs.

SelectionIndent, SelectionRightIndent, SelectionHangingIndent

These properties allow you to change the margins of individual paragraphs. The SelectionIndent property sets (or returns) the amount of the text's indentation from the left edge of the control. The SelectionRightIndent property sets (or returns) the amount of the text's indentation from the right edge of the control. The SelectionHangingIndent property is the distance between the left edge of the first line and the left edge of the following lines.



A basic knowledge of the RTF format, its commands, and how it works, will certainly help you understand how the RichTextBox control works. RTF is a document formatting language that uses simple commands to specify the formatting of a document. These commands, or tags, are ASCII strings, such as \par (the tag that marks the beginning of a new paragraph) and \b (the tag that turns on the bold style). And this is where the value of the RTF format lies.

A basic knowledge of the RTF format, its commands, and how it works, will certainly help you understand how the RichTextBox control works. RTF is a document formatting language that uses simple commands to specify the formatting of a document. These commands, or tags, are ASCII strings, such as \par (the tag that marks the beginning of a new paragraph) and \b (the tag that turns on the bold style). And this is where the value of the RTF format lies.

A basic knowledge of the RTF format, its commands, and how it works, will certainly help you understand how the RichTextBox control works. RTF is a document formatting language that uses simple commands to specify the formatting of a document. These commands, or tags, are ASCII strings, such as \par (the tag that marks the beginning of a new paragraph) and \b (the tag that turns on the bold style). And this is where the value of the RTF format lies.

SelectionBullet, BulletIndent

You use these properties to create a list of bulleted items. If you set the SelectionBullet property to True, the selected paragraphs are formatted with a bullet style, similar to the

<ul> tag in HTML. To create a list of bulleted items, assign the value True to the SelectionBullet property. To change a list of bulleted items back to normal text, make the same property False.

The paragraphs formatted with the SelectionBullet property set to True are also indented from the left by a small amount. To set the amount of the indentation, use the BulletIndent property, whose syntax is

RichTextBox1.BulletIndent

Methods

The first two methods of the RichTextBox control you will learn about are SaveFile and LoadFile:

SaveFile saves the contents of the control to a disk file.
LoadFile loads the control from a disk file.

SaveFile

The syntax of the SaveFile method is

RichTextBox1.SaveFile(path, filetype)

where *path* is the path of the file in which the current document will be saved.

Format	**Effect**
PlainText	Stores the text on the control without any formatting
RichNoOLEObjs	Stores the text without any formatting and ignores any embedded OLE objects
RichText	Stores the formatted text
TextTextOLEObjs	Stores the text along with the embedded OLE objects
UnicodePlainText	Stores the text in Unicode format

LoadFile

Similarly, the LoadFile method loads a text or RTF file to the control. Its syntax is identical to the syntax of the SaveFile method:

RichTextBox1.LoadFile(path, filetype)

The *filetype* argument is optional and can have one of the values of the RichTextBoxStreamType enumeration.

Select, SelectAll
The Select method selects a section of the text on the control, similar to setting the SelectionStart and SelectionLength properties.

```
    RichTextBox1.Select(start, length)
```

The SelectAll method accepts no arguments and selects all the text on the control.

Advanced Editing Features

CanUndo, CanRedo
These two properties are Boolean values you can read to find out whether an operation can be undone or redone. The following statements disable the Undo command if there's no action to be undone at the time, where *EditUndo* is the name of the Undo command on the Edit menu:

```
If RichTextBox1.CanUndo Then
    EditUndo.Enabled = True
  Else
    EditUndo.Enabled = False
  End If
```

UndoActionName, RedoActionName
These two properties return the name of the action that can be undone or redone. The most common value of both properties is the string "typing," which indicates that the Undo command will delete a number of characters..

The following statement sets the caption of the Undo command to a string that indicates the action to be undone:

```
EditUndo.Text = "Undo " & Editor.UndoActionName
```

Undo, Redo
These two methods undo or redo an action. The Undo method cancels the effects of the last action of the user on the control. The Redo method redoes the last action that was undone.

Cutting and Pasting

To cut, or copy, and paste text on the RichTextBox control, you can use the same techniques as with the regular TextBox control. For example, you can replace the current selection by assigning a string to the SelectedText property.

```
If Clipboard.GetDataObject.GetDataPresent(DataFormats.Text) Then
    RichTextBox.Paste(DataFormats.Text)
End If
```

Searching in a RichTextBox Control

The Find method locates a string in the control's text and is similar to the InStr() function. You can use InStr() with the control's Text property to locate a string in the text, but the Find method is optimized for the RichTextBox control and supports a couple of options that the InStr() function doesn't. The simplest form of the Find method is the following:

```
RichTextBox1.Find(string)
```

The *string* argument is the string you want to locate in the RichTextBox control.

```
RichTextBox1.Find(string, searchMode)
```

The *searchMode* argument is a member of the RichTextBoxFinds enumeration, which are shown in Table.

Table : The RichTextBoxFinds Enumeration

Value	**Effect**
MatchCase	Performs a case-sensitive search.
NoHighlight	The text found will not be highlighted.
None	Locates instances of the specified string even if they're not whole words.
Reverse	The search starts at the end of the document.
WholeWord	Locate only instances of the specified string that are whole words.

Two more forms of the Find method allow you specify the range of the text in which the search will take place:

```
RichTextBox1.Find(string, start, searchMode)
RichTextBox1.Find(string, start, end, searchMode)
```

The arguments *start* and *end* are the starting and ending locations of the search (use them to search for a string within a specified range only). If you omit the *end* argument, the search will start at the location specified by the *start* argument and will extend to the end of the text.

The String Class

The String class implements the String data type.

Properties

The String class exposes only two properties, the Length and Chars properties, which return a string's length and its characters respectively. Both properties are read-only.

Length

The Length property returns the number of characters in the string, and it's read-only. To find out the number of characters in a string vaiable, use the following statement:

```
chars = myString.Length
```

Chars

The Chars property is an array of characters that holds all the characters in the string.

Listing : Validating a Password

```
Private Function ValidatePassword(ByVal password As String) As Boolean
        If password.Length < 6 Then
            MsgBox("The password must be at least 6 characters long")
            Return False
        End If
        Dim i As Integer
        Dim valid As Boolean = False
        For i = 0 To password.Length - 1
            If Not Char.IsLetterOrDigit(password.Chars(i)) Then Return True
        Next
        MsgBox("The password must contain at least one " & _
                "character that is not a letter or a digit.")
        Return False
        End Function
```

Methods

Compare

This method compares two strings and returns a negative value if the first string is less than the second, a positive value if the second string is less than the first, and zero if the two strings are equal. The Compare method is overloaded, and the first two arguments are always the two strings to be compared.

The simplest form of the method accepts two strings as arguments:

```
String.Compare(str1, str2)
```

The following form of the method accepts a third argument, which is a True/False value and determines whether the search will be case-sensitive (if True) or not:

```
String.Compare(str1, str2, case)
```

Another form of the Compare method allows you to compare segments of two strings; its syntax is

```
String.Compare(str1, index1, str2, index2, length)
```

index1 and *index2* are the starting locations of the segment to be compared in each string.

CompareOrdinal

The CompareOrdinal method compares two strings similar to the Compare method, but it doesn't take into consideration the current locale. This method returns zero if the two strings are the same, but a positive or negative value if they're different. These values are not 1 and –1.

Concat

This method concatenates two or more strings and forms a new string. The simpler form of the Concat method has the following syntax, and it's equivalent to the & operator:

```
newString = String.Concat(string1, string2)
```

This statement is equivalent to the following:

```
newString = string1 & string2
```

A more useful from of the same method concatenates a large number of strings stored in an array:

```
newString = String.Concat(strings)
```

To use this form of the method, store all the strings you want to concatenate into a string array and then call the Concat method, as shown in this code segment:

```
Dim strings() As String = {“string1”, “string2”, “string3”, “string4”}
Dim longString As String
longString = String.Concat(strings)
```

Copy

The Copy method copies the value of one String variable to another. The last two statements in the following sample code are equivalent:

```
Dim s1, s2 As String
s1 = “some text”
s2 = s1
s2 = String.Copy(s1)
```

The following syntax will also work, because the *s1* variable is an instance of the String class, but it’s awkward—almost annoying:

```
s2 = s1.Copy(s1)
```

However, the Copy method doesn’t return a copy of the string to which it’s applied. The following statement is invalid:

```
s2 = s1.Copy ‘ INVALID STATEMENT
```

EndsWith, StartsWith

These two methods return True if the string ends or starts with a user-supplied substring. The syntax of these methods is

```
found = str.EndsWith(string)
found = str.StartsWith(string)
```

IndexOf, LastIndexOf

The IndexOf method starts searching from the beginning of the string, and the LastIndexOf method starts searching from the end of the string.

To locate a single character in a string, use the following forms of the IndexOf method:

```
String.IndexOf(ch)
```

String.IndexOf(ch, startIndex)
String.IndexOf(ch, startIndex, count)

The *startIndex* argument is the location in the string, where the search will start, and the *count* argument is the number of characters that will be examined.

IndexOfAny

This method accepts as argument an array of characters and returns the first occurrence of any of the array's characters in the string:

str.IndexOfAny(chars)

where *chars* is an array of characters. This method attempts to locate the first instance of any member of *chars* in the string.

Insert

The Insert method inserts one or more characters at a specified location in a string and returns the new string. The syntax of the Insert method is

newString = str.Insert(startIndex, subString)

startIndex is the position in the *str* variable, where the string specified by the second argument will be inserted.

Join

This method joins two or more strings and returns a single string with a separator between the original strings. Its syntax is

newString = String.Join(separator, strings)

where *separator* is the string that will be used as separator and *strings* is an array with the strings to be joined.

Split

You can split a long string into smaller ones with the Split method, whose syntax is

strings() = String.Split(delimiters, string)

where *delimiters* is an array of characters and *string* is the string to be split.

Remove

The Remove method removes a given number of characters from a string, starting at a specific location, and returns the result as a new string. Its syntax is

```
newString = str.Remove(startIndex, count)
```

where *startIndex* is the index of the first character to be removed in the *str* string variable and *count* is the number of characters to be removed.

Replace

This method replaces all instances of a specified character in a string with a new one. It creates a new instance of the string, replaces the characters as specified by its arguments, and returns this string:

```
newString = str.Replace(oldChar, newChar)
```

where *oldChar* is the character in the *str* variable to be replaced and *newChar* is the character to replace the occurrences of *oldChar*.

PadLeft, PadRight

These two methods align the string left or right in a specified field. They both return a fixed-length string with spaces to the right or to the left. Both methods accept the length of the field as argument and return a new string:

```
Dim LPString, RPString As String
LPString = "[" & "Mastering VB".PadRight(20) & "]"
RPString = "[" & "Mastering VB".PadLeft(20) & "]"
```

After the execution of these statements, the values of the *LPString* and *RPString* variables are:

```
[Mastering VB ]
          [ Mastering VB]
```

Another form of these methods allows you to specify the character to be used in padding the strings. If you change the calls to the PadLeft and PadRight methods in the last example with the following:

LPString = "Mastering VB".PadRight(20, "@")
RPString = "Mastering VB".PadLeft(20, ".")
then the two strings will be:
Mastering VB@@@@@@@@
........Mastering VB

The StringBuilder Class

The StringBuilder class stores dynamic strings and exposes methods to manipulate them much faster than the String class. To use the StringBuilder class in an application, you must import the System.Text class (unless you want to fully qualify each instance of the StringBuilder class in your code). Assuming you have imported the System.Text class in your code module, you can create a new instance of the class with the following statement:

Dim txt As New StringBuilder

There are many ways to initialize an instance of the StringBuilder class, but first I must explain the capacity of a StringBuilder object. Since the StringBuilder handles dynamic strings, it's good to declare in advance the size of the string you intend to store in the current instance of the class. The default capacity is 16 characters, and it's doubled automatically every time you exceed it. To set the initial capacity of the StringBuilder class, use the Capacity property. A related property is the Max-Capacity property, which is read-only and returns the maximum length of a string you can store in a StringBuilder variable. This value is approximately 2 billion characters; it's the length of the longest string you can store to an instance of the StringBuilder class.

To create a new instance of the StringBuilder class, you can call its constructor without any arguments, as I did in the preceding example. You can also initialize it by passing a string as argument:

Dim txt As New StringBuilder("some string")

If you can estimate the maximum length of the string you'll store in the variable, you can specify this value with the following form of the constructor, so that the variable need not be resized as you add to it:

Dim txt As New StringBuilder(initialCapacity)

The size you specify is not a hard limit; the variable may grow longer at runtime, and the String-Builder will adjust its capacity.

If you want to specify a *maximum* capacity for your StringBuilder variable, use the following constructor:

```
Dim txt As New StringBuilder(initialCapacity, maxCapacity)
```

Finally, you can initialize a new instance of the StringBuilder class using both an initial and a maximum capacity, as well as its initial value, with the following form of the constructor:

```
Dim txt As New StringBuilder(string, initialCapacity, maxCapacity)
```

All the members of the StringBuilder class are instance members. In other words, you must create an instance of the StringBuilder class before calling any of its properties or methods.

Properties

You have already seen the two basic properties of the StringBuilder class, the Capacity and Max-Capacity properties. In addition, the StringBuilder class provides the Length and Chars properties, which are the same as the corresponding properties of the String class.

Length

This property returns the number of characters in the current instance of the StringBuilder and is an integer value smaller than (or, at most, equal to) the Capacity property.

Chars

This property gets or sets the character at a specified location in the string, and it's an array of characters. Note that the index of the first character is zero.

```
ch = SB.Chars(index)
```

where *ch* is a properly declared Char variable and *SB* is an instance of the StringBuilder class. To set a character's value in the string, use the following statement:

```
SB.Chars(index) = ch
```

Methods

Many of the methods of the StringBuilder class are equivalent to the methods of the String class, but they act directly on the string to which they're applied, and they don't return a new, separate string.

Append

The Append method appends a base type to the current instance of the StringBuilder class, and its syntax is

```
SB.Append(value)
```

where the *value* argument can be a single character, a string, a date, or any numeric value. When you append numeric values to a StringBuilder, they're converted to strings; the value appended is the string returned by the type's ToString method. You can also append an object to the String-Builder—the actual string that will be appended is the object's ToString property.

Another form of the Append method allows you to append an array of characters, and it has the following syntax:

```
SB.Append(chars, startIndex, count)
```

Or, you can append a segment of a string by specifying the starting location of the substring in the longer string and the number of characters to be copied:

```
SB.Append(string, startIndex, count)
```

AppendFormat

The AppendFormat method is similar to the Append method. Before appending the string, however, it formats it. The string to be appended contains format specifications and the appropriate values. The syntax of the AppendFormat method is

```
SB.AppendFormat(string, values)
```

The first argument is a string with embedded format specifications and *values* is an array with values (objects, in general), one for each format specification in the *string*. If you have a small number of values to format, up to four, you can supply them as separate arguments separated by commas:

```
SB.AppendFormat(string, value, value, value, value)
```

The following statement appends the string "Your balance as of Thursday, May 16, 2002 is $19,950.40" to a StringBuilder variable:

```
Dim statement As New StringBuilder
statement.AppendFormat("Your balance as of {0:D} is ${1: #,###.00}", _
#5/16/2002#, 19950.40)
```

Each format specification is enclosed in a pair of curly brackets, and they're numbered sequentially (from zero). Then there's a colon followed by the actual specification. The D format specification tells the AppendFormat method to format the specified string in long date format. The second format specification, "#,###.00", uses the thousands separator and two decimal digits.

The following statements append the same string, but they pass the values through an array:

```
Dim statement As New StringBuilder
Dim values() As Object = {"5/16/2002", 19950.4}
statement.AppendFormat("Your balance as of {0:D} is ${1:#,###.00}", values)
```

In both cases, the *statement* variable will hold a string like this one:

```
Your balance as of Thursday, May 16, 2002 is $19,950.40
```

The format specifications in the original string usually contain formatting characters. For more information on date and time formatting options, see the section on the ToString method of the Date type, later in this chapter.

Insert

This method inserts a string into the current instance of the StringBuilder class, and its syntax is

SB.Insert(index, value)

The *index* argument is the location where the new string will be inserted in the current instance of the StringBuilder, and *value* is the string to be inserted.

Remove

This method removes a number of characters from the current StringBuilder, starting at a specified location; its syntax is

SB.Remove(startIndex, count)

where *startIndex* is the position of the first character to be removed from the string and *length* is the number of characters to be removed.

Replace

This method replaces all instances of a string in the current StringBuilder with another string. The syntax of the Replace method is

SB.Replace(oldValue, newValue)

where the two arguments can be either strings or characters. Another form of the Replace method limits the replacements to a specified segment of the StringBuilder instance:

SB.Replace(oldValue, newValue, startIndex, count)

This method will replace all instances of *oldValue* with *newValue* in the section starting at location *startIndex* and extending *count* characters.

Handling Dates

The DateTime Class

The DateTime class is used for storing date and time values.

```
Dim date1 As Date = #4/15/2001#
Dim date2 As Date = #4/15/2001 14:01:59#
```

Properties

The Date type exposes the following properties, which are straightforward.

Date

The Date property returns the date from a date/time value and sets the time to midnight. The statements:

```
Dim date1 As Date
date1 = Now()
Console.WriteLine(date1)
Console.WriteLine(date1.Date)
```

will print something like the following values in the Output window:

```
5/29/2001 2:30:17 PM
5/29/2001 12:00:00 AM
```

DayOfWeek, DayOfYear

These two properties return the day of the week (a number from 1 to 7) and the number of the day in the year (an integer from 1 to 365, or 366 for leap years).

Hour, Minute, Second, Millisecond

These properties return the corresponding time part of the Date value passed as arguments. If the current time is 1:35:22 P.M., the three properties of the DateTime class will return the following values when applied on the current date and time:

```
Console.WriteLine("The current time is " & Date.Now.TimeOfDay.ToString)
Console.WriteLine("The hour is " & Date.Now.Hour)
Console.WriteLine("The minute is " & Date.Now.Minute)
Console.WriteLine("The second is " & Date.Now.Second)
```

Methods

The DateTime class exposes several methods for manipulating dates. The most practical methods add and subtract time intervals to and from an instance of the DateTime class.

Compare

Compare is a shared method that compares two date/time values and returns an integer value indicating the relative order of the two values. The syntax of the Compare method is

```
order = System.DateTime.Compare(date1, date2)
```

where *date1* and *date2* are the two values to be compared. The method returns an integer, which is –1 if *date1* is less than *date2*, 0 if they're equal, and 1 if *date1* is greater than *date2*.

DaysInMonth

This shared method returns the number of days in a specific month. Because February contains a variable number of days depending on the year, the DaysInMonth method accepts as arguments both the month and the year:

```
monDays = System.DateTime.DaysInMonth(year, month)
```

To find out the number of days in February 2009, use the following expression:

```
FebDays = System.DateTime.DaysInMonth(2, 2009)
```

FromOADate

This shared method creates a date/time value from an OLE Automation Date.

```
newDate = System.DateTime.FromOADate(dtvalue)
```

The argument *dtvalue* must be a Double value in the range from –657,434 (first day of year 100) to 2,958,465 (last day of year 9999).

IsLeapYear

This shared method returns a True/False value that indicates whether the specified year is leap or not.

```
Dim leapYear As Boolean
 leapYear = System.DateTime.IsLeapYear(year)
```

Add

This method adds a TimeSpan object to the current instance of the Date class. The TimeSpan object represents a time interval and there are many methods to create a TimeSpan object, which are all discussed in the section "The TimeSpan Class" later in this chapter. The following statements create a new TimeSpan object that represents 3 days, 6 hours, 2 minutes, and 50 seconds, and add this TimeSpan to the current date and time. Depending on when these statements are executed, the two date/time values
will differ, but the difference between them will always be 3 days, 6 hours, 2 minutes, and 50 seconds:

```
Dim TS As New TimeSpan()
Dim thisMoment As Date = Now()
```

```
 TS = New TimeSpan(3, 6, 2, 50)
Console.WriteLine(thisMoment)
Console.WriteLine(thisMoment.Add(TS))
```

The values printed in the Output window when I tested this code segment were:

```
2001-04-13 16:26:38
2001-04-16 22:29:28
```

Subtract

This method is the counterpart of the Add method; it subtracts a TimeSpan object from the current instance of the Date class and returns another Date value. The following statements create a new Time-Span object that represents 3 days, 6 hours, 2 minutes, and 50 seconds, and subtracts this TimeSpan from the current date and time. Depending on when these statements are executed, the two date/time values will differ, but the difference between them will always be the same:

```
 Dim TS As New TimeSpan()
 Dim thisMoment As Date = Now()
TS = New TimeSpan(3, 6, 2, 50)
Console.WriteLine(thisMoment)
Console.WriteLine(thisMoment.Subtract(TS))
```

The values printed in the Output window when I tested this code segment were:

```
5/29/2001 2:52:03 PM
5/26/2001 8:49:13 AM
```

Adding Intervals to Dates

Various methods add specific intervals to a date/time value. Each method accepts the number of intervals to add (days, hours, milliseconds, and so on). These methods are simply listed: AddYears, AddMonths, AddDays, AddHours, AddMinutes, AddSeconds, AddMilliseconds, and AddTicks. A tick is 100 nanoseconds and is used for really fine timing operations.

To add 3 years and 12 hours to the current date, use the following statements:

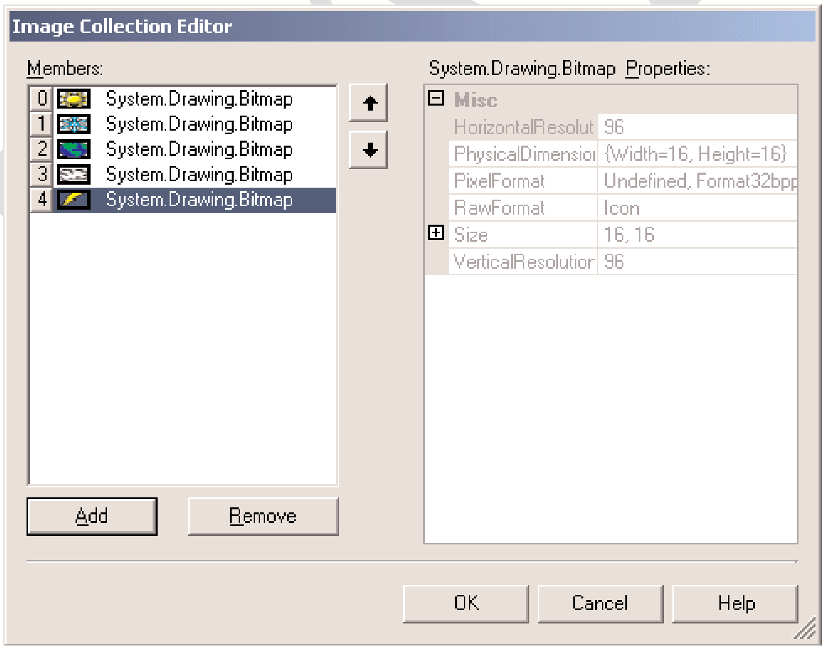
```
Dim aDate As Date
aDate = Now()
aDate = aDate.AddYears(3)
aDate = aDate.AddHours(12)
```

If the argument is a negative value, the corresponding intervals are subtracted from the current instance of the class. The following statement subtracts 2 minutes from a Date variable:

```
aDate = aDate.AddMinutes(-2)
```

The ImageList Control

The ImageList control is a really simple control that stores a number of images used by other controls at runtime. For example, a TreeView control may use a number of icons to identify its nodes. The simplest and quickest method of preparing the images to be used with the TreeView control is to create an ImageList with icons. The ImageList control maintains a series of bitmaps in memory that the TreeView control can access very quickly at runtime.



Members:
0 System.Drawing.Bitmap
1 System.Drawing.Bitmap
2 System.Drawing.Bitmap
3 System.Drawing.Bitmap
4 System.Drawing.Bitmap

System.Drawing.Bitmap Properties:

Misc	
HorizontalResolut	96
PhysicalDimensio	{Width=16, Height=16}
PixelFormat	Undefined, Format32bpp
RawFormat	Icon
Size	16, 16
VerticalResolutior	96

The other method of adding images to an ImageList control is to call the Add method of the Images collection, which contains all the images stored in the control. The Images collection provides the usual Add and Remove methods. To add an image at runtime, you must first create an Image object with the image (or icon) you want to add to the control and then call the Add method as follows:

```
ImageList1.Images.Add(image)
```

where *image* is an Image object with the desired image. You will usually call this method as follows:

```
ImageList1.Images.Add(Image.FromFile(path))
```

where *path* is the full path of the file with the image. *Images* is a collection of Image objects, not the files where the pictures are stored.

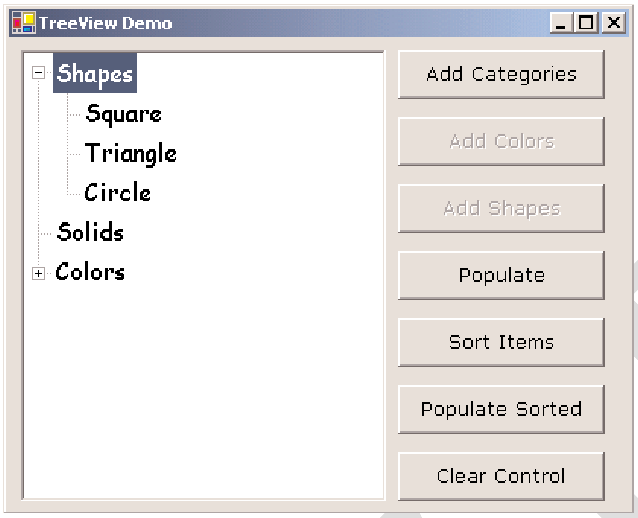
The TreeView Control

CheckBoxes If this option is enabled, a check box appears in front of each item. If the control displays check boxes, you can also select multiple items. If not, you're limited to a single selection.

FullRowSelect This True/False value determines whether the entire row of the selected item will be highlighted and whether an item will be selected even if the user clicks outside the item's text.

HideSelection This property determines whether the selected item will remain highlighted when the focus is moved to another control.

HotTracking This True/False value determines whether items are highlighted as the pointer hovers over them. When this property is True, the TreeView control behaves like a Web document with the items acting as hyperlinks—they turn blue while the pointer hovers over them. However, you can't capture this action from within your code. There's no event to report that the pointer is hovering over an item.



Indent This property indicates the indentation level in pixels. The same indentation applies to all levels of the tree—each level is indented by the same amount of pixels with respect to its parent level.

ShowLines The ShowLines property is a True/False value that determines whether the items on the control will be connected to their parent items with lines.

ShowPlusMinus The ShowPlusMinus property is a True/False value that determines whether the plus/minus button is shown next to tree nodes that have children.

ShowRootLines This is another True/False property that determines whether there will be lines between each node and root of the tree view.

Sorted This property determines whether the items in the control will be automatically sorted or not. The control sorts each level of nodes separately.

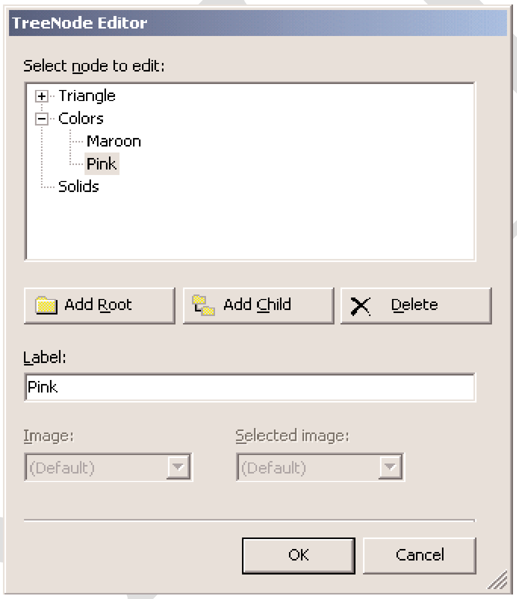
Text This is the text of the currently selected node. Use this property to retrieve the user's selection.

TopNode This is the first visible node in the TreeView control. It's the first node in the control if its contents haven't been scrolled, but it can be any node, at any level, if the control has been scrolled. The TreeNode property returns a TreeNode object, which you can use to manipulate the node from within your code.

VisibleCount This property returns the number of items that are visible on the control.

Adding New Items at Design Time

To add the root item, just click the Add Root button. The new item will be named Node0 by efault. You can change its name by selecting the item in the list. When its name appears in the Label box, change the item's name to anything you like.



You can add items at the same level as the selected one by clicking the Add Root button, or you can add items under the selected node by clicking the Add Child button. The Add Child button adds a new node under the selected node. Follow these steps to enter the root node GLOBE, a child node for Europe, and two more nodes under Europe: Germany and Italy.

Adding New Items at Runtime

Adding items to the control at runtime is a bit more involved. All the items belong to the control's Nodes collection, which is made up of TreeNode objects. To access the Nodes collection, use the following expression, where *TreeView1* is the control's name and *Nodes* is a collection of TreeNode objects:

```
TreeView1.Nodes
```

This expression returns a collection of TreeNode objects, which is called TreeNodeCollection, and it exposes the proper members for accessing and manipulating the individual nodes. The control's Nodes property is the collection of all root nodes.

The following statements will print the strings shown below them in bold (these strings are not part of the statements; they're the output the statements produce).

```
Console.WriteLine(TreeView1.Nodes(0).Text)
GLOBE
Console.WriteLine(TreeView1.Nodes(0).Nodes(0).Text)
 Europe
Console.WriteLine(TreeView1.Nodes(0).Nodes(0).Nodes(1).Text)
Italy
```

The Nodes.Add Method

The Add method adds a new node to the Nodes collection. The Add method accepts as an argument a string or a TreeNode object. The simplest form of the Add method is

```
newNode = Nodes.Add(nodeCaption)
```

where *nodeCaption* is a string that will be displayed on the control (you can't add objects to the Tree-View control). Another form of the Add method allows you to add a TreeNode object directly:

```
newNode = Nodes.Add(nodeObj)
```

To use this form of the method, you must first declare and initialize a TreeNode object:

```
Dim nodeObj As New TreeNode
nodeObj.Text = "Tree Node"
nodeObj.ForeColor = Color.BlueViolet
TreeView1.Nodes.Add(nodeObj)
```

The TreeNode object exposes a number of properties for setting its appearance. You can change its foreground and background colors, the image to be displayed in front of the node (ImageIndex property), the image to be displayed in front of the node when the node is selected (SelectedImage-Index property), and more, including the NodeFont property. You will see shortly how to assign images to the nodes of a TreeView control.

The last overloaded form of the Add method allows you to specify the index in the current Nodes collection, where the node will be added:

```
newNode = Nodes.Add(index, nodeObj)
```

The *nodeObj* Node object must be initialized as usual. The Add method inserts the new node into the current Nodes collection.

If you call the Add method on the TreeView1.Nodes collection, as we've done in the last few examples, you'll add a root item. If you call it on a child's Nodes collection, you'll add another item to the existing collection of child items. If your control contains a root item already, then this item is given by the expression

```
TreeView1.Nodes(0)
```

To add a child node to the root node, use a statement like the following:

```
TreeView1.Nodes(0).Nodes.Add("Asia")
```

The expression TreeView1.Nodes(0) is the first root node. Its Nodes property represents the nodes under the root node, and the Add method of the Nodes property adds a new node to this collection.

To add another element on the same level as the previous one, just use the same statement with a different argument. To add a country under Asia, use a statement like the following:

```
TreeView1.Nodes(0).Nodes(1).Nodes.Add("Japan")
```

This can get quite complicated, as you can understand. The proper way to add child items to a node is to create a TreeNode variable that represents the parent node, under which the child nodes will be added. The *ContinentNode* variable, for example, represents the node Europe:

```
Dim ContinentNode As TreeNode
ContinentNode = TreeView1.Nodes(0).Nodes(0)
```

The expression TreeView1.Nodes(0) is the first root node. The property Nodes(0) is the third child of the previous node (in our case, the Europe node). Then, you can add child nodes to the *ContinentNode* node:

```
ContinentNode.Nodes.Add("France")
ContinentNode.Nodes.Add("Germany")
```

To add yet another level of nodes, the city nodes, create a new variable that represents the country of the city. The Add method actually returns a TreeNode object, so you can add a country and a few cities with the following statements:

```
Dim CountryNode As TreeNode
CountryNode = ContinentNode.Nodes.Add("Germany")
CountryNode.Nodes.Add("Berlin")
CountryNode.Nodes.Add("Frankfurt")
```

Then, you can continue adding countries through the *ContinentNode* variable:

```
CountryNode = ContinentNode.Nodes.Add("Italy")
CountryNode.Nodes.Add("Rome")
```

The Count Property
This property returns the number of nodes in the Nodes collection. The expression

```
TreeView1.Nodes.Count
```

returns the number of all nodes in the first level of the control. In the case of the Globe example, it returns the value 1. The expression

```
TreeView1.Nodes(0).Nodes.Count
```

returns the number of continents in the Globe example. Again, you can simplify this expression with an intermediate TreeNode object:

```
Dim Continents As TreeNode
Continents = TreeView1.Nodes(0)
Console.WriteLine("There are " & Continents.Nodes.Count.ToString & _
                  " continents on the control")
```

The Clear Method
The Clear method removes all the child nodes from the current node. To remove all the countries under the Germany node, use a statement like the following:

```
TreeView1.Nodes(0).Nodes(2).Nodes(1).Nodes.Clear
```

The Item Property
The Item property retrieves a node specified by an index value. The expression
```
Nodes.Item(1)
```
is equivalent to the expression
```
Nodes(1)
```

The Remove Method
The Remove method removes a node from the Nodes collection. Its syntax is

```
Nodes.Remove(index)
```

where *index* is the order of the node in the current Nodes collection. To remove the selected node, call the Remove method on the SelectedNode object without arguments:

```
TreeView1.SelectedNode.Remove
```

The FirstNode, NextNode, PrevNode, and LastNode Properties

These four properties allow you to retrieve any node at the current segment of the tree. The FirstNode property will return the first city under Germany and the LastNode will return the last city under Germany. PrevNode and NextNode allow you to iterate through the nodes of the current segment:

Assigning Images to Nodes

To display an image in front of a node's caption, you must first initialize an ImageList control and populate it with all the images you plan to use with the TreeView control. The Node object exposes two image-related properties: ImageIndex and SelectedImageIndex. Both properties are the indices of an image in an ImageList control, which contains the images to be used with the control. To connect the ImageList control to the TreeView object, you must assign the name of the ImageList control to the ImageList property of the TreeView control. Then you can specify images by their index in the ImageList control.

The ImageIndex property is the index of the image you want to display in front of the node's caption. The SelectedImageIndex is the index of the image you want to display when the node is selected (expanded). Windows Explorer, for example, uses the icon of a closed folder for all collapsed nodes and the icon of an open folder for all expanded nodes. If you don't specify a value for the SelectedImageIndex property, then the image specified with the ImageIndex property will be displayed. If you haven't specified a value for this property either, then no image will be displayed for this node.

The ListView Control

The ListView control is similar to the ListBox control except that it can display its items in many forms, along with any number of subitems for each item. To use the ListView control in your project, place an instance of the control on a form and then sets its basic properties, which are described in the following sections.

The View and Arrange properties There are two properties that determine how the various items will be displayed on the control: the View property, which determines how the items will appear, and the Arrange property, which determines how the items will be aligned on the control's surface.

Table: Settings of the View Property

Setting	Description
LargeIcon	Each item is represented by an icon and a caption below the icon.
SmallIcon .	Each item is represented by a small icon and a caption that appears to the right of the icon.
List	Each item is represented by a caption.
Report	Each item is displayed in a column with its subitems in adjacent columns.

The Arrange property determines how the items will be arranged on the control, and its possible settings are show in Table 16.2.

Table : Settings of the Arrange Property

Setting	Description
Default	When an item is moved on the control, it remains where it is dropped.
Left	Items are aligned to the left side of the control.
Snap .	ToGrid Items are aligned to an invisible grid on the control. When the user moves an item, the item moves to the closest grid point on the control.
Top	Items are aligned to the top of the control.

HeaderStyle This property determines the style of the headers in Report view. It has no meaning when the View property is set to something else, because only the

Report view has columns. The possible settings for the HeaderStyle property are shown in the following table.

Table : Settings of the HeaderStyle Property

Setting	**Description**
Clickable	Visible column header that responds to clicking
Nonclickable	Visible column header that does not respond to clicking
None	No visible column header

AllowColumnReorder This property is a True/False value that determines whether the user can reorder the columns at runtime.

Activation This property specifies the action that will activate an item on the control.

Table : Settings of the Activation Property

Setting	**Description**
OneClick	Items are activated with a single click. When the cursor is over an item, it changes shape and the color of the item's text changes.
Standard .	Items are activated with a double-click. No change in the selected item's text color takes place.
TwoClick	Items are activated with a double-click and their text changes color as well

FullRowSelect This property is a True/False value indicating whether the user can select an entire row or just the item's text, and it's meaningful only in Report view.

GridLines Another True/False property. If True, then grid lines between items and subitems are drawn. This property is meaningful only in Report view.

LabelEdit The LabelEdit property lets you specify whether the user will be allowed to edit the text of the items. The default value of this property is False.

MultiSelect A True/False value indicating whether the user can select multiple items on the control or not. To select multiple items, click them with the mouse while holding down the Shift or the Control key.

Scrollable A True/False value that determines whether the scrollbars are visible or not. Even if the scrollbars are invisible, users will still be able to bring any item into view.

Sorting This property determines how the items will be sorted, and as usual it's meaningful only in Report view.

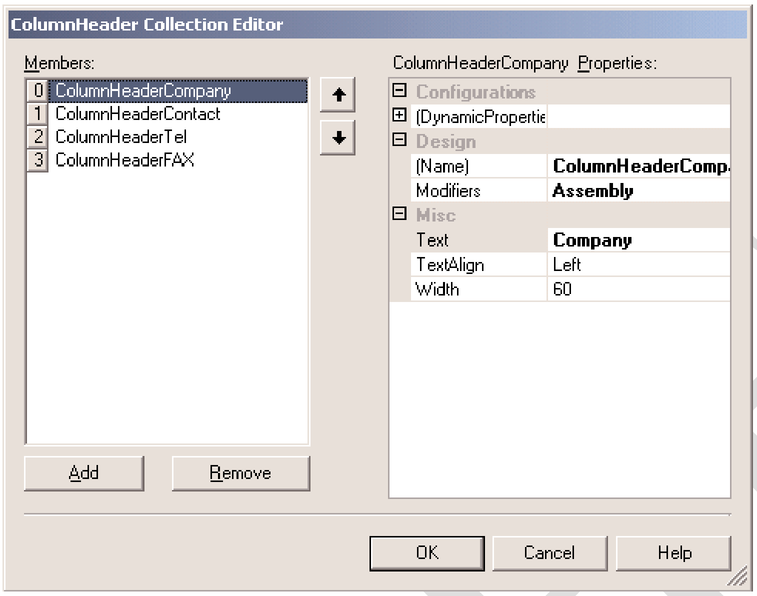
The Columns Collection

To display items in Report view, you must first set up the appropriate columns. The first column corresponds to the item, and the following columns correspond to its subitems.

It is also possible to manipulate the Columns collection from within your code, with the methods and properties discussed here.

Add method Use the Add method of the Columns collection to add a new column to the control. The syntax of the Add method is

TreeView.Columns.Add(header, width, textAlign)



ColumnHeaderCompany Properties:

Configurations	
(DynamicPropertie	
Design	
(Name)	ColumnHeaderComp
Modifiers	Assembly
Misc	
Text	Company
TextAlign	Left
Width	60

The *header* argument is the column's header. The *width* argument is the column's width in pixels, and the last argument determines how the text will be aligned. The *textAlign* argument can be Center, Justify, Left, NotSet, or Right. The NotSet setting specifies that horizontal alignment is not set.

The Add method returns a ColumnHeader object, which you can use later in your code to manipulate the corresponding column. The ColumnHeader object exposes a Name property, which can't be set with the Add method.

```
Header1 = TreeView1.Add("Column 1", 60, ColAlignment.Left)
Header1.Name = "COL1"
```

After the execution of these statements, the first column can be accessed not only by index but by name as well.

Clear method This method removes all columns.

Count property This property returns the number of columns in the ListView control. You can add more subitems than there are columns in the control, but the excess subitems will not be displayed.

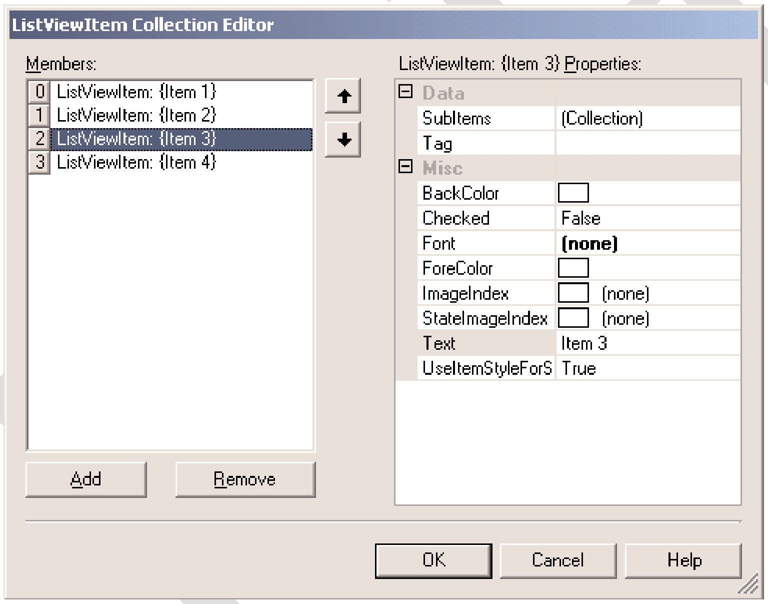
Remove method This method removes a column by its index:

ListView1.Columns(3).Remove

The indices of the following columns are automatically decreased by one.

The ListItem Object

To add items at design time, click the button with the ellipsis next to the ListItems property in the Properties window. When the ListViewItem Collection Editor window pops up, you can enter the items, including their subitems.


ListViewItem Collection Editor

Members:
0 ListViewItem: {Item 1}
1 ListViewItem: {Item 2}
2 ListViewItem: {Item 3}
3 ListViewItem: {Item 4}

ListViewItem: {Item 3} Properties:

Data	
SubItems	(Collection)
Tag	
Misc	
BackColor	☐
Checked	False
Font	**(none)**
ForeColor	☐
ImageIndex	☐ (none)
StateImageIndex	☐ (none)
Text	Item 3
UseItemStyleForS	True

Click the Add button to add a new item. Each item has subitems, which you can specify as members of the SubItems collection. To add an item with three subitems, you can populate the SubItems collection with the appropriate elements. Click the button with the ellipsis in the SubItems property on the ListViewItem Collection Editor, and the ListViewSubItem Collection Editor will appear.

BackColor property This property sets or returns the background color of the current item.

Checked property This property controls the status of an item. If it's True, then the item has been selected. You can also select an item from within your code by setting its Checked property to True. The check boxes in front of each item won't be visible unless you set the control's CheckBoxes property to True.

Font property This property sets the font of the current item. Subitems can be displayed in a different font if you specify one with the SetSubItemFont method (see the section "The SubItems Collection," later in this chapter).

Text property This property is the caption of the current item.

SubItems collection This property holds the subitems of the current ListViewItem. To retrieve a specific subitem, use a statement like the following:

```
sitem = ListView1.Items(idx1).SubItems(idx2)
```

where *idx1* is the index of the item and *idx2* the index of the desired subitem.

To add a new subitem to the SubItems collection, use the Add method, passing the text of the subitem as argument:

```
LItem.SubItems.Add("subitem's caption")
```

The argument of the Add method can also be a ListViewItem object. If you want to add a subitem at a specific location, use the Insert method. The Insert method of the SubItems collection accepts two arguments: the index of the subitem before which the new subitem will be inserted and a string or ListViewItem to be inserted:

```
LItem.SubItems.Insert(idx, subitem)
```

Like the ListViewItem objects, each subitem can have its own font, which is set with the Font property.

Remove method This method removes an item by index. When you remove an item, it takes with it all of its subitems.

SetSubItemBackColor method This method sets the background color of the current subitem.

SetSubItemForeColor method This method sets the foreground color of the current subitem. The items of the ListView control can be accessed through the ListItems property, which is a collection. As such, it exposes the standard members of a collection, which are described in the following section.

The Items Collection

All the items on the ListView control form a collection, the Items collection. This collection exposes the typical members of a collection that let you manipulate the control's items. These members are discussed next:

Add method This method adds a new item to the Items collection. The syntax of the Add method is

```
ListView1.Items.Add(caption)
```

You can also specify the index of the image to be used along with the item and a collection of subitems to be appended to the new item, with the following form of the Add method:

```
ListView1.Items.Add(caption, imageIndex)
```

where *imageIndex* is the index of the desired image on the associated ImageList control.

Finally, you can create a ListViewItem object in your code and then add it to the ListView control with the following form of the Add method:

```
ListView1.Items.Add(listItemObj)
```

The following statements create a new item, set its individual subitems, and then add the newly created ListViewItem object to the control:

```
 Dim LItem As New ListViewItem()
LItem.Text = "new item"
LItem.SetSubItem(0, "sub item 1a")
LItem.SetSubItem(1, "sub item 1b")
LItem.SetSubItem(2, "sub item 1c")
ListView1.ListItems.Add(LItem)
```

Count property Returns the number of items in the collection.
Clear method Removes all the items from the collection.
Item property Retrieves an item specified by an index value.
Remove method Removes an item from the Items collection.

The SubItems Collection

Each item in the ListView control may also have subitems. You can think of the item as the key of a record and the subitems as the other fields of the record. The subitems are displayed only in Report mode, but they are available to your code in any view. For example, you can display all items as icons, and when the user clicks on an icon, show the values of the selected item's subitems on other controls.

To access the subitems of a given item, use its SubItems collection. The following statements add an item and three subitems to the ListView1 control:

```
Dim LItem As ListViewItem
Set LItem = ListView1.Items.Add(, , "Alfreds Futterkiste")
LItem.SubItems(1) = "Maria Anders"
LItem.SubItems(2) = "030-0074321"
LItem.SubItems(3) = "030-0076545"
```