5

Windows Forms Controls: ToolStrip, MenuStrip, ContextMenuStrip, StatusStrip, and Dialog Box Controls

If you need an Immediate Solution to:	*See page:*
Using the ToolStrip Control	162
Using the MenuStrip and ContextMenuStrip Controls	165
Using the StatusStrip Control	177
Using the Dialog Box Controls	180

In Depth

You must be familiar with common Windows or Microsoft Office applications, such as Notepad, WordPad, Word, or Excel. All these applications provide a graphical user interface (GUI) with menus, toolbars, context menus, status bars, and dialog boxes to allow the users to perform various activities. Visual C# 2010 provides various controls to create such applications, along with their GUIs. Some examples of these controls are: ToolStrip, MenuStrip, ContextMenuStrip, StatusStrip, OpenFileDialog, and PrintDialog.

The ToolStrip, MenuStrip, and ContextMenuStrip controls enable you to perform common functions and make selections in an application, such as copying and pasting the text, saving a file, or changing the size or alignment of the text. You can also use the StatusStrip control in your application to show its status, for instance to show the number of lines in your document. Visual C# 2010 also provides various dialog box controls to handle the opening, saving, or printing of your files, or to set the font type or color for the text in your application. These controls include FolderBrowserDialog, OpenFileDialog, SaveFileDialog, FontDialog, ColorDialog, PrintDocument, PageSetupDialog, PrintPreviewControl, PrintPreviewDialog, and PrintDialog.

In this chapter, you learn about different Windows Forms controls. The chapter begins with the ToolStrip and MenuStrip controls, and the ToolStripItem and ToolStripMenuItem classes to create toolbars and menus in your applications. Next, you learn to create context menus and status bars using the ContextMenuStrip and StatusStrip controls, respectively. Finally, you learn about different dialog box controls to handle common tasks related to the opening, saving, or printing of files; and selecting the font type or color of the text.

The **ToolStrip** Control

The `ToolStrip` control can be used in an application by including the `ToolStrip` class, which acts as the base class for the `MenuStrip`, `StatusStrip`, and `ContextStrip` classes. It provides a graphical representation of the most commonly performed functions of any application. This control easily creates customized toolbars that support advanced user interface and layout features, such as docking, rafting, and buttons with text and images. Although tool strips are usually docked at the top of its parent window, they can also be docked to any side of a window. Figure 5.1 shows a `ToolStrip` control in the design view of Form1:

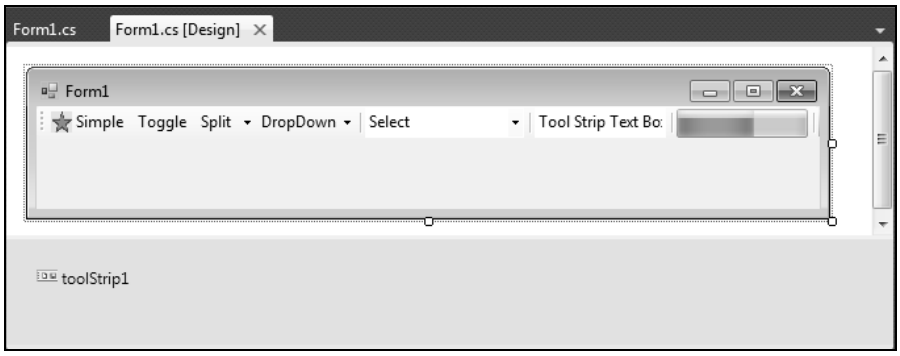

Figure 5.1: Displaying a ToolStrip Control in the Design View of Form1

The `ToolStrip` control is based on the `ToolStrip` class, which has the following class hierarchy:

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ToolStrip
```

The noteworthy properties of the `ToolStrip` class are listed in Table 5.1:

Table 5.1: Noteworthy Properties of the ToolStrip Class

Property	Description
AllowDrop	Retrieves or sets a value indicating whether or not drag-and-drop and item reordering are handled through events that you implement. The value of this property can be either `true` or `false`.
AllowItemReorder	Retrieves or sets a value indicating whether or not the `ToolStrip` class handles drag-and-drop and item reordering privately.
AllowMerge	Retrieves or sets a value indicating whether or not multiple `MenuStrip`, `ToolStripDropDownMenu`, `ToolStripMenuItem`, and other types can be combined on a `ToolStrip` control.
Anchor	Retrieves or sets the edges of the container to which a `ToolStrip` control is bound, and determines how a `ToolStrip` control is resized with its parent.
AutoSize	Specifies whether a value indicating the control is automatically resized or not.
BindingContext	Retrieves or sets the binding context for the `ToolStrip` control.
CanOverflow	Retrieves or sets a value indicating whether or not items in the `ToolStrip` control can be shown in an overflow menu.
CausesValidation	Retrieves or sets a value indicating whether or not validation is performed when an item within the `ToolStrip` control is selected.
DefaultDropDownDirection	Retrieves or sets a value representing the direction in which a `ToolStripDropDown` control is displayed relative to the `ToolStrip` control.
Dock	Retrieves or sets the `ToolStrip` control borders that are docked to the parent control, and determines how a `ToolStrip` control is resized.
Font	Retrieves or sets the font used to display text in the control.
ForeColor	Retrieves or sets the foreground color of the `ToolStrip` control.
GripDisplayStyle	Retrieves the orientation of the `ToolStrip` control move handle.
GripMargin	Retrieves or sets the space around the `ToolStrip` control move handle.
GripRectangle	Retrieves the boundaries of the `ToolStrip` control move handle.
GripStyle	Retrieves or sets whether or not the `ToolStrip` control move handle is visible or hidden.
ImageList	Retrieves or sets the image list that contains the image displayed on a `ToolStrip` item.
ImageScalingSize	Retrieves or sets the size, in pixels, of an image used on a `ToolStrip` control.
IsCurrentlyDragging	Retrieves a value indicating the current movement of the `ToolStrip` control by the user from one `ToolStripContainer` to another.
IsDropDown	Retrieves a value indicating whether or not a `ToolStrip` is a `ToolStripDropDown` control.
Items	Retrieves all the items belonging to a `ToolStrip` control.
LayoutEngine	Passes a reference to the cached `LayoutEngine` returned by the layout engine interface.
LayoutSettings	Retrieves or sets layout scheme characteristics.
LayoutStyle	Retrieves or sets a value indicating how the `ToolStrip` control places the collection of items on the screen.
Orientation	Specifies the orientation of the `ToolStripPanel control.`

Table 5.1: Noteworthy Properties of the ToolStrip Class

Property	Description
OverflowButton	Retrieves the `ToolStripItem` element that is the overflow button for a `ToolStrip` control, with overflow enabled.
Renderer	Retrieves or sets a `ToolStripRenderer` used to customize the look and feel of a `ToolStrip` control.
RenderMode	Retrieves or sets the painting styles to be applied to the `ToolStrip` control.
ShowItemToolTips	Retrieves or sets a value indicating whether or not `ToolTips` are displayed on `ToolStrip` items.
Stretch	Retrieves or sets a value indicating whether or not the `ToolStrip` control stretches from end to end in the `ToolStripContainer control`.
TabStop	Retrieves or sets a value indicating whether or not the user can give the focus to an item in the `ToolStrip` control using the `TAB` key.
TextDirection	Retrieves or sets the direction in which to draw text on a `ToolStrip` control.

The only noteworthy method of the `ToolStrip` class is the `GetNextItem()` method, which is used to retrieve the next `ToolStripItem` from a specified reference point.

Table 5.2 lists the noteworthy events of the `ToolStrip` class:

Table 5.2: Noteworthy Events of the ToolStrip Class

Event	Description
AutoSizeChanged	Occurs when the `AutoSize` property is changed
BeginDrag	Occurs when the user begins to drag the `ToolStrip` control
CausesValidationChanged	Occurs when the `CausesValidation` property changes
CursorChanged	Occurs when the `Cursor` property changes
EndDrag	Occurs when the user stops dragging the `ToolStrip` control
ForeColorChanged	Occurs when the value of the `ForeColor` property changes
ItemAdded	Occurs when a new ToolStripItem is added to the ToolStripItemCollection control
ItemClicked	Occurs when the `ToolStripItem` control is clicked
ItemRemoved	Occurs if when a ToolStripItem is removed from the ToolStripItemCollection control
LayoutCompleted	Occurs when the layout of the `ToolStrip` control is complete
LayoutStyleChanged	Occurs when the value of the `LayoutStyle` property changes
PaintGrip	Occurs when the `ToolStrip` control move handle is painted
RendererChanged	Occurs when the value of the `Renderer` property changes

Typically, the items in a `ToolStrip` control correspond to the most popular menu items in the application. If you click a tool strip item, it activates the corresponding menu item. In such cases, the code for `ToolStrip` items is easy to implement. You need to call the `PerformClick()` method of the corresponding `ToolStripMenuItem` object of the menu strip, which in turn activates the menu item.

The `Items` property of the `ToolStrip` control allows a user to add tool strip items to the tool strip. When the user points the mouse pointer at a `ToolStrip` button, tool tip is displayed.

NOTE

The ShowItemToolTips property, when set to true, can display tool tips on a ToolStrip control.

You can add various types of controls to a ToolStrip control. Some examples of these controls are buttons, push buttons, toggle buttons, drop-down buttons, and split buttons.

The `TextAlign` property specifies the alignment of the text in a button, such as the top or bottom alignment of the button. You can also add images to the tool strip items. The `DisplayStyle` property can be used to specify whether you want a text, an image, or both on the tool strip item. The `TextImageRelation` property specifies the relative placement of the image with respect to the text on a `ToolStripItem`.

The next section deals with the `ToolStripItem` class, which is used for adding items to the toolbar.

The **ToolStripItem** Class

The `ToolStripItem` class manages events and layout for all the elements present in a `ToolStrip` or `ToolStripDropDown` control. A `ToolStripItem` can be any element, such as a button, combo box, text box, or a label, present on a `ToolStrip` or `ToolStripDropDown` control. Though a toolbar is created using the `ToolStrip` control, the items present in a `ToolStrip` are actually objects of the `ToolStripItem class`. The `ToolStripItem` control must be added to a `ToolStrip`, `MenuStrip`, `StatusStrip`, or `ContextMenuStrip` control, and cannot be added directly to a form.

NOTE

A ToolStripItem can have only one parent ToolStrip control. However, the ToolStripItem control can be copied and added to other ToolStrip controls.

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

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
     System.Windows.Forms.ToolStripItem
```

The noteworthy properties of the `ToolStripItem` class are given in Table 5.3:

Table 5.3: Noteworthy Properties of the ToolStripItem Class

Property	Description
Alignment	Retrieves or sets a value that signifies whether the item aligns towards the start or the end of the `ToolStrip` control
AllowDrop	Retrieves or sets a value signifying whether drag-and-drop and item reordering is handled through events that you implement
Anchor	Retrieves or sets the edges of the container to which a `ToolStripItem` is bound, and determines how a `ToolStripItem` is resized with its parent
AutoSize	Retrieves or sets a value signifying whether or not the item is automatically resized
AutoToolTip	Retrieves or sets a value indicating whether to use the `Text` property or the `ToolTipText` property for the `ToolStripItem` ToolTip
Available	Retrieves or sets a value indicating whether or not the `ToolStripItem` control should be placed on a `ToolStrip` control
BackColor	Retrieves or sets the background color for the item
BackgroundImageLayout	Retrieves or sets the background image layout used for the `ToolStripItem` control
Bounds	Retrieves the size and location of the item
CanSelect	Retrieves a value indicating whether or not the item can be selected
ContentRectangle	Retrieves the area where content, such as text and icons, can be placed within a `ToolStripItem` without overwriting background borders

Table 5.3: Noteworthy Properties of the ToolStripItem Class

Property	Description
DisplayStyle	Retrieves or sets whether or not the text and images are displayed on a `ToolStripItem`
Dock	Retrieves or sets which `ToolStripItem` borders are docked to its parent control and determines how a `ToolStripItem` is resized with its parent
DoubleClickEnabled	Retrieves or sets a value indicating whether or not the `ToolStripItem` control can be activated by double clicking the mouse
Enabled	Retrieves or sets a value indicating whether or not the parent control of the `ToolStripItem` control is enabled
Font	Retrieves or sets the font of the text displayed by the item
ForeColor	Retrieves or sets the foreground color of the item
Height	Retrieves or sets the height, in pixels, of the `ToolStripItem` control
Image	Retrieves or sets the image displayed on the `ToolStripItem` control
ImageAlign	Retrieves or sets the alignment of the image on the `ToolStripItem` control
ImageIndex	Retrieves or sets the index value of the image that is displayed on the item
ImageScaling	Retrieves or sets a value indicating whether or not an image on the `ToolStripItem` control is automatically resized to fit in a container
ImageTransparentColor	Retrieves or sets the color to treat as transparent in a `ToolStripItem` image
IsDisposed	Retrieves a value indicating whether or not the object or control has been disposed off
IsOnDropDown	Retrieves a value indicating whether or not the container of the current control is a `ToolStripDropDown control`
IsOnOverflow	Retrieves a value indicating whether or not the `Placement` property is set to `Overflow`
Margin	Retrieves or sets the space between the item and adjacent items
MergeAction	Specifies how child menus are merged with parent menus
MergeIndex	Retrieves or sets the position of a merged item within the current `ToolStrip`
Name	Retrieves or sets the name of the item
Overflow	Retrieves or sets whether or not the item is attached to the `ToolStrip` or `ToolStripOverflowButton` control or can float between the two
Owner	Retrieves or sets the owner of the `ToolStripItem` control
OwnerItem	Retrieves the parent `ToolStripItem` class of the `ToolStripItem` control
Padding	Retrieves or sets the internal spacing, in pixels, between an item's contents and its edges
Placement	Retrieves the current layout of the item
Pressed	Retrieves a value indicating whether or not the `ToolStripItem` control is pressed
RightToLeft	Retrieves or sets a value specifying whether or not items should be placed from right to left and the text should be written from right to left
RightToLeftAutoMirrorImage	Automatically mirrors the `ToolStripItem` image when the `RightToLeft` property is set to Yes
Selected	Retrieves a value indicating whether or not the item is selected or not

Table 5.3: Noteworthy Properties of the ToolStripItem Class	
Property	**Description**
Size	Retrieves or sets the size of the item
Tag	Retrieves or sets the object that contains data about the item
Text	Retrieves or sets the text that is to be displayed on the item
TextAlign	Retrieves or sets the alignment of the text on a `ToolStripLabel`
TextDirection	Retrieves the orientation of text used on a `ToolStripItem`
TextImageRelation	Retrieves or sets the position of the `ToolStripItem` text and image relative to each other
ToolTipText	Retrieves or sets the text that appears as a `ToolTip` for a control
Visible	Retrieves or sets a value specifying whether or not the item is displayed or not
Width	Retrieves or sets the width, in pixels, of a `ToolStripItem control`

Table 5.4 lists the noteworthy methods of the `ToolStripItem` class:

Table 5.4: Noteworthy Methods of the ToolStripItem Class	
Method	**Description**
DoDragDrop ()	Starts a drag-and-drop operation
GetCurrentParent()	Retrieves a `ToolStrip` object that contains the current `ToolStripItem` control
GetPreferredSize()	Retrieves a rectangular area into which a control fits
Invalidate()	Invalidates a part or the entire surface of the control and causes the control to be redrawn
PerformClick()	Activates the `ToolStripItem` control when its `Click` event occurs
Select()	Selects the specified item

Table 5.5 lists the noteworthy events of the `ToolStripItem` class:

Table 5.5: Noteworthy Events of the ToolStripItem Class	
Event	**Description**
AvailableChanged	Occurs when the value `of the Available` property changes
BackColorChanged	Occurs when the value `of the BackColor` property changes
Click	Occurs when the `ToolStripItem` control is clicked
DisplayStyleChanged	Occurs when the `DisplayStyle property` changes
DoubleClick	Occurs when the item is double-clicked
DragDrop	Occurs when the user drags an item and then releases the mouse button, indicating that the item should be dropped into this item
DragEnter	Occurs when the user drags an item into the client area of the `ToolStripItem` control
DragLeave	Occurs when the user drags an item and the mouse pointer is no longer over the client area of this item
DragOver	Occurs when the user drags an item over the client area of this item
EnabledChanged	Occurs when the value `of the Enabled` property changes

Table 5.5: Noteworthy Events of the ToolStripItem Class	
Event	**Description**
ForeColorChanged	Occurs when the value of the `ForeColor` property changes
GiveFeedback	Occurs at a drag operation
LocationChanged	Occurs when the location of the `ToolStripItem` control is changed
MouseDown	Occurs when the mouse pointer is placed over the item and a mouse button is pressed
MouseEnter	Occurs when the mouse pointer enters the item
MouseHover	Occurs when the mouse pointer hovers over the item
MouseLeave	Occurs when the mouse pointer leaves the item
MouseMove	Occurs when the mouse pointer is moved over the item
MouseUp	Occurs when the mouse pointer is over the item and a mouse button is released
OwnerChanged	Occurs when the `Owner` property value changes
Paint	Occurs when the item is redrawn
QueryAccessibilityHelp	Occurs when an accessibility client application invokes help for the `ToolStripItem` control
RightToLeftChanged	Occurs when the value of the `RightToLeft` property changes
TextChanged	Occurs when the `Text` property value changes
VisibleChanged	Occurs when the `Visible` property value changes

Let's now learn about the MenuStrip Control

The MenuStrip Control

Menus are controls that allow a user to make selections. It also hides the selections that are not needed; therefore, saves space in Windows applications. Menus act like containers for `ToolStripMenuItem` objects and are used to display menu and menu items in a menu bar. The `MenuStrip` control represents the container for the menu structure of a form. Menus are containers of `ToolStripMenuItem` objects in general, but are flexible and can host a range of standard or custom objects. Any item that resides on a menu is known as a menu item and represents an individual part of a menu; menu items can have child menu items. The design view of a MenuStrip control is shown in Figure 5.2:

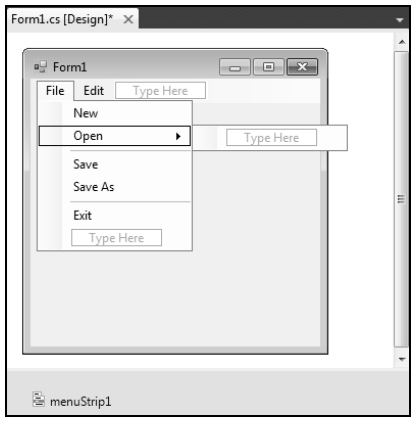

Figure 5.2: Displaying a MenuStrip Control at Design Time

There are two main classes in the standard menu handling:

- `MenuStrip`—Acts as a container for the menu structure of a form
- `ToolStripMenuItem`— Supports the items in a menu system (including the menus, such as File and Edit)

The class hierarchy of the `MenuStrip` class is as follows:

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ToolStrip
            System.Windows.Forms.MenuStrip
```

The noteworthy properties of the `MenuStrip` class are given in Table 5.6:

Table 5.6: Noteworthy Properties of the MenuStrip Class

Property	Description
CanOverflow	Retrieves or sets a value indicating whether or not the `MenuStrip` control supports overflow functionality
GripStyle	Retrieves or sets the visibility of the grip used to reposition the control
MdiWindowListItem	Retrieves or sets the `ToolStripMenuItem` control used to display a list of Multiple-Document Interface (MDI) child forms
ShowItemToolTips	Retrieves or sets a value indicating whether or not tool tips are shown for the `MenuStrip` control
Stretch	Retrieves or sets a value indicating whether or not the `MenuStrip` control stretches from end to end in its container

Table 5.7 lists the noteworthy events of the `MenuStrip` class:

Table 5.7: Noteworthy Events of the MenuStrip Class

Event	Description
MenuActivate	Occurs when the user accesses the menu with the keyboard or mouse
MenuDeactivate	Occurs when the `MenuStrip` control is deactivated

You can add submenus to the menus that pop up when the user clicks an arrow in a menu item, display check marks, create menu separator bars used to group menu items, assign shortcut keys (such as `CTRL + H`) to menu items and add images to menu items. You can also draw the appearance of menu items. These actions are supported by `ToolStripMenuItem` objects, but not by `MenuStrip` objects.

TIP

There are many menu conventions in Windows that you should adhere to if you are creating your programs for public use. Some of the menu conventions in Windows are listed as follows:

- *If a menu item opens a dialog box, you should add an ellipsis (...) after its name (such as Print...).*
- *There are many standard shortcuts already in use, such as CTRL+S for Save, CTRL+X for Cut, CTRL+V for Paste, and CTRL+C for Copy. Therefore, do not assign any different shortcuts for these functions.*
- *Menus in the menu bar that do not open a menu but instead perform some action immediately should have an exclamation point (!) after their names (such as Connect!).*
- *The File menu should always be the first menu, and an Exit item should be the last item of that menu.*

Let's now take a look at the `ToolStripMenuItem` class, which adds menus and its menu items to a standard menu bar.

The **ToolStripMenuItem** Class

The `ToolStripMenuItem` class represents an option displayed on a `MenuStrip` or `ContextMenuStrip` control. This control is a `ToolStripDropDownItem`, which is used with the `ToolStripDropDownMenu` and `ContextMenuStrip` controls for handling special highlighting, layout, and column arrangement of menus. You can create a toolbar using the `ToolStrip` control; however, note that the items in it are actually a collection of `ToolStripItem` objects.

The `ToolStripMenuItem` class supports the menus and menu items in a menu system. These menu items are objects that you can handle through the `Click` events in a menu system. The class hierarchy of the ToolStripMenuItem class is as follows:

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.ToolStripItem
        System.Windows.Forms.ToolStripDropDownItem
          System.Windows.Forms.ToolStripMenuItem
```

The noteworthy properties of the `ToolStripMenuItem` class are given in Table 5.8:

Table 5.8: Noteworthy Properties of the ToolStripMenuItem Class

Property	Description
Checked	Retrieves or sets a value indicating whether or not the `ToolStripMenuItem` control is checked
CheckOnClick	Retrieves or sets a value specifying whether or not the `ToolStripMenuItem` control should automatically appear checked and unchecked when clicked
CheckState	Retrieves or sets a value indicating whether or not a `ToolStripMenuItem` is in the checked, unchecked, or indeterminate state
Enabled	Retrieves or sets a value indicating whether or not the control is enabled
IsMdiWindowListEntry	Retrieves a value indicating whether or not the `ToolStripMenuItem` control appears on an MDI window list
ShortcutKeyDisplayString	Retrieves or sets the shortcut key text
ShortcutKeys	Retrieves or sets the shortcut keys associated with the `ToolStripMenuItem` control
ShowShortcutKeys	Retrieves or sets a value indicating whether or not the shortcut keys that are associated with the `ToolStripMenuItem` control are displayed next to the `ToolStripMenuItem` control

Table 5.9 lists the noteworthy events of the `ToolStripMenuItem` class:

Table 5.9: Noteworthy Events of the ToolStripMenuItem Class

Event	Description
`CheckedChanged`	Occurs when the value of the `Checked` property changes
`CheckStateChanged`	Occurs when the value of the `CheckState` property changes

The ToolStripMenuItem class provides properties that enable you to configure the appearance and functionality of a menu item. To display a checkmark next to a menu item, use the `Checked` property. This feature also identifies a menu item that is selected in a list of mutually exclusive menu items. The `ShortcutKeys` property defines a keyboard combination (such as `CTRL + X`) that can be pressed to select the menu item. The key combination is displayed in the menu item's caption by setting the ShowShortcutKeys property to True. `ToolStripMenuItem` objects can also have other `ToolStripMenuItem` objects attached to them, to display submenus.

To set the caption of a menu or menu item, you can use the `Text` property. Setting the `Text` property to a hyphen (-) converts the menu item into a menu separator, or the horizontal bars that group menu items together. (You can even have separators in menu bars, in which case they are vertical.) Prefixing a character in a menu item's caption with an ampersand (&) underlines that character and makes it into an access key, which means that the user can select that item by pressing the `ALT` key and underlined character on the keyboard. For example, in a menu item, the caption E&xit makes x as the access key for this menu item. You can enable and disable menu items with the `Enabled` property, and show or hide them with the `Visible` property.

In the next section, you learn about the context menus.

The ContextMenuStrip Control

Context menus are shortcut menus that pop up over controls, usually when you right-click them. They are called context menus because they appear over specific controls, i.e., in context of that control; and can be tailored to that control. Usually context menus are used to display control-specific options, such as Cut, Copy, and Paste in text boxes. You can use the `ContextMenuStrip` control to create a context menu and provide users access to frequently used menu commands. A context menu at design time is shown in Figure 5.3:

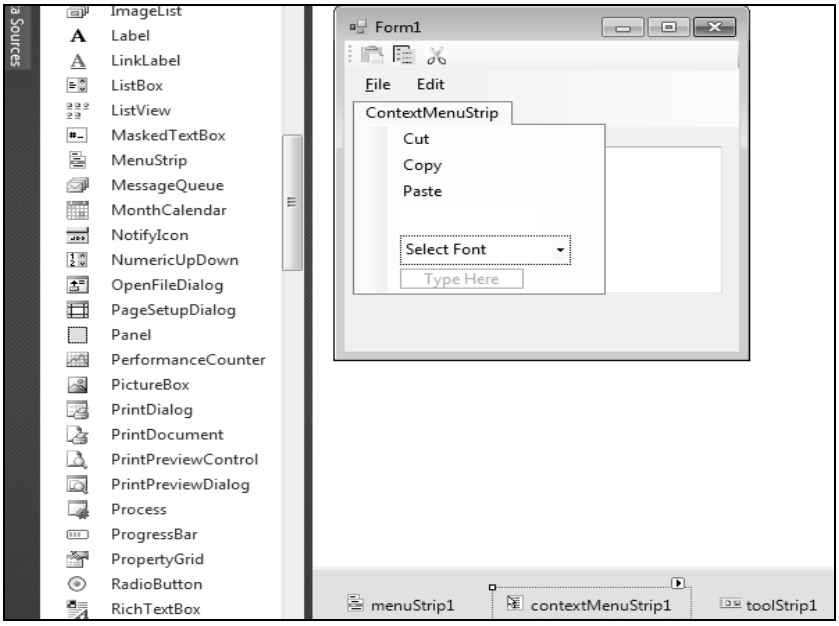

Figure 5.3: Displaying a ContextMenuStrip Control at Design Time

The `ContextMenuStrip` control is supported by the `ContextMenuStrip` class, which has the following class hierarchy:

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ToolStrip
            System.Windows.Forms.ToolStripDropDown
              System.Windows.Forms.ToolStripDropDownMenu
                System.Windows.Forms.ContextMenuStrip
```

The only noteworthy property of the ContextMenuStrip class is the SourceControl property, which is used to retrieve the last control that displayed the ContextMenuStrip control.

You associate context menus with other controls by setting the control's ContextMenuStrip property to the ContextMenuStrip control. The central property of the ContextMenuStrip control is the ToolStripMenuItems

property. You can add menu items to a context menu at design time or in code by creating ToolStripMenuItem objects and adding them to the ToolStripMenuItems collection of the context menu.

Similar to the main menus, context menu items can be disabled, hidden, or deleted. You can also show the context menus with the help of the Show() method of the `ContextMenuStrip` control. You can handle the menu item's `Click`, `Select`, and `Popup` events, as you can in main menus. In fact, the only major difference is that the context menus are not divided into separate menus, such as File, Edit, and Window.

NOTE

A context menu can be associated with a number of other controls, but each control can have only one context menu.

The next section deals with the StatusStrip control.

The StatusStrip Control

The `StatusStrip` control represents an area on Windows Form, typically at the bottom of the form, where an application can display various kinds of status information. You can use this control for performing tasks, such as showing the number of words in a document or the progress of a printing task. Typically, the `StatusStrip` control contains `ToolStripStatusLabel` objects, which may display text, an icon, or both. By default, the `StatusStrip` control has no panels. You can add panels to it by using the `ToolStripItemCollection.AddRange`() method. Alternatively, you can use the StatusStrip Items Collection Editor dialog box at design time to add panels. You can see a `StatusStrip` control in design view in Figure 5.4:

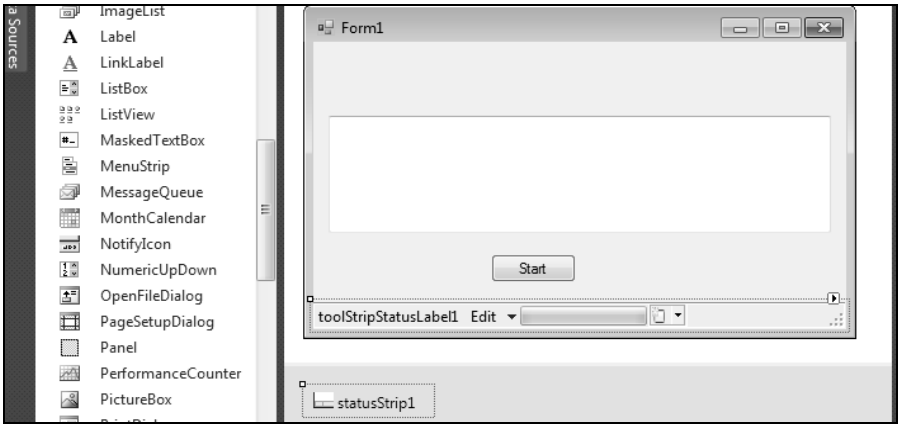

Figure 5.4: Displaying a StatusStrip Control at Design Time

The inheritance hierarchy for the `StatusStrip` class is given as follows:

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ToolStrip
              System.Windows.Forms.StatusStrip
```

Noteworthy properties of the `StatusStrip` class are given in Table 5.10:

Table 5.10: Noteworthy Properties of the StatusStrip Class

Property	Description
`CanOverflow`	Retrieves or sets a value indicating whether or not the `StatusStrip` control supports overflow functionality
`Dock`	Specifies the `StatusStrip` borders that are docked to its parent control, and determines how a `StatusStrip` control is resized with its parent control

Table 5.10: Noteworthy Properties of the StatusStrip Class	
Property	**Description**
`GripStyle`	Retrieves or sets the visibility of the grip or the sizing handle used to reposition the control
`LayoutStyle`	Retrieves or sets a value indicating how items are arranged on the `StatusStrip` control
`ShowItemToolTips`	Retrieves or sets a value indicating whether or not tooltips are shown for the `StatusStrip` control
`SizeGripBounds`	Retrieves the boundaries of the sizing handle (grip) for a `StatusStrip` control
`SizingGrip`	Retrieves or sets a value indicating whether or not a sizing handle is displayed in the lower-right corner of the control
`Stretch`	Retrieves or sets a value indicating whether or not the `StatusStrip` control stretches from end to end in its container

The `StatusStrip` control provides the following controls, which are used for displaying any type of status information in an application:

- `StatusLabel`—Displays messages
- `ProgressBar`—Displays the progress of a background task, such as saving the document or sending it to the printer
- `DropDownButton`—Displays a drop-down list
- `SplitButton`—Represents a combination of a standard button on the left and a drop-down button on the right

Now, let's take a look at the built-in dialog boxes and their corresponding controls.

The Dialog Box Controls

There are several built-in dialog boxes in Windows Forms. The built-in dialog boxes reduce the time and work required for developing commonly used dialog boxes such as file open, file save, and other dialog boxes. Some of the dialog box controls are given as follows:

- FolderBrowserDialog
- OpenFileDialog
- SaveFileDialog
- FontDialog
- ColorDialog
- PrintDocument
- PageSetupDialog
- PrintPreviewControl
- PrintPreviewDialog
- PrintDialog

The `ShowDialog`() method is used to display the dialog box at run time. You can check the return value of the `ShowDialog`() method (such as `DialogResult.OK` or `DialogResult.Cancel`) to retrieve the button clicked by a user. The possible dialog box returns values from the method to the `DialogResult` enumeration as follows:

- `Abort`—Returns an `Abort` value when the user clicks a button labeled Abort
- `Cancel`—Returns a `Cancel` value when the user clicks a button labeled Cancel
- `Ignore`—Returns an `Ignore` value when the user clicks a button labeled Ignore
- `No`—Returns a `No` value when the user clicks a button labeled No

- `None`—Returns nothing. This means that the modal dialog box continues running
- `OK`—Returns an `OK` value when the user clicks a button labeled OK
- `Retry`—Returns a `Retry` value when the user clicks a button labeled Retry
- `Yes`—Returns a `Yes` value when the user clicks a button labeled Yes

Let's take a closer look at these dialog boxes now.

The FolderBrowserDialog Control

As the name indicates, the Browse For Folder dialog box lets the user select a folder. The dialog box is shown in Figure 5.5:

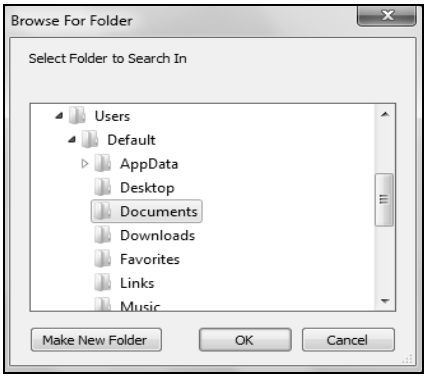

Figure 5.5: Displaying the Browse For Folder Dialog Box

The `FolderBrowserDialog` control is based on the `FolderBrowserDialog` class, which has the following class hierarchy:

```
System.Object
   System.MarshalByRefObject
     System.ComponentModel.Component
       System.Windows.Forms.CommonDialog
         System.Windows.Forms.FolderBrowserDialog
```

The noteworthy properties of the `FolderBrowserDialog` class are shown in Table 5.11:

Table 5.11: Noteworthy Properties of the FolderBrowserDialog Class

Property	Description
Description	Retrieves or sets the descriptive text displayed above the `TreeView` control in the dialog box
RootFolder	Retrieves or sets the root folder where the browsing starts
SelectedPath	Retrieves or sets the path selected by the user
ShowNewFolderButton	Retrieves or sets a value indicating whether or not the New Folder button appears in the folder browser dialog box

NOTE

You have already learned about the TreeView control in Chapter 15, Navigation Controls in ASP.NET 4.0.

The Browse For Folder dialog box is supported by the `FolderBrowserDialog` class. The dialog box also shows the Make New Folder button if the `ShowNewFolderButton` property is set to `True` (by default). The Browse For Folder dialog box prompts the user to browse and select an existing folder or create a new one and select it. Note that the `FolderBrowserDialog` class is used only to allow the user to select folders, not files and virtual folders.

The OpenFileDialog Control

The `OpenFileDialog` control allows you to select a file to open from the Open dialog box. It enables the user to check if a file exists and then opens it. You can also determine whether a read-only check box appears in the dialog box by enabling the `ShowReadOnly` property to `true`. If you enable the `ReadOnlyChecked` property to `true`, the read-only check box appears checked. You can see the Open dialog box in Figure 5.6:

Figure 5.6: Displaying an Open Dialog Box

The Open dialog box is based on the `OpenFileDialog` class, which has the following class hierarchy:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.CommonDialog
            System.Windows.Forms.FileDialog
               System.Windows.Forms.OpenFileDialog
```

The noteworthy properties of the `OpenFileDialog` class are given in Table 5.12:

Table 5.12: Noteworthy Properties of the OpenFileDialog Class

Property	Description
CheckFileExists	Retrieves or sets a value indicating if the dialog box displays a warning if the user specifies a non-existent file.
Multiselect	Retrieves or sets a value signifying whether or not the dialog box allows multiple file selections.
ReadOnlyChecked	Retrieves or sets a value signifying whether or not the read-only check box is selected on the dialog box.
SafeFileName	Retrieves the file name and extension for the selected file in the dialog box. The file name does not include the path.
SafeFileNames	Retrieves an array of file names and extensions for all the selected files in the dialog box. The file names do not include the path.
ShowReadOnly	Retrieves or sets a value signifying whether or not the dialog box displays a read-only check box.

Table 5.13 lists the noteworthy methods of the `OpenFileDialog` class:

Table 5.13: Noteworthy Methods of the OpenFileDialog Class

Method	Description
OpenFile()	Opens the file selected by the user, with read-only permission. The file is specified by the `FileName` property.
Reset()	Resets all the options to their default values.

The `OpenFileDialog` class supports the Open dialog box, which allows you to retrieve the names of files to open. If the `Multiselect` property is set to `True`, the users can also select multiple files. The `ShowReadOnly` property is used to determine if a read-only checkbox appears in the dialog box. The `ReadOnlyChecked` property indicates whether the read-only checkbox is selected. The `Filter` property sets the current file name filter string, which determines the file extension choices that appear in the dialog box. The name and path selected by the user is stored in the `FileName` property of the `OpenFileDialog` object. You can use the `OpenFile`() method to open the selected file directly.

The SaveFileDialog Control

The `SaveFileDialog` control supports the Save As dialog box that allows the user to specify the name of a file to save data to. You can use the `ShowDialog`() method to display the dialog box at run time. A Save As dialog box is shown in Figure 5.7:

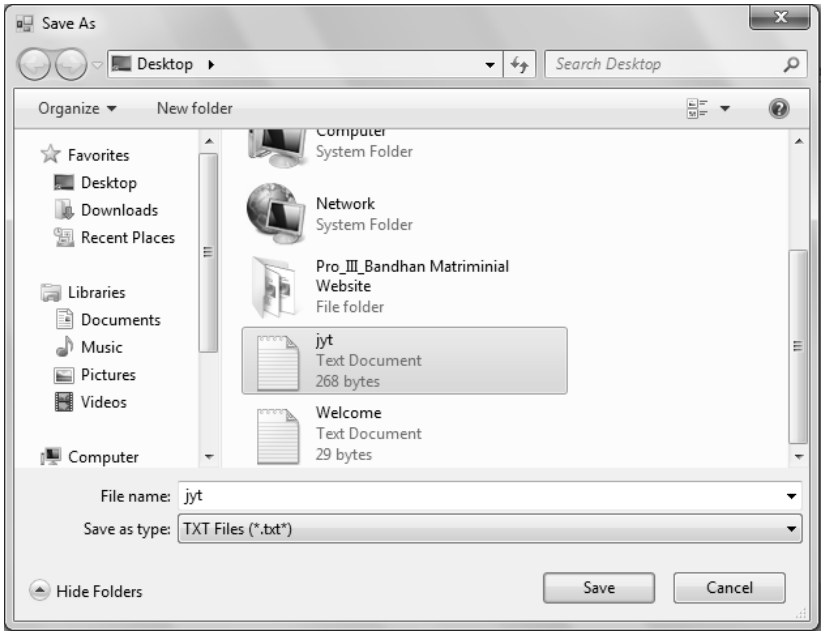

Figure 5.7: Displaying a Save As Dialog Box

The class hierarchy for the SaveFileDialog class is as follows:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.CommonDialog
            System.Windows.Forms.FileDialog
               System.Windows.Forms.SaveFileDialog
```

The noteworthy properties of the `SaveFileDialog` class are given in Table 5.14:

Table 5.14: Noteworthy Properties of the SaveFileDialog Class	
Property	**Description**
CreatePrompt	Retrieves or sets a value specifying whether or not the dialog box asks the user if it should create a file if the user specifies a non-existent file
OverwritePrompt	Retrieves or sets a value specifying whether or not the dialog box displays a warning if the user specifies a name that already exists

TIP

You can also set the CheckFileExists and CheckPathExists properties to True to check if a specified file or path already exists, or should it be created otherwise.

Table 5.15 lists the noteworthy methods of the `SaveFileDialog` class:

Table 5.15: Noteworthy Methods of the SaveFileDialog Class	
Method	**Description**
OpenFile()	Opens the file with read/write permission
Reset()	Resets all dialog box options to their default values

Let's now learn about the FontDialog control.

The FontDialog Control

The Font dialog box lets the user select the font, font size, and color. It returns `Font` and `Color` objects directly (using the properties of the same name), which can be used in controls such as rich text boxes. This saves the trouble of creating and configuring these objects from scratch. A Font dialog box is displayed, as shown in Figure 5.8:

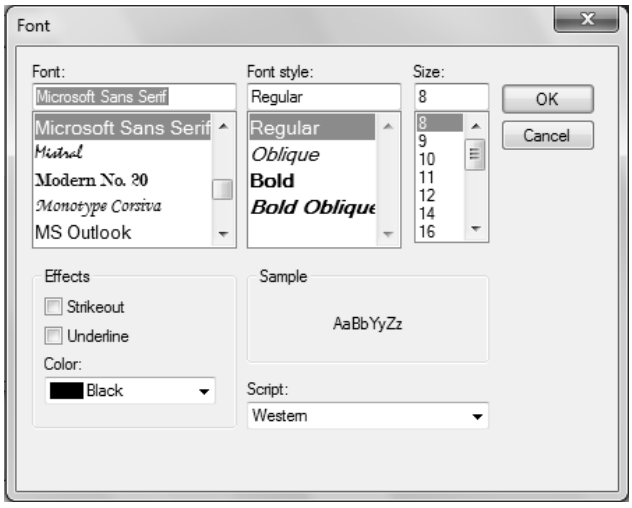

Figure 5.8: Displaying a Font Dialog Box

The `ShowDialog`() method is called to display the Font dialog box. This dialog box shows the list boxes for Font, Style, and Size, checkboxes for effects such as Strikeout and Underline, a drop-down list for Script (Script refers to different character scripts that are available for a given font—for example, Hebrew), and a sample of how the font appears. You can recover these settings using properties of the `Font` object returned by the `Font` property.

The `FontDialog` class displays a dialog box that lets the user select a font, font style, and size. It returns a `Font` object in the `Font` property, and a `Color` object in the `Color` property. The class hierarchy for the FontDialog class is as follows:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.CommonDialog
            System.Windows.Forms.FontDialog
```

The noteworthy properties of the `FontDialog` class are given in Table 5.16:

Table 5.16: Noteworthy Properties of the FontDialog Class

Property	Description
AllowSimulations	Retrieves or sets a value specifying whether or not the dialog box allows Graphics Device Interface (GDI) font simulations. GDI is responsible for drawing lines, curves, and rendering of fonts.
AllowVectorFonts	Retrieves or sets a value specifying whether or not the dialog box allows vector font selections. Vector fonts refer to the connection of lines between a series of dots that can be scaled to different sizes.
AllowVerticalFonts	Retrieves or sets a value specifying whether the dialog box displays either both vertical and horizontal fonts or only horizontal fonts.
Color	Retrieves or sets the selected font color.
FixedPitchOnly	Retrieves or sets a value specifying whether or not the dialog box allows the selection of fixed-pitch fonts, which refers to the fonts having every character with the same width.
Font	Retrieves or sets the selected font.
FontMustExist	Retrieves or sets a value specifying whether or not the dialog box specifies an error condition if the user attempts to select a font or style that does not exist.
MaxSize	Retrieves or sets the maximum font size a user can select. The default value for font size is 0.
MinSize	Retrieves or sets the minimum font size a user can select.
ShowApply	Retrieves or sets a value specifying whether or not the dialog box contains an Apply button.
ShowColor	Retrieves or sets a value specifying whether or not the dialog box displays the color choice.
ShowEffects	Retrieves or sets a value specifying whether or not the dialog box contains controls that allow the user to specify strikethrough, underline, and text color options.
ShowHelp	Retrieves or sets a value specifying whether or not the dialog box displays a Help button.

Table 5.17 lists the noteworthy methods of the `FontDialog` class:

Table 5.17: Noteworthy Methods of the FontDialog Class

Method	Description
Reset()	Resets all dialog box options to default values
ShowDialog()	Shows the dialog box

The only noteworthy event of the `FontDialog` class is the Apply event, which occurs when the user clicks the Apply button.

The ColorDialog Control

The Color dialog box lets the user select a color. The principal property you can use of this dialog box is the `Color` property, which returns a `Color` object. A Color dialog box is shown in Figure 5.9:

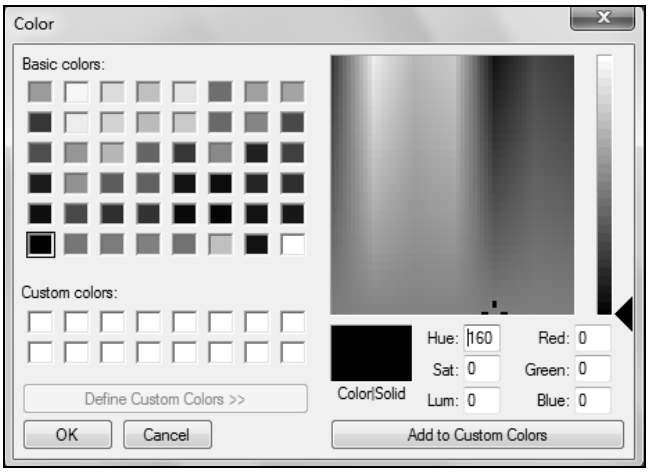

Figure 5.9: Displaying a Color Dialog Box

In Figure 5.9, you can use the Color dialog box (by clicking the Define Custom Colors button) to define your own colors with color values and hue, saturation, and luminosity. On the other hand, if you set the `AllowFullOpen` property to `False`, the Define Custom Colors button is disabled and the user can select colors only from the predefined colors in the palette.

The class hierarchy for the ColorDialog class is as follows:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.CommonDialog
            System.Windows.Forms.ColorDialog
```

The noteworthy properties of the `ColorDialog` class are given in Table 5.18:

Table 5.18: Noteworthy Properties of the ColorDialog Class

Property	Description
AllowFullOpen	Retrieves or sets a value specifying whether or not the user can use the dialog box to define custom colors
AnyColor	Retrieves or sets a value specifying whether or not the dialog box displays all available colors in the set of basic colors
Color	Retrieves or sets the color selected by the user for the text
CustomColors	Retrieves or sets the set of custom colors shown in the dialog box
FullOpen	Retrieves or sets a value specifying whether or not the controls used to create custom colors are visible when the dialog box is opened
ShowHelp	Retrieves or sets a value specifying whether or not a Help button appears in the color dialog box
SolidColorOnly	Retrieves or sets a value specifying whether or not the dialog box restrict users to selecting solid colors only and not dithered colors

Table 5.19 lists the noteworthy methods of the `ColorDialog` class:

Table 5.19: Noteworthy Methods of the ColorDialog Class

Method	Description
Reset()	Resets all the dialog box options to their default values

Table 5.19: Noteworthy Methods of the ColorDialog Class	
Method	**Description**
ShowDialog()	Shows the dialog box
ToString()	Gets a string that represents the `ColorDialog` object

The Printing Controls

The `PrintDocument` class allows you to print documents. You can add an object of this class to a project, and then handle events, such as `PrintPage`, which is called every time a new page is to be printed. When it is added to a form, the `PrintDocument` component appears in the Component Tray at the bottom of the Windows Forms Designer.

Besides the `PrintDocument` object, there are a number of dialog boxes that support printing. However, first let's have a look at the `PrintDocument` control.

The PrintDocument Control

The `PrintDocument` control supports the actual events and operations of printing in Visual C# and sets the properties that describe how to print. The `Document` property of the `PrintDialog` class needs to be set before calling the Print dialog box. This property accepts an object of the `PrintDocument` class, which obtains the printer settings and sends the output to the printer. When the `PrintDocument` control is dragged onto the form, it appears in the Component Tray at the bottom of the Windows Forms Designer, as shown in Figure 5.10:

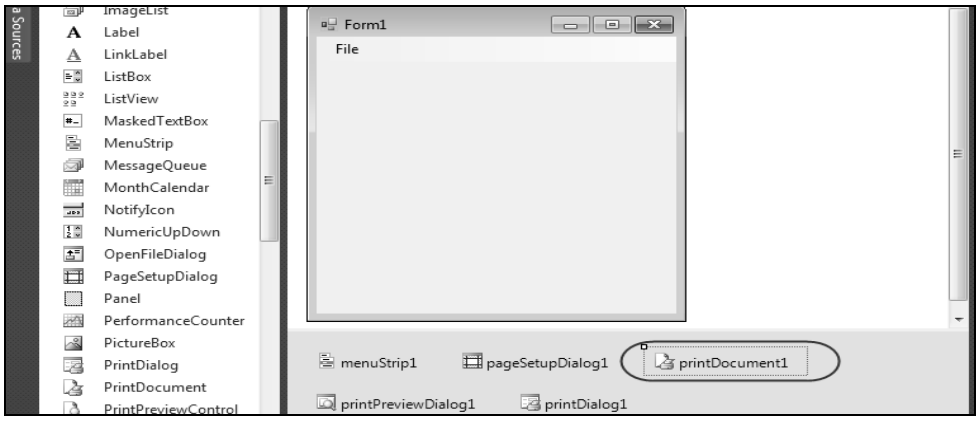

Figure 5.10: Displaying the PrintDocument Control in the Component Tray

The `PrintDocument` control is based on the `PrintDocument` class, which has the following class hierarchy:

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Drawing.Printing.PrintDocument
```

The noteworthy properties of the `PrintDocument` class are given in Table 5.20:

Table 5.20: Noteworthy Properties of the PrintDocument Class	
Property	**Description**
DefaultPageSettings	Retrieves or sets default page settings for all pages to be printed
DocumentName	Retrieves or sets the document name to display while printing the document
OriginAtMargins	Retrieves or sets a value specifying whether or not the position of a graphics object associated with a page is located just inside the user-specified margins or at the top-left corner of the printable area of the page

Table 5.20: Noteworthy Properties of the PrintDocument Class	
Property	**Description**
PrintController	Retrieves or sets the print controller for guiding the printing process
PrinterSettings	Retrieves or sets the printer that prints the document

Table 5.21 lists the noteworthy methods of the `PrintDocument` class:

Table 5.21: Noteworthy Methods of the PrintDocument Class	
Method	**Description**
Print()	Starts the printing operation
ToString()	Gives information about the print document in a string form

Table 5.22 lists the noteworthy events of the `PrintDocument` class:

Table 5.22: Noteworthy Events of the PrintDocument Class	
Event	**Description**
BeginPrint	Occurs when the `Print`() method is called and before the first page of the document is printed
EndPrint	Occurs when the last page of the document has been printed
PrintPage	Occurs when the output to print for the current page is needed
QueryPageSettings	Occurs immediately before each `PrintPage` event

The `PrinterSettings` class can also be used to configure how a document is printed, by specifying the printer to be used, number of copies to print, and the page range to print. The hierarchy of the PrinterSettings class is as follows:

```
System.Object
  System.Drawing.Printing.PrinterSettings
```

The static/shared property (which can be used with the class name, without an object, such as PrinterSettings.InstalledPrinters) of the `PrinterSettings` class is given in Table 5.23:

Table 5.23: Noteworthy Static/Shared Property of the PrinterSettings Class	
Property	**Description**
InstalledPrinters	Returns the names of all printers installed on the computer

Table 5.24 lists the noteworthy properties of the `PrinterSettings` class:

Table 5.24: Noteworthy Properties of the PrinterSettings Class	
Property	**Description**
CanDuplex	Set to `true` if the printer supports double-sided printing
Collate	Retrieves whether or not the printed document is collated
Copies	Retrieves the number of copies of the document to be printed
DefaultPageSettings	Retrieves the default page settings for the current printer
Duplex	Retrieves the printer settings for double-sided printing
FromPage	Retrieves the page number of the first page to be printed
IsPlotter	Specifies whether or not the printer is a plotter
IsValid	Specifies whether or not the printer is valid

Table 5.24: Noteworthy Properties of the PrinterSettings Class	
Property	**Description**
LandscapeAngle	Specifies the angle of the paper, in degrees, used for printing in landscape orientation
MaximumCopies	Specifies the maximum number of copies that the printer allows you to print at one time
MaximumPage	Retrieves the maximum `FromPage property` or `ToPage property` that can be selected in a `PrintDialog`
MinimumPage	Retrieves the minimum `FromPage property` or `ToPage property` that can be selected in a `PrintDialog`
PaperSizes	Specifies the paper sizes that are supported by this printer
PaperSources	Specifies the paper source trays that are available on the printer
PrinterName	Retrieves the name of the printer to be used
PrinterResolutions	Retrieves resolutions supported by the printer
PrintRange	Retrieves the page numbers to be printed, as specified by the user
PrintToFile	Retrieves a value indicating whether or not the printing output is sent to a file instead of a port
SupportsColor	Returns whether or not this printer supports color printing
ToPage	Retrieves the number of the last page to be printed

Table 5.25 lists the noteworthy methods of the `PrinterSettings` class:

Table 5.25:Noteworthy Methods of the PrinterSettings Class	
Method	**Description**
CreateMeasurementGraphics()	Retrieves a `Graphics` object that contains printer information
Clone()	Creates a copy of the `PrinterSettings class`

The PageSetupDialog Control

The Page Setup dialog box allows you to specify page orientation, paper size, and margin size. In short, the `PageSetupDialog` control sets the page related settings for printing. You can see a Page Setup dialog box, as shown in Figure 5.11:

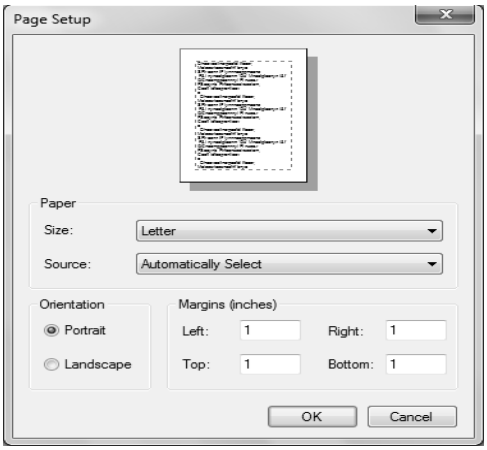

Figure 5.11: Displaying a Page Setup Dialog Box

The Page Setup dialog box is supported by the `PageSetupDialog` class, which has the following class hierarchy:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.CommonDialog
            System.Windows.Forms.PageSetupDialog
```

The noteworthy properties of the `PageSetupDialog` class are given in Table 5.26:

Table 5.26: Noteworthy Properties of the PageSetupDialog Class

Property	Description
AllowMargins	Retrieves or sets a value specifying whether or not the margins section of the dialog box is enabled.
AllowOrientation	Retrieves or sets a value specifying whether or not the orientation section of the dialog box (landscape or portrait) is enabled.
AllowPaper	Retrieves or sets a value specifying whether or not the paper section of the dialog box (paper size and paper source) is enabled.
AllowPrinter	Retrieves or sets a value specifying whether or not the Printer button is enabled.
Document	Retrieves or sets a value specifying the `PrintDocument` object which gets the page settings.
EnableMetric	Retrieves or sets a value specifying whether or not the margin settings, when displayed in millimeters, should be automatically converted to and from hundredths of an inch.
MinMargins	Retrieves or sets a value specifying the minimum margins the user is allowed to select. The value is measured in hundredths of an inch.
PageSettings	Retrieves or sets a value specifying the page settings to be modified.
PrinterSettings	Retrieves or sets the printer settings that are modified when the user clicks the Printer button in the dialog box.
ShowHelp	Retrieves or sets a value indicating whether or not the Help button is visible.
ShowNetwork	Retrieves or sets a value specifying whether or not the Network button is visible.

Table 5.27 lists the noteworthy methods of the `PageSetupDialog` class:

Table 5.27: Noteworthy Methods of the PageSetupDialog Class

Method	Description
Reset()	Resets all dialog box options to their default values
ShowDialog()	Shows the dialog box

The PrintPreviewControl Control

The `PrintPreviewControl` control displays a document to be printed; i.e., it displays a preview of the document to be printed. This control has no buttons or other user interface elements. The PrintPreviewControl control is typically used only when there is a need to write custom print-preview user interfaces. You can use `PrintPreviewControl` objects to create your own custom print previews. The `PrintPreviewControl` control is shown in Figure 5.12:

Figure 5.12: Displaying a Print Preview Control

The class hierarchy for the PrintPreviewControl class is as follows:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.Control
            System.Windows.Forms.PrintPreviewControl
```

The noteworthy properties of the `PrintPreviewControl` class are given in Table 5.28:

Table 5.28: Noteworthy Properties of the PrintPreviewControl Class

Property	Description
AutoZoom	Retrieves or sets a value that indicates whether or not the Zoom property is set automatically
Columns	Retrieves or sets the number of pages displayed horizontally
Document	Retrieves or sets a value indicating the document to be previewed
Rows	Retrieves or sets the number of pages displayed vertically
StartPage	Retrieves or sets the page number of the upper left page
UseAntiAlias	Retrieves or sets a value specifying whether or not printing uses the anti-aliasing features of the operating system
Zoom	Retrieves or sets the zoom level of the print preview

Table 5.29 lists the noteworthy methods of the `PrintPreviewControl` class:

Table 5.29: Noteworthy Methods of the PrintPreviewControl Class

Method	Description
InvalidatePreview()	Refreshes the preview of the document
ResetBackColor()	Resets the background color of the control to the color of the application workspace, which is the default color
ResetForeColor()	Resets the foreground color of the control to White, which is the default color
Show()	Shows the `PrintPreviewControl` control to the user

Table 5.30 lists the noteworthy events of the `PrintPreviewControl` class:

Table 5.30: Noteworthy Events of the PrintPreviewControl Class	
Event	**Description**
StartPageChanged	Occurs when the start page changes
TextChanged	Occurs when the value of the `Text` property changes

The next section discusses the `PrintPreviewDialog` control.

The PrintPreviewDialog Control

The `PrintPreviewDialog` control opens the Print preview dialog box and displays how a document appears when printed. This dialog box is supported by the `PrintPreviewDialog` class, and contains buttons for printing, zooming in, displaying one or multiple pages, and closing the dialog box. The Print preview dialog box is shown in Figure 5.13:

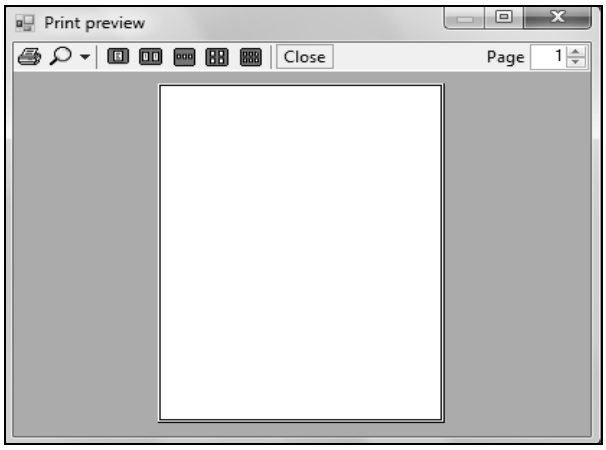

Figure 5.13: Displaying a Print Preview Dialog Box

The Print preview dialog box is supported by the `PrintPreviewDialog` class, which has the following class hierarchy:

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.Control
            System.Windows.Forms.ScrollableControl
               System.Windows.Forms.ContainerControl
                  System.Windows.Forms.Form
                     System.Windows.Forms.PrintPreviewDialog
```

The noteworthy properties of the `PrintPreviewDialog` class are given in Table 5.31:

Table 5.31: Noteworthy Properties of the PrintPreviewDialog Class	
Property	**Description**
AcceptButton	Retrieves or sets the button that is automatically clicked when the user presses the `ENTER` key
ControlBox	Retrieves or sets a value specifying whether or not a control box is displayed in the title bar of the Windows Form
Document	Retrieves or sets the document to preview
FormBorderStyle	Retrieves or sets the border style of the form
HelpButton	Retrieves or sets a value specifying whether or not a Help button should be displayed in the control box of the form

Table 5.31: Noteworthy Properties of the PrintPreviewDialog Class	
Property	**Description**
MaximizeBox	Retrieves or sets a value specifying whether or not the Maximize button is displayed in the title bar of the form
MaximumSize	Retrieves or sets the maximum size the form can be resized to
MinimizeBox	Retrieves or sets a value indicating whether or not the Minimize button is displayed in the caption bar of the form
MinimumSize	Retrieves the minimum size the form can be resized to
PrintPreviewControl	Retrieves the data contained in this form
ShowInTaskbar	Retrieves whether or not the form is displayed in the Windows taskbar
StartPosition	Retrieves or sets the starting position of the dialog box at run time
TopMost	Retrieves or sets a value specifying whether or not the form should be displayed as your application's topmost form

The main property of the dialog box is `Document`, which sets the document to be previewed. The document to be previewed must be an object of the `PrintDocument` class. The Print preview dialog box is based on the `PrintPreviewControl` object. You can also use `PrintPreviewControl` objects directly to create your own custom print preview dialog boxes.

The PrintDialog Control

The Print dialog box lets the user print documents, and this dialog box is supported by the `PrintDialog` class. A Print dialog box is shown in Figure 5.14:

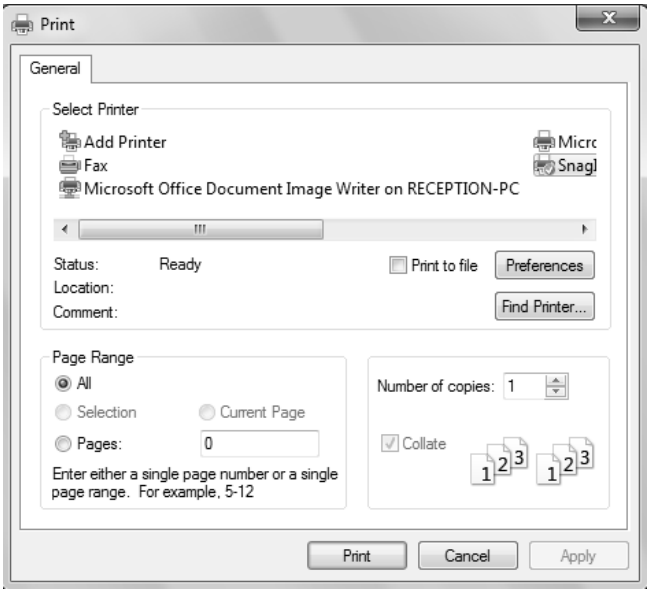

Figure 5.14: Displaying a Print dialog Box

The class hierarchy of the `PrintDialog` class is as follows:

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.CommonDialog
        System.Windows.Forms.PrintDialog
```

The noteworthy properties of the `PrintDialog` class are given in Table 5.32:

Table 5.32: Noteworthy Properties of the PrintDialog Class

Property	Description
AllowCurrentPage	Retrieves or sets a value specifying whether or not the Current Page option button is displayed. The Current Page option button prints the current page.
AllowPrintToFile	Retrieves or sets a value specifying whether or not the Print to file check box is enabled.
AllowSelection	Retrieves or sets a value specifying whether or not the Selection option button is enabled. The Selection option prints the selected text on a page.
AllowSomePages	Retrieves or sets a value specifying whether or not the Pages option button is enabled.
Document	Retrieves or sets a value specifying the `PrintDocument` control used to obtain `PrinterSettings`.
PrinterSettings	Retrieves or sets the printer settings that the dialog box modifies.
PrintToFile	Retrieves or sets a value specifying whether or not the Print to file check box is selected.
ShowHelp	Retrieves or sets a value specifying whether or not the Help button is displayed.
ShowNetwork	Retrieves or sets a value specifying whether or not the Network button is displayed.

Table 5.33 lists the noteworthy methods of the `PrintDialog` class:

Table 5.33: Noteworthy Methods of the PrintDialog Class

Method	Description
Reset()	Resets all dialog box options to their default values
ShowDialog()	Displays the dialog box

Set the `Document` property of a `PrintDialog` object with the help of a `PrintDocument` object. In the Print dialog box, a document is printed by setting the PrinterSettings property of a PrintDialog object, which indicates the number of copies to be printed. Finally, the Print() method of the PrintDocument object prints the document.

Let's now move on to the *Immediate Solutions* section, where we implement the concepts explained in the *In Depth* section.

Immediate Solutions

Using the ToolStrip Control

As mentioned previously, a `ToolStrip` control is used to create custom toolbars to support advanced user interface and layout features.

Let's understand some properties and events of the `ToolStrip` control by creating an application, named `ToolStripSample` (also available on the CD-ROM). Set the `Text` property of the Form1 form to ToolStripSample. The form contains a ToolStrip control and a Button control. Let's perform the following tasks with the ToolStrip control:

- Adding items to the `ToolStrip` control
- Handling the `Click` event of the `ToolStripButton` control
- Setting the tool tips for ToolStrip items
- Using the TAB key to Navigate through the ToolStrip items

Adding Items to the ToolStrip Control

After adding a tool strip to the Form1 form, dock it to any edge of the form—by default, the `Dock` property is set to `Top`, but it can also be set to `Bottom`, `Left`, `Right`, and `Fill`.

To add the items to a tool strip at design time, perform the following steps:

1. In the Properties window of the `ToolStrip` control, click the `Items` property at design time, which opens the Items Collection Editor dialog box, as shown in Figure 5.15:

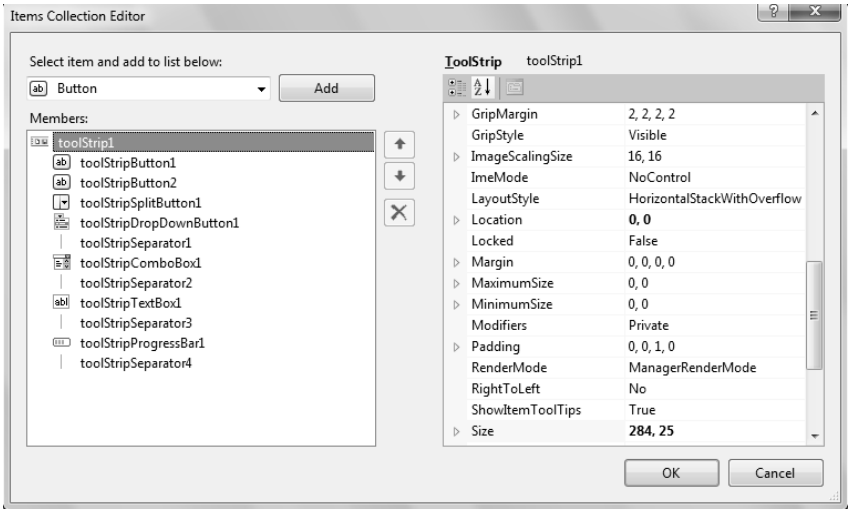

Figure 5.15: Displaying the Items Collection Editor Dialog Box

2. Select the type of items, as shown in Table 5.34, from the drop-down list beside the Add button and click the Add button. Similarly, items can be removed from the list by first selecting the item from the list and then clicking the Remove button that is denoted with a cross mark in the dialog box.

   The text for an item can also be set in the Items Collection Editor dialog box using the `Text` property. An image for the item can be added from the application resources or directly from a file using the `Image` property. To set the style for the button, use the `DisplayStyle` property and specify the arrangement of the text and image on the items using the `TextImageRelation` property. You can add one of the following items to a tool strip from the Items Collection Editor dialog box:

- `ToolStripButton`—Represents a normal button
- `ToolStripLabel`—Represents a non-selectable `ToolStripItem` that renders text, images, and hyperlinks
- `ToolStripSeparator`—Represents a line which is used to group items of a `ToolStrip` control
- `ToolStripComboBox`—Represents a combo box in a `ToolStrip` control
- `ToolStripTextBox`—Represents a text box in a `ToolStrip` control that allows the user to enter text
- `ToolStripSplitButton`—Represents a combination of a standard button on the left and a drop-down button on the right
- `ToolStripDropDownButton`—Represents a button that when clicked displays a drop-down list from which the user can select a single item

3. Set the property and value of each item in the `ToolStrip` control, as given in Table 5.34:

Table 5.34: Items to be Added to the ToolStrip Control

Item Name	Item Type	Property	Value
toolStripButton1	Button	DisplayStyle	ImageAndText
		Text	Simple
		Image	[Set an image to the `Image` property]
		TextImageRelation	ImageBeforeText
toolStripButton2	Button	DisplayStyle	Text
		Text	Toggle
		CheckOnClick	True
toolStripSplitButton1	SplitButton	DisplayStyle	Text
		Text	Split
		DropDownItems	[Add two `MenuItems` to the `DropDownItems` property]
		toolStripMenuItem1: Image property Text	[Set an image to the `Image` property] Copy
		toolStripMenuItem2: Image property Text	[Set an image to the `Image` property] Paste
toolStripDropDownButton1	DropDownButton	DisplayStyle	Text
		Text	Drop Down
		DropDownItems	[Add four `MenuItems` to the `DropDownItems` property]
		toolStripMenuItem3: Text	Red
		toolStripMenuItem4: Text	Blue
		toolStripMenuItem5: Text	Green
		toolStripMenuItem6: Text	Pick Color
toolStripSeparator1	Separator	[Nothing]	[Nothing]
toolStripComboBox1	ComboBox	Items	Verdana Tahoma

Table 5.34: Items to be Added to the ToolStrip Control			
Item Name	**Item Type**	**Property**	**Value**
		Text	Select
toolStripSeparator2	Separator	[Nothing]	[Nothing]
toolStripTextBox1	TextBox	Text	Tool Strip Text Box
toolStripSeparator3	Separator	[Nothing]	[Nothing]
toolStripProgressBar1	ProgressBar	Value	60
toolStripSeparator4	Separator	[Nothing]	[Nothing]

4. Press the F5 key on the keyboard to execute the application. Figure 5.16 shows the output of the application:

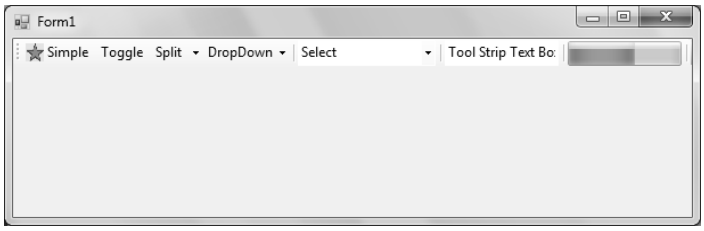

Figure 5.16: Displaying the Output of the `ToolStripSample` Application

Now, let's explain how to handle the Click event by a ToolStripControl in the next section.

Handling the Click Event of the ToolStripButton Control

When a user clicks the button item in a tool strip, the `Click` event occurs. A double-click on an item generates an event handler method for the event, similar to what you do for standard buttons. Perform the following steps in the `ToolStripSample` application, to report the text of the clicked button in a message box:

1. Double-click the Simple button on the `ToolStrip` control in the design mode of the Form1 form and modify the code of the Click event of toolStripButton1, as shown in the following code snippet:

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
   MessageBox.Show("You clicked " + sender.ToString());
}
```

2. Press the `F5` key on the keyboard to execute the application. When the form appears, click the Simple button in the `ToolStrip` control. A message box appears with a message indicating that the button has been clicked, as shown in Figure 5.17:

Figure 5.17: Displaying the Message Box

Setting the Tool Tips for ToolStrip Items

You can set tool tips for the `ToolStrip` items by first setting the `ShowItemToolTips` property of the `ToolStrip` control to `true` and then by setting the tool tip text using the `ToolTipText` property. By default, the `ShowItemToolTips` property is `true` for the `ToolStrip` control and `false` for the `MenuStrip` and `StatusStrip` controls. In addition, set the `AutoToolTip` property of the button to `false`. By default, the `AutoToolTip` property is `true` for `Button`, `DropDownButton`, and `SplitButton` items of the `ToolStrip` control.

In the ToolStripSample application, perform the following steps for setting tool tips for the items in a ToolStrip control:

1. Set the property and value of the items in the ToolStrip control, as given in Table 5.35:

Table 5.35: Setting Tool Tips for Selected Items

Items	Properties	Value
DropDownButton	AutoToolTip	false
	ToolTipText	Colors
ComboBox	ToolTipText	Font

2. Press the F5 key on the keyboard to execute the application; and when the form appears, move the cursor over the combo box. A tool tip, named Font, appears on the combo box item in the ToolStrip control, as shown in Figure 5.18:

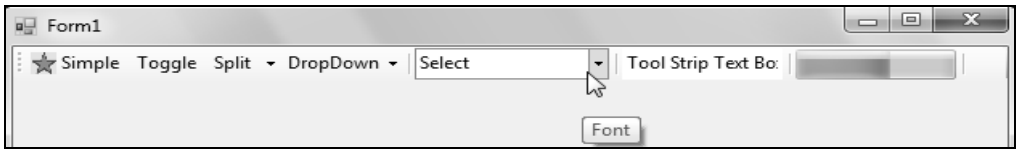

Figure 5.18: Displaying the Tool Tip for the ToolStrip Combo Box Item

Using the TAB Key to Navigate through the ToolStrip Items

To navigate the ToolStrip control, set the TabStop property of the ToolStrip control to true. When you press the TAB key, the focus moves to the ToolStrip control; after which, you can press the arrow keys to select the items within the ToolStrip control. Now, to shift the focus out of the ToolStrip control, press the TAB key second time.

Perform the following steps to enable the TAB key to navigate through the ToolStrip control:

1. Add a Button control to the Form1 form and set its Text property to Click Me in the Properties window.
2. Set the TabStop property of the ToolStrip control to true.
3. Set the TabIndex property of the ToolStrip control to 0 and that of the Button control to 1.
4. Press the F5 key on the keyboard to execute the application.

The Form1 form of the ToolStripSample application appears. As per the TabIndex set, the focus is first set on the ToolStrip control; and when you press the TAB key, the focus moves to the Button control.

Let's now learn to implement the MenuStrip and ContextMenuStrip Controls.

Using the MenuStrip and ContextMenuStrip Controls

In the *In Depth* section of the chapter, you have learned about the notable properties and events of the MenuStrip and ContextMenuStrip controls. In this section, we create an application, named MenuStripandContextMenuStripSample (also available on the CD-ROM), and then we add a Windows Form, Form1, to it. Let's perform the following tasks using the MenuStrip and ContextMenuStrip controls:

- Adding menu items to a MenuStrip control
- Adding submenus
- Disabling and enabling menu items
- Hiding and displaying menu items
- Displaying checkmarks on menu items
- Displaying separator between menu items
- Setting access keys to menu items
- Setting shortcut keys to menu items

- Adding images to menu items
- Creating context menus
- Associating a context menu with another controls
- Adding text boxes and combo boxes to menus and context menus
- Linking menu items with the ToolStrip button

Let's start by adding the menu items to a `MenuStrip` control.

Adding Menu Items

The simplest way to create menus in C# is to add a `MenuStrip` control from the Toolbox to a Windows Form, at design time. Perform the following steps to add a menu item to a `MenuStrip` control:

1. Drag and drop a `MenuStrip` control, named menustrip1, from the Toolbox to the Form1 form. When you do so, the `menustrip1` control appears.

Note: It would be more proper to call MenuStrip control a `MenuStrip` component because it does not inherit the `Control` class and appears in the Component Tray. However, still it is called a `MenuStrip` control.

2. Click the Type Here text to open a text box and enter the File text in it. Another text box with the Type Here text is added below this text box.

Now, you can add some items to the File menu you have just created.

3. Enter the name of the menu item in the newly added Type Here text box. When you enter some text in a Type Here text box, two more Type Here text boxes are added—one below and another to the right of this Type Here text box. You can see the items added to the File menu in Figure 5.19:

Figure 5.19: Adding Menu Items to the File Menu

You can make the menu items perform some actions by creating event handlers for handling their `Click` event in the Code Editor.

4. Double-click the Exit menu item of the File menu and modify the code of the Click event of exitToolStripMenuItem, as shown in the following code snippet:

```
private void exitToolStripMenuItem_Click(object sender, System.EventArgs e) {
  this.Close();
}
```

In the preceding code snippet, the `Close`() method is used to exit the program.

5. Press the `F5` key on the keyboard to execute the application and the menu appears on the Form1 form. Now, click the Exit menu item to close the application.

Let's now see how to add submenus to these menu items.

Adding Submenus

In the previous topic, you learn how to add menu items to a `MenuStrip` control. You can also create submenus, which involves adding menu items within another menu item. When a menu item has a submenu, a right pointing arrow appears in that menu item at run time. Clicking that arrow opens the submenu, displaying additional menu items. Note that submenus can also have sub submenus, which can further have other submenus as well.

To add submenus to the New menu item of the File menu, perform the following steps:

1. Click the New menu item. It opens a Type Here textbox to the right of the New menu item.
2. Click the Type Here textbox and enter the caption for the submenu item as Item1, as shown in Figure 5.20:

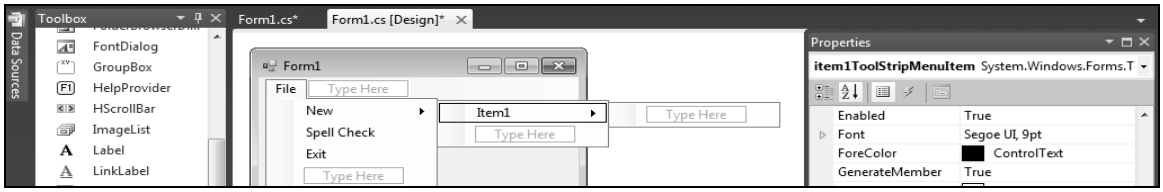

Figure 5.20: Adding Submenus to the New Menu Item

3. Similarly, add one more submenu to the New menu item as Item2. Now in the same way, add two submenus – Item3 and Item4 to the Item2 submenu.
4. Double-click the Item1 submenu and modify the code of the Click event of item1ToolStripMenuItem, as shown in the following code snippet:

```
private void item1ToolStripMenuItem_Click(object sender, System.EventArgs e)
{
   MessageBox.Show("You clicked " + sender.ToString());
}
```

In the preceding code snippet, the `Show`() method of the `MessageBox` class is used to display a message to the user.

5. Press the `F5` key on the keyboard to execute the application. When the form appears, click File → New → Item2 and the output is displayed, as shown in Figure 5.21:

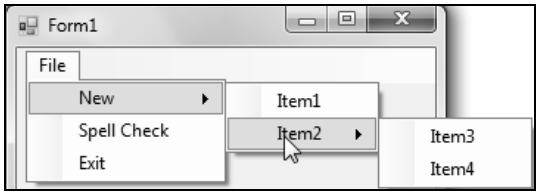

Figure 5.21: Displaying Submenus of the New Menu Item

6. Click the Item1 submenu, a message box is displayed, as shown in Figure 5.22:

Figure 5.22: Displaying the Message Box on the Selection of a Submenu

Disabling and Enabling Menu Items

To disable or gray out a menu item, so that it cannot be selected, you need to set its `Enabled` property to `false`. Perform the following steps to disable and enable the menu items:

1. Double-click the Item3 submenu in the design mode of the Form1 form and modify the code of the Click event of item3ToolStripMenuItem, as shown in the following code snippet:

```
private void item3ToolStripMenuItem_Click(object sender, System.EventArgs e)
{
   MessageBox.Show("Disabling Spell Check menu item");
   spellCheckToolStripMenuItem.Enabled = false;
}
```

2. Press the `F5` key on the keyboard to execute the application. When the form appears, click the Item3 submenu. A message box appears displaying the message, Disabling Spell Check menu item.
3. Click the OK button to close the message box. The Spell Check menu item gets disabled (Figure 5.23).

4. Enable the menu item by setting the `Enabled` property of the menu item to `true`. To do so, double-click the Item2 submenu in the design mode of the Form1 form and modify the code of the Click event of item2ToolStripMenuItem, as shown in the following code snippet:

```
private void item2ToolStripMenuItem_Click(object sender, EventArgs e)
{
 MessageBox.Show("Enabling Spell Check menu item");
 spellCheckToolStripMenuItem.Enabled = true;
}
```

5. Press the `F5` key on the keyboard to execute the application. When the form appears, first disable the Spell Check menu item by clicking the Item3 submenu and then click the Item2 submenu to enable the Spell Check menu item. A message box appears, as shown in Figure 5.23:

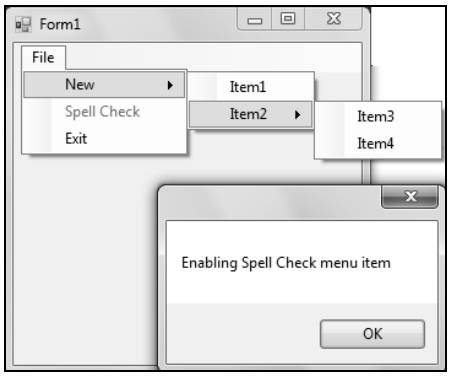

Figure 5.23: Displaying a Message Box on the Selection of the Item2 Submenu

6. Click the OK button to close the message box. The Spell Check menu item gets enabled, as shown in Figure 5.24:

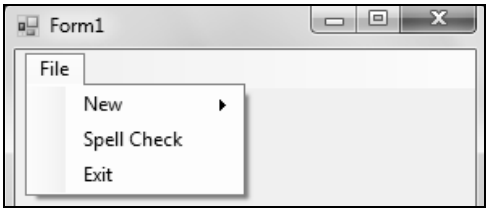

Figure 5.24: Displaying the Enabled Menu Item

Hiding and Displaying Menu Items

Using the `Visible` property, you can either show or hide menu items. Let's use the MenuStripandContextMenuStripSample application (also available on the CD-ROM). This application shows that when you select the submenu Item4, the program hides this submenu. Perform the following steps to hide and display the menu items:

1. Double-click the Item4 submenu in the design mode of the Form1 form and modify the code of the Click event of item4ToolStripMenuItem, as shown in the following code snippet:

```
private void item4ToolStripMenuItem_Click(object sender, System.EventArgs e)
{
  MessageBox.Show("Hiding Item4");
  item4ToolStripMenuItem.Visible = false;
}
```

2. Press the `F5` key on the keyboard to execute the application. When the form appears, select the Item4 submenu. A message box appears displaying the message, Hiding Item4.
3. Click the OK button to close the message box. The Item4 submenu is hidden, as shown in Figure 5.25:

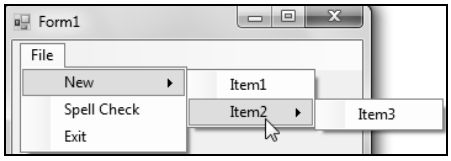

Figure 5.25: Hiding the Item 4 Submenu

4. Display the hidden submenu by setting the `Visible` property of the submenu to `true`. Modify the code of the Click event of Item1 submenu, as shown in the following code snippet:

```
private void item1ToolStripMenuItem_Click(object sender, EventArgs e)
{
 MessageBox.Show("You clicked " + sender.ToString());

 // code to display item 4
 item4ToolStripMenuItem.Visible = true;
 MessageBox.Show("Displaying Item 4");
}
```

5. Press the `F5` key on the keyboard to execute the application. When the form appears, first hide the Item4 submenu and then click the Item1 submenu. A message box appears displaying the You clicked Item1 message. After pressing the OK button, the second message box appears with the message, Displaying Item4.
6. Click the OK button to close the message box. The Item4 submenu is displayed in the menustrip1 control.

Displaying Checkmarks on Menu Items

A checkmark is used to indicate to a user that a specific option has been selected. An example of this is displayed in Figure 5.26, where a check mark appears in front of the Spell Check menu item in the `MenuStripandContextMenuStripSample` application.

You can use the `Checked` property of a `ToolStripMenuItem` object to toggle the checkmark; where `true` means the checkmark is displayed, `false` means it is hidden.

Perform the following steps to toggle the checkmark in front of the Spell Check menu item on and off when you select that item:

1. Double-click the Spell Check menu item in the design mode of the Form1 form and modify the code of the Click event of spellCheckToolStripMenuItem, as shown in the following code snippet:

```
private void spellCheckToolStripMenuItem_Click(object sender, EventArgs e)
{
   spellCheckToolStripMenuItem.Checked = !spellCheckToolStripMenuItem.Checked;
}
```

2. Press the `F5` key on the keyboard to execute the application. When the form appears, the Spell Check menu item is unchecked. Select the menu and a checkmark appears in front of it, as shown in Figure 5.26:

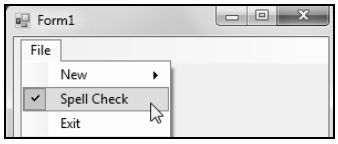

Figure 5.26: Displaying a Menu Item Checked

If you select the menu item again, the checkmark disappears from the Spell Check menu item.

Displaying a Separator between the Menu Items

A menu separator is a line used to group the items into functional groups. You can right-click the menu item and select the Separator option from the context menu to create a menu separator.

To create a menu separator between the New, Spell Check, and Exit menu items, perform the following steps:

1. Right-click the Spell Check menu item and select the Insert → Separator option from the context menu.

A menu separator is added before the Spell Check menu item, as shown in Figure 5.27:

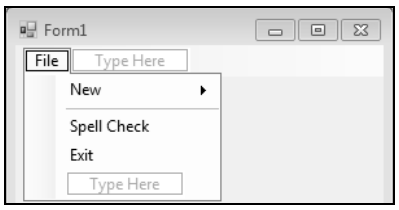

Figure 5.27: Adding a Separator between the New and Spell Check Menu Items

2. Press the `F5` key on the keyboard to execute the application. When the form appears, click the File menu, you can see the separators between the New and Spell Check menu items.

Let's now learn to set the access keys for menu items.

Setting Access Keys for Menu Items

Access keys make it possible to select menu items from the keyboard by using the `ALT` key. For example, if you make the character F in the File menu's caption an access key, you can use the `ALT + F` keys to open the File menu. Similarly, you can make the x in the Exit menu item as an access key. As learned earlier, to give an item an access key, you precede the access key in its text with an ampersand (&). For example, in this case, use the text &File and E&xit.

Note that you still have to open a menu item's menu to be able to use its access key. If you want to assign a key to a menu item that can be used without first opening that item's menu, use shortcut keys instead of access keys.

In the `MenuStripandContextMenuStripSample` application, perform the following steps to set access keys for the File menu and Exit menu items:

1. Select the File menu in the design mode of the Form1 form and in the Properties window set the `Text` property of the File menu to &File and press the Enter key on the keyboard. You can see the character F underlined (Figure 5.28).
2. Similarly, set x as the access key for Exit menu item.
3. Press the `F5` key on the keyboard to execute the application. When the form appears, you cannot see the access keys. To make them visible, press the `ALT` key on the keyboard and you can see the access keys underlined, as shown in Figure 5.28:

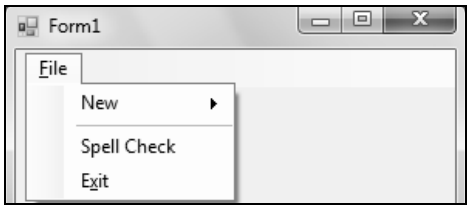

Figure 5.28: Displaying the Access Keys

Setting the Shortcut Keys for Menu Items

When you press the shortcuts for menu items, the associated menu item is selecting, resulting in the occurrence of its `Click` event. You set a shortcut with the `ShortcutKeys` property. To display the shortcut next to the menu item's caption at run time, you set the `ShowShortcutKeys` property to `true` (it is `true` by default).

In the `MenuStripandContextMenuStripSample` application, perform the following steps to assign the shortcut key `CTRL+E` for the Exit menu item:

1. Select the Exit menu item and in the Properties window, select Ctrl as the Modifier and E as the Key, as shown in Figure 5.29:

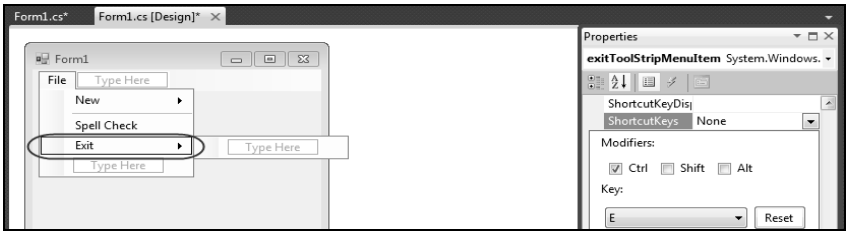

Figure 5.29: Setting the Shortcut keys

2. Press the `F5` key on the keyboard to execute the application and the output is displayed, as shown in Figure 5.30:

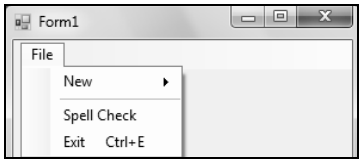

Figure 5.30: Displaying the Shortcut Keys

When the user presses the `CTRL+E` keys, the Exit menu item gets activated. You can also assign a short cut key to to a tool strip menu item by using the members of the `ShortcutKeys` enumeration. The following code snippet shows how to assign a short cut key to a tool strip menu item:

```
    ToolStripMenuItem.ShortcutKeys = Shortcut.CtrlE;
```

You need to use the preceding code snippet in the Load event of a form

NOTE

Shortcuts select their corresponding menu items even if no menu is open at the time. If you want to make sure the user must first open the item's menu, use access keys instead.

Adding Images to the Menu Items

You can use the Image property of the menu item to set images to the menu items. In the `MenuStripandContextMenuStripSample` application, perform the following steps to add an image to the New menu item:

1. Click the … (ellipse) button next to the `Image` property in the properties window of the New menu item. The Select Resource dialog box opens, where you can import an image or add one to the application resources, as shown in Figure 5.31:

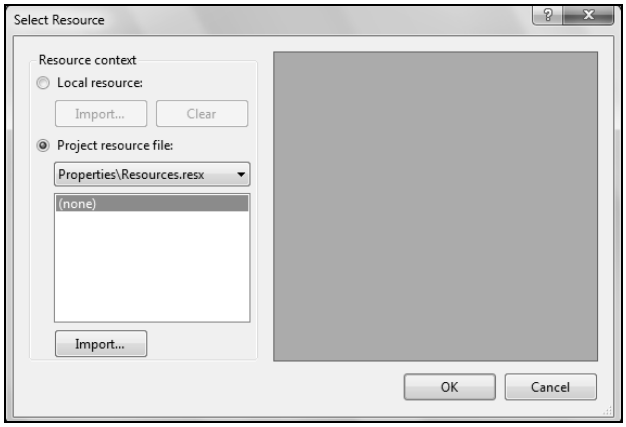

Figure 5.31: Displaying the Select Resource Dialog Box

2. Select the Project resource file radio button and click the Import button, the Open dialog box appears from where you can add images to your project resource file.
3. Select the images and click the OK button to add the images to the project resource file. The images appear in the Select Resource dialog box. In this case, the NEW.jpg image is added.
4. Select the NEW.jpg image from the list and click the OK button.
5. Press the `F5` key on the keyboard to execute the application. When the form appears, the image appears in front of the New menu item, as shown in Figure 5.32:

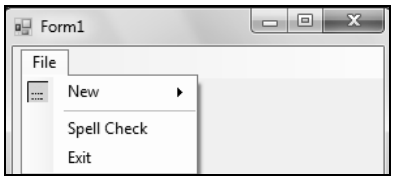

Figure 5.32: Displaying an Image beside the New Menu Item

Some images have a background that you may not want to display when the user selects the menu option. To make this background disappear, set the background color of the image to `Transparent` using the `ImageTransparentColor` property.

Creating Context Menus

Creating context menus is similar to creating standard menus—you only need to add a `ContextMenuStrip` control to the Windows Form. The Text property for this context menu is `contextMenuStrip1`, the rest of the coding and everything else is same as creating any standard menu. You can add menu items to the context menu by following the same procedure which was used for creating standard menus.

Perform the following steps to add context menu items, such as Cut, Copy, and Paste:

1. Drag and drop a `ContextMenuStrip` control on the Form 1 form and then add three menu items to this context menu, such as Cut, Copy, and Paste, as shown in Figure 5.33:

Figure 5.33: Adding Menu Items to the Context Menu

2. Double-click the Cut menu item and modify the code of the Click event of cutToolStripMenuItem, as shown in the following code snippet:

```
private void cutToolStripMenuItem_Click(object sender, EventArgs e)
{
   MessageBox.Show("You clicked the Cut menu item");
}
```

In the code, at the `Click` event of the Cut menu item, a message box is displayed.

3. Double-click the Copy menu item and modify the code of the Click event of copyToolStripMenuItem, as shown in the following code snippet:

```
private void copyToolStripMenuItem_Click(object sender, EventArgs e)
{
   MessageBox.Show("You clicked the Copy menu item");
}
```

In the code, at the `Click` event of the Copy menu item, a message box is displayed.

4. Double-click the Paste menu item and modify the code of the Click event of pasteToolStripMenuItem, as shown in the following code snippet:

```
private void pasteToolStripMenuItem_Click(object sender, EventArgs e)
{
  MessageBox.Show("You clicked the Paste menu item");
}
```

In the code, at the `Click` event of the Paste menu item, a message box is displayed.

5. Press the `F5` key on the keyboard to execute the application. Now, if you right-click the form, you do not see any context menu. This is because we have just defined menu items for the `ContextMenuStrip` control. To display the context menu, you need to associate the `ContextMenuStrip` control with either any Windows Forms control or with the Windows Form.

Associating a Context Menu with a Control

To associate a context menu with a control, you need to set the ContextMenuStrip property of the control, such as a button or a text box, with a context menu control, for instance ContextMenuStrip1.

Perform the following steps to associate a context menu with the text box:

1. Drag and drop a TextBox control to the Form 1 form and in the Properties window, set its `Multiline` property to `true`.
2. Assign `ContextMenuStrip1` to the `ContextMenuStrip` property of the text box.
3. Add the following highlighted code in the `Click` event of the Cut menu item:

```
private void cutToolStripMenuItem_Click(object sender, EventArgs e)
{
  MessageBox.Show("You clicked the Cut menu item");
  textBox1.Cut();
}
```

In the code, the `Cut`() method of the text box moves the selected text in the text box to the clipboard.

4. Add the following highlighted code in the `Click` event of the Copy menu item:

```
private void copyToolStripMenuItem_Click(object sender, EventArgs e)
{
  MessageBox.Show("You clicked the Copy menu item");
  textBox1.Copy();
}
```

In the code, the `Copy`() method of the text box copies the selected text in the text box to the clipboard.

5. Add the following highlighted code in the `Click` event of the Paste menu item:

```
private void pasteToolStripMenuItem_Click(object sender, EventArgs e)
{
  MessageBox.Show("You clicked the Paste menu item");
  textBox1.Paste();
}
```

In the code, the `Paste`() method of the text box replaces the selected text in the text box with the contents in the clipboard.

6. Press the `F5` key on the keyboard to execute the application. Now enter some text in the text box, select it, and right-click the text box, the context menu appears, as shown in Figure 5.34:

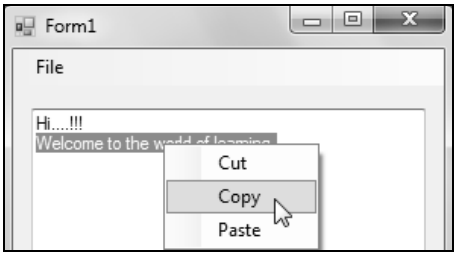

Figure 5.34: Using the ContextMenuStrip Control

Now, you can select any of the items from the context menu and perform the respective task.

Adding Text Boxes and Combo Boxes to Menus and Context Menus

Apart from hosting ToolStripMenuItem objects, MenuStrip can host a number of other standard menu item objects, such as ToolStripComboBox, ToolStripSeparator, and ToolStripTextBox objects. You can add any of these items to the menu using the Type Here combo box.

Perform the following steps to add a text box and a combo box to the `MenuStrip` control:

1. Add an Edit menu to the `MenuStrip` control. Then, add a Replace With menu item to the Edit menu.
2. Add a text box beside the Replace With option by selecting the `TextBox` option from the Type Here combo box, as shown in Figure 5.35:

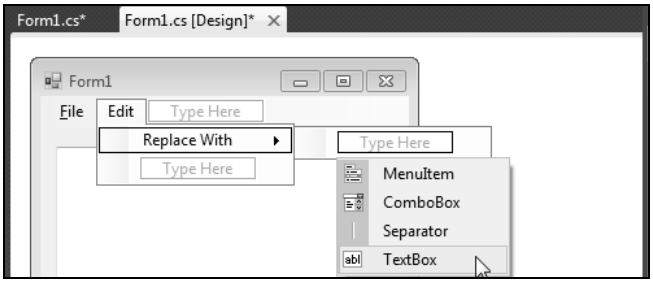

Figure 5.35: Adding a Text Box to the Menu

3. Add a Font menu item under the Replace With option and add combo box below the Font menu item by selecting the `ComboBox` option from the Type Here combo box.
4. Select the `Items` property of the combo box. A String Collection Editor dialog box appears.
5. Add these seven items in the dialog box – Arial, Calibri, Comic Sans MS, Impact, Tahoma, Times New Roman, and Verdana and set the `Text` property of the combo box to Select.

All these items work like the standard `ToolStripMenuItem` objects only, but with added advantages. For example, in the menu that opens when you click the Replace With submenu, you can type something in the text box and press the Enter key. The item that you have typed is automatically added to the menu item.

6. Modify the code of the SelectedIndexChanged event of toolStripComboBox1 to set the selected font on the newly added item, as shown in the following code snippet:

```
private void toolStripComboBox1_SelectedIndexChanged(object sender, System.EventArgs e)
{
   toolStripTextBox1.Font = new Font(toolStripComboBox1.SelectedItem.ToString(),
    (float)(Font.Size), Font.Style);
}
```

7. Press the `F5` key on the keyboard to execute the application, when the form appears, select the Edit → Replace With option and enter the text C# 2010 in the text box beside the Replace With option.
8. Select Comic Sans MS option from the combo box in the Edit menu, the text entered in the text box changes to Comic Sans MS, as shown in Figure 5.36:

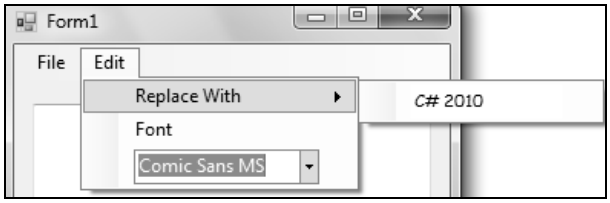

Figure 5.36: Adding a Combo Box to a Menu

Similarly, you can also add a text box and a combo box to the context menu. Let's perform the following steps to do so:

1. Add a text box to the context menu below the Paste option by selecting the `TextBox` option from the Type Here combo box, as shown in Figure 5.37:

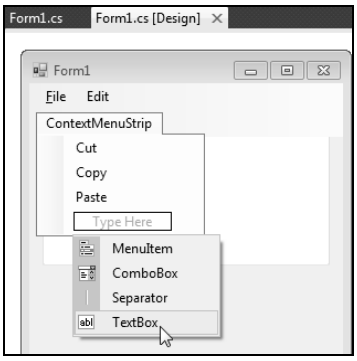

Figure 5.37: Adding Text Box to the Context Menu

2. Add a combo box to the context menu below the text box by selecting the `ComboBox` option from the Type Here combo box.
3. Select the `Items` property of the combo box. A String Collection Editor dialog box appears.
4. Add these seven items in the dialog box – Arial, Calibri, Comic Sans MS, Impact, Tahoma, Times New Roman, and Verdana and set the `Text` property of the combo box to Select Font.
5. Modify the code of the SelectedIndexChanged event of toolStripComboBox2 to set the selected font on the text box font, as shown in the following code snippet:

```
private void toolStripComboBox2_SelectedIndexChanged(object sender, System.EventArgs e)
{
   textBox1.Font = new Font(toolStripComboBox2.SelectedItem.ToString(),
    (float)(Font.Size), Font.Style);
}
```

6. Press the `F5` key on the keyboard to execute the application, when the form appears, enter the text `Welcome to the world of .NET. Enjoy learning Visual C#` and right-click on the Form1 form (Figure 5.38).
7. Type Undo in the text box of the context menu, the menu item is added in the context menu. In addition, also select Impact option from the combo box of the context menu, the font of the text in the text box changes, as shown in Figure 5.38:

Figure 5.38: Adding Combo Box and Text Box Controls to a ContextMenu Control

Linking Menu Items with the ToolStrip Button

Typically, `ToolStrip` buttons correspond to frequently used menu items. To connect a `ToolStrip` to a menu item, you can use the menu item's `PerformClick`() method.

In the MenuStripandContextMenuStripSample application, perform the following steps to link menu items with the ToolStrip button:

1. Drag and drop a ToolStrip control to the Form1 form.
2. Add the items, given in Table 5.36, to the ToolStrip control and set the properties and values of the ToolStrip items, as given in Table 5.36:

Table 5.36: Items to be Added to the ToolStrip Control

Item Name	Item Type	Property	Value
toolStripButton1	Button	DisplayStyle	Image
		Image	[Set an image to the Image property]
		ToolTipText	Cut
toolStripButton2	Button	DisplayStyle	Image
		Image	[Set an image to the Image property]
		ToolTipText	Copy
toolStripButton3	Button	DisplayStyle	Image
		Image	[Set an image to the Image property]
		ToolTipText	Paste

3. Add three menu items to the Edit menu, Cut, Copy, and Paste; and modify the code of their respective Click events, as shown in Listing 5.1:

Listing 5.1: Showing the Code of the Cut, Copy, and Paste Menu Items

```
private void CopyToolStripMenuItem1_Click(object sender, System.EventArgs e)
{
  textBox1.Copy();
}
private void PasteToolStripMenuItem1_Click(object sender, System.EventArgs e)
{
  textBox1.Paste();
}
private void cutToolStripMenuItem1_Click(object sender, EventArgs e)
{
 textBox1.Cut();
}
```

4. Modify the code of the Click events of the button items of the ToolStrip control, as given in Listing 5.2:

Listing 5.2: Showing the Code of the Cut, Copy, and Paste Button Items of the ToolStrip Control

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
 pasteToolStripMenuItem1.PerformClick();
}
private void toolStripButton2_Click(object sender, EventArgs e)
{
 copyToolStripMenuItem1.PerformClick();
}
private void toolStripButton3_Click(object sender, EventArgs e)
{
 cutToolStripMenuItem1.PerformClick();
}
```

5. Press the F5 key on the keyboard to execute the application. When the form appears, type Hello World in the text box. Select the entire text in the text box and click the Copy icon on the ToolStrip control. Then paste the text anywhere in the text box by clicking on the Paste icon on the ToolStrip control, as shown in Figure 5.39:

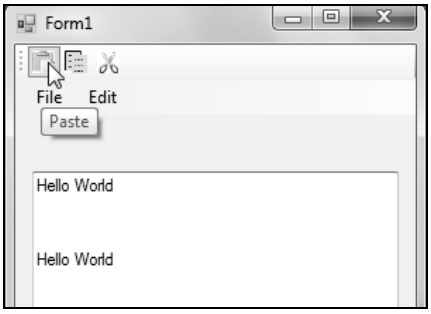

Figure 5.39: Displaying the Linking of Menu Items with ToolStrip Buttons

The Copy button on the `ToolStrip` control corresponds to the Copy item in the Edit menu, and clicking that button is equivalent to clicking the Copy menu item.

This application demonstrates that a tool strip is a shortcut to the most frequently used menu items.

Let's now learn to implement the StatusStrip Control.

Using the StatusStrip Control

As mentioned in the *In Depth* section, the `StatusStrip` control is used to display any type of status information about an application. Let's create an application named `StatusStripSample` (also available in the CD-ROM). Let's perform the following tasks using the StatusStrip control:

- Adding a label to the `StatusStrip` control
- Displaying an icon in the `StatusStrip` control
- Adding split buttons and progress bars in the `StatusStrip` control

Let's start with adding a label to the `StatusStrip` control.

Adding a Label to a StatusStrip Control

To display a label in a `StatusStrip` control, you need to add a `StatusLabel` item to it (as discussed in the In Depth section). Perform the following steps to add a label to a `StatusStrip` control:

1. Drag and drop a TextBox control to the Form1 form and set its `Multiline` property to `true in the Properties window`.
2. Add a `StatusStrip` control to the form. Click the ellipsis (…) button of the `Items` property of the `StatusStrip` control to open the Items Collection Editor dialog box, as shown in Figure 5.40:

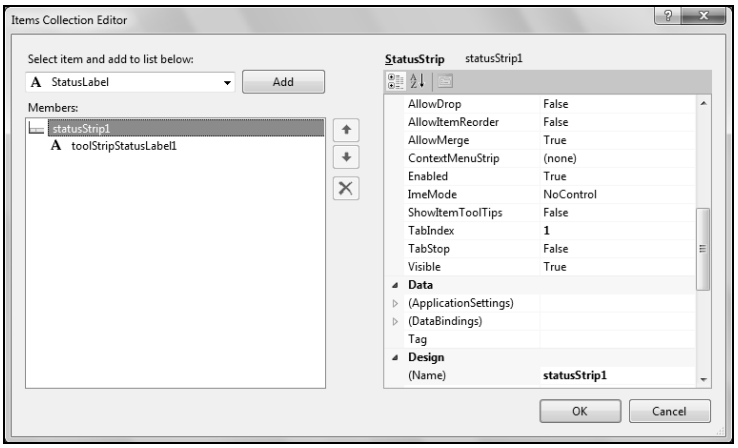

Figure 5.40: Displaying the Items Collection Editor Dialog Box

3. Add a `StatusLabel` item to it by selecting the `StatusLabel` item from the combo box and clicking the Add button.
4. Modify the code of the TextChanged event of textBox1, as shown in the following code snippet:

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
 string[] str = textBox1.Text.Split(' ');
 toolStripStatusLabel1.Text = ("Words: " +
        str.Length + ", " + "Characters (with spaces): " + textBox1.TextLength);
 }
```

In the preceding code snippet, the `Text` property of the `StatusLabel` item is used to set the text. Whenever the user enters any text in the text box, the number of characters and words are recorded in the status strip of the form.

5. Press the `F5` key on the keyboard to execute the application and the output is shown in Figure 5.41:

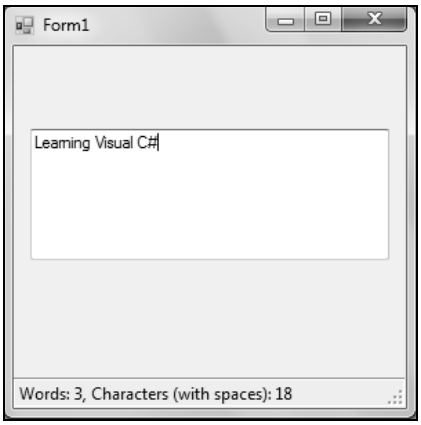

Figure 5.41: Displaying Label in a StatusStrip Control

Now, we have created something similar to a Word processor. If you notice the status bar of Microsoft Word, it displays the number of pages and words in its status bar. Let's now see how to add an icon to the `StatusStrip` control.

Displaying an Icon in the StatusStrip Control

You can also add an icon to a `StatusStrip` control. At design time, just add a `StatusLabel` item to it and set its `DisplayStyle` property to `Image`. Then add an image to the item using the `Image` property. You can also display an icon to a `StatusStrip` control by using the `FromFile`() method of the `Bitmap` class, as shown in the following code snippet:

```
toolStripStatusLabel2.Image = Bitmap.FromFile("D:\\Books\\1.bmp");
```

Adding Split Buttons and Progress Bars in StatusStrip Controls

Split buttons and progress bars can be added to the `StatusStrip` control in the same way as a label is added. Let's perform the following steps to add split buttons to the `StatusStrip` control:

1. Add a `SplitButton` item to the `StatusStrip` control by selecting the `SplitButton` item from the combo box and clicking the Add button.
2. Set the `DisplayStyle` property of the SplitButton item to `Text` and set its `Text` property to Edit.
3. Click the ellipsis (…) button of the `DropDownItems` property of the `SplitButton` item in the Properties window. An Items Collection Editor dialog box appears (Figure 5.42).
4. Add the Copy item to the `SplitButton` item by selecting the `MenuItem` option from the combo box and clicking the Add button. Set the `Text` property of the Copy item to Copy, as shown in Figure 5.42:

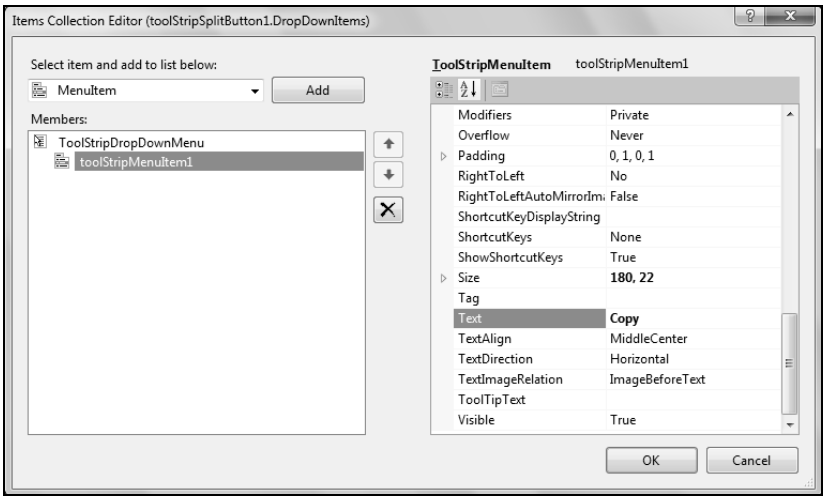

Figure 5.42: Displaying the Items Collection Editor Dialog Box

5. Add the Paste item to the `SplitButton` item in the same manner as we did for the Copy item.
6. Modify the code of the Click events of toolStripMenuItem1 and toolStripMenuItem2, as shown in Listing 5.3:

Listing 5.3: Showing the Code of the Click Events of toolStripMenuItem1 and toolStripMenuItem2

```
private void toolStripMenuItem1_Click(object sender, EventArgs e)
{
 textBox1.Copy();
}
private void toolStripMenuItem2_Click(object sender, EventArgs e)
{
 textBox1.Paste();
}
```

7. Press the `F5` key on the keyboard to execute the application. When the form appears, type the Welcome to Visual C# 2010 text in the text box. Select the entire text in the text box and then select the Copy option from the Edit split button on the `StatusStrip` control. Paste the text anywhere on the text box by clicking the Paste option from the Edit split button on the `StatusStrip` control, as shown in Figure 5.43:

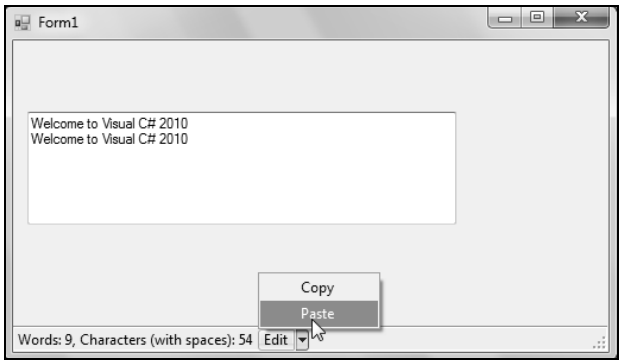

Figure 5.43: Displaying an Icon and a SplitButton Item in a Status Strip

Similarly, you can add a progress bar to the StatusStrip control by performing the following steps:

1. Add a `ProgressBar` item to the `StatusStrip` control by selecting the `ProgressBar` item from the combo box and clicking on the Add button.

2. Drag and drop a `Button` control in the Form1 form with the Text property set as Start.
3. Double-click the button and modify the code of the Click event of button1, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e)
{
  int x;
  toolStripProgressBar1.Minimum = 1;
  toolStripProgressBar1.Maximum = 100000;
  toolStripProgressBar1.Value = 1;
  toolStripProgressBar1.Step = 1;
  for (x = 1; x < 100000; x++)
  {
        toolStripProgressBar1.PerformStep();
  }
  textBox1.Text = "Welcome to C# 2010";
}
```

4. Press the `F5` key on the keyboard to execute the application and click the Start button, the output is displayed, as shown in Figure 5.44:

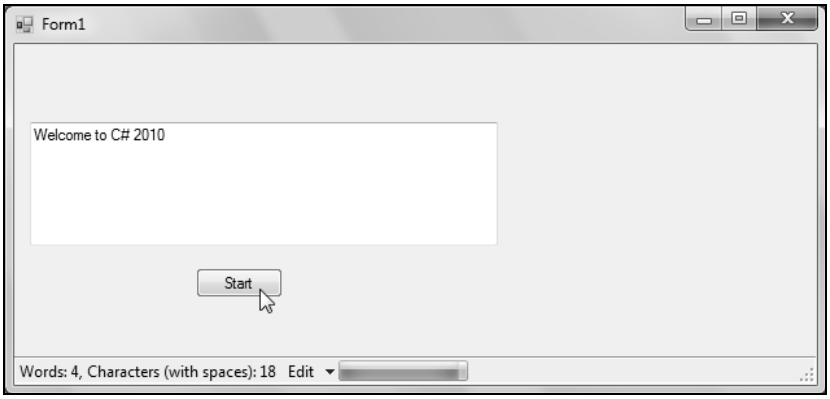

Figure 5.44: Displaying a Progress Bar in Action

The status strip covers almost all the items that can possibly be added on a status bar.

Let's now learn to implement the dialog box controls.

Using the Dialog Box Controls

Visual C# provides some built-in dialog box controls, such as the FolderBrowserDialog, OpenFileDialog, and ColorDialog controls. These dialog box controls help you to browse a folder, open a file or select a color. You can use these built-in dialog box controls to ensure the same appearance and behavior as other Windows-based programs perform. Let's perform the following tasks using the different dialog box controls:

- Using the `FolderBrowserDialog` control to browse through folders
- Using the `OpenFileDialog` control to open files
- Using the `SaveFileDialog` control to save files
- Using the `FontDialog` control to select fonts
- Using the `ColorDialog` control to select colors
- Using the printing controls

Let's see how to use each of these dialog box controls that are available on the Dialogs tab of the Toolbox.

Using the FolderBrowserDialog Control to Browse Through Folders

`The FolderBrowserDialog` control is a powerful searching tool that lets the user select a folder on which the user can search for files based on the criteria set in the user interface.

Let's create an application named `FolderBrowserDialogSample` (also available on the CD-ROM). In this application, perform the following steps to use the `FolderBrowserDialog` control:

1. Drag and drop the controls, given in Table 5.37, to the Form1 form and set their properties and values, as given in Table 5.37:

Table 5.37: Controls to be Added to the Form1 form

Controls	Property	Value
Label	Anchor	Top, Left, Right
	AutoSize	False
	BorderStyle	Fixed3D
	Text	[Blank]
Button	Text	Browse
	Anchor	Top, Right
Button	Text	Find Files
ListBox	Anchor	Top, Bottom, Left, Right
FolderBrowserDialog	Description	Select Folder to Search In

2. Double-click the `button1` control and modify the code of the Click event of button1, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e)
{
  folderBrowserDialog1.RootFolder = Environment.SpecialFolder.MyComputer;
  folderBrowserDialog1.SelectedPath="D:\\Books";
  if (folderBrowserDialog1.ShowDialog()==DialogResult.OK)
  {
        label1.Text = folderBrowserDialog1.SelectedPath;
  }
}
```

In the preceding code snippet, a FolderBrowserDialog control, named folderBrowserDialog1, is used with its two properties RootFolder and SelectedPath. The RootFolder property is used to specify the root path as `Environment.SpecialFolder.MyComputer`; and the SelectedPath property is used to specify a directory path as `D:\\Books`. If the OK button of the folderBrowserDialog1 control is clicked, the directory path is assigned to a Label control, named label1, by using its Text property.

3. Add a reference of the `Microsoft.VisualBasic` assembly to the application and add the following namespaces to the application:

```
using Microsoft.VisualBasic.FileIO;
using System.Collections.ObjectModel;
```

4. Double-click the `button2` control and modify the code of the Click event of button2, as shown in the following code snippet:

```
private void button2_Click(object sender, EventArgs e)
{
  ReadOnlyCollection<string> files = null;
  files = FileSystem.GetFiles(label1.Text);
  listBox1.DataSource = files;
}
```

In the preceding code snippet, an object of the `ReadOnlyCollection<string>` generic class is used to store the name of the files returned by the `GetFiles`() method of the `FileSystem` class. These files are later stored in a list box.

5. Press the `F5` key on the keyboard to execute the application. When the form appears, click the Browse button, the Browse For Folder dialog box appears, as shown in Figure 5.45:

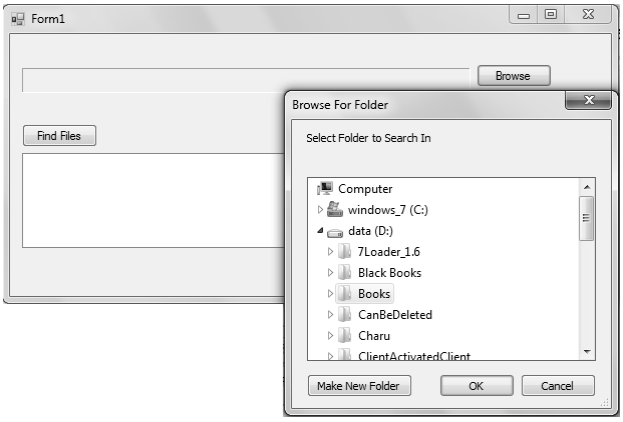

Figure 5.45: Displaying the Browse For Folder Dialog Box with Root Folder Set to Computer

In Figure 5.45, you can see that the Browse For Folder dialog box lists only the subdirectories of Computer. When you select a folder from which you want to retrieve the files, the selected folder path appears on the label, as shown in Figure 5.46:

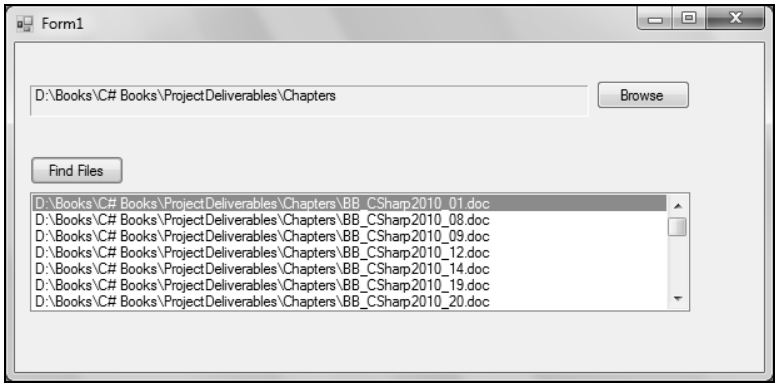

Figure 5.46: Displaying the Files in the Books Folder

6. Click the Find Files button. All the files within that folder appear in the list box (Figure 5.46).

Let's now see how the `OpenFileDialog` control works.

Using the OpenFileDialog Control to Open Files

`The OpenFileDialog` control lets you get the name and the path of files you want to open. Let's create a Windows Forms Application, named `OpenFileDialogSample` (also available on the CD-ROM). In this application, you can open an image in the picture box.

Perform the following steps use the `OpenFileDialog` control:

1. Drag and drop a `PictureBox` control and a `Button` control on the Form1 form and set the Text property of the `Button` control to Load Image.
2. Drag and drop an `OpenFileDialog` control to the Form1 form to configure the Open dialog box. To specify that we want the user to be able to open JPEG or GIF files, we use the `Filter` property of this control, which sets the all possible file types that this dialog can open.
3. Set the `Filter` property of the `OpenFileDialog` control to JPEG files `(*.jpg)|*.jpg`|GIF files `(*.gif)|*.gif`|All files `(*.*)|*.*`. This gives the user three prompts, JPEG files `(*.jpg)`, GIF files `(*.gif)`, and All files `(*.*)` in the Files of type box in the Open dialog box, and informs the user about the file extensions to use by separating the information with upright bars (|).

4. Modify the code of the Click event of button1 to determine the file that a user wants to open from the `FileName` property and load the corresponding image into the picture box, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e)
{
openFileDialog1.InitialDirectory = "D:\\Pictures";
 if (openFileDialog1.ShowDialog() != DialogResult.Cancel)
 {
        pictureBox1.Image = Image.FromFile(openFileDialog1.FileName);
 }
}
```

5. Press the `F5` key on the keyboard to execute the application. When the form appears, click the Load Image button. An Open dialog box appears, as shown in Figure 5.47:

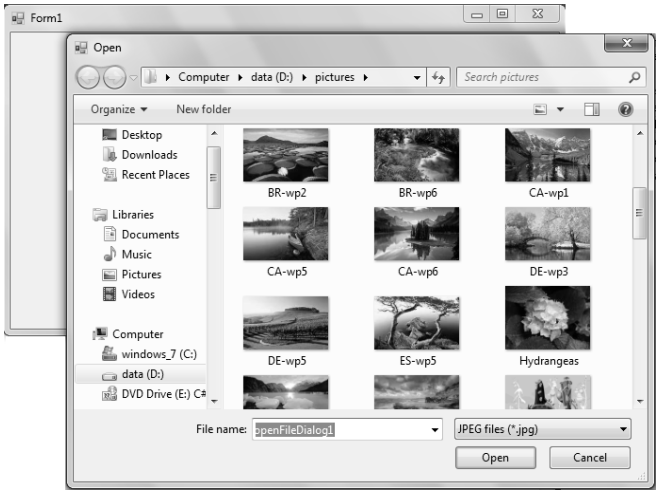

Figure 5.47: The Open Dialog Box after clicking the Load Image Button

6. Select an image and it appears in the picture box, as shown in Figure 5.48:

Figure 5.48: Displaying an Image Using the OpenFileDialog Control

Let's now see how to use the `SaveFileDialog` control.

Using the SaveFileDialog Control to Save Files

You can use the `SaveFileDialog` control to display the Save As dialog box and get the name of the file the user wants to save. Let's create a Windows Forms Application, named `SaveFileDialogSample` (also available on the CD-ROM). Perform the following steps to save the data entered by the user into a file using the `SaveFileDialog` control:

1. Set the Text property of the Form1 form to Save File Dialog Sample.
2. Drag and drop a `Label` control, a `TextBox` control, a `Button` control, and a `SaveFileDialog` control to the form.
3. Set the Text property of the Label control to Enter text to save.
4. Set the text box to multiline text box by setting its `Multiline` property to `true`.
5. Set the Text property of the Button control to Save.
6. Double-click the Save button and modify the code of the Click event of button1, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e)
{
  saveFileDialog1.Filter = "TXT Files (*.txt*)|*.txt*";
  if (saveFileDialog1.ShowDialog() == DialogResult.OK)
  {
        Microsoft.VisualBasic.FileIO.FileSystem.WriteAllText(saveFileDialog1.FileName,
         textBox1.Text, true);
  }
}
```

In the preceding code snippet, with the `Filter` property of the `SaveFileDialog` control, the dialog box is set to display text file types having `.txt` extensions only. The `WriteAllText`() method of the `FileSystem` class writes text to a file and takes three parameters, which are as follows:

- Name of the file
- Text to be saved

Boolean value (true or false), which allows a user to append the file

NOTE

You can use the CreatePrompt property to display a prompt asking the user if a file that does not exist should be created, and the OverwritePrompt property to ask the user whether an existing file should be overwritten.

7. Add a reference of the `Microsoft.VisualBasic` assembly to the application.
8. Press the `F5` key on the keyboard to execute the application. When the form appears, type the Welcome to Visual C# 2010 text in the text box and click the Save button to save the text as a `.txt` file. A Save As dialog box appears, as shown in Figure 5.49:

Figure 5.49: Showing the Use of a SaveFileDialog Control

Figure 5.49 displays the Save as type as TXT Files (*.txt*), which we have set in the code using the `Filter` property.

9. Type a name for the file to be saved, and click the Save button (Figure 5.49). The File is saved on the Desktop with the name entered in the File name text box. In this case, it is saved with the `Welcome.txt` file name.

TIP

You can set the title of a dialog box using the Title property.

Using the FontDialog Control to Select Fonts

A FontDialog control is used to set the font type, font color, or font size of your text in a text accepting control, such as a text box, text area, or rich text box. It also uses a Color dialog box to select a particular color for your text. Let's create a Windows Forms Application, named FontDialogSample (also available on the CD-ROM), to set a font type and a font color for the text of a rich text box control.

Perform the following steps to learn how to use a FontDialog control:

1. Set the Text property of the Form1 form to Font Dialog Sample.
2. Drag and drop a `RichTextBox` control, a `Button` control, and a `FontDialog` control to the form.
3. Set the Text property for the rich text box as Hello and for the button as Select Font.
4. Set the `ShowColor` property of the `FontDialog` control to `true` which makes it show a Color combo box with some standard windows colors to choose from (this combo box gives you only the very basic colors to choose from; for a wider range of colors, use the Color dialog box, discussed in the next topic).
5. Double-click the Select Font button and modify the code of the Click event of button1, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e)
{
   if (fontDialog1.ShowDialog() != DialogResult.Cancel)
   {
         richTextBox1.Font = fontDialog1.Font;
         richTextBox1.ForeColor = fontDialog1.Color;
   }
}
```

In the preceding code snippet, the `Font` and `Color` properties of the `FontDialog` control is used to set the font and its color selected by the user on the text set in the rich text box.

6. Press the `F5` key on the keyboard to execute the application. When the form appears, click the Select Font button. A Font dialog box appears, as shown in Figure 5.50:

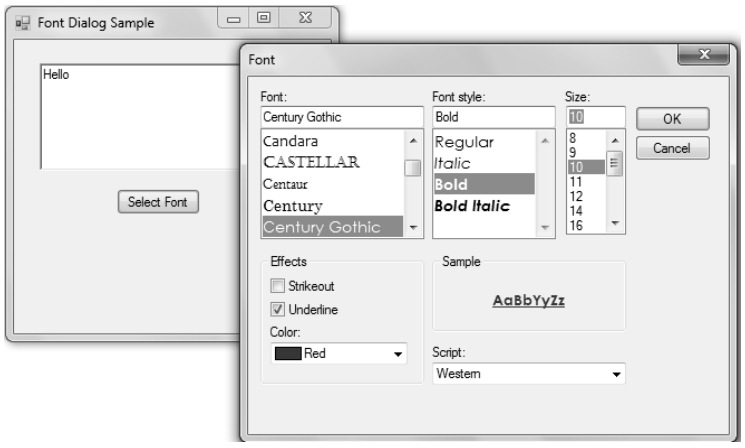

Figure 5.50: Displaying the Use of the FontDialog Control

7. Select the font, font style, size, effects, and color of the font from the Font dialog box.

8. Click the OK button and the new font and color is assigned to the text in the rich text box, as shown in Figure 5.51:

Figure 5.51: Setting the Font in a Rich Text Box

Next, we discuss about the `ColorDialog` control.

Using the ColorDialog Control to Select Colors

Let's create a Windows Forms Application, named `ColorDialogSample` (also available on the CD-ROM), to use a ColorDialog control to set the background color of a `Label` control. Perform the following steps to use the ColorDialog control:

1. Set the Text property of the Form1 form as Color Dialog Sample.
2. Drag and drop a `Label` control, a `Button` control, and a `ColorDialog` control to the form.
3. Set the Text property of the Button and Label controls to Choose Color and Change my color...!!!, respectively.
4. Set the font of the Label control to Bookman Old Style, font style to Bold, and its font size to 12.
5. Double-click the Choose Color button and modify the code of the Click event of button1, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e)
{
  if (colorDialog1.ShowDialog() != DialogResult.Cancel)
  {
        label1.Text = "Here's my new color!";
        label1.BackColor = colorDialog1.Color;
  }
}
```

In the preceding code, the color chosen by the user in the Color dialog box is set to the `BackColor` property of the label using the `Color` property of the `ColorDialog` control.

6. Press the `F5` key on the keyboard to execute the application and when the form appears, click the Choose Color button. A Color dialog box appears, as shown in Figure 5.52:

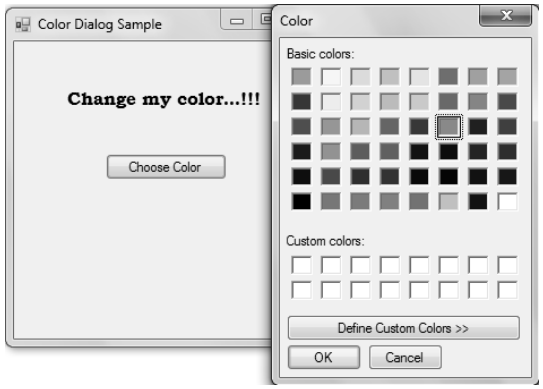

Figure 5.52: Displaying the Use of a Color Dialog Box

7. Select a color and click the OK button (Figure 5.52). The selected color is applied to the background color for the label, as shown in Figure 5.53:

Figure 5.53: Setting a Label's Background Color

The label's original text Change my color…!!! is also being changed to Here's my new color!, as shown in Figure 5.53. Let's now see how you can print your document using the various printing controls.

Using the Printing Controls

In the *In Depth* section, you have already learned about various controls related to the printing of document, such as PrintDocument, PageSetupDialog, PrintPreviewControl, and PrintDialog controls. Let's create a Windows Forms Application, named PrintingSample (also available on the CD-ROM), to learn how to use the following controls in the context of printing a document:

- The PrintDocument control
- The PageSetupDialog control
- The PrintPreviewControl control
- The PrintPreviewDialog control
- The PrintDialog Control

Using the PrintDocument Control

A `PrintDocument` control can be added to an application that generates printouts. In the PrintingSample application, we are using a PrintDocument control, which represents a document that needs to be printed by a printer. The `PrintDocument` object exposes a `Graphics` object that provides methods for drawing objects to be displayed on a display device. However, to print text, the `DrawString`() method is used; and to print frames around the text the `DrawLine`() or `DrawRectangle`() method is used.

The `PrintDocument` control is not visible at runtime. However, its icon can be seen in the Component Tray at design time. The `Print`() method is called to print documents. The PrintDocument control raises the `BeginPrint` and `PrintPage` events of the control. When the print job starts, a `BeginPrint` event occurs. After that, the `PrintPage` event is fired for each page (if you want to indicate that there are more pages to print, set the HasMorePages property of the object to true). Finally, the `EndPrint` event is fired when the whole job is completed.

Using the PageSetupDialog Control

The Page Setup dialog box lets the user specify the format for the pages that are to be printed, such as setting page orientation (portrait or landscape) and margin size. You can use a Page Setup dialog box to modify both the `PrinterSettings` and `PageSettings` objects in a `PrintDocument` object to record the settings the user wants to use for printing. In the PrintingSample application, we are using the objects of both classes, PrinterSettings and PageSettings. The PrinterSettings class is used to specify the printer related settings for the printing of a document. These settings are stored in the PrinterSettings property of the PageSetupDialog class which returns a PrinterSettings object. This object is assigned to the PrinterSettings property of the PrintDocument object. This makes sure that the settings the user wants are the same settings that are assigned to the document for printing.

The PageSettings class is used to specify the page related settings for the printing of a document. The main properties of the `PageSettings` class are as follows:

- `Bounds`—Retrieves the bounds of the page

- `Color`—Retrieves a value indicating whether the page should be printed in color
- `Landscape`—Retrieves a value indicating whether the page is printed in landscape or portrait orientation
- `Margins`— Retrieves the margins for this page
- `PaperSize`— Retrieves the paper size for the page
- `PaperSource`— Retrieves the page's paper source
- `PrinterResolution`— Retrieves the printer resolution for the page
- `PrinterSettings`— Retrieves the printer settings associated with the page

You can let the user display a Page Setup dialog box in the `PrintingSample` application, and record the new settings in the `PrintDocument` object. To do so, let's perform the following steps:

1. Set the `Text` property of the Form1 form to Printing Sample.
2. Drag and drop a `MenuStrip` control, a `PrintDocument` control, and a `PageSetupDialog` control to the form.
3. Add a File menu to the `MenuStrip` control, and a Page Setup menu item to the File menu.
4. Double-click the Page Setup menu item and modify the code of the Click event of menu item, as shown in the following code snippet::

```
private void pageSetupToolStripMenuItem_Click(object sender, EventArgs e)
{
  pageSetupDialog1.Document = printDocument1;
  pageSetupDialog1.PrinterSettings = printDocument1.PrinterSettings;
  pageSetupDialog1.PageSettings = printDocument1.DefaultPageSettings;
  if (pageSetupDialog1.ShowDialog() == DialogResult.OK)
  {
        printDocument1.PrinterSettings = pageSetupDialog1.PrinterSettings;
        printDocument1.DefaultPageSettings = pageSetupDialog1.PageSettings;
  }
}
```

4. Press the F5 key on the keyboard to execute the application. When the form appears, click the File → Page Setup option. A Page Setup dialog box appears, as shown in Figure 5.54:

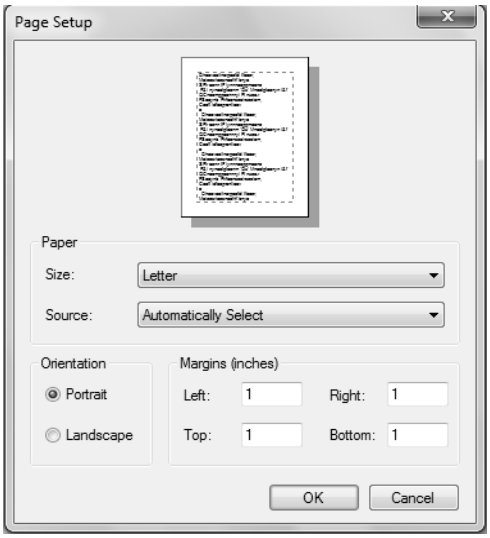

Figure 5.54: Displaying the Use of the PageSetupDialog Control

After setting the various options for page setting in the Page Setup dialog box, it is preferred to preview the document before printing. There are two options – the `PrintPreviewControl` control and the `PrintPreviewDialog` control. Let's see each of these in detail.

Using the PrintPreviewControl Control

The `PrintPreviewControl` control displays print previews. You can also use it to create your own custom print preview windows. You need to assign a document for printing by setting the `Document` property and implementing the `PrintPage` event handler of the PrintPreviewControl control. To do so, let's perform the following steps:

1. Add a second form, `Form2`, with the Text property set to Printing Preview Control, in the `PrintingSample` application to show how to create a custom print preview.
2. Drag and drop a `PrintPreviewControl` control on the `Form2 form` and set its scope to `internal` in the `Form2.Designer.cs`. In addition, also add a `Button` control on `Form2` and set its `Text` property to Close. This button is used to close the form.
3. Double-click the Close button in the design mode of the Form2 form and modify the code of the Click event of button1, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e)
{
  this.Close();
}
```

4. Add a Custom Print Preview menu item to the File menu of Form1.
5. Double-click the Custom Print Preview menu item in the design mode and modify the code of the Click event of menu item, as shown in the following code snippet:

```
private void customPrintPreviewToolStripMenuItem_Click(object sender, EventArgs e)
{
  Form2 preview = new Form2();
  preview.printPreviewControl1.Document = printDocument1;
  preview.Show();
}
```

Till now, you have set a `PrintPreviewControl` control to preview a document. However, a document also needs to be set for preview.

6. Modify the code in the Form2.cs file to set a document for printing, as shown in Listing 5.4:

Listing 5.4: Showing the Code to Set a Document for Printing

```
int pageNumber = 0;
private void printDocument1_BeginPrint(object sender,
  System.Drawing.Printing.PrintEventArgs e)
{
  pageNumber = 0;
}
private void printDocument1_PrintPage(object sender,
 System.Drawing.Printing.PrintPageEventArgs e)
{
  pageNumber = pageNumber+1;
  switch (pageNumber)
  {
        case 1:
        {
        e.Graphics.FillRectangle(Brushes.Red, new Rectangle(200, 200, 500, 500));
        e.HasMorePages = true;
        break;
        }
        case 2:
        {
        e.Graphics.FillRectangle(Brushes.Blue, new Rectangle(200, 200, 500, 500));
        e.HasMorePages = false;
        break;
        }
  }
}
```

NOTE

This application does not print selected ranges of pages—it just prints the whole document. If you want to handle print ranges, please refer to the PrintDocument.PrinterSettings.PrintRange property, which holds the range of pages to be printed.

7. Press the `F5` key on the keyboard to execute the application. Select the Custom Print Preview option from the File menu in Form1. You see the Printing Preview Control window, as shown in Figure 5.55:

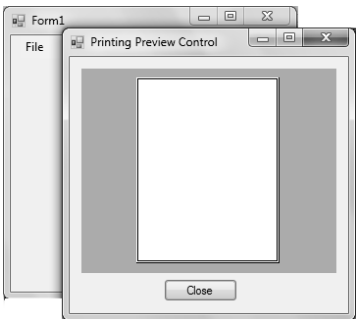

Figure 5.55: Displaying the Use of PrintPreviewControl

If you want to print some text, you should call the `DrawString`() method of the `Graphics` object. Let's see the next control for previewing a document, the `PrintPreviewDialog` control.

Using the PrintPreviewDialog Control

To display a print preview, a `PrintDocument` needs to be assigned to a print preview dialog's `Document` property and should also implement at least the `PrintPage` event handler of the `PrintDocument object`. Now to display the print preview dialog box, use the `ShowDialog`() method. Now, let's perform the following steps:

1. Add a Print Preview menu item to the File menu.
2. Drag and drop a `PrintPreviewDialog` control to the Form1 form.
3. Double-click the menu item and modify the code of the Click event of menu item, as shown in the following code snippet:

```
private void printPreviewToolStripMenuItem_Click(object sender, EventArgs e)
{
   printPreviewDialog1.Document = printDocument1;
   printPreviewDialog1.ShowDialog();
}
```

4. Press the `F5` key on the keyboard to execute the application. Select the Print Preview option from the File menu in Form1. You see the Printing preview window, as shown in Figure 5.56:

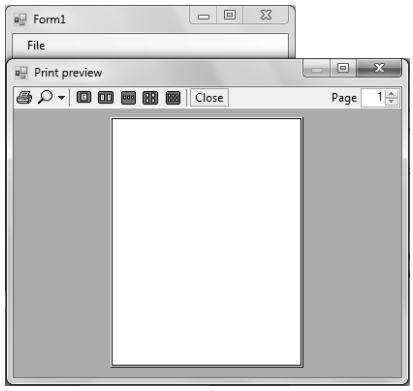

Figure 5.56: Displaying the Use of the Print preview Window

After previewing the document, in the next section you learn to print the document.

Using the PrintDialog Control

The `PrintDialog` control shows the standard Print dialog box. This allows a user to select a printer and set its properties. If you do not show this dialog box, the output is sent automatically to the default printer and it uses the default settings of the printer. To display the Print dialog box, the `ShowDialog`() method of the `PrintDialog` control is called. To do so, let's perform the following steps:

1. Add a Print menu item to the File menu.
2. Drag and drop a `PrintDialog` control to the Form1 form.
3. Double-click the Print menu item and modify the code of the Click event of menu item, as shown in the following code snippet:

```
private void printToolStripMenuItem_Click(object sender, EventArgs e)
{
  printDialog1.Document = printDocument1;
  printDialog1.PrinterSettings = printDocument1.PrinterSettings;
  printDialog1.AllowSomePages = true;
  if (printDialog1.ShowDialog()== DialogResult.OK)
  {
        printDocument1.PrinterSettings = printDialog1.PrinterSettings;
        printDocument1.Print();
  }
}
```

The `PrinterSettings` property of the control should be set or else an exception is thrown.

4. Press the `F5` key on the keyboard to execute the application. Select the Print option from the File menu in Form1. You see the Print dialog box, as shown in Figure 5.57:

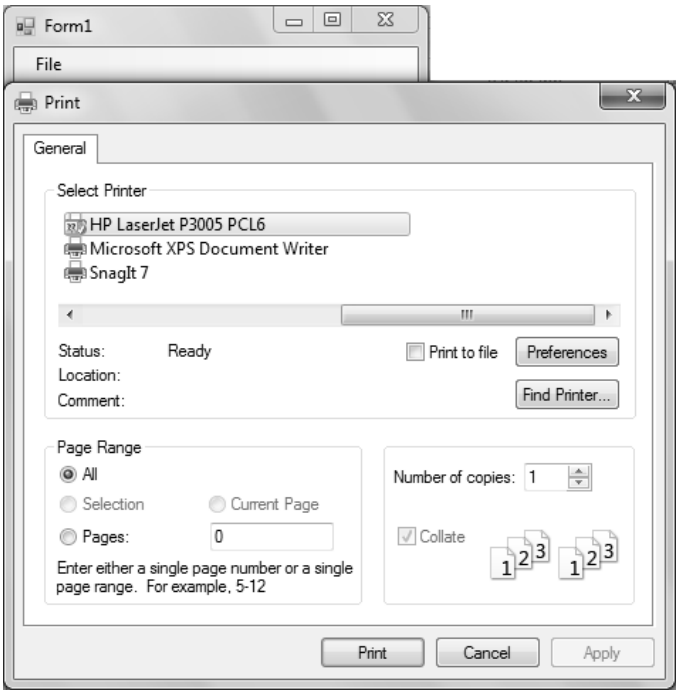

Figure 5.57: Displaying the Print Dialog Box

Let's now recap the points in the summary section.

Summary

In this chapter, you have learned about various Windows Forms controls, such as ToolStrip, MenuStrip, OpenFileDialog, and PrintDialog. The chapter has started by exploring the toolbars and menus controls and classes, which include the ToolStrip and MenuStrip controls, and the ToolStripItem and ToolStripMenuItem classes. Next, you have learned about the ContextMenuStrip control to create a short cut menu for the right mouse click on a control; and the StatusStrip control to create status bar showing the status of an application. Then, you have learned about various dialog box controls, such as SaveFileDialog and ColorDialog, to perform common Windows tasks related to opening or saving of a file or selecting a color for your text. You have also learned about the controls related to the process of printing a document. These controls include PrintDocument, PageSetupDialog, PrintPreviewControl, PrintPreviewDialog, and PrintDialog.

In the next chapter, you learn about various validation controls, such as RequiredFieldValidator, CompareFieldValidator, RangeValidator, RegularExpressionValidator, CustomValidator, and ValidationSummary.