# 1

# Windows Forms Controls: Button, Label, TextBox, RichTextBox, and MaskedTextBox

| *If you need an Immediate Solution to:* | *See page:* |
|---|---|
| Using the Button Control | 21 |
| Using the Label Control | 23 |
| Using the TextBox Control | 24 |
| Using RichTextBox Controls | 26 |
| Using MaskedTextBox Controls | 29 |

# *In Depth*

You have already learned to create a Windows Forms application and add Windows Forms controls to the Windows Forms applications in Chapter 8 of the book, *Windows Forms in C# 2010*. This chapter further elaborates on the use of specific and most frequently used Windows Forms controls—`Button`, `Label`, `TextBox`, `RichTextBox`, and `MaskedTextBox`. In simple words, the `Button` control lets you generate and handle a click event and the `Label` control lets you display some text, such as caption for a text box, on a Windows Form. Similarly, the last three controls—`TextBox`, `RichTextBox`, and `MaskedTextBox`—are used to accept user input.

We begin this chapter by discussing the `Control` class, which is the base class for all the Windows Forms controls. Then we discuss important properties, methods, and events of the `Button`, `Label`, `TextBox`, `RichTextBox`, and `MaskedTextBox` controls. In the Immediate Solutions section, we learn how to perform various operations on each of these controls.

## Exploring the Control Class

The `Control` class is the base class for all the Windows Forms controls. This class implements basic functionalities, such as handling user input through keyboard and mouse, and defining position and size of a component (control) required by the classes that display information to users. The `Control` class is placed inside the `System.Windows.Forms` namespace of the .NET Framework class library. The inheritance hierarchy of the `Control` class is given as follows:

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

You can notice in the preceding inheritance hierarchy that at the top of the hierarchy is the `System.Object` class, which is the base class for every type defined in .NET.

The `Control` class defines properties, methods, and events, which are common to all the Windows Forms controls. Table 1.1 lists some useful public properties of the `Control` class:

**Table 1.1: Noteworthy Public Properties of the Control Class**

| Property | Description |
|---|---|
| `AllowDrop` | Retrieves or sets a value indicating whether drag-and-drop operations are allowed in the control or not |
| `Anchor` | Retrieves or sets a value indicating which edges of the control are anchored |
| `BackColor` | Retrieves or sets the background color of the control |
| `BackgroundImage` | Retrieves or sets the background image in the control |
| `Bottom` | Retrieves or sets the distance (in pixels) between the bottom edge of the control and the top of the client area of its container, such as a Windows Form |
| `BackgroundImageLayout` | Retrieves or sets the background image layout as defined in the `ImageLayout` enumeration |
| `Bounds` | Retrieves or sets the size and location of the control including its nonclient elements, such as scroll bars, borders, title bars, and menus, related to the parent control |
| `CanFocus` | Returns a value specifying if the control can receive the focus |
| `CanSelect` | Returns a value specifying if the control can be selected |
| `ClientRectangle` | Retrieves or sets the rectangle that represents the client area of the control |
| `ClientSize` | Retrieves or sets the height and width of the client area of the control |

**Table 1.1: Noteworthy Public Properties of the Control Class**

| Property | Description |
|---|---|
| `ContainsFocus` | Returns a value specifying if the control has the input focus |
| `ContextMenu` | Retrieves or sets the shortcut menu associated with the control |
| `ContextMenuStrip` | Retrieves or sets the `ContextMenuStrip` control associated with the control |
| `Controls` | Retrieves or sets the collection of controls contained within the control |
| `Created` | Retrieves or sets a value indicating whether the control has been created |
| `Cursor` | Retrieves or sets the cursor displayed when the user moves the mouse pointer over this control |
| `DataBindings` | Retrieves or sets the data bindings for the control |
| `DefaultBackColor` | Retrieves or sets the default background color of the control |
| `DefaultFont` | Retrieves or sets the default font of the control |
| `DefaultForecolor` | Retrieves or sets the default foreground color of the control |
| `DefaultMargin` | Retrieves or sets the space (in pixels) that is specified by default between two controls |
| `DefaultMaximumSize` | Retrieves or sets the length and height (in pixels) that is specified as the default maximum size of the control |
| `DefaultMinimumSize` | Retrieves or sets the length and height (in pixels) that is specified as the default minimum size of a control |
| `DefaultPadding` | Retrieves or sets the internal spacing (in pixels) of the contents of a control |
| `DefaultSize` | Retrieves or sets the default size of the control |
| `DisplayRectangle` | Retrieves or sets the rectangle that represents the display area of the control |
| `Disposing` | Retrieves or sets a value indicating whether the base control class is in the process of disposing |
| `Dock` | Retrieves or sets a value indicating which edge of the parent control, a control is docked to |
| `Enabled` | Retrieves or sets a value specifying if the control is enabled |
| `Focused` | Returns a value specifying if the control has focus |
| `Font` | Retrieves or sets the current font for the control |
| `FontHeight` | Retrieves or sets the height of the font of the control |
| `ForeColor` | Retrieves or sets the foreground color of the control |
| `HasChildren` | Returns a value specifying if the control contains child controls |
| `Height` | Retrieves or sets the height of the control |
| `IsDisposed` | Retrieves or sets a value indicating whether the control has been disposed of |
| `IsMirrored` | Retrieves or sets a value indicating whether the control is mirrored |
| `LayoutEngine` | Retrieves or sets a cached instance of the control's layout engine |
| `Left` | Retrieves or sets the X-coordinate (in pixels) of a control's left edge |
| `Location` | Retrieves or sets the coordinates of the upper-left corner of the control with respect to the upper-left corner of its container control, such as a Windows Form |

**Table 1.1: Noteworthy Public Properties of the Control Class**

| Property | Description |
|---|---|
| `Margin` | Retrieves or sets the space between the controls |
| `MaximumSize` | Retrieves or sets the maximum size of the control |
| `MinimumSize` | Retrieves or sets the minimum size of the control |
| `ModifierKeys` | Retrieves or sets a value indicating which of the modifier keys is in a pressed state |
| `MouseButtons` | Retrieves or sets a value indicating which of the mouse button is in a pressed state |
| `MousePosition` | Retrieves or sets the coordinates of the mouse cursor relative to the upper-left corner of the screen |
| `Name` | Retrieves or sets the control's name |
| `Padding` | Retrieves or sets the padding within the control |
| `Parent` | Retrieves or sets the control's parent container |
| `Region` | Retrieves or sets the window region associated with the control |
| `Right` | Returns the distance, in pixels, between the right edge of the control and the left edge of its container's client area |
| `RightToLeft` | Retrieves or sets a value indicating if the alignment of the control's elements is reversed to support right-to-left fonts |
| `Site` | Retrieves or sets the site of the controls |
| `Size` | Retrieves or sets the height and width of the control |
| `TabIndex` | Retrieves or sets the tab order of a control in its container control |
| `TabStop` | Retrieves or sets a value specifying if the user can tab to this control with the TAB key |
| `Tag` | Retrieves or sets an object that contains data about the control |
| `Text` | Retrieves or sets the text connected to this control |
| `Top` | Retrieves or sets the distance, in pixels, between the top edge of the control and the top edge of its container's client area |
| `TopLevelControl` | Retrieves or sets the parent control that is not parented by another Windows Forms controls |
| `UseWaitCursor` | Retrieves or sets a value indicating whether the control and all its parents controls are displayed |
| `Visible` | Retrieves or sets a value specifying if the control is visible |
| `Width` | Retrieves or sets the width of the control |

Some important public methods of the `Control` class are listed in Table 1.2:

**Table 1.2: Noteworthy Public Methods of the Control Class**

| Method | Description |
|---|---|
| `BringToFront()` | Brings the control in front of the stacking order |
| `Contains()` | Retrieves a value indicating if the control passed to this method is a child of the control |

**Table 1.2: Noteworthy Public Methods of the Control Class**

| Method | Description |
|---|---|
| `CreateGraphics()` | Creates a `Graphics` object for the control |
| `Dispose()` | Releases the resources used by the control |
| `DoDragDrop()` | Starts a drag-and-drop operation |
| `DrawToBitmap()` | Supports rendering to the specified bitmap |
| `Equals(Object)` | Compares two objects for their equality |
| `FindForm()` | Retrieves the form on which the control is placed |
| `Focus()` | Gives the focus to the control |
| `GetContainerControl()` | Returns the immediate parent control of a control in the inheritance hierarchy of the control |
| `GetChildAtPoint()` | Retrieves or sets the child control of a control at the specified coordinates |
| `GetNextControl()` | Retrieves the next control forward or backward in the tab order of child controls |
| `GetPreferredSize()` | Retrieves the size of a rectangular area into which a control can be fitted |
| `GetStyle()` | Retrieves the value for the specified control style bit for the control |
| `Hide()` | Hides the control |
| `Invalidate()` | Invalidates a part of the control and sends a paint message to the control |
| `Invoke()` | Executes a delegate on the thread that owns the control's underlying window handle |
| `IsKeyLocked()` | Determines whether the Caps Lock, Num Lock, and Scroll Lock keys are in effect |
| `IsMnemonic()` | Determines if the specified character is the mnemonic character for the control in the string |
| `Refresh()` | Forces the control to invalidate its client area and repaint itself (and any child control) |
| `ResetBackColor()` | Resets the `BackColor` property to its default value |
| `ResetBindings()` | Bounds the control to the `BindingSource class` and then reread all the items in the list and in turn refresh their values |
| `ResetCursor()` | Resets the `Cursor` property to its default value |
| `ResetFont()` | Resets the `Font` property to its default value |
| `ResetForeColor()` | Resets the `ForeColor` property to its default value |
| `Scale()` | Scales the control and any child control |
| `SendToBack()` | Sends the control to the back of the stacking |
| `Select()` | Activates (or selects) a control |
| `Show()` | Displays the control and sets its visible property to `true` |

Table 1.3 lists the most notable public events of the `Control` class:

**Table 1.3: Noteworthy Public Events of the Control Class**

| Event | Description |
|---|---|
| `BackColorChanged` | Occurs when the value of the `BackColor` property is changed |
| `BackgroundImageChanged` | Occurs when the `BackgroundImage` property is changed |
| `BackgroundImageLayoutChanged` | Occurs when the value for the `BackgroundImageLayout` property changes |
| `BindingContextChanged` | Occurs when the value for the `BindingContext` property changes |
| `CausesValidationChanged` | Occurs when the value for the `CausesValidation` property changes |
| `ClientSizeChanged` | Occurs when the value for the `ClientSize` property changes |
| `ContextMenuStripChanged` | Occurs when the value for the `ContextMenuStrip` property changes |
| `Click` | Occurs when the control is clicked |
| `ContextMenuChanged` | Occurs when the `ContextMenu` property value is changed |
| `ControlAdded` | Occurs when a new control is added |
| `ControlRemoved` | Occurs when a control is removed |
| `CursorChanged` | Occurs when the `Cursor` property value is changed |
| `DockChanged` | Occurs when the `Dock` property value is changed |
| `DoubleClick` | Occurs when the control is double clicked |
| `DragDrop` | Occurs when a drag-and-drop operation is completed |
| `DragEnter` | Occurs when an object is dragged into the control's bounds |
| `DragLeave` | Occurs when an object has been dragged into and out of the control's bounds |
| `DragOver` | Occurs when an object has been dragged over the control's bounds |
| `EnabledChanged` | Occurs when the `Enabled` property value is changed |
| `Enter` | Occurs when the control is entered |
| `FontChanged` | Occurs when the `Font` property value is changed |
| `ForeColorChanged` | Occurs when the `ForeColor` property value is changed |
| `GiveFeedback` | Occurs during a drag operation |
| `GotFocus` | Occurs when the control receives focus |
| `Invalidated` | Occurs when a control's display is updated |
| `KeyDown` | Occurs when a key is pressed down, while the control has focus |
| `KeyPress` | Occurs when a key is pressed, while the control has focus |
| `KeyUp` | Occurs when a key is released, while the control has focus |
| `Layout` | Occurs when a control has to lay out its child controls |
| `Leave` | Occurs when the control is left |
| `LocationChanged` | Occurs when the `Location` property value is changed |
| `LostFocus` | Occurs when the control loses focus |
| `MarginChanged` | Occurs when the margins of the control changes |
| `MouseCaptureChanged` | Occurs when the control loses or gains mouse capture |
| `MouseClick` | Occurs when the control is clicked by the mouse |

**Table 1.3: Noteworthy Public Events of the Control Class**

| Event | Description |
|---|---|
| `MouseDoubleClick` | Occurs when the control is double clicked by the mouse |
| `MouseDown` | Occurs when the mouse pointer is over the control and a mouse button is pressed |
| `MouseEnter` | Occurs when the mouse pointer enters the control |
| `MouseHover` | Occurs when the mouse pointer hovers over the control |
| `MouseLeave` | Occurs when the mouse pointer leaves the control |
| `MouseMove` | Occurs when the mouse pointer is moved over the control |
| `MouseUp` | Occurs when the mouse pointer is over the control and a mouse button is released |
| `MouseWheel` | Occurs when the mouse wheel moves, while the control has focus |
| `Move` | Occurs when the control is moved |
| `PaddingChanged` | Occurs when the controls padding changes |
| `Paint` | Occurs when the control is redrawn |
| `ParentChanged` | Occurs when the `Parent` property value is changed |
| `Resize` | Occurs when the control is resized |
| `RightToLeftChanged` | Occurs when the `RightToLeft` property value is changed |
| `SizeChanged` | Occurs when the value of the Size property changes |
| `StyleChanged` | Occurs when the control style changes |
| `SystemsColorsChanged` | Takes place when the system colors changes |
| `TabIndexChanged` | Occurs when the value of the `TabIndex` property changes |
| `TabStopChanged` | Occurs when the value of the `TabStop` property changes |
| `TextChanged` | Occurs when the value of the Text property changes |
| `Validated` | Occurs when the control is done validating |
| `Validating` | Occurs when the control is validating |
| `VisibleChanged` | Occurs when the `Visible` property value is changed |

Next, let's learn about the Button control.

## Describing the Button Control

The `Button` control is one of the most basic Windows Forms controls, which lets you generate a `Click` event. You can make the user perform some action at the runtime by handling the `Click` event of a `Button` control. The inheritance hierarchy of the `Button` class is given as follows:

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

Table 1.4 lists the notable public properties of the `Button` class:

**Table 1.4: Noteworthy Public Properties of the Button Class**

| Property | Description |
|---|---|
| `AutoSizeMode` | Retrieves or sets the mode in which a `Button` control can automatically resize itself. |
| `DialogResult` | Retrieves or sets a value that is returned to the parent form when the button is clicked. This property is often used when creating dialog boxes. |

Table 1.5 lists the noteworthy public methods of the `Button` class:

**Table 1.5: Noteworthy Public Methods of the Button Class**

| Method | Description |
|---|---|
| `NotifyDefault()` | Notifies the button whether it is a default button, so that it can adjust its appearance |
| `PerformClick()` | Causes a `Click` event for a button |
| `ToString()` | Returns the string which contains the name of the component, if it is having any |

Now, in the following sections, you perform the following tasks to learn more about the Button control:

- Set the caption of a button
- Set the background and foreground colors of a button
- Add an image to a button
- Handle the events of a button

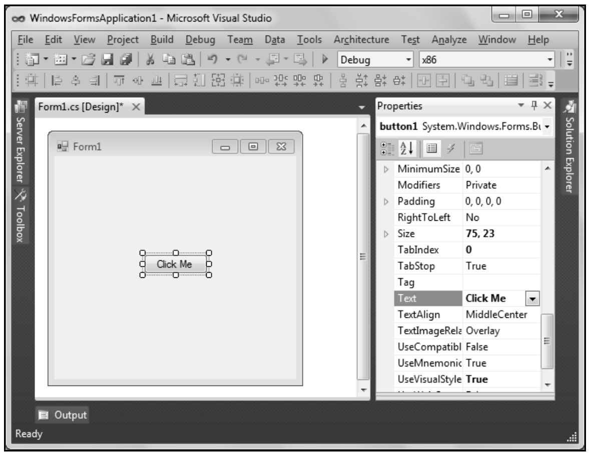
Let's discuss these tasks one by one in detail.

## Setting the Caption of a Button

You use a button's `Text` property to set its caption. You can set this property either at design time using the Properties window or at runtime. After you add a button to a form, you can set the caption text of the button by placing the appropriate text for the `Text` property of the button control using the Properties window.

A button control shows its default text, for instance button1, when you drag it from the Toolbox to a form.

To set a new caption for the button control, first select the button control and then in the Properties window, type a new caption for the button control in front of the `Text` property and press the ENTER key on the keyboard. This changes the caption of the button control to the new value you have specified for the `Text` property of the button control, as shown in Figure 1.1:


WindowsFormsApplication1 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Debug
x86
Server Explorer
Toolbox
Form1.cs [Design]*
Form1
Click Me
Properties
Solution Explorer
Output
Ready

button1 System.Windows.Forms.Bu

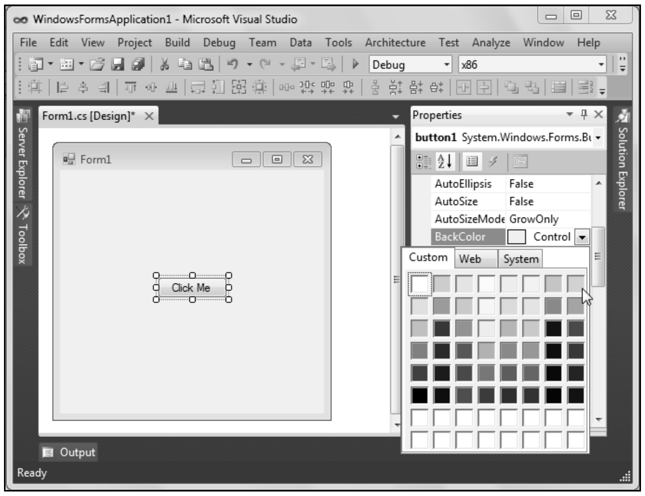
| | |
|---|---|
| MinimumSize | 0, 0 |
| Modifiers | Private |
| Padding | 0, 0, 0, 0 |
| RightToLeft | No |
| Size | 75, 23 |
| TabIndex | 0 |
| TabStop | True |
| Tag | |
| Text | Click Me |
| TextAlign | MiddleCenter |
| TextImageRel | Overlay |
| UseCompatibl | False |
| UseMnemonic | True |
| UseVisualStyle | True |

**Figure 1.1: Setting the Caption of a Button**

In Figure 1.1, you can see that we have specified `Click Me` as the new caption for the button.

## Setting the Background and Foreground Colors of a Button

You can add a background color to a button by setting its `BackColor` property. This property can also be set either at the design time or at the runtime. To set the `BackColor` property of the button present on `Form1`, first select the button and then in the Properties window, click the down arrow in front of the `BackColor` property. This displays a small window with three tabs, named System, Web, and Custom. Click the Custom tab and select a color from the color palette displayed under this tab, as shown in Figure 1.2:


WindowsFormsApplication1 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Debug
x86
Form1.cs [Design]*
Server Explorer
Toolbox
Form1
Click Me
Properties
Solution Explorer
Custom Web System
Output
Ready

button1 System.Windows.Forms.Bu

AutoEllipsis False
AutoSize False
AutoSizeMode GrowOnly
BackColor Control

Figure 1.2: Setting the Background Color of a Button

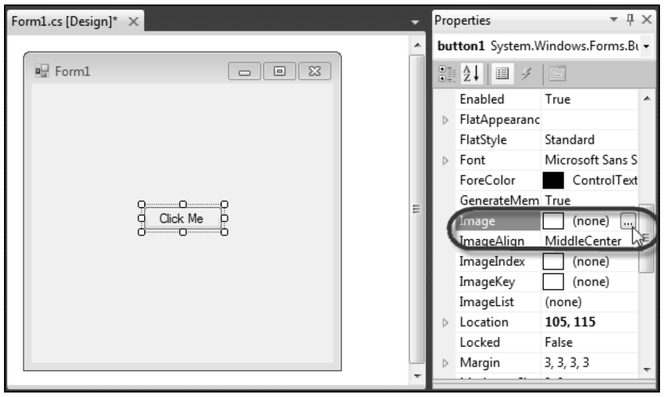
The background color of the button changes according to the color you have selected in the color palette.

Similar to changing the background color, you can also change the foreground color of a button both at the design time or runtime. To change the foreground color of the button present on `Form1` at design time, first select the button. Then in the Properties window, select a new color for the `ForeColor` property.

## Adding an Image to a Button

You can add images to buttons at design time and at run time. Let's perform the following steps to add an image to the button control:

1. Select the button and then in the Properties window, click the ellipsis button in front of the `Image` property, as shown in Figure 1.3:



button1 System.Windows.Forms.Bu

Enabled True
FlatAppearanc
FlatStyle Standard
Font Microsoft Sans S
ForeColor ControlText
GenerateMem True
Image (none)
ImageAlign MiddleCenter
ImageIndex (none)
ImageKey (none)
ImageList (none)
Location 105, 115
Locked False
Margin 3, 3, 3, 3

Figure 1.3: Opening the Select Resource Dialog Box

The Select Resource dialog box opens (Figure 1.4).

2. Select the Local resource radio button and click the Import button, as shown in Figure 1.4:

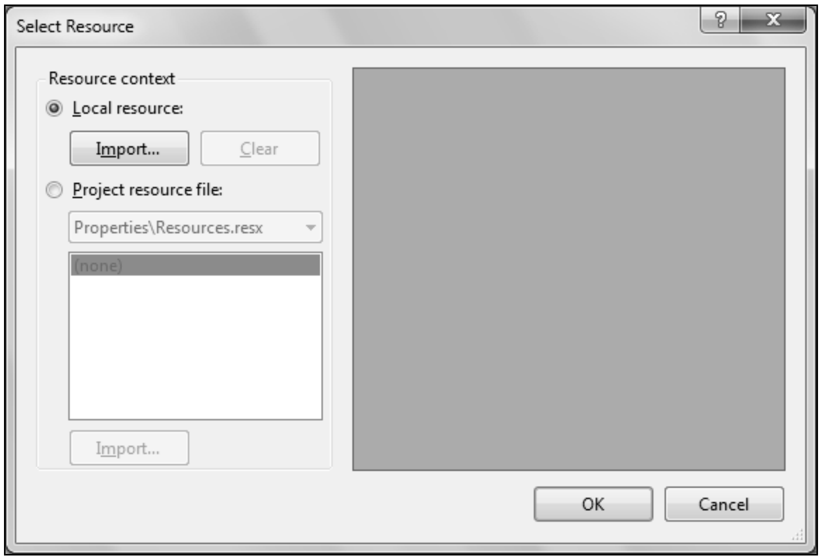


**Figure 1.4: Displaying the Select Resource Dialog Box**

3. Click the Import button in the Select Resource dialog box to import an image (Figure 1.4). The Open dialog box opens (Figure 1.5).
4. Select an image and click the Open button, as shown in Figure 1.5:

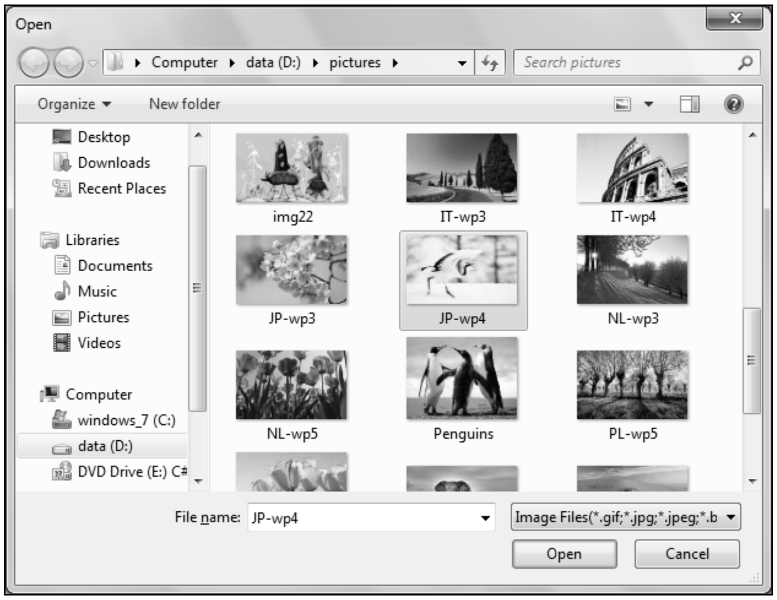


**Figure 1.5: Selecting an Image in the Open Dialog Box**

The Open dialog box closes and adds the image to the Select Resource dialog box, as shown in Figure 1.6:

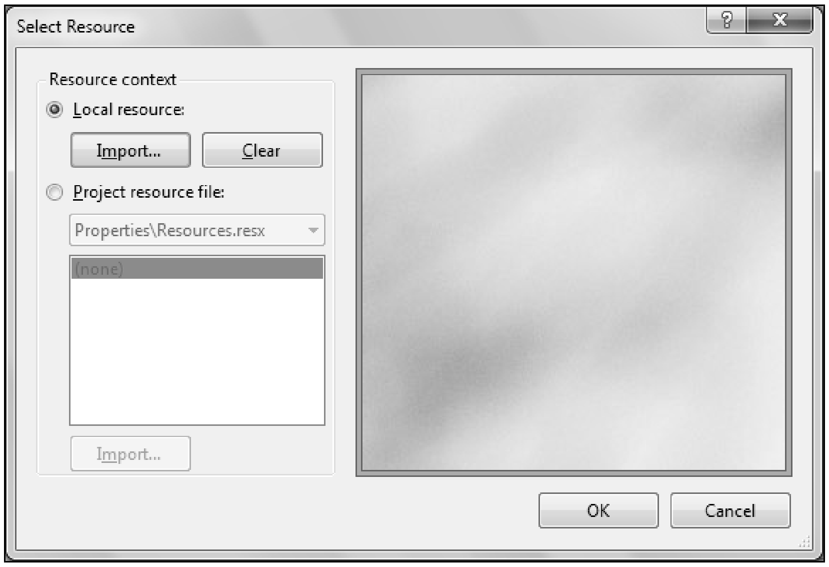


**Figure 1.6: The Selected Image Added to the Select Resource Dialog Box**

5. Click the OK button to close the Select Resource dialog box (Figure 1.6). The selected image is added as the background image of the button, as shown in Figure 1.7:

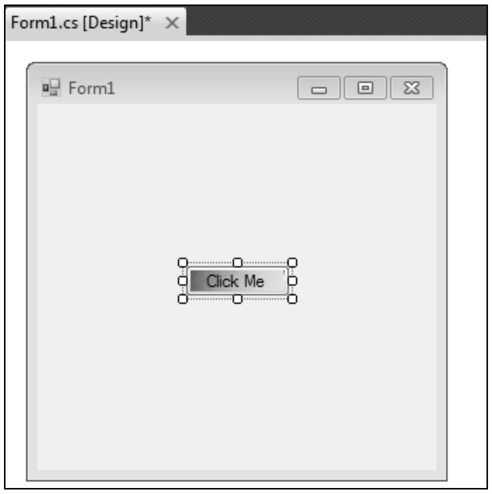


**Figure 1.7: Adding an Image to a Button**

## *Handling the Events of a Button*

You always respond to button clicks with the button's `Click` event. To handle the click event of the Button control, double-click the button at design time. This adds an event handler for handling the `Click` event of the button in the Code Editor window, as shown in the following code snippet:

```
    private void button1_Click(object sender, EventArgs e)
    {
    }
```

In the preceding code snippet, the `sender` argument is the button object itself that caused the event, and the `e` argument is a simple `EventArgs` object that doesn't contain any additional useful information. Now, you can add your code in this click event of the button1 control, as shown in the following code snippet:

```
private void button1_Click(object sender, EventArgs e) {
   button1.Text = "You Clicked Me";
}
```

In the preceding code snippet, we are changing the `Text` property of the button to `You Clicked Me` when the button is clicked at runtime. Now, if you execute the application and click the button, the `Click Me` text is changed to the `You Clicked Me` text.

A button control also supports the `MouseDown`, `MouseMove`, `MouseUp`, `KeyDown`, `KeyPress`, and `KeyUp` events. You learn more about the event of a button control in the Immediate Solutions section of this chapter.

Next, let's learn about the Label control.

## Describing the Label Control

The `Label` control is one of the most commonly used Windows Forms controls. This control is generally used to display the text that is not supposed to be changed by the user, such as caption text for a TextBox. The inheritance hierarchy of the `Label` class is given as follows:

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

Table 1.6 lists the noteworthy public properties of the `Label` class:

**Table 1.6: Noteworthy Public Properties of the Label Class**

| Property | Description |
|---|---|
| `AutoSize` | Retrieves or sets a value specifying if the control should be automatically resized to display all its contents |
| `BorderStyle` | Retrieves or sets the border style for the control |
| `FlatStyle` | Retrieves or sets the flat style appearance of the `Label` control |
| `PreferredHeight` | Retrieves or sets the preferred height of the control |
| `PreferredWidth` | Retrieves or sets the preferred width of the control |
| `TabStop` | Retrieves or sets the value that indicates whether the user can tab to the `Label` control |
| `Text` | Retrieves or sets the text content of the `Label` control |
| `TextAlign` | Retrieves or sets the alignment of text in the `Label` control |

Table 1.7 lists the noteworthy public methods of the `Label` class:

**Table 1.7: Noteworthy Public Methods of the Label Class**

| Method | Description |
|---|---|
| `GetPreferredSize()` | Retrieves the size of the rectangular area into which a control can be easily fit |
| `ToString()` | Returns a string that contains the name of the control |

Table 1.8 lists the noteworthy public events of the `Label` class:

**Table 1.8: Noteworthy Public Events of the Label Class**

| Event | Description |
|---|---|
| `AutoSizeChanged` | Occurs when the value of the `AutoSize` property changes |
| `TabStopChanged` | Occurs when the `TabStop` property changes |
| `TextAlignChanged` | Occurs when the `TabAlign` property has changed |

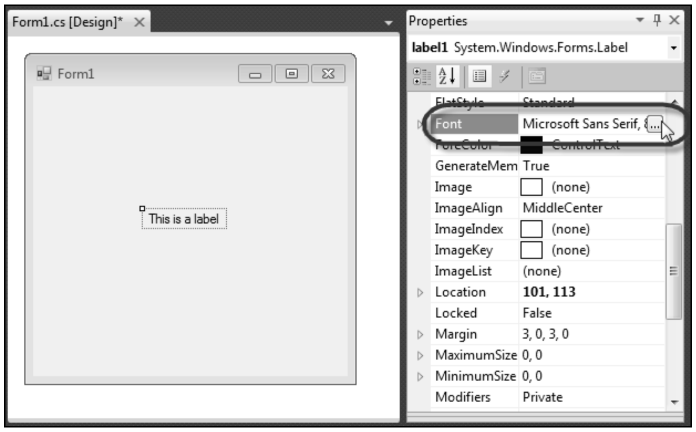
Now, in the following section, let's perform the following tasks to learn more about a label control:

- Formatting the text in a label control
- Handling the events of a label control

## Formatting the Text in a Label Control

You can format the text in a label by setting the `Font` property of the label using the Properties window. To format the text in the labels, perform the following steps:

1. Provide a suitable caption to the label by setting the `Text` property of the label using the Properties window. Then select the label and click the ellipsis button in front of the `Font` property in the Properties window (to open the Font dialog box), as shown in Figure 1.8:

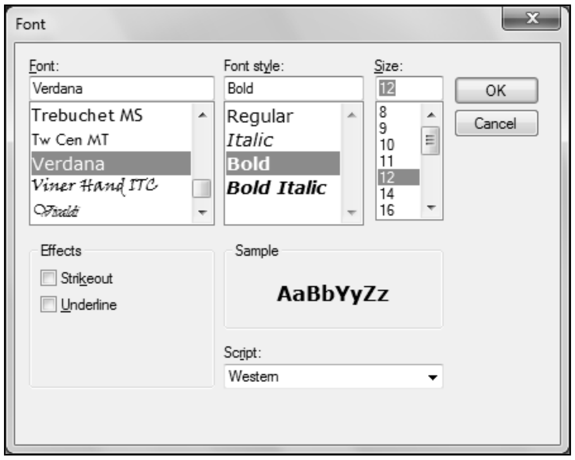


| | |
|---|---|
| FlatStyle | Standard |
| Font | Microsoft Sans Serif, 8 |
| ForeColor | ControlText |
| GenerateMem | True |
| Image | (none) |
| ImageAlign | MiddleCenter |
| ImageIndex | (none) |
| ImageKey | (none) |
| ImageList | (none) |
| Location | 101, 113 |
| Locked | False |
| Margin | 3, 0, 3, 0 |
| MaximumSize | 0, 0 |
| MinimumSize | 0, 0 |
| Modifiers | Private |

Figure 1.8: Opening the Font Dialog Box

The Font dialog box opens (Figure 1.9).

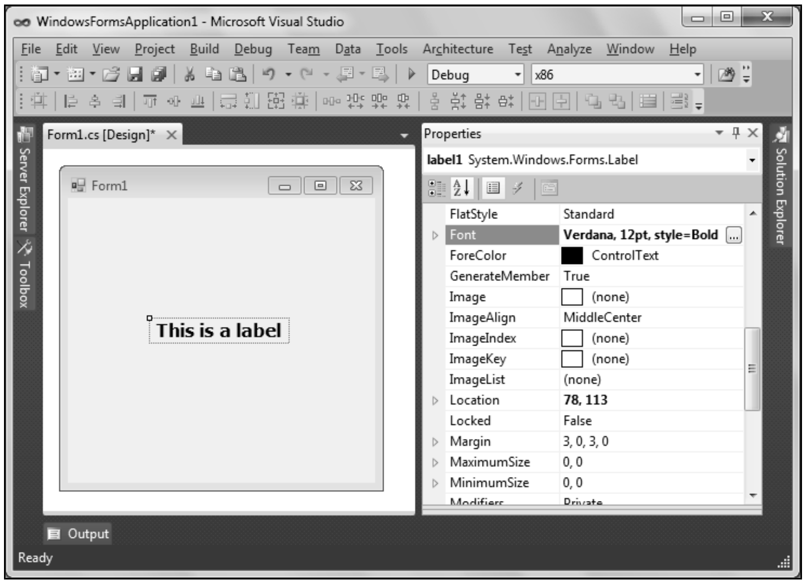
2. Select a font, font style, and size for the label and click the OK button in the Font dialog box, as shown in Figure 1.9:



Font: Verdana
Font style: Bold
Size: 12
Script: Western

Figure 1.9: The Font Dialog Box

This closes the Font dialog box and the selected font, font style, and size is applied to the label present on the Form, as shown in Figure 1.10:


WindowsFormsApplication1 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Debug
x86
Form1.cs [Design]*
Server Explorer
Toolbox
Solution Explorer
Form1
Properties
Output
Ready

This is a label

label1 System.Windows.Forms.Label

| | |
|---|---|
| FlatStyle | Standard |
| Font | Verdana, 12pt, style=Bold |
| ForeColor | ControlText |
| GenerateMember | True |
| Image | (none) |
| ImageAlign | MiddleCenter |
| ImageIndex | (none) |
| ImageKey | (none) |
| ImageList | (none) |
| Location | 78, 113 |
| Locked | False |
| Margin | 3, 0, 3, 0 |
| MaximumSize | 0, 0 |
| MinimumSize | 0, 0 |
| Modifiers | Private |

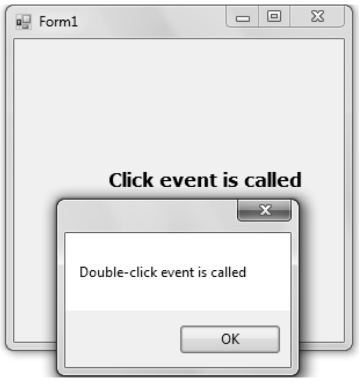
**Figure 1.10: Formatting the Text of a Label**

## *Handling the Events of a Label*

You can use various events of the `Label` control, such as `Click` and `DoubleClick`, to perform some actions at runtime. The event handlers for the `Click` and `DoubleClick` events of a Label control are given as follows:

```
    private void label1_Click(object sender, EventArgs e)
    {
            label1.Text="Click event is called";
     }

     private void label1_DoubleClick(object sender, EventArgs e)
     {
                MessageBox.Show("Double-click event is called");
    }
```

In the preceding code snippet, the Click event of label1 changes the text of the label, and the DoubleClick event of label1 shows a message box, as shown in Figure 1.11:



Click event is called

Double-click event is called

**Figure 1.11: Displaying the Effects of Events of a Label**

Next, let's learn about the TextBox control.

## Describing the TextBox Control

The `TextBox` control is a Windows Forms control that lets you enter text on a Windows Form at runtime. By default, a `TextBox` control accepts only a single line of text. However, you can make a `TextBox` control to accept multiple lines of text, add scroll bars to the control, and disable text editing in the control, by setting various properties of the `TextBox` control. The inheritance hierarchy of the `TextBox` class is given as follows:

```
System.Object
  System.MarshalByRefObject
        System.ComponentModel.Component
                System.Windows.Forms.Control
                        System.Windows.Forms.TextBoxBase
                                System.Windows.Forms.TextBox
```

**NOTE**

*Most of the functionality of the TextBox control is simply inherited from the TextBoxBase class, which is also a base class for the RichTextBox control.*

Windows Forms text boxes are used to get input from the user and display the text, although they can also be made read-only. Text boxes can display multiple lines, wrap text to the size of the control, and add basic formatting, such as quotation marks and masking characters for passwords. You can display a text in the control by setting its Text property at design time in the Properties window or at run time in code.

You can limit the length of the text that has to be entered into a `TextBox` control by setting its `MaxLength` property. By default, the `MaxLength` property is specified with the 32767 value. If it is set to zero, the `TextBox` control lets the user enter a maximum of 2147483646 characters or an amount based on available memory—whichever is smaller. For multiline `TextBox` controls (`TextBox` with its `Multiline` property set to `true`), the maximum number of characters, the user can enter, is 4294967295 or an amount based on available memory—whichever is smaller. The `TextBox` controls also can be used to accept passwords, if you use the `PasswordChar` property to mask characters.

You can also restrict text from being entered in a `TextBox` control by creating an event handler for the `KeyDown` event, and letting you validate each character entered in the control. Moreover, you can restrict any data entry in a `TextBox` control by setting the `ReadOnly` property to `true`. Table 1.9 lists the noteworthy public properties of the `TextBox` class:

**Table 1.9: Noteworthy Public Properties of the TextBox Class**

| Property | Description |
|---|---|
| `AcceptsReturn` | Retrieves or sets the value that indicates whether pressing ENTER in a multiline `TextBox` control creates a new line of text in the control or it activates the default button for the form |
| `AutoCompleteCustomSource` | Retrieves or sets a custom the `System.Collections.Specialized.StringCollection` namespace to use when the `AutoCompleteSource` property is changed to `CustomSource` |
| `AutoCompleteMode` | Retrieves or sets the option that controls how automatic completion works for the `TextBox` control |
| `AutoCompleteSource` | Retrieves or sets a value that specifies the source of complete strings used for automatic completion |
| `CharacterCasing` | Retrieves or sets a value that indicates whether the `TextBox` control changes the case of characters as they are typed |
| `PasswordChar` | Retrieves or sets the character that is used to mask characters for a password in a single-line text box |
| `ScrollBars` | Retrieves or sets what scroll bars should appear in a `multiline` text box |
| `TextAlign` | Retrieves or sets how text is aligned in a text box control |

**Table 1.9: Noteworthy Public Properties of the TextBox Class**

| Property | Description |
|---|---|
| `UseSystemPasswordChar` | Retrieves or sets the value that indicates whether the text in the `TextBox` control appear as default password character |

Table 1.10 lists the noteworthy public event of the `TextBox` class:

**Table 1.10: Noteworthy Public Event of the TextBox Class**

| Event | Description |
|---|---|
| `TextAlignChanged` | Occurs when the value of the `TextAlign` property changes |

Next, let's learn about the RichTextBox control.

## Describing the RichTextBox Control

The `RichTextBox` control is used for displaying, entering, and manipulating rich text with formatting. It does everything that the `TextBox` control does, but in addition, it lets you make the text of a `RichTextBox` control bold, italic, and underlined, change the color of the text, select font design and font sizes, save the text of a `RichTextBox` control to a Rich Text Format (RTF) file, and load the text of a RTF file in a `RichTextBox` control. The inheritance hierarchy of the `RichTextBox` class is given as follows:

```
System.Object
  System.MarshalByRefObject
        System.ComponentModel.Component
               System.Windows.Forms.Control
                      System.Windows.Forms.TextBoxBase
                             System.Windows.Forms.RichTextBox
```

Note that the preceding inheritance hierarchy is as similar to the `TextBox` control, as the `RichTextBox` control also derives from the `TextBoxBase` class.

Table 1.11 lists the noteworthy public properties of the `RichTextBox` class:

**Table 1.11: Noteworthy Public Properties of the RichTextBox Class**

| Property | Description |
|---|---|
| `AllowDrop` | Retrieves or sets a value that indicates whether the control will enable drag-and-drop operations |
| `AutoSize` | Retrieves or sets a value specifying if the size of the rich text box automatically adjusts when the font changes |
| `AutoWordSelection` | Retrieves or sets a value specifying if the user is allowed to select an entire word in the text of a `RichTextBox` control by double-clicking the word |
| `BulletIndent` | Retrieves or sets the indentation used in the rich text box when the bullet style is applied to the text |
| `CanRedo` | Retrieves or sets a value indicating if there are actions in rich text box that can be reapplied |
| `DetectUrls` | Retrieves or sets a value specifying if the rich text box should detect URLs when typed into the `RichTextBox` control |
| `EnableAutoDragDrop` | Retrieves or sets the value that enables drag-and-drop operation on pictures, text, and other data |
| `Font` | Retrieves or sets the font used when displaying text in the control |
| `ForeColor` | Retrieves or sets the forecolor used when displaying text in the control |

**Table 1.11: Noteworthy Public Properties of the RichTextBox Class**

| Property | Description |
|---|---|
| `LanguageOption` | Retrieves or sets the value that indicates RichTextBox settings for Input Method Editor (IME) and Asian language support |
| `MaxLength` | Retrieves or sets the maximum number of characters the user can type into the rich text box |
| `Multiline` | Retrieves or sets a value specifying if this is a multiline `RichTextBox` control |
| `RedoActionName` | Retrieves or sets the name of the action that can be reapplied to the control when the Redo`()` method is called |
| `RightMargin` | Retrieves or sets the size of a single line of text within the `RichTextBox` control |
| `Rtf` | Retrieves or sets the text of the `RichTextBox` control, including all RTF codes |
| `ScrollBars` | Retrieves or sets the kind of scroll bars to display in the `RichTextBox` control |
| `SelectedRtf` | Retrieves or sets the currently selected RTF formatted text in the control |
| `SelectedText` | Retrieves or sets the selected text within the rich text box |
| `SelectionAlignment` | Retrieves or sets the alignment to apply to the current selection or insertion point |
| `SelectionBackColor` | Retrieves or sets the color of the text when the text is selected in a `RichTextBox` control |
| `SelectionBullet` | Retrieves or sets a value specifying if the bullet style is applied to the current selection or insertion point |
| `SelectionCharOffset` | Retrieves or sets if text in the `RichTextBox` control appears on the baseline, as a superscript, or as a subscript |
| `SelectionColor` | Retrieves or sets the text color of the current text selection or insertion point |
| `SelectionFont` | Retrieves or sets the font of the current text selection or insertion point |
| `SelectionHangingIndent` | Retrieves or sets the distance between the left edge of the first line of text in the selected paragraph and the left edge of the next lines in the same paragraph |
| `SelectionIndent` | Retrieves or sets the distance in pixels between the left edge of the rich text box and the left edge of the current text selection or text added after the insertion point |
| `SelectionLength` | Retrieves or sets the number of characters selected in control |
| `SelectionProtected` | Retrieves or sets a value that indicates whether the current text selection is protected |
| `SelectionRightIndent` | Retrieves or sets a value that specifies the distance in pixels between the right edge of the `RichTextBox` control and the right edge of the text that is selected |
| `SelectionType` | Retrieves or sets the selection type within the control |
| `Text` | Retrieves or sets the current text in the rich text box |
| `TextLength` | Retrieves or sets the length of text in the `RichTextBox` control |

Table 1.12 lists the noteworthy public methods of the `RichTextBox` class:

**Table 1.12: Noteworthy Public Methods of the RichTextBox Class**

| Method | Description |
|---|---|
| `CanPaste()` | Determines if you can paste information from the Clipboard |
| `Find()` | Searches for text within the contents of the rich text box |

**Table 1.12: Noteworthy Public Methods of the RichTextBox Class**

| Method | Description |
|---|---|
| `GetLineFromCharIndex()` | Retrieves or sets the line number from the specified character position within the text of the `RichTextBox` control |
| `GetCharIndexFromPosition()` | Retrieves or sets the location within the control at the specified character index |
| `GetPositionFromCharIndex()` | Helps in retrieving the location at a specified character index within the control |
| `LoadFile()` | Loads the contents of a file into the `RichTextBox` control |
| `SaveFile()` | Saves the contents of the rich text box to a file |

Table 1.13 lists the noteworthy public events of the `RichTextBox` class:

**Table 1.13: Noteworthy Public Events of the RichTextBox Class**

| Event | Description |
|---|---|
| `ContentsResized` | Occurs when the contents within the control are resized |
| `HScroll` | Occurs when the user clicks the horizontal scroll bar of the control |
| `SelectionChanged` | Occurs when a change is made in the selected text within the control |
| `VScroll` | Occurs when the user clicks the vertical scroll bar |

Next, let's learn about the MaskedTextBox control.

## Describing the MaskedTextBox Control

The `MaskedTextBox` control is an improvement over the `TextBox` control as it uses a declarative syntax (mask) to distinguish between proper and improper user input. This control lets you specify a format for input and avoid wrongly formatted or unexpected inputs. The control can be used to mask a phone number, date, time, Social Security Number (SSN), and zip code. The inheritance hierarchy of the `MaskedTextBox` class is given as follows:

```
System.Object
  System.MarshalByRefObject
        System.ComponentModel.Component
                System.Windows.Forms.Control
                        System.Windows.Forms.TextBoxBase
                                System.Windows.Forms.MaskedTextBox
```

Table 1.14 lists the noteworthy public properties of the `MaskedTextBox` class:

**Table 1.14: Noteworthy Public Properties of the MaskedTextBox Class**

| Property | Description |
|---|---|
| `BeepOnError` | Retrieves or sets a value indicating whether the `MaskedTextBox` control should raise a beep for each invalid key stroke made by the user. |
| `CanUndo` | Retrieves or sets a value indicating whether the user can undo the previous operation. This property is not supported by the `MaskedTextBox` control. |
| `CutCopyMaskFormat` | Retrieves or sets a value that determines whether literals and prompt characters are copied to the clipboard. |
| `IsOverwriteMode` | Retrieves or sets a value that specifies whether new user input overwrites existing input. |
| `Lines` | Retrieves or sets the lines of text in multiline configurations. This property is not supported by the `MaskedTextBox` control. |
| `Mask` | Retrieves or sets the input mask to use at run time. |

**Table 1.14: Noteworthy Public Properties of the MaskedTextBox Class**

| Property | Description |
|---|---|
| `MaskCompleted` | Retrieves or sets a value indicating whether all required inputs have been entered into the input mask. |
| `MaskedTextProvider` | Retrieves or sets a clone of the mask provider associated with this instance of the `MaskedTextBox` control. |
| `MaskFull` | Retrieves or sets a value indicating whether all required and optional inputs enter into the input mask. |
| `MaxLength` | Retrieves or sets the maximum number of characters the user can type or paste into the `TextBox` control. This property is not supported by the `MaskedTextBox` control. |
| `Multiline` | Retrieves or sets a value indicating whether this is a multiline text box control. This property is not fully supported by the `MaskedTextBox` control. |
| `SelectedText` | Retrieves or sets the current selection in the `MaskedTextBox` control. |
| `Text` | Retrieves or sets the text as it is currently displayed to the user. |
| `TextAlign` | Retrieves or sets how text is aligned in a `MaskedTextBox` control. |
| `TextLength` | Retrieves or sets the length of the displayed text. |
| `TextMaskFormat` | Retrieves or sets a value that determines whether literals and prompt characters are included in the formatted string. |
| `WordWrap` | Retrieves or sets a value indicating whether a multiline text box control automatically wraps words to the beginning of the next line when necessary. This property is not supported by the `MaskedTextBox` control. |

Table 1.15 lists the noteworthy public methods of the `MaskedTextBox` class:

**Table 1.15: Noteworthy Public Methods of the MaskedTextBox Class**

| Method | Description |
|---|---|
| `GetCharFromPosition()` | Retrieves the character that is closest to the specified location within the control. |
| `GetCharIndexFromPosition()` | Retrieves the index of the character nearest to the specified location. |
| `GetPositionFromCharIndex()` | Retrieves the location within the control at the specified character index. |
| `GetFirstCharIndexFromLine()` | Retrieves the index of the first character of a given line. This method is not supported by the `MaskedTextBox` control. |
| `GetFirstCharIndexOfCurrentLine()` | Retrieves the index of the first character of the current line. This method is not supported by the `MaskedTextBox` control. |
| `GetLineFromCharIndex()` | Retrieves the line number from the specified character position within the text of the control. This method is not supported by the `MaskedTextBox` control. |
| `ScrollToCaret()` | Scrolls the contents of the control to the current caret position. This method is not supported by the `MaskedTextBox` control. |
| `ToString()` | Returns the string that represents the current `MaskedTextBox` control. |
| `Undo()` | Undoes the last edit operation in the text box. This method is not supported by the `MaskedTextBox` control. |
| `ValidateText()` | Converts the user input string to an instance of the validating type. |

Table 1.16 lists the noteworthy public events of the `MaskedTextBox` class:

**Table 1.16: Noteworthy Public Events of the MaskedTextBox Class**

| Event | Description |
|---|---|
| `AcceptsTabChanged` | Occurs when the value of the `AcceptsTab` property has changed. This event is not raised by the `MaskedTextBox` control. |
| `IsOverwriteModeChanged` | Occurs after the insert mode has changed. |
| `MaskChanged` | Occurs after the input mask is changed. |
| `MaskInputRejected` | Occurs when the user's input or assigned character does not match the corresponding format element of the input mask. |
| `MultilineChanged` | Occurs when the value of the Multiline property is changed. |
| `TextAlignChanged` | Occurs when the text alignment is changed. |
| `TypeValidationCompleted` | Occurs when the `MaskedTextBox` control has finished parsing the current value using the `ValidatingType` property. |

As far, you have covered variety of Windows Forms controls, such as Button, Label, TextBox, and RichTextBox, on conceptual basis. Now, it's time to learn  practically about the different aspects of all these controls as discussed in the In Depth  section.

# Immediate Solutions

## Using the Button Control

We presented an overview of buttons in the *In Depth* section of this chapter. Now, let's learn how we can use buttons in our Windows applications. We start by creating a new Windows application, named `ButtonControl`, which is also available in the CD-ROM provided with this book.

Let's perform the following steps to create the `ButtonControl` application:

1. Create a Windows Forms application, named `ButtonControl`.
2. Add six button controls from Toolbox to the Form1 (in Designer mode).
3. Set the Text property of the Form1 as Buttons Form and set the Text property of these button controls, as follows:
    - button1 to Click Me
    - button2 to Red Foreground
    - button3 to Green background
    - button4 to Image Button
    - button5 to Keep mouse over me
    - button6 to Move mouse over me
4. Modify the ForeColor property of the button2 control to the red color, the BackColor property of the button3 control to the green color. In addition, set an image on the button4 control through its Image property. Now, arrange the controls on Form1, as shown in Figure 1.12:

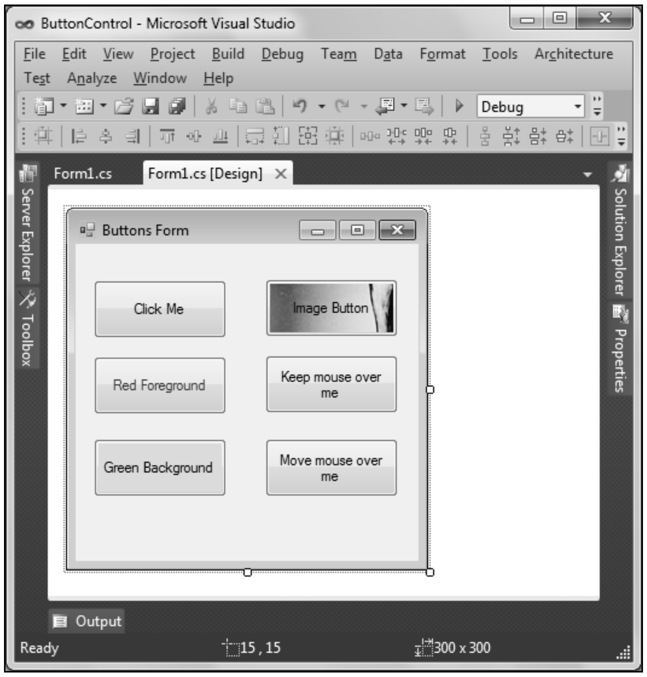


Figure 1.12: Displaying the Button Controls with Various Properties

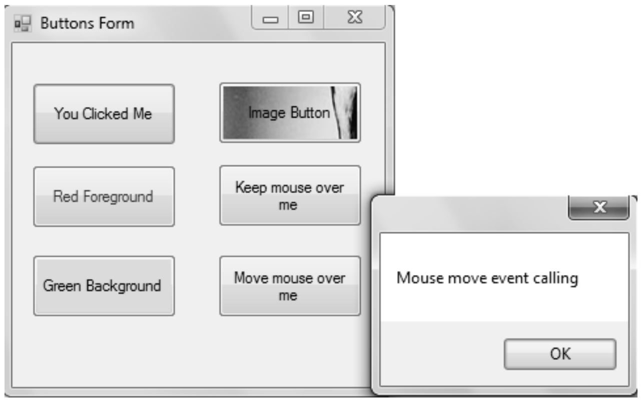
6. Add the code, given in Listing 1.1, to the Form1.cs file:

**Listing 1.1:** Adding the Code to Use Button Controls

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace ButtonControl
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            button1.Text = "You Clicked Me";
        }
        private void button5_MouseHover(object sender, EventArgs e)
        {
            button1.Text = "Click Me";
        }
        private void button6_MouseMove(object sender, MouseEventArgs e)
        {
            MessageBox.Show("Mouse move event calling");
        }
    }
}
```

In Listing 1.1, we are adding code for the events of the button1, button5, and button6 controls. The Click event of button1 displays a text message, You Clicked Me, on the button itself. The MouseHover event of button5 displays the Click Me text again on the button1 control. Finally, the button6 control displays a message box with displaying the Mouse move event calling message. Note that button2, button3, and button4 are used in the application for the designing purpose. The button2 control displays the text in red color, the button3 control displays the background of the button as green color, and button3 displays an image on the button itself.

7. Press the F5 key on the keyboard to run the `ButtonControl` application. Now, click the Click Me button in Buttons Form, so that the text of the button is changed to the `You Clicked Me` text. If you move mouse over the `Move mouse over me` button, it displays a message box, as shown in Figure 1.13:



Mouse move event calling

**Figure 1.13: Displaying the Output of the ButtonControl Application**

When you keep mouse over the `Keep mouse over me` button, the text of the button1 control again changes to the `Click Me` text, as shown in Figure 1.14:

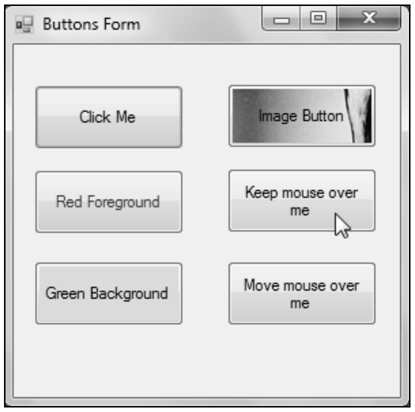


**Figure 1.14: Changing the Text of the button1 Control**

Next, you create an application using the Label control.

## Using the Label Control

A variety of operations can be performed on the `Label` control, such as formatting the text and handling of events. In this section, we are creating a Windows Forms application, LabelControl (also available in the CD), to display the use of Label controls. Let's perform the following steps to create the LabelControl application:

1. Create a Windows Forms application, named LabelControl.
2. Add three labels and one text box controls from Toolbox to Form1 (in Designer mode).
3. Set the Text property of the Form1 as Labels Form and also set the properties of the Label controls, as shown in Table 1.17:

**Table 1.17: Control Properties**

| Control | Text property | Font property | BackColor property | ForeColor property |
|---|---|---|---|---|
| label1 | Enter you name: | Calibri,14 | Default | Default |
| label2 | Displaying status | Verdana,10, Underlined | Default | Green |
| label3 | Double-click Me | Calibri,14 | 255,192,192 | Default |

4. Now, arrange the controls on Form1, as shown in Figure 1.15:

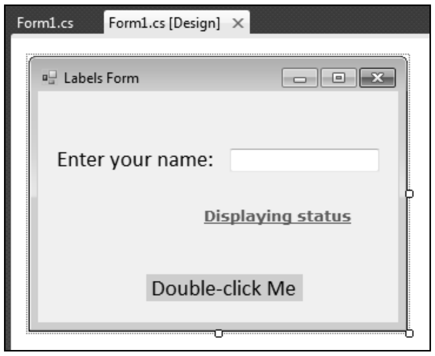


**Figure 1.15: Displaying the Controls after Specifying their Properties**

5. Add the following code snippet in the DoubleClick event of the label3 control:

```
private void label3_DoubleClick(object sender, EventArgs e)
{
 string nm = textBox1.Text;
 label2.Text="Your name is: " + nm;
}
```

6. Press the F5 key on the keyboard to run the LabelControl application. Figure 1.16 shows the output of the LabelControl application:

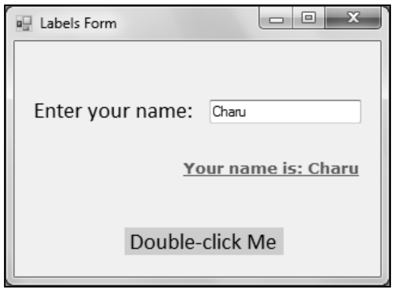


**Figure 1.16: Displaying the Output of the LabelControl Application**

Next, let's create an application using the TextBox control.

## Using the TextBox Control

In this section, we are creating a Windows Forms application, TextBoxControl (also available in the CD), to display the text alignments in a text box control. Let's perform the following steps to create the TextBoxControl application:

1. Create a Windows Forms application, named TextBoxControl.
2. Set the Text property of the Form1 as Text boxes Form.
3. Add two TextBox, one Label, and one Button control from Toolbox to the Form1 (in Designer mode). Set the Text property of the label1 control to Enter a Number and the button1 control to Generate the Table. Change the TextAlign property of the textBox1 control to Right and the textBox2 control to Center. You should also set the Multiline property of textBox2 to True. Now, arrange the controls on Form1, as shown in Figure 1.17:

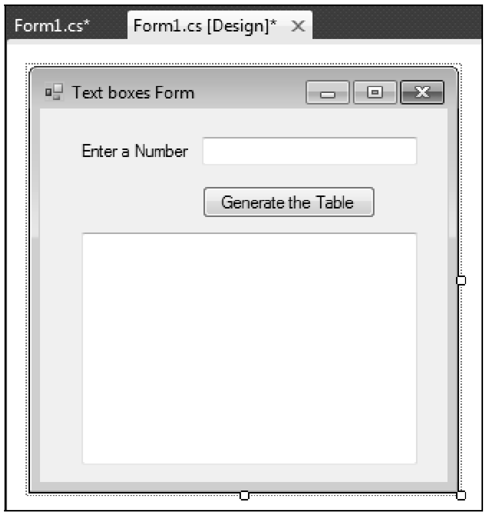


**Figure 1.17: Displaying the Controls Arrangement**

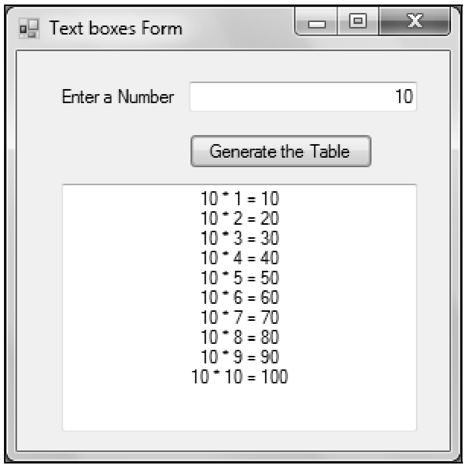
4. Add the code, given in Listing 1.2, to the Form1.cs file:

**Listing 1.2:** Modifying the Code of the Form1.cs File

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;

using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace TextBoxControl
{
    public partial class Form1 : Form
    {

        public Form1()
        {
            InitializeComponent();
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {

            if (e.KeyChar < '0' | e.KeyChar > '9')
            {
                MessageBox.Show("Please enter only digits!");
                e.Handled = true;
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            int x = Convert.ToInt32(textBox1.Text);
            int z;
            for (int i = 1; i <= 10; i++)

            {
                z = x * i;
                textBox2.AppendText(x+" * "+i+" = "+z.ToString()+"\n");
            }

        }
    }

}
```

In Listing 1.2, the KeyPress event of the textbox1 control shows a message box, if you press a key other than a numeric key. The Click event of the button1 generates the mathematical table of the entered number in the textbox1 control to the textBox2 control.

5. Press the F5 key on the keyboard to run the TextBoxControl application.
6. Enter a number in the Enter a Number text box (in our case 10 in entered) and click the Generate the Table button. Figure 1.18 shows the output after clicking the Generate the Table button:



```
10
```

```
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50
10 * 6 = 60
10 * 7 = 70
10 * 8 = 80
10 * 9 = 90
10 * 10 = 100
```

**Figure 1.18: Displaying the Output of the TextBoxControl Application**

In Figure 1.18, you can see the text alignment in the first text box as right and in the second text box as center. You should also note that if you try to enter any character other than a number in the first text box, you get the Please enter only digits! message, as shown in Figure 1.19:

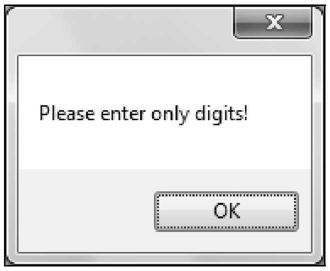

Please enter only digits!

**Figure 1.19: Displaying a Message Box**

Next, let's  create an application using the RichTextBox control.

## Using RichTextBox Controls

In this section, we are creating a Windows Forms application RichTextBoxControl (also available in the CD) to display the use of rich text box control. Let's perform the following steps to create the RichTextBoxControl application:

1. Create a Windows Forms application, named RichTextBoxControl.
2. Set the Text property of the Form1 as Rich text boxes Form.
3. Add three rich text boxes and five button controls from Toolbox to the Form1(in Designer mode). Set the Text property of the button controls given as follows:
   - button1 to Format Text
   - button2 to Copy Text
   - button3 to Indent Text
   - button4 to Add Bullets
   - button5 to Color Text

Now, arrange the controls on Form1, as shown in Figure 1.20:

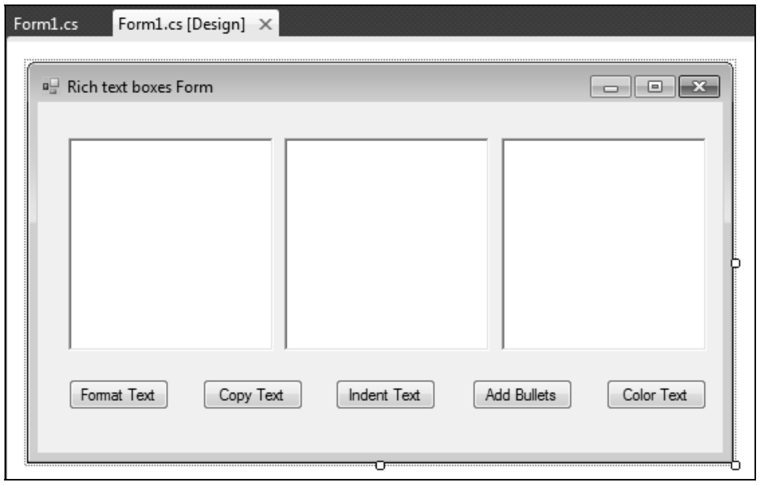


**Figure 1.20: Displaying Controls Arrangement**

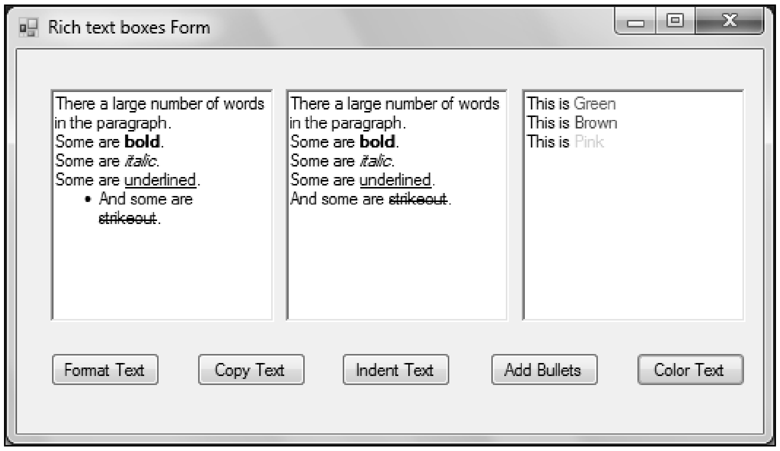
4. Add the code, given in Listing 1.3, to the Form1.cs file:

**Listing 1.3:** Adding the Code to Use the RichTextBox Control

```
 using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace RichTextBoxControl
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            richTextBox1.SelectionStart = richTextBox1.Find("bold");
            Font boldFont = new Font(richTextBox1.Font, FontStyle.Bold);
            richTextBox1.SelectionFont = boldFont;
            richTextBox1.SelectionStart = richTextBox1.Find("italic");
            Font italicFont = new Font(richTextBox1.Font, FontStyle.Italic);
            richTextBox1.SelectionFont = italicFont;
            richTextBox1.SelectionStart = richTextBox1.Find("underlined");
            Font underlineFont = new Font(richTextBox1.Font, FontStyle.Underline);
            richTextBox1.SelectionFont = underlineFont;
            richTextBox1.SelectionStart = richTextBox1.Find("strikeout");
            Font strikeoutFont = new Font(richTextBox1.Font, FontStyle.Strikeout);
            richTextBox1.SelectionFont = strikeoutFont;
        }

        private void button2_Click(object sender, EventArgs e)
        {
            richTextBox2.Rtf = richTextBox1.Rtf;
        }

        private void button3_Click(object sender, EventArgs e)
        {
            richTextBox1.SelectionIndent = 20;
            richTextBox1.SelectionHangingIndent = -25;
            richTextBox1.SelectionRightIndent = 10;
        }

        private void button4_Click(object sender, EventArgs e)
        {
            richTextBox1.SelectionIndent = 20;
            richTextBox1.BulletIndent = 10;
            richTextBox1.SelectionBullet = true;
        }

        private void button5_Click(object sender, EventArgs e)
        {
            richTextBox3.SelectionStart = richTextBox3.Find("Green");
            richTextBox3.SelectionColor = Color.Green;
            richTextBox3.SelectionStart = richTextBox3.Find("Brown");
            richTextBox3.SelectionColor = Color.Brown;
            richTextBox3.SelectionStart = richTextBox3.Find("Pink");
            richTextBox3.SelectionColor = Color.Pink;
        }

    }
}
```

In Listing 1.3, we have added the code in the Click event of the button1 control to change the format of the specified text of the richTextbox11 control. It converts the `bold` text into bold, `italic` into italic, `underlined` into underlined, and `strikeout` into strikeout format. The Click event of the button2 control copies the text of the richTextBox1 control to the richTextBox2 control. The Click events of the button3 and button4 controls perform the task of text indentation and bullets insertion. Finally, the Click event of the button5 control changes the color of the text in the richTextBox3 control.

5. Press the F5 key on the keyboard to run the RichTextBoxControl application. Figure 1.21 shows the output of the RichTextBoxControl application:



There a large number of words in the paragraph.
Some are **bold**.
Some are *italic*.
Some are underlined.
- And some are ~~strikeout~~.

There a large number of words in the paragraph.
Some are **bold**.
Some are *italic*.
Some are underlined.
And some are ~~strikeout~~.

This is Green
This is Brown
This is Pink

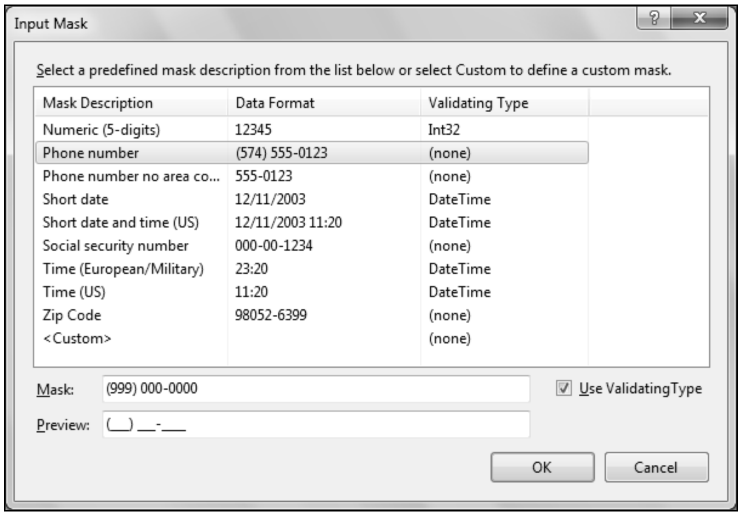
**Figure 1.21: Displaying the Output of the RichTextBoxControl Application**

Next, let’s  create an application using the MaskedTextBox control.

## Using MaskedTextBox Controls

In this section, we are creating a Windows Forms application, MaskedTextBox (also available in the CD), to display the use of a masked text box control. In this application, you can see how a phone number is validated with the help of a masked text box. Let's perform the following steps to create the MaskedTextBox application:

1. Create a Windows Forms application, named MaskedTextBox.
2. Add a label and a masked text box controls from Toolbox to the Form1 (in Designer mode). Set the Text property of the label1 control to Enter your phone number. Now, set the Mask property of the maskedTextBox1 control with the help of the Input Mask dialog box, as shown in Figure 1.22:


Input Mask

Select a predefined mask description from the list below or select Custom to define a custom mask.

| Mask Description | Data Format | Validating Type |
|---|---|---|
| Numeric (5-digits) | 12345 | Int32 |
| Phone number | (574) 555-0123 | (none) |
| Phone number no area co... | 555-0123 | (none) |
| Short date | 12/11/2003 | DateTime |
| Short date and time (US) | 12/11/2003 11:20 | DateTime |
| Social security number | 000-00-1234 | (none) |
| Time (European/Military) | 23:20 | DateTime |
| Time (US) | 11:20 | DateTime |
| Zip Code | 98052-6399 | (none) |
| <Custom> | | (none) |

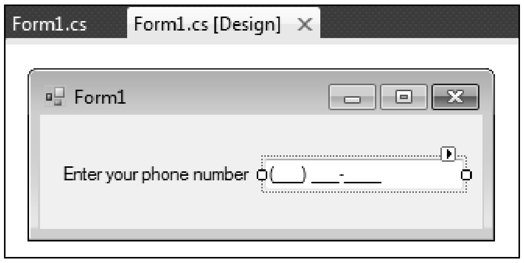
Mask: (999) 000-0000

☑ Use ValidatingType

Preview: (___) ___-____

**Figure 1.22: Selecting a Predefined Mask Format from the Input Mask Dialog Box**

3. Select the Phone number option in the predefined masks and the Ok button available in the Input Mask dialog box (Figure 1.22). Now, arrange the controls on Form1, as shown in Figure 1.23:



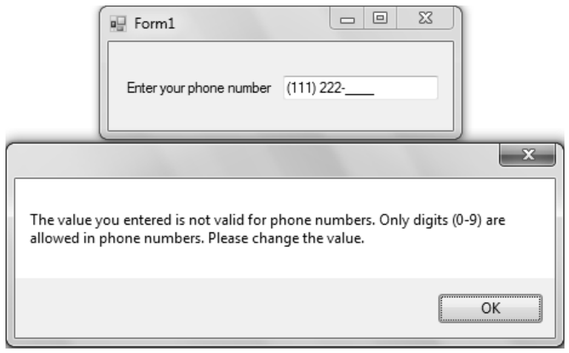
Enter your phone number (___) ___-____

**Figure 1.23: Displaying Controls Arrangement in the Application**

4. Add the following code snippet to the MaskInputRejected event of the maskedTextbox1 control:

```
private  void  maskedTextBox1_MaskInputRejected(object  sender,  MaskInputRejectedEventArgs
 e)
{
 MessageBox.Show("The value you entered is not valid for phone numbers. Only digits (0-9)
 are allowed in phone numbers. Please change the value.");
}
```

5. Press the F5 key on the keyboard to run the MaskedTextBox application. Figure 1.24 shows the output of the MaskedTextBox application:



Enter your phone number (111) 222-____

The value you entered is not valid for phone numbers. Only digits (0-9) are allowed in phone numbers. Please change the value.

**Figure 1.24: Displaying the Output of the MaskedTextBox Application**

In Figure 1.24, you should note that, when you try to enter a character other than a numeric character, a message box appears and displays an error message.

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

## Summary

In this chapter, you have learned how to work with some basic Windows Forms controls including `Button`, `Label`, `TextBox`, `RichTextBox`, and `MaskedTextBox`. We have started the chapter with a description of the `Button` control, which lets you generate and handle a `Click` event. Next, the `Label` control has explored, which can display captions for other controls. Then, the remaining three controls—`TextBox`, `RichTextBox`, and `MaskedTextBox`— have been explained, which perform the basic task of taking inputs from the users.

In the next chapter, you learn how to work with some other Windows Forms controls, including `RadioButton`, `CheckBox`, `ListBox`, `CheckedListBox`, and `ComboBox`.