Event Driven Programming

Elements of Event Driven Programming

Chalew Tesfaye
Department of Computer Science
Debre Berhan University
2017

Objectives

- At the end of the lesson students should be able to:
  - ✓ Understand events and delegates
  - ✓ Create an event handler for a control
  - ✓ Work with windows controls
  - ✓ Work with properties and events of controls
  - ✓ Design and build windows based application

Contents

- Working on events and delegates
- Creating an Event Handler for a Control
- Using Windows Forms
- Working with Windows Forms Controls
- Using Dialog Boxes in a Windows Forms Application
- Creating Menus
- Adding Controls at Run Time

Working with Controls

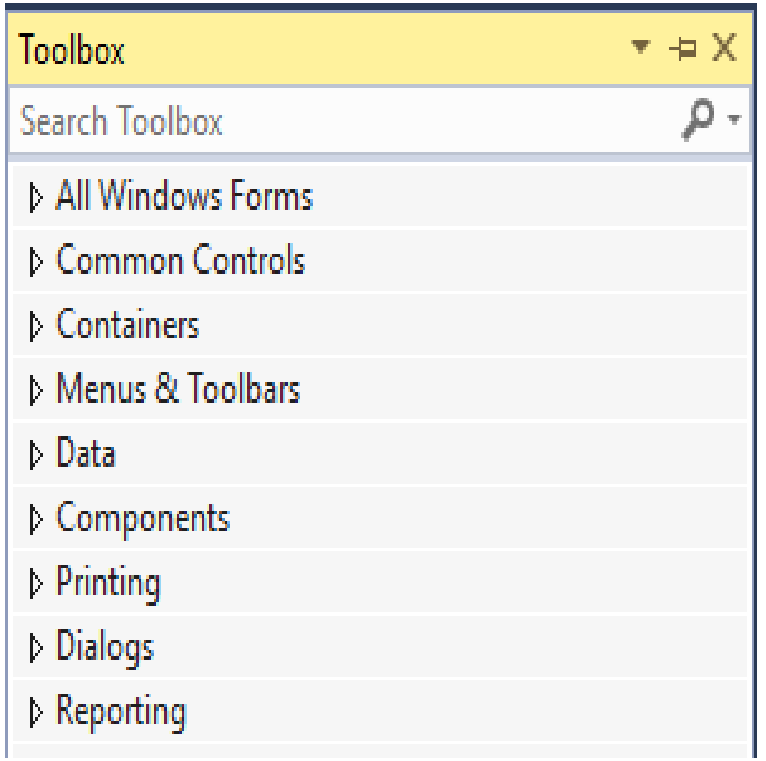
Building blocks of GUI based application

Controls …

- Controls are classes, hence have
  - ✓ Properties
  - ✓ Methods
  - ✓ Events – special methods

Controls …

- Depending on the functionality that you want to provide in the user interface of your application,
  - ✓ you will select a control from one of the following categories:



Controls …

- Command controls
  - ✓ Button
  - ✓ LinkLabel
  - ✓ NotifyIcon
  - ✓ ToolTip
- Text controls enable users to enter text and edit the text contained in these controls at run time:
  - ✓ Textbox
  - ✓ Rich Textbox
- The following additional text controls can be used to display text but do not allow application users to directly edit the text content that they display:
  - ✓ Label
  - ✓ StatusBar

Controls …

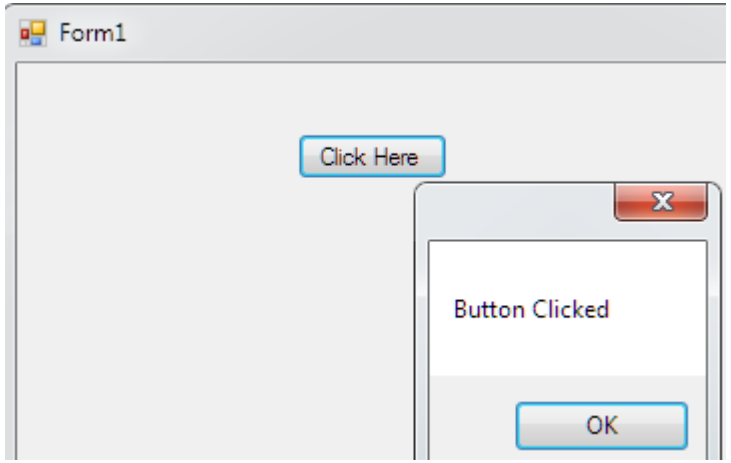
- The following selection controls allow users to select a value from a list:
  - ✓ CheckedListBox
  - ✓ ComboBox
  - ✓ DomainUpDown
  - ✓ ListBox
  - ✓ ListView
  - ✓ NumericUpDown
  - ✓ TreeView
- The following are the categories of menu controls:
  - ✓ MenuStrip
  - ✓ ContextMenu

Controls …

- Container controls can be used to group other controls on a form
  - ✓ Panel
  - ✓ GroupBox
  - ✓ TabControl
- The following are the categories of graphic controls:
  - ✓ ImageList
  - ✓ PictureBox
- Visual Studio .NET provides a set of common dialog boxes. These include
  - ✓ ColorDialog
  - ✓ FontDialog
  - ✓ PageSetupDialog
  - ✓ PrintDialog
  - ✓ OpenFileDialog
  - ✓ SaveFileDialog
  - ✓ FolderBrowtherDialog

Button Control

- Is an interactive component that enables users to communicate with an application
- The Button class inherits directly from the ButtonBase class.
- A Button control is a child control placed on a Form and used to process click event
- A Button can be clicked by using the mouse, ENTER key, ESC Key or SPACEBAR if the button has focus.
- It can be also used to start, stop and interrupt



Button Control …

- **Button Properties:**
  - ✓ Set button properties in properties windows, such name, text, size, color,…
- **Button Event:**
  - ✓ Button control has events, Click, DoubleClick, MouseDown, and lots more.
  - ✓ Use properties window to select appropriate event by click the lightning bolt.
- **Button Methods:**

Button Control …

- **Creating a Button**
  - ✓ To create a Button control, you simply drag and drop a Button control from Toolbox to Form in Visual Studio.
  - ✓ Once a Button is on the Form, you can move it around and resize it using mouse.

Button Control …

Setting Button Properties

- ✓ After you place a Button control on a Form, the next step is to set button properties.
- ✓ The easiest way to set a Button control properties is by using the Properties Window.
- ✓ You can open Properties window by pressing F4 or right click on a control and select Properties menu item
- ✓ **Background and Foreground**
  - > BackColor and ForeColor properties are used to set background and foreground color of a Button respectively.
  - > If you click on these properties in Properties window, the Color Dialog pops up.
  - > Alternatively, you can set background and foreground colors at run-time.
  - > The following code snippet sets BackColor and ForeColor properties.

```
// Set background and foreground
dynamicButton.BackColor = Color.Red;
dynamicButton.ForeColor = Color.Blue;
```

- **AutoEllipsis**:
  - ✓ enables the automatic handling of text that extends beyond the width of the button
- **Image in Button**
  - ✓ The Image property of a Button control is used to set a button background as an image.
  - ✓ The Image property needs an Image object.
  - ✓ The Image class has a static method called FromFile that takes an image file name with full path and creates an Image object.
  - ✓ You can also align image and text.
  - ✓ The ImageAlign and TextAlign properties of Button are used for this purpose.
  - ✓ The following code snippet sets an image as a button background.

```
// Assign an image to the button.
dynamicButton.Image = Image.FromFile(@"C:\Images\Dock.jpg");
// Align the image and text on the button.
dynamicButton.ImageAlign = ContentAlignment.MiddleRight;
dynamicButton.TextAlign = ContentAlignment.MiddleLeft;
// Give the button a flat appearance.
dynamicButton.FlatStyle = FlatStyle.Flat;
```

✓ Text and Font

- The Text property of Button represents the contents of a Button.
- The TextAlign property if used to align text within a Button that is of type ContentAlignment enumeration.
- The Font property is used to set font of a Button.
- The following code snippet sets Text and Font properties of a Button control.

```
dynamicButton.Text = "I am Dynamic Button";
dynamicButton.TextAlign = ContentAlignment.MiddleLeft;
dynamicButton.Font = new Font "Georgia", 16);
```

✓ Adding Button Click Event Hander

- A Button control is used to process the button click event.
- We can attach a button click event handler at run-time by setting its Click event to an EventHandler object.
- The EventHandler takes a parameter of an event handler.

```
// Add a Button Click Event handler
dynamicButton.Click += new EventHandler(DynamicButton_Click);
```

- The signature of Button click event handler is listed in the following code snippet.

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

Creating a Button Dynamically

- Creating a Button control at run-time is merely a work of
  - creating an instance of Button class,
  - set its properties and
  - add Button class to the Form controls.

```
private void CreateDynamicButton()
{
    // Create a Button object
    Button dynamicButton = new Button();
    // Set Button properties
    dynamicButton.Height =40;
    dynamicButton.Width = 300;
    dynamicButton.BackColor = Color.Red;
    dynamicButton.ForeColor = Color.Blue;
    dynamicButton.Location = new Point(10, 10);
    dynamicButton.Text = "I am Dynamic Button";
    dynamicButton.Name = "DynamicButton";
    dynamicButton.Font = new Font("Georgia", 16);
    // Add a Button Click Event handler
    dynamicButton.Click += new EventHandler(dynamicButton_Click);
    // Add Button to the Form. Placement of the Button
    // will be based on the Location and Size of button
    Controls.Add(dynamicButton);
}
private void dynamicButton_Click(object sender, EventArgs e)
{
    MessageBox.Show("Dynamic button is clicked");
}
```

Label Control

- A Label control is used as a display medium for text on Forms.
- Label control does not participate in user input or capture mouse or keyboard events.
- So it only displays text that users cannot directly edit.

Creating a Label

- There are two ways to create a control.
  - ✓ **Design-time**
    - First, we can use the Form designer of Visual Studio to create a control at design-time.
    - In design-time mode, we can use visual user interfaces to create a control properties and write methods.
    - To create a Label control at design-time, you simply drag and drop a Label control from Toolbox to a Form.
    - Once a Label is on the Form, you can move it around and resize it using mouse and set its properties and events.

Run-time

- ✓ Label class represents a Label control.
- ✓ We simply create an instance of Label class, set its properties and add this in to the Form controls.
- ✓ In the first step, we create an instance of the Label class. `Label dynamicLabel = new Label();`
- ✓ In the next step, we set properties of a Label control.

```
dynamicLabel.BackColor = Color.Red;
dynamicLabel.ForeColor = Color.Blue;
dynamicLabel.Text = "I am a Dynamic Label";
dynamicLabel.Name = "DynamicLabel";
dynamicLabel.Font = new Font("Georgia", 16);
```

- ✓ In the last step, we need to add a Label control to the Form by calling Form.Controls.Add method.

```
this.Controls.Add(dynamicLabel);
```

Setting Label Properties

- After you place a Label control on a Form, the next step is to set properties.
- The easiest way to set properties is from the Properties Window.
- You can open Properties window by pressing F4 or right click on a control and select Properties menu item.
- **Name**: represents a unique name of a Label control.
  - ✓ It is used to access the control in the code.

```
dynamicLabel.Name = "DynamicLabel";
string name = dynamicLabel.Name;
```

Location, Height, Width, and Size

- ✓ The Location property takes a Point that specifies the starting position of the Label on a Form.
- ✓ The Size property specifies the size of the control.
- ✓ We can also use Width and Height property instead of Size property.

```
dynamicLabel.Location = new Point (20, 150);
dynamicLabel.Height = 40;
dynamicLabel.Width = 300;
```

Background, Foreground, BorderStyle

- BackColor and ForeColor properties are used to set background and foreground color of a Label respectively.
- If you click on these properties in Properties window, the Color Dialog pops up.
- Alternatively, you can set background and foreground colors at run-time.

```
dynamicLabel.BackColor = Color.Red;
dynamicLabel.ForeColor = Color.Blue;
```

- The BorderStyle property has three values FixedSingle, Fixed3D, and None.
- The default value of border style is Fixed3D.

```
dynamicLabel.BorderStyle = BorderStyle.FixedSingle;
```

- **Font**
  - ✓ Font property represents the font of text of a Label control.
  - ✓ If you click on the Font property in Properties window, you will see Font name, size and other font options.

```
dynamicLabel.Font = new Font("Georgia", 16);
```

- **Text and TextAlign, and TextLength**
  - ✓ Text property of a Label represents the current text of a Label control.
  - ✓ The TextAlign property represents text alignment that can be Left, Center, or Right.
  - ✓ The TextLength property returns the length of a Label contents.

```
dynamicLabel.Text = "I am Dynamic Label";
dynamicLabel.TextAlign  = HorizontalAlignment.Center;
int size = dynamicLabel.TextLength;
```

Append Text

- We can append text to a Label by simply setting Text property to current text plus new text you would want to append something like this.

```
dynamicLabel.Text += " Appended text";
```

AutoEllipsis

- An ellipsis character (...) is used to give an impression that a control has more characters but it could not fit in the current width of the control.
- If AutoEllipsis property is true, it adds ellipsis character to a control if text in control does not fit.
- You may have to set AutoSize to false to see the ellipses character.

Image in Label

- The Image property of a Label control is used to set a label background as an image.
- The Image property needs an Image object.
- The Image class has a static method called FromFile that takes an image file name with full path and creates an Image object.
- You can also align image and text.
- The ImageAlign and TextAlign properties of Button are used for this purpose.
- The following C# code snippet sets an image as a Label background.

```
dynamicLabel.Image = Image.FromFile(@"C:\Images\Dock.jpg");
dynamicLabel.ImageAlign = ContentAlignment.MiddleRight;
dynamicLabel.TextAlign = ContentAlignment.MiddleLeft;
dynamicLabel.FlatStyle = FlatStyle.Flat;
```

LinkLabel control

- A LinkLabel control is a label control that can display a hyperlink.
- A LinkLabel control is inherited from the Label class so it has all the functionality provided by the Windows Forms Label control.
- LinkLabel control does not participate in user input or capture mouse or keyboard events.
  - Do all what you have did in **Label control** *except* change the class from **Label** to **LinkLabel** and use the following special property …

Hyperlink Properties

- Here are the hyperlink related properties available in the LinkLabel control.
  - **Links and LinkArea**
    - A LinkLabel control can display more than one hyperlink.
    - The Links property a type of *LinkCollection* represents all the hyperlinks available in a LinkLabel control.
    - The Add method of LinkColleciton is used to add a link to the collection.
    - The Remove and RemoveAt methods are used to remove a link from the LinkCollection.
    - The Clear method is used to remove all links from a LinkCollection.
    - LinkArea property represents the range of text that is treated as a part of the link.
    - It takes a starting position and length of the text.

...

- The following code snippet adds a link and sets LinkArea and a link click event handler.

```
dynamicLinkLabel.LinkArea = new LinkArea(0, 22);
dynamicLinkLabel.Links.Add(24, 9, "http://www.c-sharpcorner.com");
dynamicLinkLabel.LinkClicked += new LinkLabelLinkClickedEventHandler
    (LinkedLabel_Clicked);
```

- Here is the code for the LinkLabel click event handler and uses Process.
- Start method to open a hyperlink in a browser.

```
private void LinkedLabel_Clicked (object sender, LinkLabelLinkClickedEventArgs e)
{
   dynamicLinkLabel.LinkVisited = true;
   System.Diagnostics.Process.Start("http://www.c-sharpcorner.com");
}
```

LinkColor, VisitedLinkColor, ActiveLinkColor and DisabledLinkColor

- LinkColor, VisitedLinkColor, ActiveLinkColor and DisabledLinkColor properties represent colors when a hyperlink is in normal, visited, active, or disabled mode.
- The following code snippet sets these colors.

```
dynamicLinkLabel.ActiveLinkColor = Color.Orange;
dynamicLinkLabel.VisitedLinkColor = Color.Green;
dynamicLinkLabel.LinkColor = Color.RoyalBlue;
dynamicLinkLabel.DisabledLinkColor = Color.Gray;
```

TextBox Control

- A TextBox control accepts user input on a Form.
- It displays text entered at design time that can be edited by users at run time, or changed programmatically.
- **Creating a TextBox**
  - ✓ We can create a TextBox control using a Forms designer at design-time or using the TextBox class in code at run-time (also known as dynamically).
  - ✓ To create a TextBox control at design-time, you simply drag and drop a TextBox control from Toolbox to a Form in Visual Studio.
  - ✓ Once a TextBox is on the Form, you can move it around and resize it using mouse and set its properties and events.

...

- Creating a TextBox control at **run-time** is merely a work of creating an instance of TextBox class, set its properties and add TextBox class to the Form controls.

```
// Create a TextBox object
TextBox dynamicTextBox = new TextBox();
// Set background and foreground
dynamicTextBox.BackColor = Color.Red;
dynamicTextBox.ForeColor = Color.Blue;
dynamicTextBox.Text = "I am Dynamic TextBox";
dynamicTextBox.Name = "DynamicTextBox";
dynamicTextBox.Font = new Font("Georgia", 16);
//add the TextBox control to the Form
Controls.Add( dynamicTextBox );
```

Setting TextBox Properties

Location, Height, Width, and Size

- ✓ The Location property takes a Point that specifies the starting position of the TextBox on a Form.
- ✓ The Size property specifies the size of the control.
- ✓ We can also use Width and Height property instead of Size property.
- ✓ The following code snippet sets Location, Width, and Height properties of a TextBox control.

```
// Set TextBox properties
dynamicTextBox.Location = new Point(20, 150);
dynamicTextBox.Height = 40;
dynamicTextBox.Width = 300;
```

Multiline TextBox

- By default, a TextBox control accepts input in a single line only.
- To make it multi-line, you need to set Multiline property to true.
- By default, the Multiline property is false.
- When you drag and drop a TextBox control from Toolbox to a Form, you cannot change the height of a TextBox control.
- But if you select a TextBox control and click on Tasks handle and check MultiLine CheckBox,
  - you will see height *resizing grip handles* are available on a TextBox and you can resize the height of a control.
- You can do this dynamically by setting Multiline property to true.

```
// Make TextBox multiline
dynamicTextBox.Multiline = true;
```

Background, Foreground, BorderStyle

- BackColor and ForeColor properties are used to set background and foreground color of a TextBox respectively.
- Alternatively, you can set background and foreground colors at run-time.

```
// Set background and foreground
dynamicTextBox.BackColor = Color.Red;
dynamicTextBox.ForeColor = Color.Blue;
```

- The BorderStyle property has three values FixedSingle, Fixed3D, and None.
- The default value of border style is Fixed3D.
- The following code snippet sets the border style of a TextBox to FixedSingle.

```
dynamicTextBox.BorderStyle = BorderStyle.FixedSingle;
```

Name

- ✓ Name property represents a unique name of a TextBox control. It is used to access the control in the code.

```
dynamicTextBox.Name = "DynamicTextBox";
string name = dynamicTextBox.Name;
```

Text, TextAlign, and TextLength

- ✓ **Text** property of a TextBox represents the current text of a TextBox control.
- ✓ The **TextAlign** property represents text alignment that can be Left, Center, or Right.
- ✓ The **TextLength** property returns the length of a TextBox contents.

```
dynamicTextBox.Text = "I am Dynamic TextBox";
dynamicTextBox.TextAlign  = HorizontalAlignment.Center;
int size = dynamicTextBox.TextLength;
```

Append Text

- One way to append text to a TextBox is simply set Text property to current text plus new text you would want to append something like this.

```
textBox1.Text += " Appended text";
```

- TextBox also has the **AppendText** method to do the same.
- The **AppendText** method appends text at the end of a TextBox.
- The following code snippet uses AppendText method to append text to the textBox1 contents.

```
textBox1.AppendText(" Appended text");
```

AcceptsReturn and AcceptsTab

- In a Multiline TextBox control, you need to press CTRL+ENTER to create a new line.
- The AcceptsReturn property sets TextBox control to move to new line by simply pressing ENTER key.
- By default, AcceptsReturn property of a TextBox control is false.

```
// accepts ENTER key
dynamicTextBox.AcceptsReturn = true
```

- If a TextBox control is set to multiline, the AcceptsTab property is used to set so the TextBox control accepts TAB key.
- If this property is not set, pressing TAB key simply move to the next control on a Form.
- By default, AcceptsTab property value of a TextBox control is false.

```
// accepts TAB key
dynamicTextBox.AcceptsTab = true;
```

WordWrap

- If WordWrap property is true, the text in the TextBox control automatically wraps to the next line if required.
- If this property is set to true, horizontal scroll bars are not displayed regardless of the ScrollBars property setting.

```
// Wordwrap
dynamicTextBox.WordWrap = true;
```

Font

- Font property represents the font of text of a TextBox control.
- If you click on the Font property in Properties window, you will see Font name, size and other font options.

```
dynamicTextBox.Font = new Font("Georgia", 16);
```

ScrollBars

- ✓ A Multiline TextBox control can have scrollbars.
- ✓ The ScrollBars property of TextBox control is used to show scrollbars on a control.
- ✓ The ScrollBars property is represented by a ScrollBars enumeration that has four values Both, Vertical, Horizontal, and None.
- ✓ The following code snippet makes both vertical and horizontal scrollbars active on a TextBox control and they will be visible when the scrolling is needed on a TextBox control.

```
dynamicTextBox.ScrollBars = ScrollBars.Both;
```

Password Character and Character Casing

- PasswordChar property is used to apply masking on a TextBox when you need to use it for a password input and do now what password to be readable.
  - For example, you can place a star (*) for password characters.
- The following code snippet sets a dollar ($) symbol as any character entered in a TextBox.

```
dynamicTextBox.PasswordChar = '$';
```

- UseSystemPasswordChar property is used to full default system password If the UseSystemPasswordChar is set to true,
  - the default system password character is used and any character set by **PasswordChar** is ignored.
- CharacterCasing property of TextBox sets the case of text in a TextBox, It has three values Upper, Lower, and Normal.

```
dynamicTextBox.CharacterCasing = CharacterCasing.Upper;
```

Read TextBox Contents

- The simplest way of reading a TextBox control contents is using the Text property.
- The following code snippet reads contents of a TextBox in a string.

```
string textBoxContents = dynamicTextBox.Text;
```

- In a multiline TextBox,
  - if the TextBox contents are separated by multiple lines and you want to read contents of a TextBox line by line, you can use the Lines property of the TextBox.
- The Lines property returns an array of strings where each element of the returned array is a line.
- The following code snippet reads a TextBox contents line by line.

```
string [] textBoxLines = dynamicTextBox.Lines;
foreach (string line in textBoxLines)
{
    MessageBox.Show(line);
}
```

Maximum Length

- You can restrict number of characters in a TextBox control by setting **MaxLength** property.
- The following code snippet sets the maximum length of a TextBox to 50 characters.

```
dynamicTextBox.MaxLength = 50;
```

ReadOnly

- You can make a TextBox control read-only (non-editable) by setting the ReadOnly property to true.
- The following code snippet sets the ReadOnly property to true.

```
dynamicTextBox.ReadOnly = true;
```

- **Enabling and Disabling Shortcuts**
  - ✓ **ShortcutsEnabled** property of the TextBox is used to enable or disable shortcuts.
  - ✓ By default, shortcuts are enabled.
  - ✓ The following code snippet disables shortcuts in a TextBox.

```
dynamicTextBox.ShortcutsEnabled = false;
```

- ShortcutsEnabled property applies to the following shortcut key combinations:
  - ✓ CTRL+Z
  - ✓ CTRL+E
  - ✓ CTRL+C
  - ✓ CTRL+Y
  - ✓ CTRL+X
  - ✓ CTRL+BACKSPACE
  - ✓ CTRL+V
  - ✓ CTRL+DELETE
  - ✓ CTRL+A
  - ✓ SHIFT+DELETE
  - ✓ CTRL+L
  - ✓ SHIFT+INSERT
  - ✓ CTRL+R

Selection in TextBox

- **SelectedText** property returns the selected text in a TextBox control.

```
string selectedText = dynamicTextBox.SelectedText;
```

- You may also use **SelectionStart** and **SelectionLength** properties to get and set the selected text in a TextBox.
- The SelectionStart property represents the starting index of the selected text and
- SelectionLength property represents the number of characters to be selected after the starting character.
- The following code snippet sets the selection on a TextBox.

```
dynamicTextBox.SelectionStart = 10;
dynamicTextBox.SelectionLength = 20;
```

Clear, SelectAll and DeselectAll

- Clear method removes the contents of a TextBox. The following code snippet uses Clear method to clear the contents of a TextBox.

```
textBox1.Clear();
```

- TextBox class provides SelectAll and DeselectAll methods to select and deselect all text of a TextBox control.
- The following code snippet shows how to use SelectAll and DeselectAll methods.

```
private void selectAllToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (textBox1.TextLength > 0)
        textBox1.SelectAll();
}
private void deselectAllToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (textBox1.TextLength > 0)
        textBox1.DeselectAll();
}
```

Cut, Copy, Paste, Undo Operations in TextBox

- TextBox class provides *Cut*, *Copy*, *Paste*, and *Undo* methods to cut, copy, paste, and undo clipboard operations.
- The following code snippet shows how to use Cut, Copy, Paste, and Undo methods.

```
private void cutToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (textBox1.SelectionLength > 0)
        textBox1.Cut();
}
private void copyToolStripMenuItem_Click(object sender, EventArgs e)
{
        if (textBox1.SelectionLength > 0)
            textBox1.Copy();
}
```

...

```
private void pasteToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (Clipboard.GetDataObject().GetDataPresent( DataFormats.Text ))
    {
        textBox1.Paste();
    }
}
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (textBox1.CanUndo)
    {
        textBox1.Undo();
        textBox1.ClearUndo();
    }
}
```

RichTextBox Control

- A RichTextBox control is an advanced text box that provides text editing and advanced formatting features including loading rich text format (RTF) files.
- **Creating a RichTextBox**
  - ✓ We can create a RichTextBox control using a Forms designer at design-time or using the RichTextBox class in code at run-time.
  - ✓ To create a RichTextBox control at design-time, you simply drag and drop a RichTextBox control from the Toolbox onto a Form in Visual Studio.
  - ✓ Once a RichTextBox is added to a Form, you can move it around and resize it using the mouse and set it's properties and events.

...

- Creating a RichTextBox control at run-time is merely a work of
  - ✓ creating an instance of RichTextBox class,
  - ✓ setting it's properties and
  - ✓ adding the RichTextBox object to the Form's Controls collection.
- The first step to create a dynamic RichTextBox is to create an instance of the RichTextBox class.
  - ✓ The following code snippet creates a RichTextBox control object.

```
// Create a RichTextBox object
RichTextBox dynamicRichTextBox = new RichTextBox();
```

...

- In the next step, you may set properties of a RichTextBox control.
  - ✓ The following code snippet sets size, location, background color, foreground color, Text, Name, and Font properties of a RichTextBox.

```
dynamicRichTextBox.Location = new Point(20, 20);
dynamicRichTextBox.Width = 300;
dynamicRichTextBox.Height = 200;
// Set background and foreground
dynamicRichTextBox.BackColor = Color.Red;
dynamicRichTextBox.ForeColor = Color.Blue;
dynamicRichTextBox.Text = "I am Dynamic RichTextBox";
dynamicRichTextBox.Name = "DynamicRichTextBox";
dynamicRichTextBox.Font = new Font("Georgia", 16);
```

...

- Once a RichTextBox control is ready with its properties, the next step is to add the RichTextBox control to the Form.
- To do so, we use *Form.Controls.Add* method.
- The following code snippet adds a RichTextBox control to the current Form.

```
Controls.Add(dynamicRichTextBox);
```

Setting RichTextBox Properties

Location, Height, Width, and Size

- The Location property takes a Point that specifies the starting position of the RichTextBox on a Form.
- The Size property specifies the size of the control.
- We can also use Width and Height property instead of Size property.
- The following code snippet sets Location, Width, and Height properties of a RichTextBox control.

```
dynamicRichTextBox.Location = new Point(20, 20);
dynamicRichTextBox.Width = 300;
dynamicRichTextBox.Height = 200;
```

✓ Background, Foreground, BorderStyle

- BackColor and ForeColor properties are used to set the background and foreground color of a RichTextBox respectively.
- Alternatively, you can set background and foreground colors at run-time.
- The following code snippet sets BackColor and ForeColor properties.

```
// Set background and foreground
dynamicRichTextBox.BackColor = Color.Red;
dynamicRichTextBox.ForeColor = Color.Blue;
```

- The BorderStyle has three values FixedSingle, Fixed3D, and None.
- The default value of border style is Fixed3D.
- The following code snippet sets the border style of a RichTextBox to FixedSingle.

```
dynamicRichTextBox.BorderStyle = BorderStyle.FixedSingle;
```

- **Name**
  - ✓ The Name property represents a unique name of a RichTextBox control. It is used to access the control in the code.
  - ✓ The following code snippet sets and gets the name and text of a RichTextBox control.
  - ✓ dynamicRichTextBox.Name = "DynamicRichTextBox";
- **Text and TextLength**
  - ✓ The Text property of a RichTextBox represents the current text of a RichTextBox control.
  - ✓ The TextLength property returns the length of a RichTextBox contents.
  - ✓ The following code snippet sets the Text and TextAlign properties and gets the size of a RichTextBox control.

```
dynamicRichTextBox.Text = "I am Dynamic RichTextBox";
int size = dynamicRichTextBox.TextLength;
```

Append Text

- One way to append text to a RichTextBox is simply set Text property to current text plus new text you would want to append something like this.

```
RichTextBox1.Text += " Appended text";
```

- RichTextBox also has the AppendText method to do the same.
- The *AppendText* method appends text at the end of a RichTextBox.

```
RichTextBox1.AppendText(" Appended text");
```

AcceptsTab

- If a RichTextBox control is set to multiline, the AcceptsTab property is used to set the RichTextBox control to accept the TAB key as text.
- If this property is not set, pressing the TAB key simply moves to the next control on the Form.
- By default, the AcceptsTab property value of a RichTextBox control is false.

```
// accepts TAB key
dynamicRichTextBox.AcceptsTab = true;
```

WordWrap

- If WordWrap property is true, the text in the RichTextBox control automatically wraps to the next line if required.
- If this property is set to true, horizontal scroll bars are not displayed regardless of the ScrollBars property setting.

```
// Wordwrap
dynamicRichTextBox.WordWrap = true;
```

ScrollBars

- A Multiline RichTextBox control can have scrollbars.
- The ScrollBars property of RichTextBox control is used to show scrollbars on a control.
- The ScrollBars property is represented by a RichTextBoxScrollBars enumeration that has four values Both, Vertical, Horizontal, and None.
  dynamicRichTextBox.ScrollBars = RichTextBoxScrollBars.Both;

Font

- ✓ Font property represents the font of text of a RichTextBox control.
- ✓ The following code snippet sets Font property at run-time.

```
dynamicRichTextBox.Font = new Font("Georgia", 16);
```

Maximum Length

- ✓ You can restrict the number of characters in a RichTextBox control by setting MaxLength property.
- ✓ The following code snippet sets the maximum length of a RichTextBox to 50 characters. dynamicRichTextBox.MaxLength = 50;

ReadOnly

- ✓ You can make a RichTextBox control read-only (non-editable) by setting the ReadOnly property to true.
- ✓ The following code snippet sets the ReadOnly property to true.

```
dynamicRichTextBox.ReadOnly = true;
```

- **Enabling and Disabling Shortcuts**
  - ✓ **ShortcutsEnabled**
    - property of the RichTextBox is used to enable or disable shortcuts.
  - ✓ By default, shortcuts are enabled.
  - ✓ The following code snippet disables shortcuts in a RichTextBox.

```
dynamicRichTextBox. ShortcutsEnabled = false;
```

- ShortcutsEnabled property applies to the following shortcut key combinations:
  - ✓ CTRL+Z
  - ✓ CTRL+E
  - ✓ CTRL+C
  - ✓ CTRL+Y
  - ✓ CTRL+X
  - ✓ CTRL+BACKSPACE
  - ✓ CTRL+V
  - ✓ CTRL+DELETE
  - ✓ CTRL+A
  - ✓ SHIFT+DELETE
  - ✓ CTRL+L
  - ✓ SHIFT+INSERT
  - ✓ CTRL+R

Read RichTextBox Contents

- The simplest way of reading a RichTextBox control contents is using the **Text** property.
- Note however that the Text property has no formatting; it has only text.
- See the Rtf property for the text including the formatting.
- The following code snippet reads contents of a RichTextBox in a string.

```
string RichTextBoxContents = dynamicRichTextBox.Text;
```

- In a multiline RichTextBox,
  - if the RichTextBox contents are separated by multiple lines and
  - you want to read contents of a RichTextBox line by line,
  - you can use the **Lines** property of the RichTextBox.

...

- ✓ The Lines property returns an array of strings where each element of the returned array is a line.
- ✓ The following code snippet reads a RichTextBox contents line by line.

```
string [] RichTextBoxLines = dynamicRichTextBox.Lines;
foreach (string line in RichTextBoxLines)
{
    MessageBox.Show(line);
}
```

Clear, SelectAll and DeselectAll

- The Clear method removes the contents of a RichTextBox.
- The following code snippet uses Clear method to clear the contents of a RichTextBox.

```
RichTextBox1.Clear();
```

- RichTextBox class provides SelectAll and DeselectAll methods to select and deselect all text of a RichTextBox control.
- The following code snippet shows how to use SelectAll and DeselectAll methods.

```
private void selectAllToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (RichTextBox1.TextLength > 0)
        RichTextBox1.SelectAll();
}
private void deselectAllToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (RichTextBox1.TextLength > 0)
        RichTextBox1.DeselectAll();
}
```

Cut, Copy, Paste, Undo Operations in RichTextBox

- RichTextBox class provides Cut, Copy, Paste, and Undo methods to cut, copy, paste, and undo clipboard operations.
- The following code snippet shows how to use Cut, Copy, Paste, and Undo methods.

```
private void cutToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (RichTextBox1.SelectionLength > 0)
        RichTextBox1.Cut();
}
private void copyToolStripMenuItem_Click(object sender, EventArgs e)
{
        if (RichTextBox1.SelectionLength > 0)
            RichTextBox1.Copy();
}
```

...

```
private void pasteToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (Clipboard.GetDataObject().GetDataPresent( DataFormats.Text ))
    {
        RichTextBox1.Paste();
    }
}
private void undoToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (RichTextBox1.CanUndo)
    {
        RichTextBox1.Undo();
        RichTextBox1.ClearUndo();
    }
}
```

Load and Save RTF Files

- **LoadFile** method of RichTextBox control is used to load an RTF file and displays its contents.
- **SaveFile** method is used to save the contents of a RichTextBox to an RTF file.
- The following code snippet loads an RTF file using an OpenFileDialog and saves back its contents.

```
private void LoadRTFButton_Click (object sender, EventArgs e)
{
   OpenFileDialog ofd = new OpenFileDialog();
   ofd.InitialDirectory = "c:\\";
   ofd.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
   ofd.FilterIndex = 2;
   ofd.RestoreDirectory = true;
   if (ofd.ShowDialog() == System.Windows.Forms.DialogResult.OK) {
      dynamicRichTextBox.LoadFile ( ofd.FileName );
      dynamicRichTextBox.Find ("Text", RichTextBoxFinds.MatchCase );
      dynamicRichTextBox.SelectionFont = new Font("Verdana", 12, FontStyle.Bold);
      dynamicRichTextBox.SelectionColor = Color.Red;
      dynamicRichTextBox.SaveFile(@"C:\Data\SavedRTF.rtf",  RichTextBoxStreamType.RichText);
   }
}
```

- **BulletIndent**
  - **BulletIndent** property gets or sets the indentation used in the RichTextBox control when the bullet style is applied to the text.
    dynamicRichTextBox.BulletIndent = 10;
- **Redo and CanRedo**
  - Redo method can be used to reapply the last undo operation to the control.
  - CanRedo property represents whether there are actions that have occurred within the RichTextBox that can be reapplied.

```
if (dynamicRichTextBox.CanRedo == true)
{
    if (dynamicRichTextBox.RedoActionName != "Delete")
        dynamicRichTextBox.Redo();
}
```

...

- **DetectUrls**
  - ✓ If set true, the DetectUrls property will automatically format a Uniform Resource Locator (URL) when it is typed into the control.
- **EnableAutoDragDrop**
  - ✓ RichTextBox control supports drag and drop operations that allow us to drag and drop text, picture, and other data.
  - ✓ EnableAutoDragDrop property enables drag-and-drop operations on text, pictures, and other data.

    ```
    dynamicRichTextBox.EnableAutoDragDrop = true;
    ```
- **Rtf and SelectedRtf**
  - ✓ Rtf property is used to get and set rich text format (RTF) text in a RichTextBox control.
  - ✓ SelectedRtf property is used to get and set selected text in a control.
  - ✓ RTF text is the text that includes formatting.

- **RightMargin, AutoWordSelection, and ZoomFactor**
  - ✓ *RightMargin* property represents the size of a single line of text within a RichTextBox control.
  - ✓ *AutoWordSelection* property represents if a word is automatically selected when a text is double clicked within a RichTextBox control.
  - ✓ ZoomFactor represents the current zoom level of the RichTextBox.
  - ✓ Value 1.0 means there is no zoom applied on a control.

```
private void ZoomButton_Click ( object sender, EventArgs e)
{
    dynamicRichTextBox.AutoWordSelection = true;
    dynamicRichTextBox.RightMargin = 5;
    dynamicRichTextBox.ZoomFactor = 3.0f;
}
```

CheckBox Control

- A CheckBox control allows users to select a single or multiple options from a list of options.
- **Creating a CheckBox**
  - ✓ We can create a CheckBox control using a Forms designer at **design-time** or using the CheckBox class in code at **run-time** (also known as *dynamically*).
  - ✓ To create a CheckBox control at design-time, you simply drag and drop a CheckBox control from Toolbox to a Form in Visual Studio.
  - ✓ Once a CheckBox is on the Form, you can move it around and resize it using mouse and set its properties and events.
  - ✓ Creating a CheckBox control at **run-time** is merely a work of creating an instance of CheckBox class, set its properties and add CheckBox class to the Form controls.

...

- First step to create a dynamic CheckBox is to create an instance of CheckBox class.

```
// Create a CheckBox object
CheckBox dynamicCheckBox = new CheckBox();
```

- In the next step, you may set properties of a CheckBox control.

```
dynamicCheckBox.Left = 20;
dynamicCheckBox.Top = 20;
dynamicCheckBox.Width = 300;
dynamicCheckBox.Height = 30;
// Set background and foreground
dynamicCheckBox.BackColor = Color.Orange;
dynamicCheckBox.ForeColor = Color.Black;
dynamicCheckBox.Text = "I am a Dynamic CheckBox";
dynamicCheckBox.Name = "DynamicCheckBox";
dynamicCheckBox.Font = new Font("Georgia", 12);
```

- Next step is to add the CheckBox control to the Form.

```
Controls.Add(dynamicCheckBox);
```

Setting CheckBox Properties

- After you place a CheckBox control on a Form, the next step is to set properties.
- **Location, Height, Width, and Size**

```
// Set CheckBox properties
dynamicCheckBox.Location = new Point (20, 150);
dynamicCheckBox.Height = 40;
dynamicCheckBox.Width = 300;
```

- **Background, Foreground, BorderStyle**

```
// Set background and foreground
dynamicCheckBox.BackColor = Color.Red;
dynamicCheckBox.ForeColor = Color.Blue;
```

- **Name**
  - ✓ Name property represents a unique name of a CheckBox control. It is used to access the control in the code.

```
dynamicCheckBox.Name = "DynamicCheckBox";
string name = dynamicCheckBox.Name;
```

- **Text and TextAlign**

  dynamicCheckBox.Text = "I am a Dynamic CheckBox";

  dynamicCheckBox.TextAlign = ContentAlignment.MiddleCenter;

- **Check Mark Alignment**
  - ✓ CheckAlign property is used to align the check mark in a CheckBox.
  - ✓ By using CheckAlign and TextAlign properties, we can place text and check mark to any position on a CheckBox we want.

```
dynamicCheckBox.CheckAlign = ContentAlignment.MiddleCenter;
dynamicCheckBox.TextAlign = ContentAlignment.TopRight;
```

- **Font**

```
dynamicCheckBox.Font = new Font("Georgia", 16);
```

- **Read CheckBox Contents**
  - ✓ The simplest way of reading a CheckBox control contents is using the Text property.

```
string CheckBoxContents = dynamicCheckBox.Text;
```

- **Appearance**
  - ✓ Appearance property of CheckBox can be used to set the appearance of a CheckBox to a Button or a CheckBox.

```
dynamicCheckBox.Appearance = Appearance.Button;
```

- **AutoEllipsis**
  - ✓ An ellipsis character (...) is used to give an impression that a control has more characters but it could not fit in the current width of the control.
  - ✓ If AutoEllipsis property is true, it adds ellipsis character to a control if text in control does not fit.
  - ✓ You may have to set AutoSize to false to see the ellipses character.

```
dynamicCheckBox.AutoEllipsis = true;
```

Image in CheckBox

- ✓ The Image property of a CheckBox control is used to set the background as an image.
- ✓ The Image property needs an Image object.
- ✓ The Image class has a static method called FromFile that takes an image file name with full path and creates an Image object.
- ✓ You can also align image and text.
- ✓ The ImageAlign and TextAlign properties of CheckBox are used for this purpose.
- ✓ The following code snippet sets an image as a CheckBox background.

```
// Assign an image to the CheckBox.
dynamicCheckBox.Image = Image.FromFile(@"C:\Images\Dock.jpg");
// Align the image and text on the CheckBox.
dynamicCheckBox.ImageAlign = ContentAlignment.MiddleRight;
// Give the CheckBox a flat appearance.
dynamicCheckBox.FlatStyle = FlatStyle.Flat;
```

CheckBox States

- A typical CheckBox control has two possible states Checked and Unchecked.
- Checked state is when the CheckBox has check mark on and Unchecked is when the CheckBox is not checked.
- Checked property is true when a CheckBox is in checked state.

  dynamicCheckBox.Checked = true;
- CheckState property represents the state of a CheckBox.
- It can be checked or unchecked.
- Usually, we check if a CheckBox is checked or not and decide to take an action on that state something

```
if (dynamicCheckBox.Checked) {  // Do something when CheckBox is checked }
else {  // Do something here when CheckBox is not checked  }
```

...

- ✓ ThreeState is a new property added to the CheckBox in latest versions of Windows Forms.
- ✓ When this property is true, the CheckBox has three states - Checked, Unchecked, and Indeterminate.

```
dynamicCheckBox.CheckState = CheckState.Indeterminate;
if (dynamicCheckBox.CheckState == CheckState.Checked) { }
else if (dynamicCheckBox.CheckState == CheckState.Indeterminate) { }
else {}
```

...

- **AutoCheck** property represents whether the Checked or CheckState values and the CheckBox's appearance are automatically changed when the CheckBox is clicked.
- By default this property is true but if set to false.

```
dynamicCheckBox.AutoCheck = false;
```

- **CheckBox Checked Event Hander**
  - ✓ CheckedChanged and CheckStateChanged are two important events for a CheckBox control.
  - ✓ The CheckedChanged event occurs when the value of the Checked property changes.
  - ✓ The CheckStateChanged event occurs when the value of the CheckState property changes.
  - ✓ To add these event handlers, you go to Events window and double click on CheckedChanged and CheckedStateChanged events

...

- The following code snippet defines and implements these events and their respective event handlers.

```
dynamicCheckBox.CheckedChanged += new
System.EventHandler(CheckBoxCheckedChanged);
dynamicCheckBox.CheckStateChanged += new
System.EventHandler(CheckBoxCheckedChanged);
private void CheckBoxCheckedChanged( object sender, EventArgs e)
{
}
private void CheckBoxCheckedChanged( object sender, EventArgs e)
{
}
```

RadioButton Control

- A RadioButton control provides a round interface to select one option from a number of options.
- Radio buttons are usually placed in a group on a container control such as a Panel or a GroupBox and one of them is selected.

Creating a RadioButton

- We can create a RadioButton control using a Forms designer at design-time or using the RadioButton class in code at **run-time** (also known as **dynamically**).
- To create a RadioButton control at **design-time**, you simply drag and drop a RadioButton control from Toolbox to a Form in Visual Studio.
- Once a RadioButton is on the Form, you can move it around and resize it using mouse and set its properties and events.

...

- Creating a **RadioButton** control at run-time
  - ✓ First step to create a dynamic RadioButton is to create an instance of RadioButton class.

    ```
    RadioButton dynamicRadioButton = new RadioButton ();
    ```
  - ✓ In the next step, you may set properties of a RadioButton control.

    ```
    dynamicRadioButton.Left = 20;
    dynamicRadioButton.Top = 100;
    dynamicRadioButton.Width = 300;
    dynamicRadioButton.Height = 30;
    dynamicRadioButton.BackColor = Color.Orange;
    dynamicRadioButton.ForeColor = Color.Black;
    dynamicRadioButton.Text = "I am a Dynamic RadioButton";
    dynamicRadioButton.Name = "DynamicRadioButton";
    dynamicRadioButton.Font = new Font ("Georgia", 12);
    ```
  - ✓ Next step is to add the RadioButton control to the Form.

    ```
    Controls.Add( dynamicRadioButton );
    ```

Setting RadioButton Properties

- After you place a RadioButton control on a Form, the next step is to set properties.
- **Location, Height, Width, and Size**

```
dynamicRadioButton.Location = new Point (20, 150);
dynamicRadioButton.Height = 40;
dynamicRadioButton.Width = 300;
```

- **Background, Foreground, BorderStyle**

```
dynamicRadioButton.BackColor = Color.Red;
dynamicRadioButton.ForeColor = Color.Blue;
```

- **Name**

```
dynamicRadioButton.Name = "DynamicRadioButton";
string name = dynamicRadioButton.Name;
```

...

- **Text and TextAlign**

```
dynamicRadioButton.Text = "I am a Dynamic RadioButton";
dynamicRadioButton.TextAlign = ContentAlignment.MiddleCenter;
```

- **Font**

```
dynamicRadioButton.Font = new Font ("Georgia", 16);
```

- **Read RadioButton Contents**

```
string RadioButtonContents = dynamicRadioButton.Text;
```

- **Appearance**
  - ✓ Appearance property of RadioButton can be used to set the appearance of a RadioButton to a Button or a RadioButton.
  - ✓ The Button look does not have a round select option.

```
dynamicRadioButton.Appearance = Appearance.Button;
```

- **Check Mark Alignment**

```
dynamicRadioButton.CheckAlign = ContentAlignment.MiddleRight;
dynamicRadioButton.TextAlign = ContentAlignment.MiddleRight;
```

AutoEllipsis

- An ellipsis character (...) is used to give an impression that a control has more characters but it could not fit in the current width of the control.
- If AutoEllipsis property is true, it adds ellipsis character to a control if text in control does not fit.
- You may have to set AutoSize to false to see the ellipses character.

```
dynamicRadioButton.AutoEllipsis = true;
```

Image in RadioButton

```
dynamicRadioButton.Image = Image.FromFile(@"C:\Images\Dock.jpg");
dynamicRadioButton.ImageAlign = ContentAlignment.MiddleRight;
dynamicRadioButton.FlatStyle = FlatStyle.Flat;
```

- **RadioButton States**
  - ✓ Checked property is true when a RadioButton is in checked state.
    dynamicRadioButton.Checked = true;
  - ✓ Usually, we check if a RadioButton is checked or not and decide to take an action on that state something

```
if (dynamicRadioButton.Checked)
{   // Do something when RadioButton is checked}
else
{   // Do something here when RadioButton is not checked}
```

- **AutoCheck** property represents whether the Checked or CheckState values and the RadioButton's appearance are automatically changed when the RadioButton is clicked.
- By default this property is true but if set to false.

  dynamicRadioButton.AutoCheck = false;

...

- **RadioButton Checked Event Hander**
- CheckedChanged event occurs when the value of the Checked property changes.
- To add this event handler, you go to Events window and double click on CheckedChanged events
- You can write this code to implement CheckedChanged event dynamically.

```
dynamicRadioButton.CheckedChanged += new System.EventHandler (
    RadioButtonCheckedChanged );
private void RadioButtonCheckedChanged (object sender, EventArgs e)
{
}
```

ComboBox Control

- A ComboBox control is a combination of a TextBox and a ListBox control. Only one list item is displayed at one time in a ComboBox and other available items are loaded in a drop down list.
- **Creating a ComboBox**
  - ✓ We can create a ComboBox control using a Forms designer at design-time or using the ComboBox class in code at run-time.
  - ✓ To create a ComboBox control at design-time, you simply drag and drop a ComboBox control from Toolbox to a Form in Visual Studio.
  - ✓ Once a ComboBox is on the Form, you can move it around and resize it using mouse and set its properties and events.

...

- Creating a ComboBox control at run-time is merely a work of creating an instance of ComboBox class, set its properties and adds ComboBox class to the Form controls.
- First step to create a dynamic ComboBox is to create an instance of ComboBox class.

```
ComboBox comboBox1 = new ComboBox();
```

- In the next step, you may set properties of a ComboBox control.

```
comboBox1.Location = new System.Drawing.Point(20, 60);
comboBox1.Name = "comboBox1";
comboBox1.Size = new System.Drawing.Size (245, 25);
comboBox1.BackColor = System.Drawing.Color.Orange;
comboBox1.ForeColor = System.Drawing.Color.Black;
```

- Once the ComboBox control is ready with its properties, the next step is to add the ComboBox to a Form.

```
Controls.Add(comboBox1);
```

Setting ComboBox Properties

- After you place a ComboBox control on a Form, the next step is to set properties.
- **Name**

```
comboBox1.Name = "comboBox1";
```

- **Location, Height, Width and Size**

```
comboBox1.Location = new Point (12, 12);
comboBox1.Size = new Size (300, 25);
comboBox1.Width = 300;
comboBox1.Height = 25;
```

DropDownHeight and DropDownWidth

- ✓ You can control the size of the dropdown area of a ComboBox.
- ✓ The DropDownHeight and DropDownWidth properties represent the height and width of the dropdown area in pixel respectively.
- ✓ If the DropDownWidth and DropDownHeight properties are less than the Width and Height values, they will not be applicable.
- ✓ If all the items do not fit in the size of the dropdown area, the scrollbars will appear
- ✓ The following code snippet sets the height and width of the dropdown area of a ComboBox.

```
comboBox1.DropDownHeight = 50;
comboBox1.DropDownWidth = 300;
```

...

- **Font**

  comboBox1.Font = new Font("Georgia", 16);

- **Background and Foreground**

  comboBox1.BackColor = System.Drawing.Color.Orange;

  comboBox1.ForeColor = System.Drawing.Color.Black;

- **ComboBox Items**
  - ✓ The Items property is used to add and work with items in a ComboBox.
  - ✓ We can add items to a ComboBox at design-time from Properties Window by clicking on Items Collection

...

- When you click on the Collections, the String Collection Editor window will pop up where you can type strings.
- Each line added to this collection will become a ComboBox item.
- You can add same items at **run-time** by using the following code snippet.

```
comboBox1.Items.Add(“first item");
comboBox1.Items.Add(“second item");
comboBox1.Items.Add(“third item");
comboBox1.Items.Add(“fourth item");
```

Getting All Items

- To get all items, we use the Items property and loop through it to read all the items.

```
private void GetItemsButton_Click ( object sender, EventArgs e)
{
   StringBuilder sb = new StringBuilder ();
   foreach (string name in Combo1.Items)
   {
      sb.Append (name);
      sb.Append (" ");
   }
   MessageBox.Show ( sb.ToString() );
}
```

Selected Text and Item

- Text property is used to set and get text of a ComboBox.

```
comboBox1.Text = “here is the text value of the combobox";
MessageBox.Show(comboBox1.Text);
```

...

- We can also get text associated with currently selected item by using Items property.

```
string selectedItem  = comboBox1.Items[comboBox1.SelectedIndex].ToString();
```

- **Why the value of ComboBox.SelectedText is Empty?**
  - ✓ SelectedText property gets and sets the selected text in a ComboBox only when a ComboBox has focus on it.
  - ✓ If the focus moves away from a ComboBox, the value of SelectedText will be an empty string.
  - ✓ To get current text in a ComboBox when it does not have focus, use Text property.
- **DataSource**
  - ✓ DataSource property is used to get and set a data source to a ComboBox.
  - ✓ The data source can be a collection or object that implements IList interface such as an array, a collection, or a DataSet.
  - ✓ The following code snippet binds an enumeration converted to an array to a ComboBox.
  - ✓ comboBox1.DataSource = System.Enum.GetValues (typeof (ComboBoxStyle));

...

- **DropDownStyle**
  - ✓ DropDownStyle property is used to gets and sets the style of a ComboBox. It is a type of ComboBoxStyle enumeration.
  - ✓ The ComboBoxStyle enumeration has following three values.
    - > **Simple** - List is always visible and the text portion is editable.
    - > **DropDown** - List is displayed by clicking the down arrow and that the text portion is editable.
    - > **DropDownList** - List is displayed by clicking the down arrow and that the text portion is not editable.
  - ✓ The following code snippet sets the DropDownStyle property of a ComboBox to DropDownList. comboBox1.DropDownStyle = ComboBoxStyle.DropDownList;
- **DroppedDown**
  - ✓ If set true, the dropped down portion of the ComboBox is displayed.
  - ✓ By default, this value is false.
- **Sorting Items**
  - ✓ The Sorted property set to true, the ComboBox items are sorted.
  - ✓ The following code snippet sorts the ComboBox items.

    comboBox1.Sorted = true;

Find Items

- ✓ The FindString method is used to find a string or substring in a ComboBox.
- ✓ The following code snippet finds a string in a ComboBox and selects it if found.

```
private void FindButton_Click ( object sender, EventArgs e)
{
    int index  = comboBox1.FindString(textBox1.Text);
    if (index < 0)
    {
        MessageBox.Show("Item not found.");
        textBox1.Text = String.Empty;
    }
    else
    {
            comboBox1.SelectedIndex = index;
    }
}
```

- **ComboBox SelectedIndexChanged Event Hander**
  - ✓ CheckedChanged and CheckStateChanged are two important events for a ComboBox control.
  - ✓ The CheckedChanged event occurs when the value of the Checked property changes.
  - ✓ The CheckStateChanged event occurs when the value of the CheckState property changes.
- The following code snippet defines and implements these events and their respective event handlers.

```
comboBox1.SelectedIndexChanged += new System.EventHandler
    (ComboBox1_SelectedIndexChanged);
private void ComboBox1_SelectedIndexChanged( object sender,
System.EventArgs e)
{
    MessageBox.Show(comboBox1.Text);
}
```

ListBox Control

- A ListBox control provides an interface to display a list of items.
- Users can select one or more items from the list.
- A ListBox may be used to display multiple columns and these columns may have images and other controls.
- **Creating a ListBox**
  - ✓ There are two approaches to create a ListBox control in Windows Forms.
  - ✓ Either we can use the Forms designer to create a control at design-time or we can use the ListBox class to create a control at run-time.
- **Design-time**
  - ✓ To create a ListBox control at design-time, we simply drag a ListBox control from the Toolbox and drop it to a Form in Visual Studio.
  - ✓ Once a ListBox is on the Form, you can move it around and resize it using the mouse and set its properties and events.

Run-time

- The first step to create a dynamic ListBox is to create an instance of the ListBox class

```
ListBox listBox1 = new ListBox ();
```

- In the next step, you may set the properties of a ListBox control.

```
listBox1.Location = new System.Drawing.Point(12, 12);
listBox1.Name = "ListBox1";
listBox1.Size = new System.Drawing.Size(245, 200);
listBox1.BackColor = System.Drawing.Color.Orange;
listBox1.ForeColor = System.Drawing.Color.Black;
```

- Once the ListBox control is ready with its properties, the next step is to add the ListBox to a Form. to the current Form:

```
Controls.Add(listBox1);
```

Setting ListBox Properties

- **Name**

```
listBox1.Name = "ListBox1";
```

- **Location, Height, Width and Size**

```
listBox1.Location = new System.Drawing.Point(12, 12);
listBox1.Size = new System.Drawing.Size(245, 200);
```

- **Font**

```
listBox1.Font = new Font ("Georgia", 16);
```

Background, Foreground and BorderStyle

```
listBox1.BackColor = System.Drawing.Color.Orange;
listBox1.ForeColor = System.Drawing.Color.Black;
listBox1.BorderStyle = BorderStyle.FixedSingle;
```

✓ ListBox Items

- The **Items** property is used to add and work with items in a ListBox.
- You can add items both at **design-time** and **run-time**
- You can add the same items at run-time by using the following code snippet:

```
listBox1.Items.Add("item 1");
listBox1.Items.Add("item 2");
listBox1.Items.Add("item 3");
listBox1.Items.Add("item 4");
```

✓ Getting All Items

- To get all items, we use the Items property and loop through it to read all the items.

```
private void GetItemsButton_Click (object sender, EventArgs e) {
   System.Text.StringBuilder sb = new System.Text.StringBuilder();
   foreach (object item in listBox1.Items)
   {
      sb.Append (item.ToString());
      sb.Append(" ");
   }
   MessageBox.Show(sb.ToString());
}
```

- **Selected Text and Item**
  - ✓ The Text property is used to set and get text of a ListBox.
    MessageBox.Show(listBox1.Text);
  - ✓ We can also get text associated with the currently selected item using the Items property:
    string selectedItem = listBox1.Items[listBox1.SelectedIndex].ToString();
- **Why is the value of ListBox.SelectedText Empty?**
  The SelectedText property gets and sets the selected text in a ListBox only when a ListBox has focus on it.
- If the focus moves away from a ListBox then the value of SelectedText will be an empty string.
- To get the current text in a ListBox when it does not have focus, use the Text property.

Selection Mode and Selecting Items

- The SelectionMode property defines how items are selected in a ListBox.
- The SelectionMode value can be one of the following four SelectionMode enumeration values:
  - *None*: No item can be selected.
  - *One*: Only one item can be selected.
  - *MultiSimple*: Multiple items can be selected.
  - *MultiExtended*: Multiple items can be selected, and the user can use the SHIFT, CTRL, and arrow keys to make selections.
- To select an item in a ListBox, we can use the SetSelect method that takes an item index and a true or false value where the true value represents the item to be selected.
- The following code snippet sets a ListBox to allow multiple selection and selects the second and third items in the list:

```
listBox1.SelectionMode = SelectionMode.MultiSimple;
listBox1.SetSelected(1, true);
listBox1.SetSelected(2, true);
```

- We can clear all selected items by calling the ClearSelected method, as in:

```
listBox1.ClearSelected();
```

- **How to disable item selection in a ListBox?**
  Just set the SelectionMode property to None.
- **Sorting Items**
  - ✓ If the Sorted property is set to true then the ListBox items are sorted. The following code snippet sorts the ListBox items:
    listBox1.Sorted = true;
- **Find Items**

```
private void FindItemButton_Click (object sender, EventArgs e)
{
   listBox1.ClearSelected();
   int index = listBox1.FindString(textBox1.Text);
   if (index < 0)
   {
      MessageBox.Show("Item not found.");
      textBox1.Text = String.Empty;
   }
   else
   {
      listBox1.SelectedIndex = index;
   }
}
```

ListBox SelectedIndexChanged Event Hander

- ✓ The SelectedIndexChanged event is fired when the item selection is changed in a ListBox.
- ✓ You can add the event handler using the Properties Widow and selecting the Event icon and double-clicking on SelectedIndexChanged
- ✓ The following code snippet defines and implements these events and their respective event handlers.
- ✓ You can use this same code to implement an event at run-time.

```
listBox1.SelectedIndexChanged += new
    EventHandler(listBox1_SelectedIndexChanged);
private void listBox1_SelectedIndexChanged (object sender, System.EventArgs e)
{
   MessageBox.Show(listBox1.SelectedItem.ToString());
}
```

- **Data Binding**
  - ✓ The DataSource property is used to bind a collection of items to a ListBox.
  - ✓ The following code snippet is a simple data binding example where an ArrayList is bound to a ListBox:

```
private void DataBindingButton_Click (object sender, EventArgs e)
{
    ArrayList authors = new ArrayList();
    authors.Add("Mikiyo");
    authors.Add("Mora");
    authors.Add("Tesfish");
    authors.Add("Haftish");
    listBox1.Items.Clear();
    listBox1.DataSource = authors;
}
```

- If you are binding an object with multiple properties, you must specify which property you are displaying by using the DisplayMember property, as in:

```
listBox1.DataSource = GetData();
listBox1.DisplayMember = "Name";
```

PictureBox Control

- **PictureBox** control is used to display images in Windows Forms.
- **Creating a PictureBox**
  - ✓ PictureBox class represents a PictureBox control.
  - ✓ The following code snippet creates a PictureBox, sets its width and height and adds control to the Form by calling Controls.Add() method.

```
PictureBox imageControl = new PictureBox();
 imageControl.Width = 400;
 imageControl.Height = 400;
 Controls.Add (imageControl);
```

Display an Image

- Image property is used to set an image to be displayed in a PictureBox control.

```
private void DisplayImage()
{
    PictureBox imageControl = new PictureBox();
    imageControl.Width = 400;
    imageControl.Height = 400;
    Bitmap image = new Bitmap("C:\\Images\\Creek.jpg");
    imageControl.Dock = DockStyle.Fill;
    imageControl.Image = (Image)image;
    Controls.Add(imageControl);
}
```

SizeMode

- **SizeMode** property is used to position an image within a PictureBox.
- It can be Normal, StretchImage, AutoSize, CenterImage, and Zoom.
- The following code snippet sets SizeMode property of a PictureBox control.

```
imageControl.SizeMode = PictureBoxSizeMode.CenterImage;
```

ImageList Control

- An ImageList is a supporting control that is typically used by other controls, such as a ListView but is exposed as a component to developers.
- We can use this component in our applications when we are building our own controls such as a photo gallery or an image rotator control.
- **Creating an ImageList**
  - ✓ ImageList class represents the ImageList First step to create a dynamic ImageList is to create an instance of ImageList class.
  - ✓ The following code snippet creates an ImageList control object.

```
ImageList photoList = new ImageList();
```

...

- In the next step, you may set properties of an ImageList control.
- The following code snippet sets a few properties of an ImageList.

```
photoList.TransparentColor = Color.Blue;
photoList.ColorDepth = ColorDepth.Depth32Bit;
photoList.ImageSize = new Size (200, 200);
```

- Unlike other Windows Forms control, you can't add ImageList control to a Form.
- You need to draw an ImageList control using the Draw method.
- The Draw method takes a Graphics object that is the handle of the container control that will be used as a drawing canvas.

```
photoList.Draw (g, new Point (20, 20), count);
```

- **Setting ImageList Properties**
  - ✓ ColorDepth property represents the color depth of the image list.
  - ✓ ImageSize property represents the size of the images in the image list.
  - ✓ Images property represents all images in an ImageList as an ImageCollection object.

...

- The following code snippet sets these properties and adds three images to the ImageList control and later loops through the images and displays them on a Form.

```
Graphics g = Graphics.FromHwnd (this.Handle);
ImageList photoList = new ImageList();
photoList.TransparentColor = Color.Blue;
photoList.ColorDepth = ColorDepth.Depth32Bit;
photoList.ImageSize = new Size (200, 200);
photoList.Images.Add (Image.FromFile(@"C:\Images\Garden.jpg"));
photoList.Images.Add (Image.FromFile(@"C:\Images\Tree.jpg"));
photoList.Images.Add (Image.FromFile(@"C:\Images\Waterfall.jpg"));
for (int count = 0; count < photoList.Images.Count; count++)
{
   photoList.Draw(g, new Point (20, 20), count);
   // Paint the form and wait to load the image
   Application.DoEvents();
   System.Threading.Thread.Sleep (1000);
}
```

More information on

- John Sharp. Microsoft Visual C# 2013 Step by Step, 2015 Microsoft Press USA
- Joel Murach, Anne Boehm. Murach C# 2012, Mike Murach & Associates Inc USA, 2013
- Microsoft Corporation. Microsoft Visual Studio 2012 Product Guide, 2012

QUESTIONS