Unit – 4 Working With GUI

4.1 Explain Visual studio IDE:

- IDE stands for the ***integrated development environment (IDE)***. IDE Provides an Environment consolidated for the Programmer to write a Computer Program.
- IDE helps in combining common activities of writing software into a single application such as editing source code, building executables, and debugging. IDE is basically an Environment or Combination of tools like a text editor, debugger, and Compiler.
- In this article, we will try to understand IDEs and Debuggers. An IDE provides virtually all of the tools a Programmer needs to write and build a program from end to end. Programmers or Developers use an IDE to write, manage, and execute code while running their applications. It makes the development process much easier by abstracting different aspects of editing code into a Single program.
- Most IDEs can run programming languages, such as Python or Java, but many IDEs are language-specific(ie; Pycharm is for developing python applications). For text editing capabilities, and possessor allow the insertion of frameworks and element libraries to build upon base-level code.

Why is an IDE important?
Throughout the process of writing, creating, and testing software, developers employ a variety of tools. Text editors, code libraries, bug tracking software, compilers, and test platforms are among the most common development tools. It combines several of these development-related technologies into a single framework. It is also helpful for new developers who may use an IDE to learn about a team's standard tools and practices. The main aim is to make software development easier while also detecting and reducing code errors and typos.

Components:
Menu Bar:
A menu bar is a thin, horizontal bar containing the labels of menus in a GUI. The menu bar provides the user with a place in a window to find program's essential functions. These functions include opening and closing files, editing text, and quitting the program.

Standard Toolbar:
A toolbar is a classic control container. It can host text, buttons, etc. It is up to you to decide whether your application needs a toolbar and what you want to position on it.

Toolbox:
You can drag and drop different controls onto the surface of the designer you are using, and resize and position the controls. Toolbox appears in conjunction with designer views, such as the designer view of a XAML file or a Windows Forms App project. Toolbox displays only those controls that can be used in the current designer. You can search within Toolbox to further filter the items that appear.

Forms Designer:
Windows Forms Designer in Visual Studio provides a rapid development solution for creating Windows Forms-based applications. Windows Forms Designer lets you easily add controls to a form, arrange them, and write code for their events.

Output Window:
The Output window displays status messages for various features in the integrated development environment (IDE). To open the Output window, on the menu bar, choose View > Output, or press Ctrl+Alt+O.

Solution Explorer:
The Solution Explorer in Visual Studio Code assists programmers and developers in managing their projects better. It enables navigating files, making the necessary changes, and keeping track of them. It also helps developers view the project structure, add or remove files from the project, and open new ones.

Properties Window:
The Properties window is used to display properties for objects selected in the two main types of windows available in the Visual Studio integrated development environment (IDE).

Microsoft Visual Studio is a powerful IDE that ensures quality code throughout the entire application lifecycle, from design to deployment. Some windows are used for writing code, some for designing interfaces, and others for getting a general overview of files or classes in your application. Microsoft Visual Studio includes a host of visual designers to aid in the development of various types of applications. These tools include such as Windows Forms Designer, WPF (Windows Presentation

Foundation) Designer, Web development, Class designer, Data designer and Mapping designer.

4.2 Working with windows forms:

- https://www.youtube.com/watch?v=O_9DdPNo4RI&list=PLjC4UKOOcfDRsRI6xXOLzyGiGcBHVARlr&index=1
- https://www.youtube.com/watch?v=c7pPqY72pS0
- https://www.youtube.com/watch?v=CdH8z_JNi_U

4.3 MDI AND SDI FORM:

MDI and SDI are interface designs for handling documents within a single application. MDI stands for "Multiple Document Interface" while SDI stands for "Single Document Interface". Both are different from each other in many aspects. One document per window is enforced in SDI while child windows per document are allowed in MDI. SDI contains one window only at a time but MDI contain multiple document at a time appeared as child window. MDI is a container control while SDI is not container control. MDI supports many interfaces means we can handle many applications at a time according to user's requirement. But SDI supports one interface means you can handle only one application at a time.

What is MDI?
MDI stands for Multiple Document Interface. It is an interface design for handling documents within a single application. When application consists of an MDI parent form containing all other window consisted of the app, then MDI interface can be used. Switch focus to a specific document can be easily handled in MDI. For maximizing all documents, parent window is maximized by MDI.

What is SDI?
SDI stands for Single Document Interface. It is an interface design for handling documents within a single application. SDI exists independently from others and thus is a stand-alone window. SDI supports one interface means you can handle only one application at a time. For grouping, SDI uses special window managers.

Key Differences between MDI and SDI:
- MDI stands for "Multiple Document Interface" while SDI stands for "Single Document Interface".

- One document per window is enforced in SDI while child windows per document are allowed in MDI.
- MDI is a container control while SDI is not container control.
- SDI contains one window only at a time but MDI contains multiple documents at a time appeared as child window.
- MDI supports many interfaces means we can handle many applications at a time according to user's requirement. But SDI supports one interface means you can handle only one application at a time.
- For switching between documents MDI uses special interface inside the parent window while SDI uses Task Manager for that.
- In MDI grouping is implemented naturally but in SDI grouping is possible through special window managers.
- For maximizing all documents, parent window is maximized by MDI but in case of SDI, it is implemented through special code or window manager.
- Switch focus to the specific document can be easily handled while in MDI but it is difficult to implement in SDI.

4.4 Message box show:

- It is always required that a message is displayed to the user as a token of information or confirmation so that the user is aware of the status of the operation he performed. The message can be anything ranging from "The payment is successful" or a warning type like "Do you want to continue" etc.
- This is achieved in C# with the help of Message Box. A message box can be considered as an interface between the user and the application.
- It is nothing but a window that has text, images, or symbols to guide or convey something to the user. Until appropriate action is performed, and the message box is closed, it will not allow other actions to be performed.

Syntax:
Message Box is a class in the "Systems.Windows. Forms" Namespace and the assembly it is available is "System.Windows.Forms.dll". The show method available in the class is used to display the message along with action buttons. The action buttons can be anything ranging from Yes to No, Ok to Cancel.

Type of Methods:

Method Signature	Description
MessageBox.Show(string text)	Displays a message box with the specified text and an OK button.
MessageBox.Show(string text, string caption)	Displays a message box with the specified text and caption, and an OK button.
MessageBox.Show(string text, string caption, MessageBoxButtons buttons)	Displays a message box with the specified text, caption, and custom buttons.
MessageBox.Show(string text, string caption, MessageBoxButtons buttons, MessageBoxIcon icon)	Displays a message box with the specified text, caption, custom buttons, and custom icon.
MessageBox.Show(string text, string caption, MessageBoxButtons buttons, MessageBoxIcon icon, MessageBoxDefaultButton defaultButton)	Displays a message box with the specified text, caption, custom buttons, custom icon, and default button.
MessageBox.Show(string text, string caption, MessageBoxButtons buttons, MessageBoxIcon icon, MessageBoxDefaultButton defaultButton, MessageBoxOptions options)	Displays a message box with the specified text, caption, custom buttons, custom icon, default button, and options.

Types of MessageBox Buttons :
The following are the types of Buttons that are available in the MessageBox.Show() method. They are as given below

- **OK:** It is defined as MessageBoxButtons.OK
- **OK and Cancel:** It is defined as MessageBoxButtons.OkCancel.
- **Abort Retry and Ignore**: It is defined as MessageBoxButtons.AbortRetryIgnore.
- **Yes No and Cancel:** It is defined as MessageBoxButtons.YesNoCancel.
- **Yes and No:** It is defined as MessageBoxButtons.YesNo.
- **Retry and Cancel:** It is defined as MessageBoxButtons.RetryCancel.

Types of MessageBox Icons :
The following are the types of MessageBox icons method are:

- **None:** No icons are displayed in the Message box.
- **Hand:** A hand icon is displayed. It is defined as MessageBoxIcon.Hand.
- **Question:** A question mark is displayed. It is defined as MessageBoxIcon.Question.
- **Exclamation:** An exclamation mark is displayed. It is defined as MessageBoxIcon.Exclamation.
- **Asterisk:** An asterisk symbol is displayed. It is defined as MessageBoxIcon.Asterisk.
- **Stop:** A stop icon is displayed. It is defined as MessageBoxIcon.Stop.
- **Error:** An error icon is displayed. It is defined as MessageBoxIcon.Error.
- **Warning:** A warning icon is displayed. It is defined as MessageBoxIcon.Warning.
- **Information:** An info symbol is displayed. It is defined as MessageBoxIcon.Information.

- **Types of MessageBox Options-chat gpt karvanu che**
  The following are the various Message Box options that are available.
- **ServiceNotification:** It is defined as
- **MessageBoxOptions.ServiceNotification.** This is used to display the message box on the current desktop which is active. The message box is displayed even when no user is logged on to the desktop.
- **DefaultDesktopOnly:** It is defined as MessageBoxOptions.DefaultDesktopOnly. This also displays on the currently active desktop. The difference between this and service notification is that here the message is displayed on the interactive window.
- **RightAlign:** It is defined as MessageBoxOptions.RightAlign. This is used to format the message in right alignment.
- **RtlReading:** It is defined as MessageBoxOptions.RtlReading. This denotes that message is displayed from right to left order.

Example ;
```
using System;
using System.Windows.Forms;
class msgbox_example
{

  static void Main()
```

```
    {
      string box_msg = "A Message Box with OK Button";

       string box_title = "Message Box Demo";

       MessageBox.Show(box_msg, box_title);
    }

}
```

4.5 Standard controls in c# :

1.The Textbox control:

- This is an input control which is used to take user input.
- This is server side control; asp provides own tag to create it.
- Here are some important properties and events associated with the Textbox control.

Properties:
Text: Gets or sets the text displayed in the Textbox.
Multiline: Specifies whether the Textbox should allow multiple lines of text.
MaxLength: Specifies the maximum number of characters that can be entered.
Password Char: Specifies a character to be displayed instead of the actual text (for password fields).
Read-only: Specifies whether the Textbox is read-only or editable.
Border Style: Specifies the border style of the Textbox.
Back Color: Specifies the background color of the Textbox.
Forecolor: Specifies the text color of the Textbox.

Events:
Text Changed: Triggered when the content of the Textbox changes.
Keypress: Triggered when a key is pressed while the Textbox has focus.
Key Down: Triggered when a key is pressed down.
Key Up: Triggered when a key is released.

Example:

```
using System;
using System.Windows.Forms;

namespace TextBoxExample
```

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

        private void textBox1_TextChanged(object sender, EventArgs e)
        {
        // Handle the TextChanged event
        string enteredText = textBox1.Text;
        label1.Text = "Entered Text: " + enteredText;
        }
        }
}
```

2. The Label control:

- The Label control in C# is used to display static text on a form or other containers. It's often used to provide descriptions, headings, or any non-editable text that provides information to the user.
- Here are some important properties and events associated with the Label control.

Properties:

- **Text:** Gets or sets the text displayed in the Label.
- **AutoSize:** Specifies whether the Label adjusts its size to fit its content.
- **BackColor:** Specifies the background color of the Label.
- **ForeColor:** Specifies the text color of the Label.
- **Font:** Specifies the font used for the Label's text.
- **BorderStyle:** Specifies the border style of the Label.
- **ContentAlignment:** Specifies the alignment of the Label's text within its bounds.

Events:

- **Click:** Triggered when the Label is clicked.
- **DoubleClick:** Triggered when the Label is double-clicked.

Example:

```
using System;
using System.Windows.Forms;

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

        private void label1_Click(object sender, EventArgs e)
        {
        // Handle the Click event of the label
          label1.Text = "Label Clicked!";
        }
        }
}
```

3. The Button control:

- The Button control in C# is used to create a clickable button that users can interact with to trigger actions or events in a Windows Forms application.
- Here are some important properties and events associated with the Button control.

Properties:

- **Text:** Gets or sets the text displayed on the button.
- **Image:** Specifies an image to display on the button.
- **ImageAlign:** Specifies the alignment of the image relative to the text.
- **BackColor:** Specifies the background color of the button.
- **ForeColor:** Specifies the text color of the button.
- **Font:** Specifies the font used for the button's text.
- **FlatStyle:** Specifies the appearance style of the button (e.g., Flat, Popup, Standard).
- **Enabled:** Specifies whether the button is enabled or disabled.

Events:

- **Click:** Triggered when the button is clicked.
- **MouseEnter:** Triggered when the mouse enters the button's area.
- **MouseLeave:** Triggered when the mouse leaves the button's area.

Example :

```
using System;
using System.Windows.Forms;

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

        private void button1_Click(object sender, EventArgs e)
        {
        // Handle the Click event of the button
         MessageBox.Show("Button Clicked!");
        }
        }
}
```

4. The List box control:

- The ListBox control in C# is used to display a list of items from which users can select one or more items. It's a common control for scenarios where users need to choose items from a predefined list.
- Here are some important properties and events associated with the ListBox control.

Properties:

- **Items:** Represents the collection of items displayed in the ListBox.
- **SelectionMode:** Specifies how items are selected (Single, MultiSimple, MultiExtended).
- **SelectedIndex:** Gets or sets the index of the currently selected item.
- **SelectedItems:** Provides a collection of selected items.

- **BackColor:** Specifies the background color of the ListBox.
- **ForeColor**: Specifies the text color of the ListBox.
- **Font:** Specifies the font used for the ListBox's items.

Events:

- **SelectedIndexChanged:** Triggered when the selected index changes.
- **DoubleClick:** Triggered when an item is double-clicked.
- **MouseDown:** Triggered when the mouse button is pressed while over the control.

Example :

```
using System;
using System.Windows.Forms;

namespace ListBoxExample
{
        public partial class Form1 : Form
        {
        public Form1()
        {
        InitializeComponent();
        // Add items to the ListBox during initialization
         listBox1.Items.Add("Item 1");
         listBox1.Items.Add("Item 2");
         listBox1.Items.Add("Item 3");
        }

        private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
        // Handle the SelectedIndexChanged event
        if (listBox1.SelectedIndex != -1)
        {
                string selectedItem = listBox1.SelectedItem.ToString();
                MessageBox.Show("Selected Item: " + selectedItem);
        }
        }
        }
}
```

5. The Check box Control :

- The CheckBox control in C# is used to provide a two-state selection mechanism, where users can either check or uncheck a box to indicate a choice or option.
- It's commonly used for scenarios where users need to make binary selections.Here are some important properties and events associated with the CheckBox control.

Properties:

- **Text:** Gets or sets the text displayed next to the CheckBox.
- **Checked:** Gets or sets a value indicating whether the CheckBox is checked.
- **CheckState:** Gets or sets the state of the CheckBox (Checked, Unchecked, Indeterminate).
- **AutoCheck:** Specifies whether the control's Checked property changes automatically when clicked.
- **ThreeState:** Specifies whether the CheckBox can have three states (Checked, Unchecked, Indeterminate).
- **BackColor:** Specifies the background color of the CheckBox.
- **ForeColor:** Specifies the text color of the CheckBox.

Events:

- **CheckedChanged:** Triggered when the Checked property changes.
- **CheckStateChanged:** Triggered when the CheckState property changes.
- **Click:** Triggered when the CheckBox is clicked.

Example :

```
using System;
using System.Windows.Forms;

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

        private void checkBox1_CheckedChanged(object sender, EventArgs e)
        {
```

```
        // Handle the CheckedChanged event
        if (checkBox1.Checked)
        {
            label1.Text = "Checkbox is checked";
        }
          else
        {
            label1.Text = "Checkbox is unchecked";
        }
        }
        }
}
```

6. Combo box Control :

- The ComboBox control in C# is used to provide a dropdown list of items from which users can select a single option.
- It's commonly used for scenarios where users need to choose one item from a predefined list of options.

Properties:

- **Items:** Represents the collection of items displayed in the ComboBox.
- **SelectedIndex:** Gets or sets the index of the currently selected item.
- **SelectedItem:** Gets or sets the currently selected item.
- **Text:** Gets or sets the text displayed in the ComboBox.
- **DropDownStyle:** Specifies the behavior of the dropdown portion (DropDown, DropDownList, Simple).
- **AutoCompleteMode:** Specifies the automatic completion behavior (None, Suggest, Append, SuggestAppend).
- **BackColor:** Specifies the background color of the ComboBox.
- **ForeColor:** Specifies the text color of the ComboBox.
- **Font:** Specifies the font used for the ComboBox's text.

Events:

- **SelectedIndexChanged:** Triggered when the selected index changes.
- **SelectedValueChanged:** Triggered when the selected value changes.
- **DropDown:** Triggered when the dropdown portion is shown.

- **DropDownClosed:** Triggered when the dropdown portion is closed.

Example :

```
using System;
using System.Windows.Forms;

namespace ComboBoxExample
{
   public partial class Form1 : Form
        {
      public Form1()
      {
         InitializeComponent();
         // Add items to the ComboBox during initialization
         comboBox1.Items.Add("Option 1");
         comboBox1.Items.Add("Option 2");
         comboBox1.Items.Add("Option 3");
      }
   private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
      {
         // Handle the SelectedIndexChanged event
         if (comboBox1.SelectedIndex != -1)
         {
                string selectedItem = comboBox1.SelectedItem.ToString();
                label1.Text = "Selected Item: " + selectedItem;
         }
      }
        }
}
```

7. The Radio button control :

- The RadioButton control in C# is used to provide a set of mutually exclusive choices where users can select only one option from a group of options. It's commonly used when you have multiple options and you want the user to choose one from those options.
- Here are some important properties and events associated with the RadioButton control.

Properties:

- **Text:** Gets or sets the text displayed next to the RadioButton.
- **Checked:** Gets or sets a value indicating whether the RadioButton is checked.
- **AutoCheck:** Specifies whether the control's Checked property changes automatically when clicked.
- **ForeColor:** Specifies the text color of the RadioButton.
- **Font:** Specifies the font used for the RadioButton's text.

Events:

- **CheckedChanged:** Triggered when the Checked property changes.
- **Click:** Triggered when the RadioButton is clicked.

Example:

```
using System;
using System.Windows.Forms;

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

    private void radioButton1_CheckedChanged(object sender, EventArgs e)
    {
      // Handle the CheckedChanged event for RadioButton 1
```

```
            if (radioButton1.Checked)
            {
                label1.Text = "Option 1 selected";
            }
        }

        private void radioButton2_CheckedChanged(object sender, EventArgs e)
        {
            // Handle the CheckedChanged event for RadioButton 2
            if (radioButton2.Checked)
            {
                label1.Text = "Option 2 selected";
            }
        }

        private void radioButton3_CheckedChanged(object sender, EventArgs e)
        {
            // Handle the CheckedChanged event for RadioButton 3
            if (radioButton3.Checked)
            {
                label1.Text = "Option 3 selected";
            }
        }
          }
}
```

8. The Picture box Control :

- The PictureBox control in C# is used to display images on a Windows Forms application. It provides a way to load and display image files, such as JPEG, PNG, GIF, and more.
- Here are some important properties and events associated with the PictureBox control.

Properties:

- **Image:** Gets or sets the image displayed in the PictureBox.
- **SizeMode:** Specifies how the image is displayed (Normal, StretchImage, AutoSize, CenterImage, Zoom).
- **BorderStyle:** Specifies the border style of the PictureBox.
- **BackColor:** Specifies the background color of the PictureBox.

Events:

- **Click:** Triggered when the PictureBox is clicked.
- **DoubleClick:** Triggered when the PictureBox is double-clicked.

Example:

```
using System;
using System.Windows.Forms;

namespace PictureBoxExample
{
   public partial class Form1 : Form
        {
      public Form1()
      {
         InitializeComponent();
         // Load an image and set it to the PictureBox
         pictureBox1.Image = Properties.Resources.sample_image;
         pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
      }

      private void pictureBox1_Click(object sender, EventArgs e)
      {
         // Handle the Click event of the PictureBox
         MessageBox.Show("Image clicked!");
      }
        }
}
```

9. The Panel control :

- The Panel control in C# is used as a container to group and organize other controls within a form. It provides a way to group controls visually and manage their layout within a specific area.
- Here are some important properties and events associated with the Panel control

Properties:

- **BackColor:** Specifies the background color of the Panel.
- **BorderStyle:** Specifies the border style of the Panel (None, FixedSingle, Fixed3D).
- **AutoScroll:** Specifies whether scrollbars are automatically displayed when controls exceed the panel's visible area.
- **Dock:** Specifies how the panel is docked within its parent control.
- **Controls:** Represents the collection of controls contained within the Panel.

Events:

- **Click:** Triggered when the Panel is clicked.
- **DoubleClick:** Triggered when the Panel is double-clicked.

Example:

```
using System;
using System.Windows.Forms;

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

      private void button1_Click(object sender, EventArgs e)
      {
            // Create a new button and add it to the panel
            Button newButton = new Button();
            newButton.Text = "Dynamic Button";
            newButton.Click += DynamicButton_Click;

            panel1.Controls.Add(newButton);
        }

        private void DynamicButton_Click(object sender, EventArgs e)
        {
            // Handle the click event of the dynamically added button
            MessageBox.Show("Dynamic Button Clicked!");
        }
          }
}
```

10. The Scrollbar Control:

The ScrollBar control in C# is used to provide a user interface element for scrolling content within a container that is larger than the visible area.

Here are some important properties and events associated with the ScrollBar control.

Properties:

- **Orientation:** Specifies the orientation of the ScrollBar (Horizontal or Vertical).
- **Minimum:** Specifies the minimum value of the ScrollBar.
- **Maximum:** Specifies the maximum value of the ScrollBar.
- **Value:** Specifies the current value of the ScrollBar.
- **LargeChange:** Specifies the amount by which the value changes when the user clicks on the track or presses the page-up/page-down keys.
- **SmallChange:** Specifies the amount by which the value changes when the user clicks the arrows or uses the arrow keys.
- **Enabled:** Specifies whether the ScrollBar is enabled or disabled.

Events:

- **Scroll:** Triggered when the ScrollBar's value changes.
- **ValueChanged:** Triggered when the Value property changes.

Example :

```
using System;
using System.Windows.Forms;

namespace ScrollBarExample
{
        public partial class Form1 : Form
        {
      public Form1()
      {
         InitializeComponent();
         // Set ScrollBar properties
         hScrollBar1.Minimum = 0;
         hScrollBar1.Maximum = 100;
         hScrollBar1.LargeChange = 10;
         hScrollBar1.SmallChange = 1;
      }

      private void hScrollBar1_Scroll(object sender, ScrollEventArgs e)
      {
         // Handle the Scroll event of the horizontal ScrollBar
         label1.Text = "Value: " + hScrollBar1.Value;
      }
        }
}
```

11. The Timer control :

- The Timer control in C# is used to execute a particular piece of code at predefined intervals. It's commonly used for tasks that need to be performed periodically, such as updating the UI, polling for data, or triggering events.
- Here are some important properties and events associated with the Timer control

Properties:

- Interval: Specifies the time interval (in milliseconds) between each Tick event.
- Enabled: Specifies whether the Timer is running or stopped.

Events:

- Tick: Triggered at each interval specified by the Interval property.

Example:

```
using System;
using System.Windows.Forms;

namespace TimerExample
{
   public partial class Form1 : Form
        {
      private int counter = 0;

      public Form1()
      {
         InitializeComponent();
         // Set Timer properties
         timer1.Interval = 1000; // 1 second
         timer1.Enabled = true; // Start the Timer
              timer1.Tick += Timer1_Tick; // Subscribe to the Tick event
      }

      private void Timer1_Tick(object sender, EventArgs e)
      {
         // Handle the Tick event of the Timer
         counter++;
         label1.Text = "Counter: " + counter;
      }
        }
}
```

12. The Date time picker Control :

- The DateTimePicker control in C# is used to allow users to select dates and/or times using a graphical user interface. It provides a convenient way for users to pick specific dates and times without having to manually input them.
- Here are some important properties and events associated with the DateTimePicker control.

Properties:

- **Value:** Gets or sets the selected date and time.
- **Format:** Specifies how the date and/or time is displayed (Long, Short, Time, Custom).
- **CustomFormat:** Specifies a custom format string when the Format property is set to Custom.
- **MinDate:** Specifies the minimum selectable date.
- **MaxDate:** Specifies the maximum selectable date.
- **ShowUpDown:** Specifies whether the control displays a dropdown for selecting times.
- **Enabled:** Specifies whether the DateTimePicker is enabled or disabled.

Events:

- **ValueChanged:** Triggered when the selected date and/or time changes.

Example:

```
using System;
using System.Windows.Forms;

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

        private void dateTimePicker1_ValueChanged(object sender, EventArgs e)
        {
            // Handle the ValueChanged event of the DateTimePicker
            label1.Text = "Selected Date: " + dateTimePicker1.Value.ToString("yyyy-
MM-dd");
        }
          }
}
```

13. The Notify icon Control:

- The NotifyIcon control in C# is used to display an icon in the system tray (notification area) of the Windows taskbar. It's commonly used to provide quick access to application features, show notifications, or run tasks in the background while keeping the application minimized.
- Here are some important properties and events associated with the NotifyIcon control.

Properties:

- **Icon:** Specifies the icon displayed in the system tray.
- **Text:** Specifies the tooltip text displayed when hovering over the icon.
- **ContextMenu:** Specifies the context menu to display when right-clicking the icon.
- **Visible:** Specifies whether the NotifyIcon is visible in the system tray.

Events:

- **MouseClick:** Triggered when the user clicks the icon.
- **MouseDoubleClick:** Triggered when the user double-clicks the icon.
- **BalloonTipClicked:** Triggered when the user clicks the balloon notification.
- **BalloonTipClosed:** Triggered when the balloon notification is closed.

Example:

```
using System;
using System.Windows.Forms;

namespace NotifyIconExample
{
   public partial class Form1 : Form
        {
       public Form1()
       {
          InitializeComponent();
          // Set NotifyIcon properties
          notifyIcon1.Icon = Properties.Resources.app_icon;
          notifyIcon1.Text = "My Application";
          notifyIcon1.Visible = true;
          // Subscribe to events
          notifyIcon1.MouseClick += NotifyIcon1_MouseClick;
       }

       private void NotifyIcon1_MouseClick(object sender, MouseEventArgs e)
       {
          // Handle the MouseClick event of the NotifyIcon
          if (e.Button == MouseButtons.Left)
          {
                 MessageBox.Show("Icon clicked!");
          }
       }
         }
}
```

14. The Image List control:

- The ImageList control in C# is used to manage and store a collection of images that can be used in other controls, such as ListView, TreeView, or menus. It provides a way to centralize image resources and efficiently display them in various UI elements.
- Here are some important properties and events associated with the ImageList control.

Properties:

- **Images:** Represents the collection of images stored in the ImageList.
- **ColorDepth:** Specifies the color depth of the images (Depth8Bit, Depth16Bit, Depth24Bit, Depth32Bit).
- **ImageSize:** Specifies the size of the images in the ImageList.
- **TransparentColor:** Specifies the transparent color for the images.
- **TransparentColorChanged:** Triggered when the TransparentColor property changes.

Events:

- Disposed: Triggered when the ImageList is disposed.

Example:

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace ImageListExample
{
   public partial class Form1 : Form
        {
      public Form1()
      {
         InitializeComponent();
         // Create an ImageList
         ImageList imageList = new ImageList();
         imageList.ImageSize = new Size(32, 32);
         imageList.Images.Add("icon1", Properties.Resources.icon1);
         imageList.Images.Add("icon2", Properties.Resources.icon2);
```

```
            // Assign the ImageList to a ListView
            listView1.LargeImageList = imageList;

            // Add items to the ListView with assigned images
            listView1.Items.Add(new ListViewItem("Item 1", "icon1"));
            listView1.Items.Add(new ListViewItem("Item 2", "icon2"));
        }
    }
}
```

15. The Link Label control :

- The LinkLabel control in C# is used to display hyperlinks in a Windows Forms application. It provides a way to create clickable links that can open web pages, files, or perform other actions when clicked.
- Here are some important properties and events associated with the LinkLabel control.

Properties:

Text: Gets or sets the text displayed as the link.
LinkArea: Specifies the area of the text that is treated as a link.
LinkBehavior: Specifies the behavior when the link is clicked (AlwaysUnderline, HoverUnderline, NeverUnderline).
LinkColor: Specifies the color of the link text.
VisitedLinkColor: Specifies the color of visited link text.
ActiveLinkColor: Specifies the color of the link text when clicked.
Visited: Specifies whether the link has been visited.

Events:

- LinkClicked: Triggered when the link is clicked.

Example:

```
using System;
using System.Diagnostics;
using System.Windows.Forms;

namespace LinkLabelExample
{
   public partial class Form1 : Form
        {
       public Form1()
       {
          InitializeComponent();
          linkLabel1.Text = "Click here to visit Google";
          linkLabel1.LinkClicked += LinkLabel1_LinkClicked;
       }

       private void LinkLabel1_LinkClicked(object sender,
LinkLabelLinkClickedEventArgs e)
       {
          // Handle the LinkClicked event of the LinkLabel
          Process.Start("https://www.google.com");
       }
        }
}
```

16. The List view control :

- The ListView control in C# is used to display a collection of items in a tabular format. It allows you to show data in various views such as details, list, icons, or tiles. The ListView is a versatile control often used to display lists of items, files, or data records.
- Here are some important properties and events associated with the ListView control:

Properties:

- **Items:** Represents the collection of items displayed in the ListView.
- **View:** Specifies the view style (Details, LargeIcon, SmallIcon, List, Tile).
- **Columns:** Represents the collection of columns in Details view.
- **SmallImageList:** Specifies the ImageList for small icons in SmallIcon view.
- **LargeImageList:** Specifies the ImageList for large icons in LargeIcon view.
- **Groups:** Represents the collection of groups for organizing items.
- **SelectedItems:** Provides a collection of selected items.
- **MultiSelect:** Specifies whether multiple items can be selected.
- **FullRowSelect:** Specifies whether the entire row is highlighted when an item is selected.

Events:

- **SelectedIndexChanged:** Triggered when the selected item(s) change.
- **ItemActivate:** Triggered when an item is double-clicked or activated.
- **ColumnClick:** Triggered when a column header is clicked.

Example:

```
using System;
using System.Windows.Forms;

namespace ListViewExample
{
  public partial class Form1 : Form
      {
    public Form1()
    {
      InitializeComponent();
      // Set the view style to Details
      listView1.View = View.Details;

      // Add columns to the ListView
      listView1.Columns.Add("Name");
      listView1.Columns.Add("Age");
```

```
            // Add items with sub-items
            ListViewItem item1 = new ListViewItem("John");
            item1.SubItems.Add("30");

            ListViewItem item2 = new ListViewItem("Jane");
            item2.SubItems.Add("25");

            listView1.Items.Add(item1);
            listView1.Items.Add(item2);
        }
          }
}
```

17. The Tree view control :

- The TreeView control in C# is used to display hierarchical data in a tree-like structure, where each item can have child items. It's commonly used to show relationships between items or to organize data in a nested manner.
- Here are some important properties and events associated with the TreeView control.

Properties:

- **Nodes:** Represents the collection of tree nodes displayed in the TreeView.
- **ImageList:** Specifies the ImageList used for node icons.
- **SelectedNode:** Gets or sets the currently selected tree node.
- **CheckBoxes:** Specifies whether checkboxes are displayed next to nodes.
- **FullRowSelect:** Specifies whether the entire row is highlighted when a node is selected.
- **ShowLines:** Specifies whether lines are displayed to connect nodes.
- **ShowPlusMinus:** Specifies whether expand/collapse buttons are shown.
- **HideSelection:** Specifies whether the selection is hidden when the TreeView loses focus.

Events:

- **AfterSelect:** Triggered after a tree node is selected.
- **BeforeExpand:** Triggered before a tree node is expanded.
- **BeforeCollapse:** Triggered before a tree node is collapsed.

Example :

```
using System;
using System.Windows.Forms;

namespace TreeViewExample
{
  public partial class Form1 : Form
      {
    public Form1()
    {
      InitializeComponent();
      // Add root nodes to the TreeView
      TreeNode rootNode = new TreeNode("Root Node");
      TreeNode childNode1 = new TreeNode("Child Node 1");
      TreeNode childNode2 = new TreeNode("Child Node 2");

      treeView1.Nodes.Add(rootNode);
      rootNode.Nodes.Add(childNode1);
      rootNode.Nodes.Add(childNode2);

      // Expand the root node
      rootNode.Expand();
    }

    private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
    {
      // Handle the AfterSelect event of the TreeView
      label1.Text = "Selected Node: " + e.Node.Text;
    }
```

```
        }
}
```

18 .The Tool bar control :

- The ToolBar control in C# is used to create a customizable toolbar within a Windows Forms application. It provides a way to group various command buttons or items that perform specific actions when clicked.
- Here are some important properties and events associated with the ToolBar control.

Properties:

- **Buttons:** Represents the collection of toolbar buttons.
- **ImageList:** Specifies the ImageList used for button icons.
- **ButtonSize:** Specifies the size of the toolbar buttons.
- **DropDownArrows:** Specifies whether dropdown arrows are displayed next to buttons.
- **Appearance:** Specifies the appearance of the toolbar (Normal, Flat, Gradient).
- **BorderStyle:** Specifies the border style of the toolbar (None, Flat, Raised, Etched).
- **Dock:** Specifies how the toolbar is docked within its parent control.

Events:

- **ButtonClick:** Triggered when a toolbar button is clicked.

Example:

```
using System;
using System.Windows.Forms;

namespace ShortToolBarExample
{
   public partial class MainForm : Form
   {
      private ToolStrip toolStrip;
      private ToolStripButton saveButton;
      private ToolStripButton openButton;
```

```
        public MainForm()
        {
            InitializeComponent();
            InitializeToolBar();
        }

        private void InitializeToolBar()
        {
            // Create a ToolStrip control
            toolStrip = new ToolStrip();

            // Create ToolStripButton controls
            saveButton = new ToolStripButton("Save");
            openButton = new ToolStripButton("Open");

            // Add ToolStripButton controls to the ToolStrip
            toolStrip.Items.Add(saveButton);
            toolStrip.Items.Add(openButton);

            // Add the ToolStrip to the form's controls
            this.Controls.Add(toolStrip);
        }

        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }
}
```

19. The Status bar control :

- The StatusBar control in C# is used to display status information at the bottom of a form. It's commonly used to provide users with context-sensitive information or updates about the current state of the application.
- Here are some important properties and events associated with the StatusBar control.

Properties:

- **Panels:** Represents the collection of panels within the StatusBar.
- **ShowPanels:** Specifies whether panels are visible.
- **SizingGrip:** Specifies whether a sizing grip is displayed.
- **Text:** Gets or sets the text displayed in the default panel.

Events:

- **PanelClick:** Triggered when a panel is clicked.

Example :

```
private void CreateMyStatusBar()
{
   // Create a StatusBar control.
   StatusBar statusBar1 = new StatusBar();
   // Create two StatusBarPanel objects to display in the StatusBar.
   StatusBarPanel panel1 = new StatusBarPanel();
   StatusBarPanel panel2 = new StatusBarPanel();

   // Display the first panel with a sunken border style.
   panel1.BorderStyle = StatusBarPanelBorderStyle.Sunken;
   // Initialize the text of the panel.
   panel1.Text = "Ready...";
   // Set the AutoSize property to use all remaining space on the StatusBar.
   panel1.AutoSize = StatusBarPanelAutoSize.Spring;

   // Display the second panel with a raised border style.
   panel2.BorderStyle = StatusBarPanelBorderStyle.Raised;

   // Create ToolTip text that displays time the application was started.
   panel2.ToolTipText = "Started: " +
System.DateTime.Now.ToShortTimeString();
   // Set the text of the panel to the current date.
   panel2.Text = System.DateTime.Today.ToLongDateString();
   // Set the AutoSize property to size the panel to the size of the contents.
   panel2.AutoSize = StatusBarPanelAutoSize.Contents;

   // Display panels in the StatusBar control.
   statusBar1.ShowPanels = true;

   // Add both panels to the StatusBarPanelCollection of the StatusBar.

   statusBar1.Panels.Add(panel1);
   statusBar1.Panels.Add(panel2);

   // Add the StatusBar to the form.
   this.Controls.Add(statusBar1);
}
```

4.6 Types of dialog box in c# :

1. The Openfiledialog control :

- It is used to create a standard Windows file open dialog for selecting files from the user's file system.

Properties:

- **Filter:** Specifies the file type filter for the dialog. For example, you can set it to "Text files (*.txt)|*.txt|All files (*.*)|*.*" to allow the user to select text files.
- **FileName:** Gets or sets the name of the selected file, including the path.
- **FileNames:** Gets an array of selected file names (useful when the dialog allows multiple selections).
- **Multiselect:** Determines whether the user can select multiple files. If set to true, FileNames will contain an array of selected file names.
- **CheckFileExists:** Specifies whether the dialog should check if the selected file exists. If set to true, it will display a warning if the selected file doesn't exist.

Example:

```
using System;
using System.Windows.Forms;

namespace OpenFileDialogExample
{
   public partial class MainForm : Form
        {
      public MainForm()
      {
         InitializeComponent();
      }

      private void openButton_Click(object sender, EventArgs e)
      {
         // Create an instance of OpenFileDialog
         OpenFileDialog openFileDialog = new OpenFileDialog();

         // Set properties
         openFileDialog.Title = "Open File";
         openFileDialog.Filter = "Text Files (*.txt)|*.txt|All Files (*.*)|*.*";

         // Show the dialog and check if the user clicked OK
         if (openFileDialog.ShowDialog() == DialogResult.OK)
         {
                // Get the selected file path
                string selectedFilePath = openFileDialog.FileName;

                // Handle the selected file here, e.g., display it in a textbox
                selectedFileTextBox.Text = selectedFilePath;
         }
      }
        }
}
```

2. The Savefiledialog Control:

- In C#, the "SaveFileDialog" is a class that provides a common dialog for allowing users to specify a file name and location for saving a file. It's part of the Windows Forms library and is commonly used in Windows applications to prompt the user for a destination when they want to save a file.
- The SaveFileDialog is similar in concept to the OpenFileDialog but is specifically designed for file save operations.

Properties:

- **Filter:** Specifies the file type filter for the dialog. You can define a filter using a string in the format "Description|File Extension", where multiple filters are separated by vertical bars (|). For example, "Text files (*.txt)|*.txt|All files (*.*)|*.*".
- **Title:** Sets the title or caption of the save file dialog box.

- **InitialDirectory:** Defines the initial directory displayed when the dialog is opened.
- **FileName:** Gets or sets the name of the selected file, including the path.
- **CheckPathExists:** Specifies whether the dialog should check if the selected folder path exists.

Example:

```
using System;
using System.Windows.Forms;

namespace SaveFileDialogExample
{
   public partial class MainForm : Form
        {
       public MainForm()
       {
          InitializeComponent();
       }

       private void saveButton_Click(object sender, EventArgs e)
       {
          // Create an instance of SaveFileDialog
          SaveFileDialog saveFileDialog = new SaveFileDialog();

          // Set properties
          saveFileDialog.Title = "Save File";
          saveFileDialog.Filter = "Text Files (*.txt)|*.txt|All Files (*.*)|*.*";
          saveFileDialog.DefaultExt = "txt";

          // Show the dialog and check if the user clicked OK
          if (saveFileDialog.ShowDialog() == DialogResult.OK)
          {
              // Get the selected file path
              string selectedFilePath = saveFileDialog.FileName;

              // Handle the selected file here, e.g., save data to the file
              // In this example, we'll just display the selected file path
              selectedFileTextBox.Text = selectedFilePath;
          }
       }
        }
}
```

3. The Fontdialog control :

- In C#, the FontDialog is a standard Windows Forms dialog that allows users to select a font and customize its attributes for text display in an application.
- It is commonly used in Windows applications when you want to give users the ability to change the font settings for text.

Properties:

- **Font:** Gets or sets the selected font. You can use it to set an initial font or retrieve the user's font selection.
- **Color:** Gets or sets the selected color. You can use it to set an initial color or retrieve the user's color selection.
- **ShowColor:** Determines whether to show or hide the color selection controls in the dialog. If set to true, the user can choose a font color as well.
- **MaxSize:** Sets the maximum font size that the user can select.

- **MinSize:** Sets the minimum font size that the user can select.

Example:

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace FontDialogExample
{
   public partial class MainForm : Form
         {
      public MainForm()
      {
         InitializeComponent();
      }

      private void changeFontButton_Click(object sender, EventArgs e)
      {
         // Create an instance of the FontDialog
         FontDialog fontDialog = new FontDialog();

         // Set properties of the FontDialog
         fontDialog.Font = displayLabel.Font; // Set the initial font from a control
(optional)
         fontDialog.Color = displayLabel.ForeColor; // Set the initial font color
from a control (optional)

         // Show color selection controls in the dialog
         fontDialog.ShowColor = true;

         // Show the FontDialog and capture the result
         DialogResult result = fontDialog.ShowDialog();

         // Check if the user clicked OK in the dialog
         if (result == DialogResult.OK)
```

```
            {
                // Apply the selected font and color to a label control
                displayLabel.Font = fontDialog.Font;
                displayLabel.ForeColor = fontDialog.Color;
            }
        }
          }
}
```

4. The Colordialog Control:

- In C#, the ColorDialog is a standard Windows Forms dialog that allows users to select a color. It's commonly used in Windows applications when you want to give users the ability to choose a color for various purposes, such as text color, background color, or graphics drawing.

Properties:

- **Color:** Gets or sets the selected color. You can use it to set an initial color or retrieve the user's color selection.
- **CustomColors:** An array of custom colors that you can predefine for the user to select from. You can populate this array with specific colors to give users quick access to commonly used colors.
- **FullOpen:** If set to true, the dialog box displays all available colors in a larger area. If set to false, it shows only the basic colors.
- **AnyColor:** Determines whether the dialog should allow the user to define a custom color by clicking outside the predefined color palette.
- **SolidColorOnly:** If set to true, restricts color selection to solid (non-transparent) colors.
- **ShowHelp:** Determines whether the Help button is displayed in the dialog.

Example:

```
using System;
using System.Windows.Forms;

namespace ColorDialogExample
{
   public partial class MainForm : Form
```

```
    {
        public MainForm()
        {
            InitializeComponent();
        }

        private void chooseColorButton_Click(object sender, EventArgs e)
        {
            // Create a new ColorDialog instance
            ColorDialog colorDialog = new ColorDialog();

            // Show the color dialog and check if the user selected a color
            if (colorDialog.ShowDialog() == DialogResult.OK)
            {
                // Set the selected color to a TextBox (or any other control)
                selectedColorTextBox.BackColor = colorDialog.Color;
            }
        }
    }
}
```

5. The PrintDialog Control:

- C#, the PrintDialog is a standard Windows Forms dialog that allows users to select a printer and configure printing options before sending a document or content to the printer.

Properties :

- **Displaying the Dialog:** Show the PrintDialog using the ShowDialog method. This method displays the dialog as a modal dialog box.
- **PrinterSettings:** Gets or sets the printer settings, including the selected printer, page orientation, paper size, and more.
- **AllowSelection:** If set to true, allows the user to select a portion of the document or content to print (useful for applications that support page selection).
- **AllowSomePages:** If set to true, allows the user to specify a range of pages to print.
- **PrintToFile:** If set to true, allows the user to print to a file instead of a physical printer.

Example :

```
using System;
using System.Windows.Forms;

namespace PrintDialogExample
{
   public partial class MainForm : Form
         {
       public MainForm()
       {
          InitializeComponent();
       }

       private void printButton_Click(object sender, EventArgs e)
       {
          // Create an instance of the PrintDialog
          PrintDialog printDialog = new PrintDialog();

          // Configure properties of the PrintDialog
          printDialog.AllowSelection = true; // Allow user to select a portion of the
content
          printDialog.AllowSomePages = true; // Allow user to specify a page range

          // Show the PrintDialog and capture the result
          DialogResult result = printDialog.ShowDialog();

          // Check if the user clicked OK in the dialog
          if (result == DialogResult.OK)
          {
                 // Retrieve the printer settings
                 PrinterSettings printerSettings = printDialog.PrinterSettings;

                 // Handle the selected printer and print settings here
                 // In this example, we'll just display the selected printer's name
                 selectedPrinterLabel.Text = "Selected Printer: " +
printerSettings.PrinterName;
             }
```

```
        }
          }
}
```

4.7 Menu controls in c#:

1. The Context Menu Control:

- In C#, a context menu is a type of menu that appears when a user right-clicks on a control or an area of a user interface.
- It provides a set of context-specific options and actions that the user can perform based on the context of their interaction. You can create a context menu using the ContextMenuStrip class in Windows Forms applications.

Properties:

- **Items:** Gets a collection of items (e.g., ToolStripMenuItem) contained within the ContextMenuStrip. You can add and remove items dynamically.
- **BackColor:** Sets or gets the background color of the ContextMenuStrip.
- **Font:** Sets or gets the font used for text in the ContextMenuStrip.
- **ForeColor:** Sets or gets the color of the text in the ContextMenuStrip.
- **Name:** Specifies the name of the ContextMenuStrip control.

Example:

```
using System;
using System.Windows.Forms;

namespace ContextMenuExample
{
   public partial class MainForm : Form
   {
      private ContextMenuStrip contextMenuStrip1;
      private ToolStripMenuItem menuItemCopy;
      private ToolStripMenuItem menuItemCut;
      private ToolStripMenuItem menuItemPaste;

      public MainForm()
```

```
{
    InitializeComponent();
    InitializeContextMenu();
}

private void InitializeContextMenu()
{
    // Create a ContextMenuStrip
    contextMenuStrip1 = new ContextMenuStrip();

    // Create menu items and add them to the ContextMenuStrip
    menuItemCopy = new ToolStripMenuItem("Copy");
    menuItemCopy.Click += CopyMenuItem_Click;
    contextMenuStrip1.Items.Add(menuItemCopy);

    menuItemCut = new ToolStripMenuItem("Cut");
    menuItemCut.Click += CutMenuItem_Click;
    contextMenuStrip1.Items.Add(menuItemCut);

    menuItemPaste = new ToolStripMenuItem("Paste");
    menuItemPaste.Click += PasteMenuItem_Click;
    contextMenuStrip1.Items.Add(menuItemPaste);

    // Associate the ContextMenuStrip with a control
    textBox1.ContextMenuStrip = contextMenuStrip1;
}

private void CopyMenuItem_Click(object sender, EventArgs e)
{
    // Implement copy action here
    textBox1.Copy();
}

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

```
            // Implement cut action here
            textBox1.Cut();
        }

        private void PasteMenuItem_Click(object sender, EventArgs e)
        {
            // Implement paste action here
            textBox1.Paste();
        }
    }
}
```

2. The MenuStrip Control :

- In C#, the MenuStrip is a Windows Forms control that provides a traditional menu system similar to the menus found in many desktop applications. It allows you to create top-level menus and submenus, and you can associate these menus with various actions or commands in your application.

Properties :

- **Items:** Gets a collection of menu items (ToolStripMenuItem) contained within the MenuStrip.
- **RenderMode:** Specifies how the MenuStrip is rendered. Options include Professional, System, and ManagerRenderMode (custom rendering).
- **BackgroundImage:** Sets or gets the background image of the MenuStrip.
- **Dock:** Determines how the MenuStrip is docked within its parent container (e.g., Top, Bottom, Left, Right, or None).
- **Enabled:** Indicates whether the MenuStrip is enabled or disabled.

Example:

```
using System;
using System.Windows.Forms;

namespace SimpleMenuStripExample
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
            InitializeMenuStrip();
        }

        private void InitializeMenuStrip()
        {
            // Create a MenuStrip control
            MenuStrip menuStrip = new MenuStrip();

            // Create a single menu item
            ToolStripMenuItem menuItem = new ToolStripMenuItem("File");

            // Add the menu item to the MenuStrip
            menuStrip.Items.Add(menuItem);

            // Associate the MenuStrip with the form
            this.MainMenuStrip = menuStrip;

            // Add the MenuStrip to the form's controls
            this.Controls.Add(menuStrip);
        }

                static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }
}
```

3. The Status strip Control :

In C#, a StatusStrip is a Windows Forms control used to display status information, progress, or other relevant details to the user typically at the bottom of a form. It is often used to provide feedback or to show the state of an application.

Properties :

- **Items:** Gets a collection of items (e.g., ToolStripStatusLabel controls) contained within the StatusStrip.
- **SizingGrip:** Gets or sets whether a sizing grip is displayed in the lower-right corner of the StatusStrip to allow the user to resize the form.
- **RenderMode:** Specifies how the StatusStrip is rendered. Options include Professional, System, and ManagerRenderMode (custom rendering).
- **BackColor:** Sets or gets the background color of the StatusStrip.
- **Font:** Sets or gets the font used for text in the StatusStrip.

Example:

```
using System;
using System.Windows.Forms;

namespace SimpleStatusStripExample
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
            InitializeStatusStrip();
        }

        private void InitializeStatusStrip()
        {
            // Create a StatusStrip control
            StatusStrip statusStrip = new StatusStrip();

            // Create a ToolStripStatusLabel
            ToolStripStatusLabel toolStripStatusLabel = new ToolStripStatusLabel();
            toolStripStatusLabel.Text = "Ready";

            // Add the ToolStripStatusLabel to the StatusStrip
            statusStrip.Items.Add(toolStripStatusLabel);

            // Add the StatusStrip to the form's controls
            this.Controls.Add(statusStrip);
        }

        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }
}
```

4. The Tool strip control :

- In C#, a ToolStrip is a Windows Forms control that provides a toolbar-like interface with various tools and controls for performing actions in an application.
- It's commonly used to create toolbars, menu bars, and other interactive elements in a Windows Forms application.

Properties:

- **Items:** Gets a collection of items (e.g., ToolStripButton, ToolStripLabel) contained within the ToolStrip.
- **ImageList:** Gets or sets the ImageList used for assigning images to ToolStrip items.
- **RenderMode:** Specifies how the ToolStrip is rendered. Options include Professional, System, and ManagerRenderMode (custom rendering).
- **GripStyle:** Determines whether and how the grip (the moveable area) is displayed in the ToolStrip.

- **BackColor:** Sets or gets the background color of the ToolStrip.

Example :

```
using System;
using System.Windows.Forms;

namespace SimpleToolStripExample
{
   public partial class MainForm : Form
   {
      private ToolStrip toolStrip;
      private ToolStripButton toolStripButton;

      public MainForm()
      {
         InitializeComponent();
         InitializeToolStrip();
      }

      private void InitializeToolStrip()
      {
         // Create a ToolStrip control
         toolStrip = new ToolStrip();

         // Create a ToolStripButton
         toolStripButton = new ToolStripButton("Click Me");
         toolStripButton.Click += ToolStripButton_Click;

         // Add the ToolStripButton to the ToolStrip
         toolStrip.Items.Add(toolStripButton);

         // Add the ToolStrip to the form's controls
         this.Controls.Add(toolStrip);
      }

      private void ToolStripButton_Click(object sender, EventArgs e)
```

```
        {
            // Implement button action here
            MessageBox.Show("Button clicked!");
        }
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }}
```

4.8 How to add third party control in c#...? :

https://www.youtube.com/watch?v=QJNuOHjIS6Y