Unit 2

Benefits

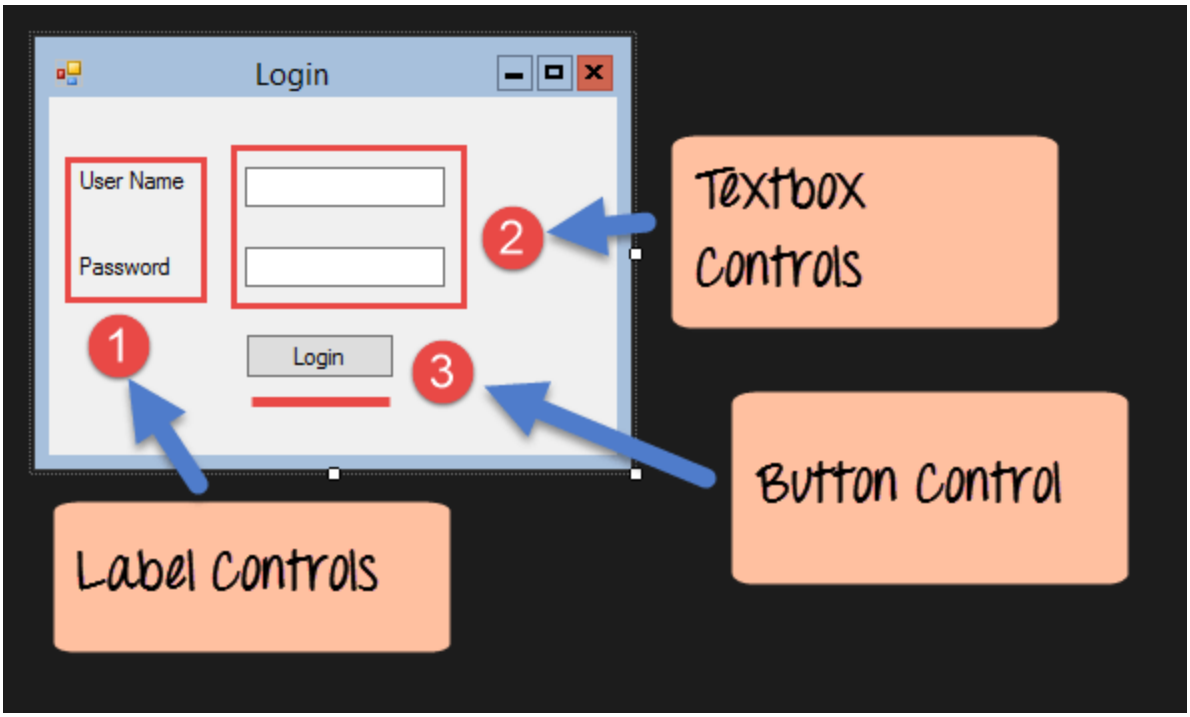
- It helps in developing better applications.
- It brings a much better user interface.
- It helps in making the look of the application more appealing.
- It creates a balance between the functionality as well as the appearance of the application.
- It reduces the task of merging various GUI components to the application.
- It comes with rigid support for GUI components.
- It helps in bringing the best out of the application developer.
- It makes the task of adding GUI components very easy.
- It creates an easy way of getting the GUI in place appropriate for the application.
- Efficient way to provide efficiently working GUI also for older computers.

Windows Form

Windows Forms Basics

A Windows forms application is one that runs on the desktop computer. A Windows forms application will normally have a collection of controls such as labels, textboxes, list boxes, etc.

Below is an example of a simple Windows form application C#. It shows a simple Login screen, which is accessible by the user. The user will enter the required credentials and then will click the Login button to proceed.



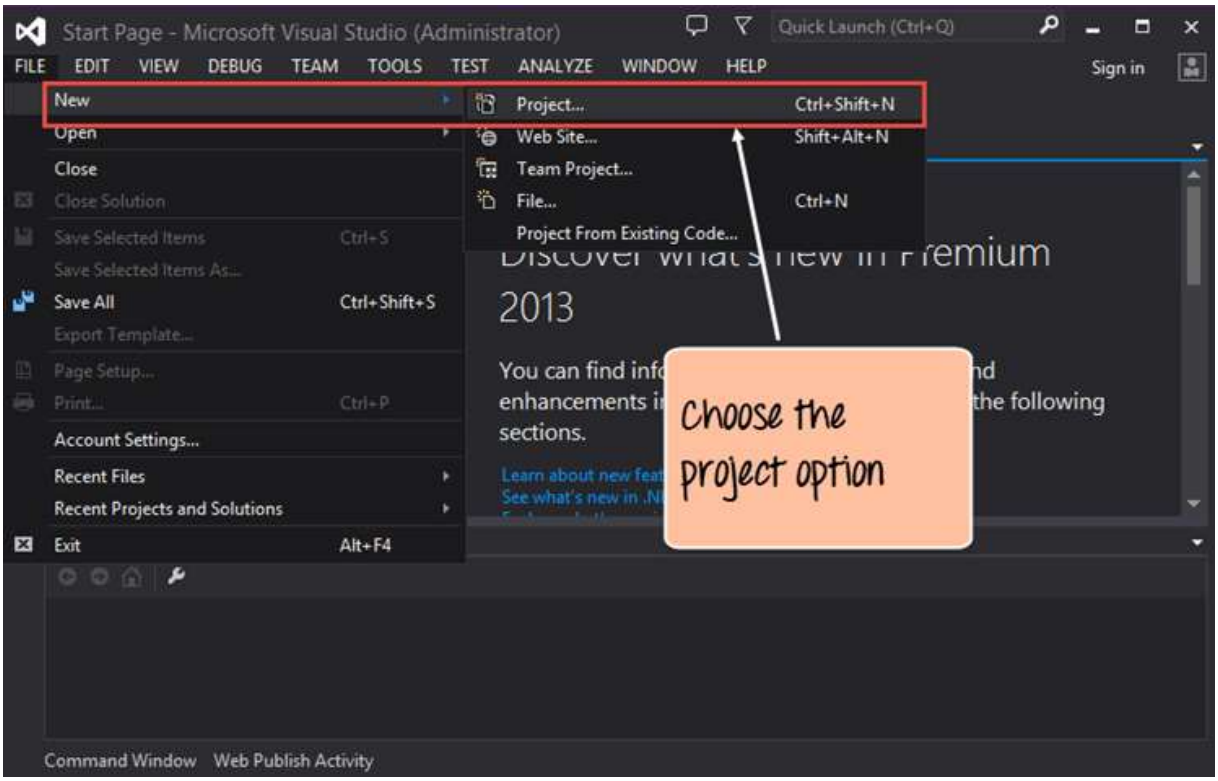
So an example of the controls available in the above application

1. This is a collection of label controls which are normally used to describe adjacent controls. So in our case, we have 2 textboxes, and the labels are used to tell the user that one textbox is for entering the user name and the other for the password.
2. The 2 textboxes are used to hold the username and password which will be entered by the user.
3. Finally, we have the button control. The button control will normally have some code attached to perform a certain set of actions. So for example in the above case, we could have the button perform an action of validating the user name and password which is entered by the user.

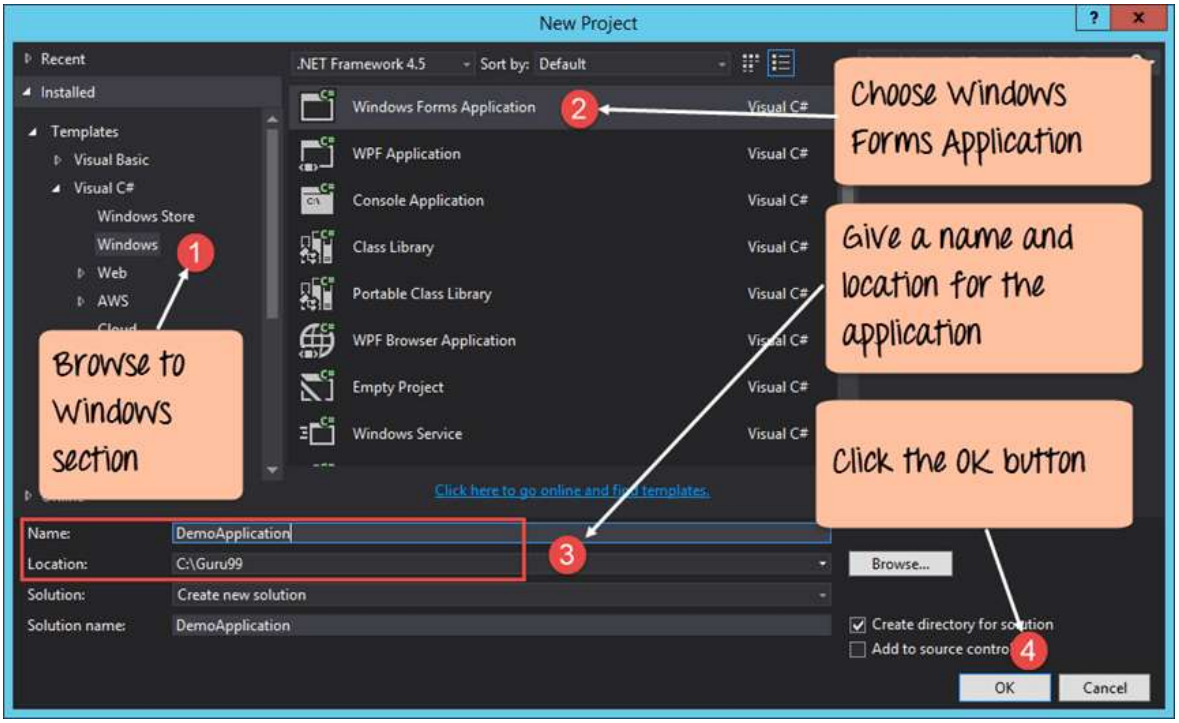
C# Hello World

Now let's look at an example of how we can implement a simple 'hello world' application in Visual Studio. For this, we would need to implement the below-mentioned steps

Step 1) The first step involves the creation of a new project in Visual Studio. After launching Visual Studio, we need to choose the menu option New->Project.



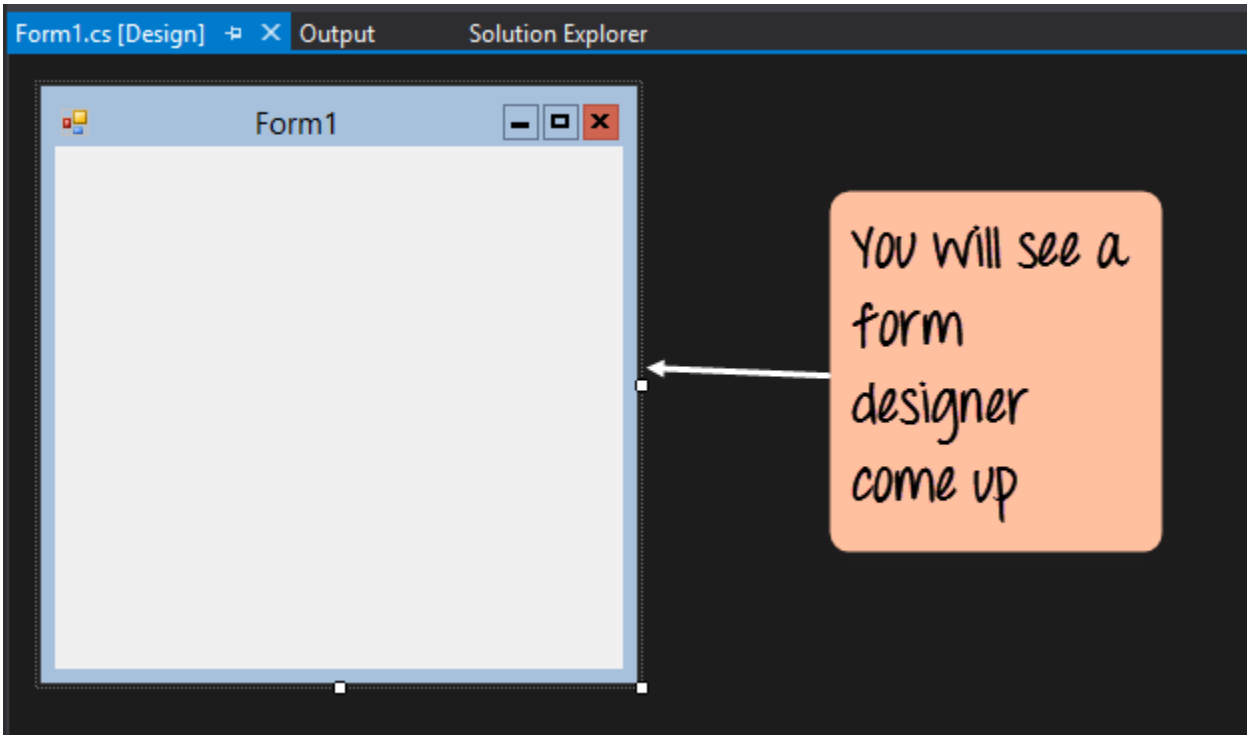
Step 2) The next step is to choose the project type as a Windows Forms application. Here we also need to mention the name and location of our project.



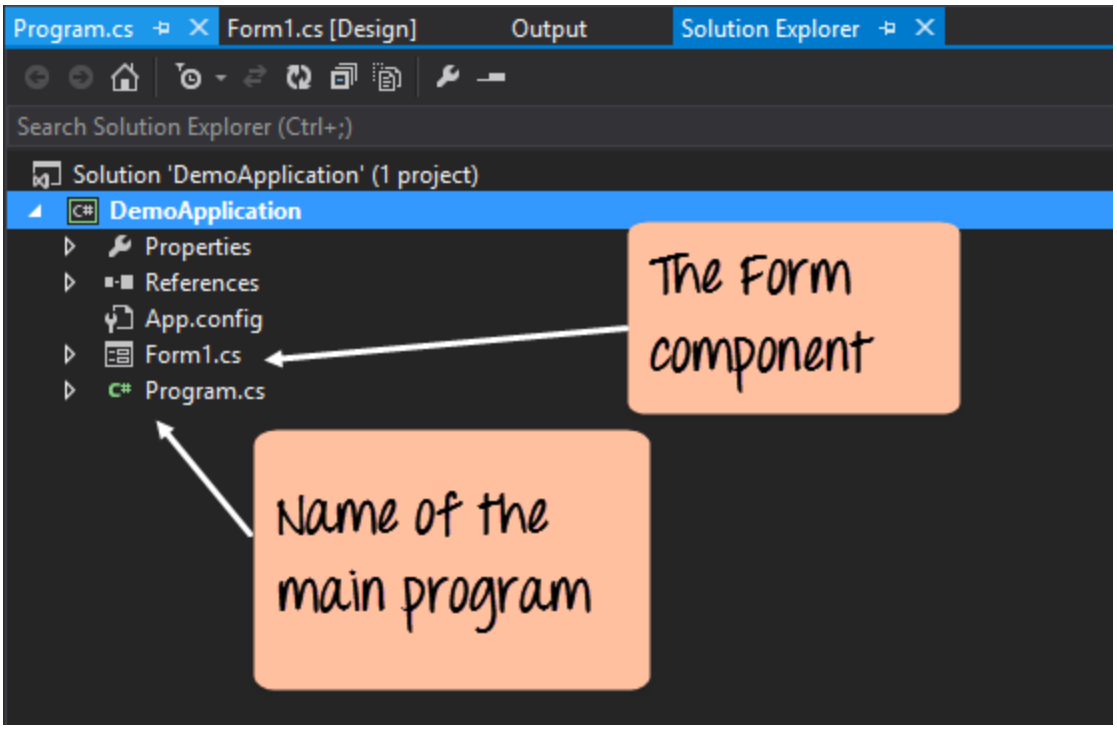
1. In the project dialog box, we can see various options for creating different types of projects in Visual Studio. Click the Windows option on the left-hand side.
2. When we click the Windows options in the previous step, we will be able to see an option for Windows Forms Application. Click this option.
3. We will give a name for the application. In our case, it is DemoApplication. We will also provide a location to store our application.
4. Finally, we click the 'OK' button to let Visual Studio create our project.

If the above steps are followed, we will get the below output in Visual Studio.

Output:-



We will see a Form Designer displayed in Visual Studio. It's in this Form Designer that we will start building our Windows Forms application.

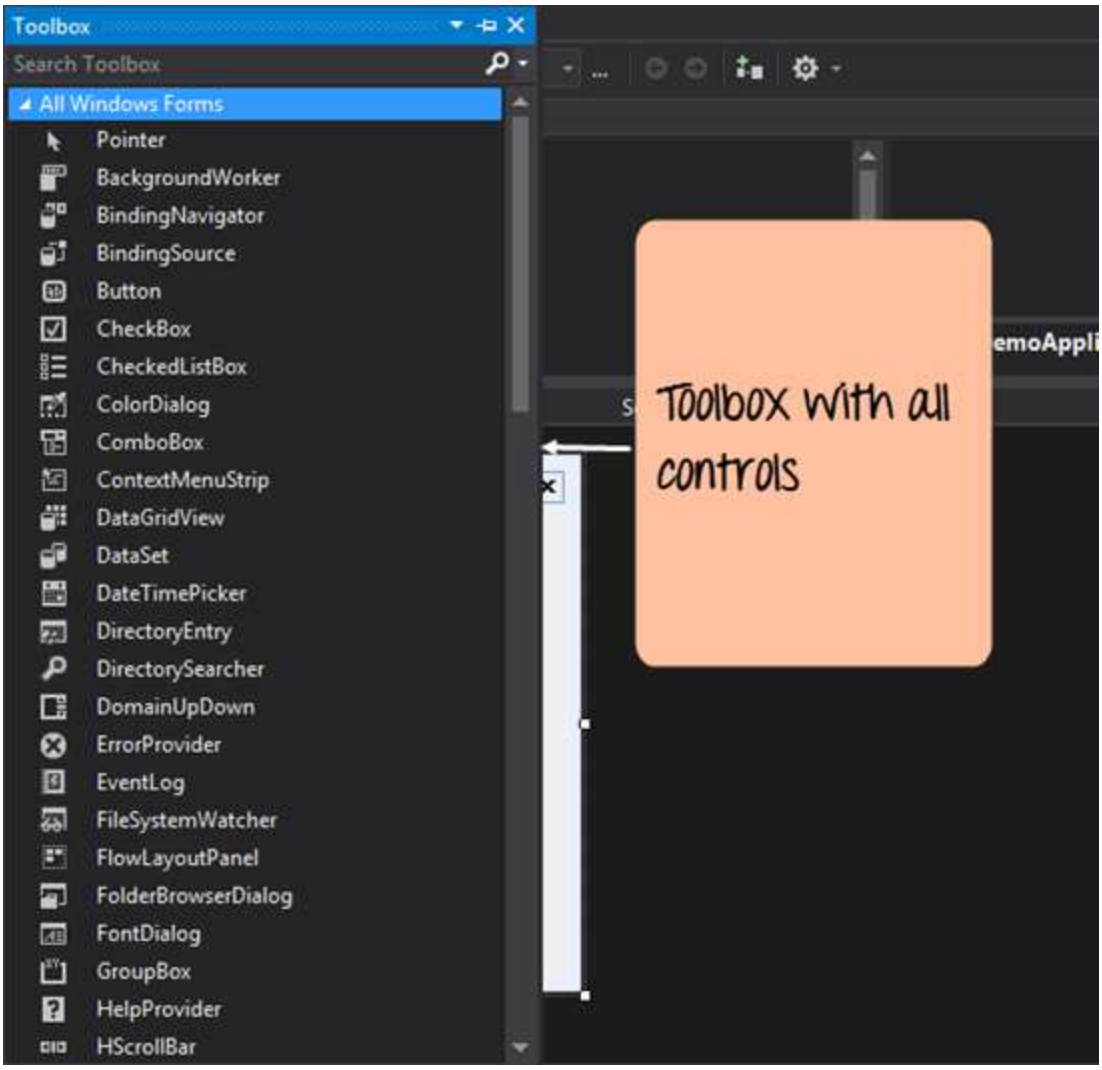


In the Solution Explorer, we will also be able to see the DemoApplication Solution. This solution will contain the below 2 project files

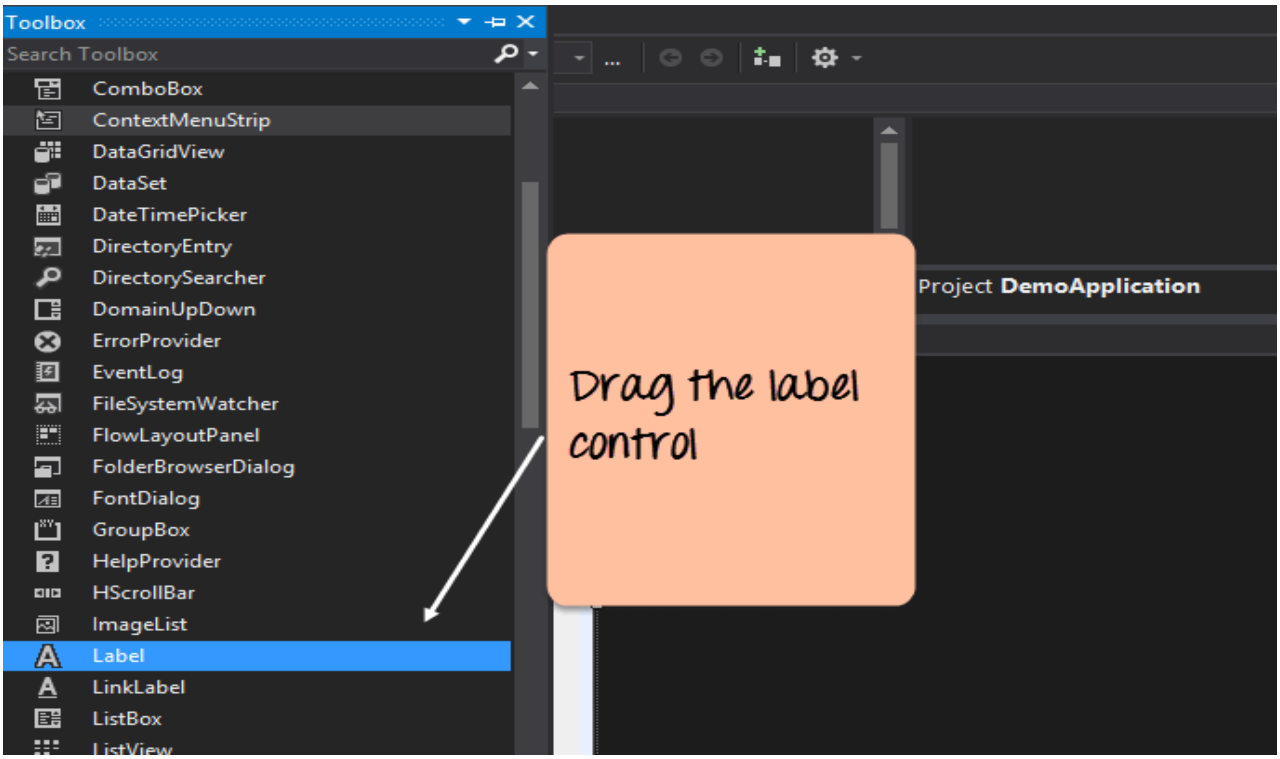
1. A Form application called Forms1.cs. This file will contain all of the code for the Windows Form application.
2. The Main program called Program.cs is default code file which is created when a new application is created in Visual Studio. This code will contain the startup code for the application as a whole.

On the left-hand side of Visual Studio, we will also see a ToolBox. The toolbox contains all the controls which can be added to a Windows Forms. Controls like a text box or a label are just some of the controls which can be added to a Windows Forms.

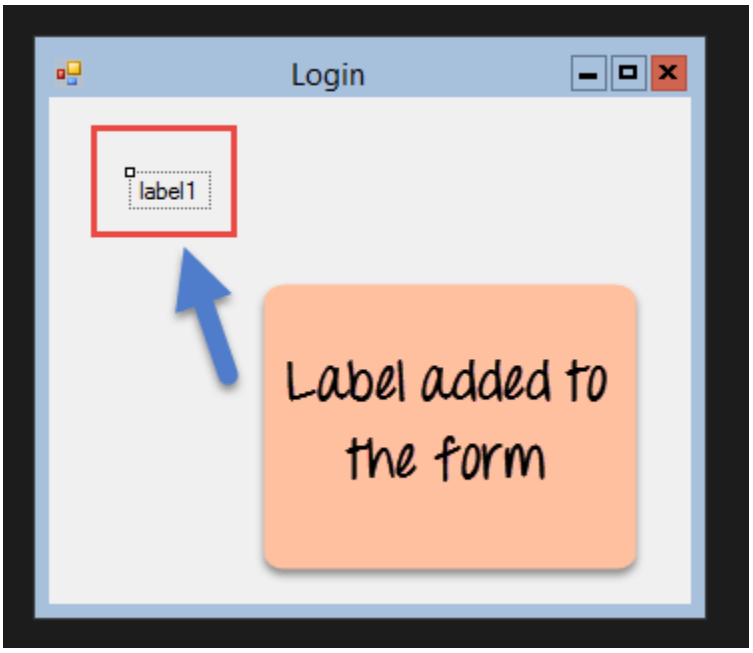
Below is a screenshot of how the Toolbox looks like.



Step 3) In this step, we will now add a label to the Form which will display "Hello World." From the toolbox, we will need to choose the Label control and simply drag it onto the Form.

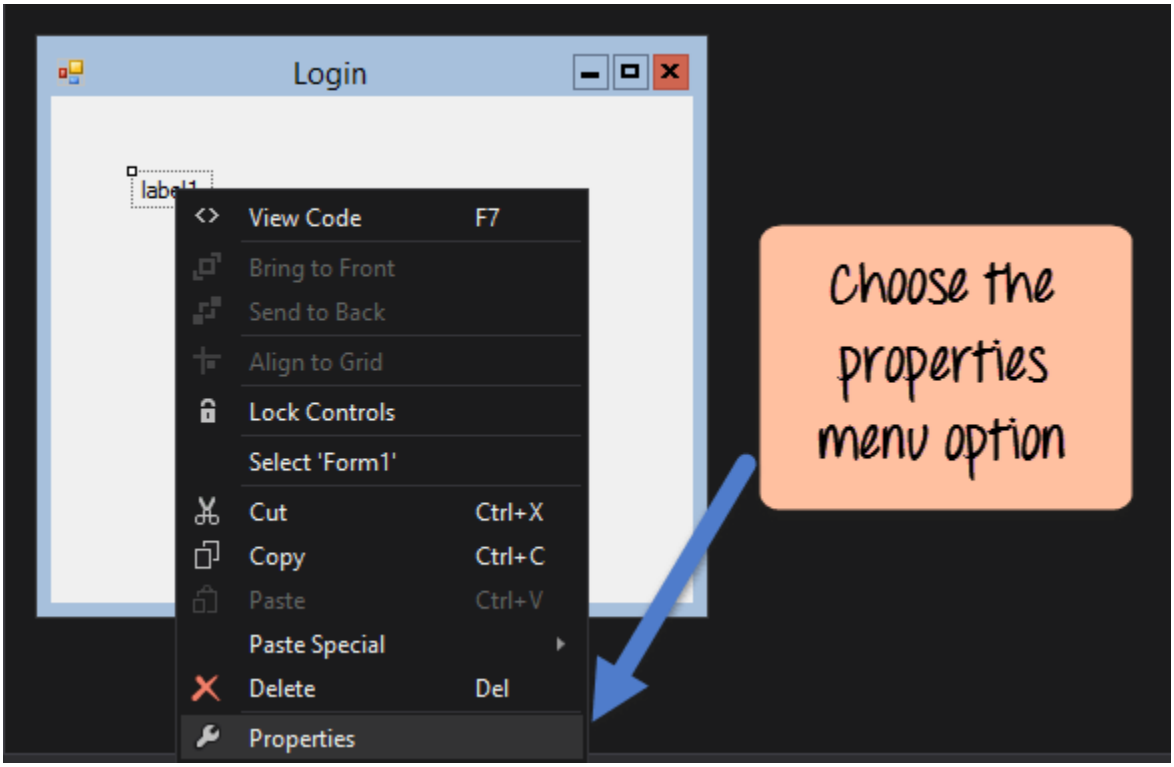


Once we drag the label to the form, we can see the label embedded on the form as shown below.

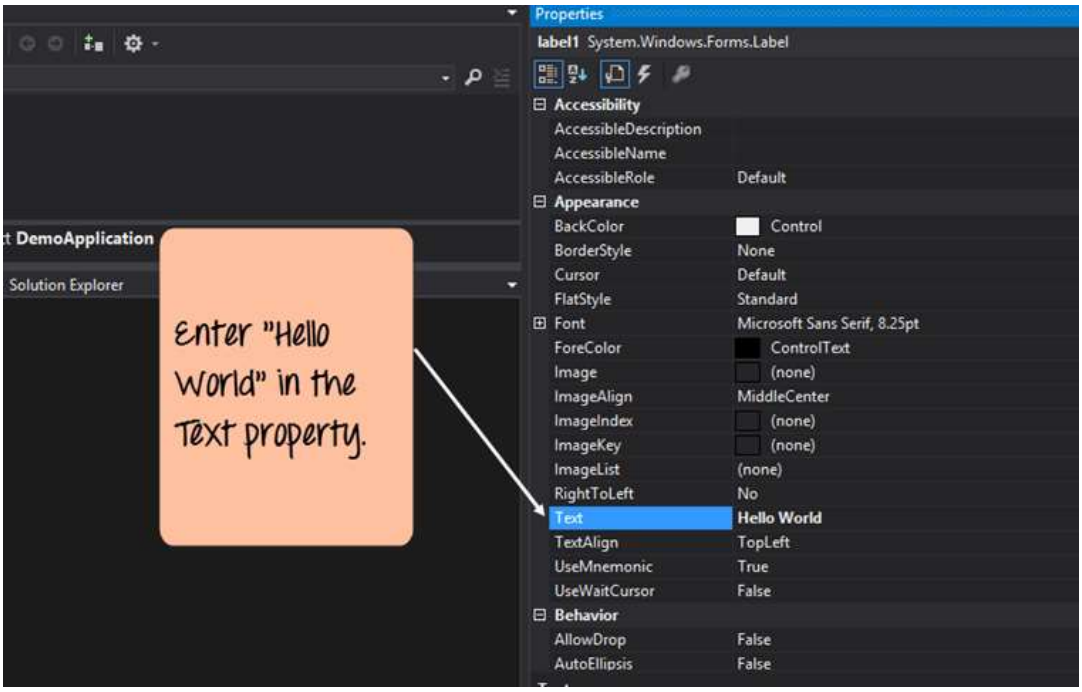


Step 4) The next step is to go to the properties of the control and Change the text to 'Hello World'.

To go to the properties of a control, we need to right-click the control and choose the Properties menu option

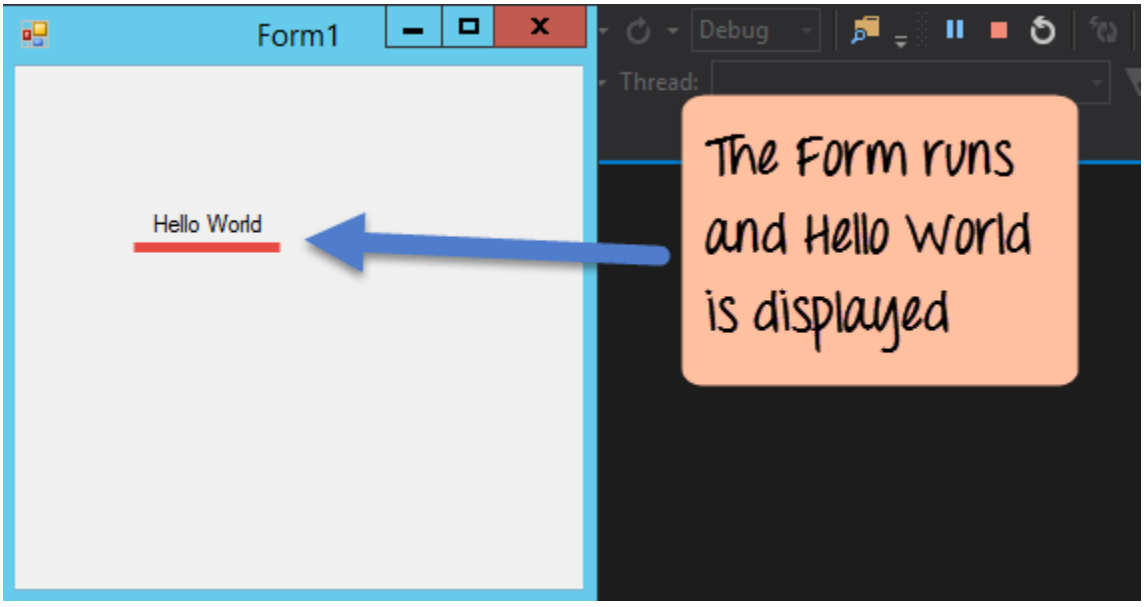


- The properties panel also shows up in Visual Studio. So for the label control, in the properties control, go to the Text section and enter “Hello World”.
- Each Control has a set of properties which describe the control.



If we follow all of the above steps and run our program in Visual Studio, we will get the following output

Output:-



In the output, we can see that the Windows Form is displayed. We can also see 'Hello World' is displayed on the form.

Adding Controls to a form

We had already seen how to add a control to a form when we added the label control in the earlier section to display “Hello World.”

Let’s look at the other controls available for Windows forms and see some of their common properties.

In our Windows form application in C# examples, we will create one form which will have the following functionality.

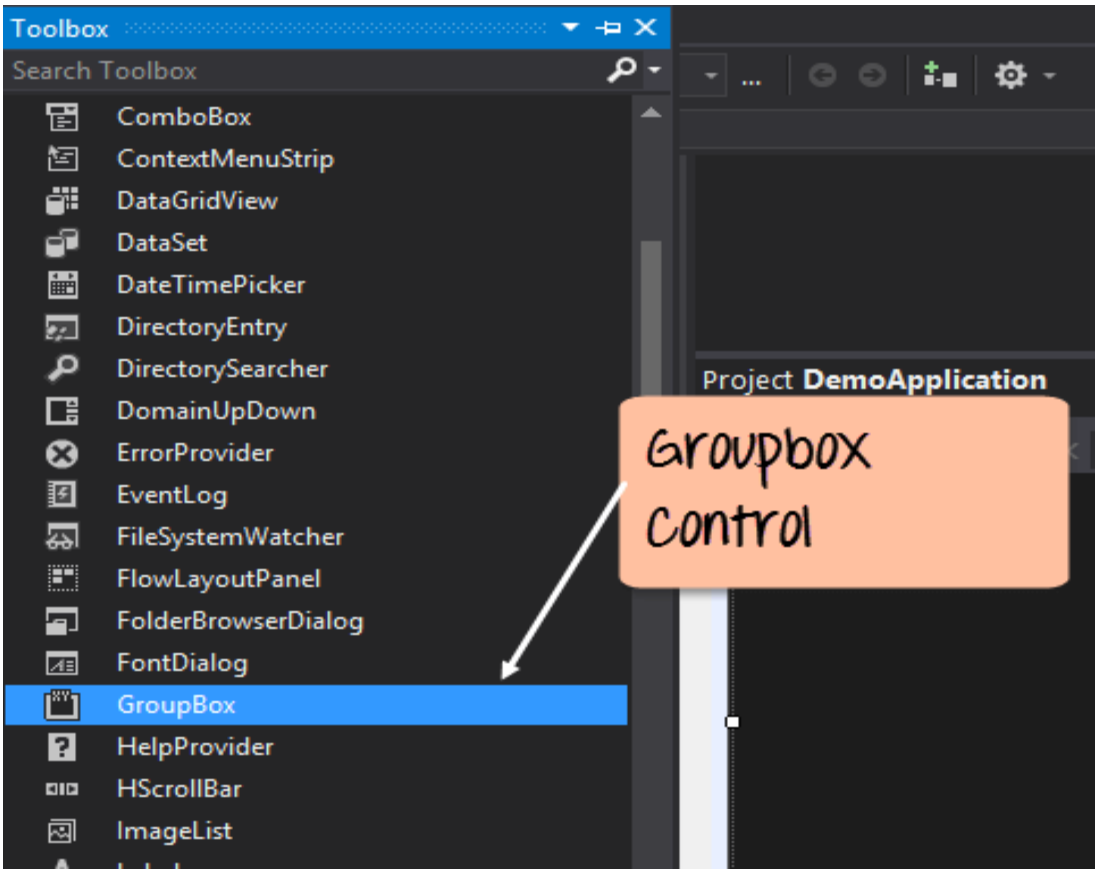
1. The ability for the user to enter name and address.
2. An option to choose the city in which the user resides in
3. The ability for the user to enter an option for the gender.
4. An option to choose a course which the user wants to learn. There will make choices for both C# and ASP.Net

So let’s look at each control in detail and add them to build the form with the above-mentioned functionality.

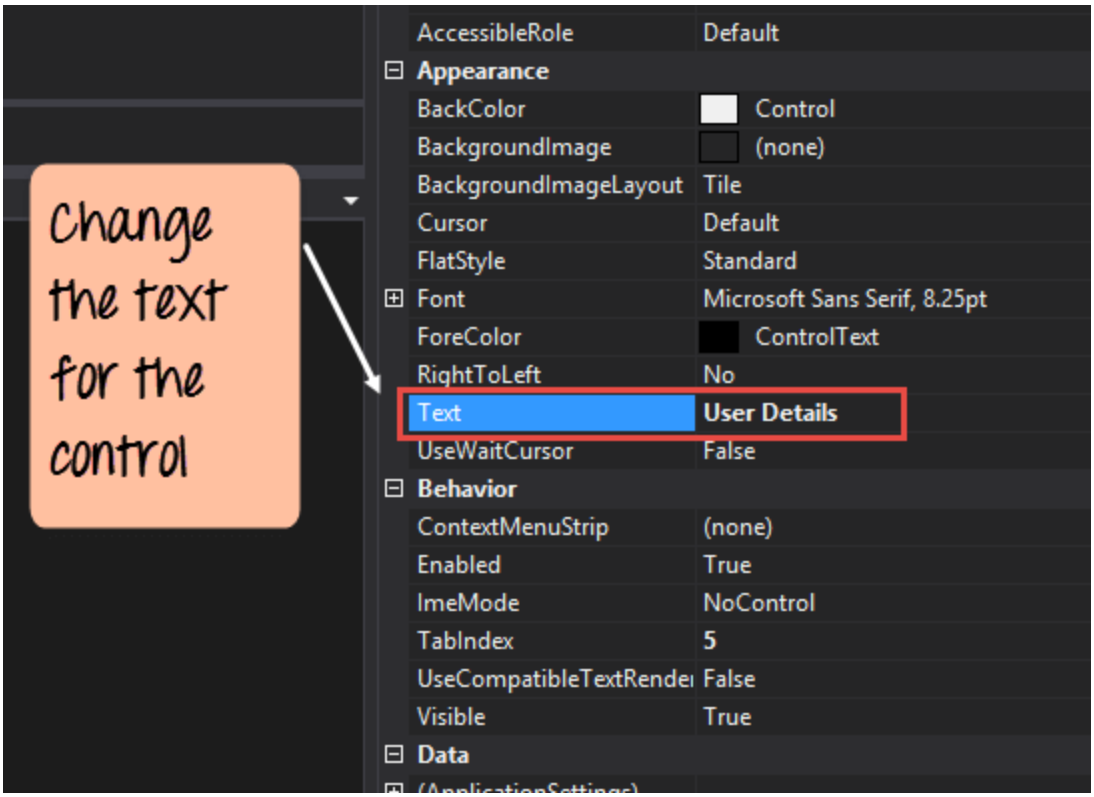
Group Box

A group box is used for logical grouping controls into a section. Let’s take an example if we had a collection of controls for entering details such as name and address of a person. Ideally, these are details of a person, so we would want to have these details in a separate section on the Form. For this purpose, we can have a group box. Let’s see how we can implement this with an example shown below

Step 1) The first step is to drag the Groupbox control onto the Windows Form from the toolbox as shown below

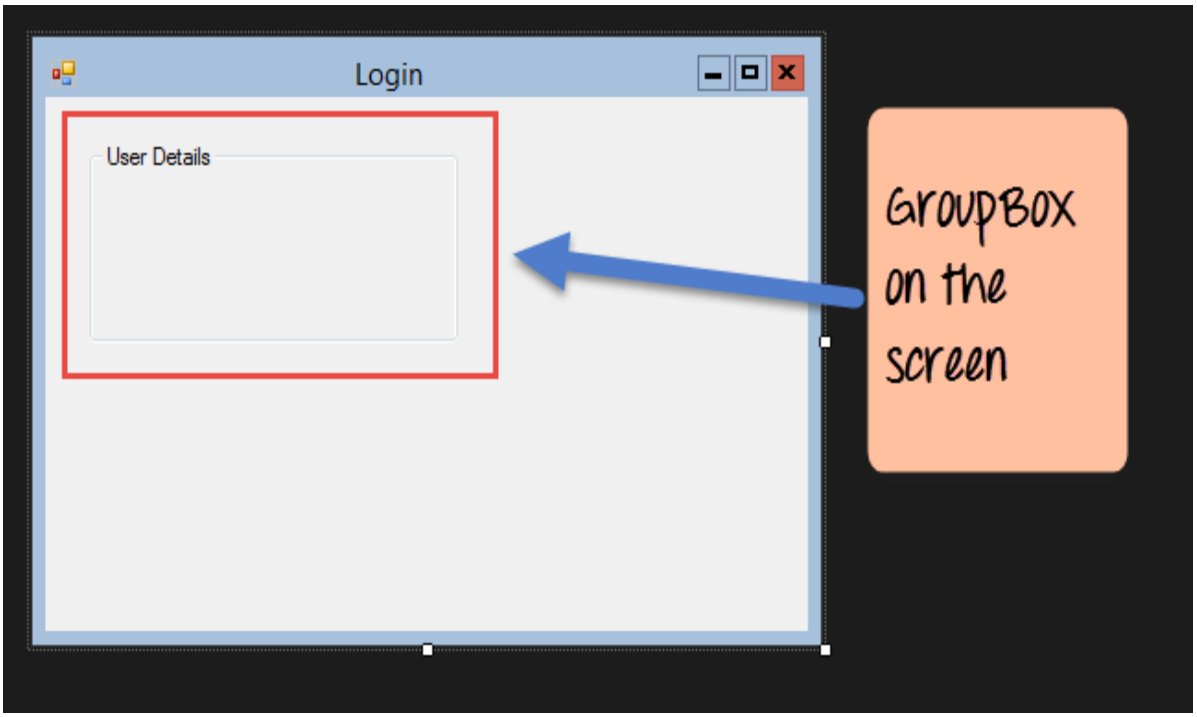


Step 2) Once the groupbox has been added, go to the properties window by clicking on the groupbox control. In the properties window, go to the Text property and change it to "User Details".



Once we make the above changes, we will see the following output

Output:-



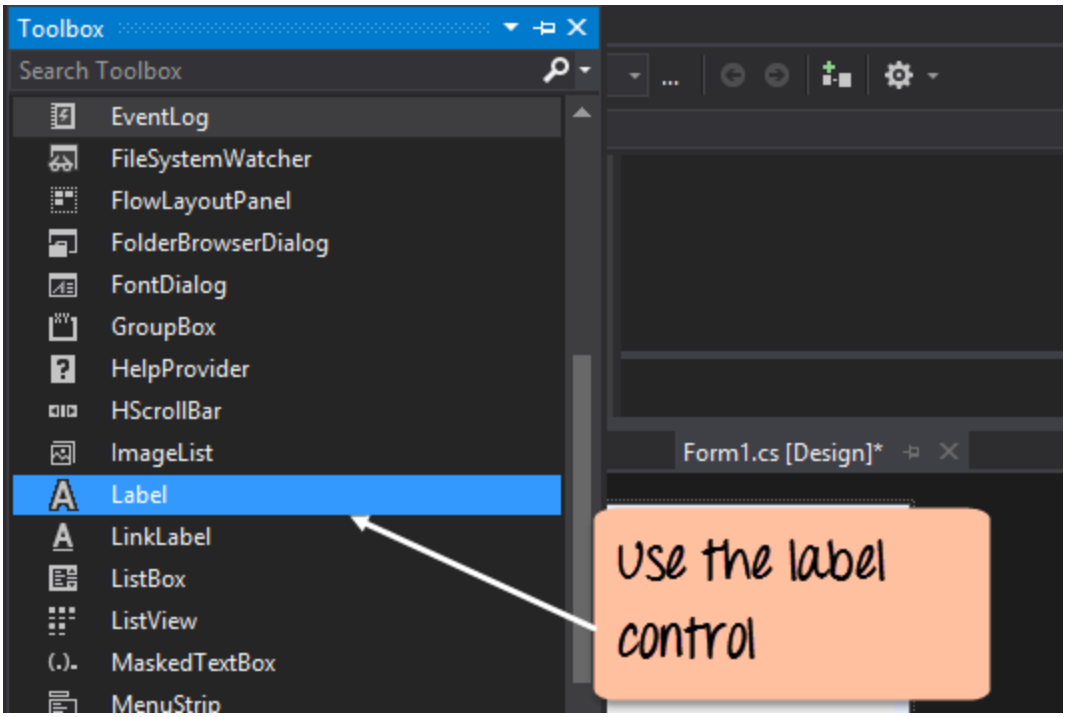
In the output, we can clearly see that the Groupbox was added to the form. We can also see that the text of the groupbox was changed to “User Details.”

Label Control

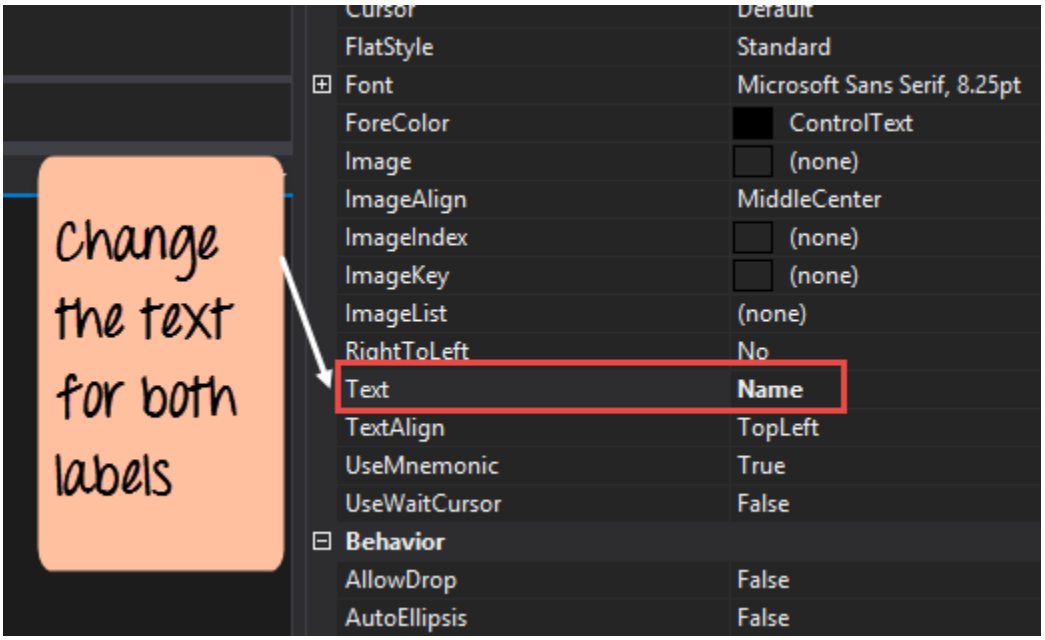
Next comes the Label Control. The label control is used to display a text or a message to the user on the form. The label control is normally used along with other controls. Common examples are wherein a label is added along with the textbox control.

The label indicates to the user on what is expected to fill up in the textbox. Let’s see how we can implement this with an example shown below. We will add 2 labels, one which will be called ‘name’ and the other called ‘address.’ They will be used in conjunction with the textbox controls which will be added in the later section.

Step 1) The first step is to drag the label control on to the Windows Form from the toolbox as shown below. Make sure we drag the label control 2 times so that we can have one for the ‘name’ and the other for the ‘address’.

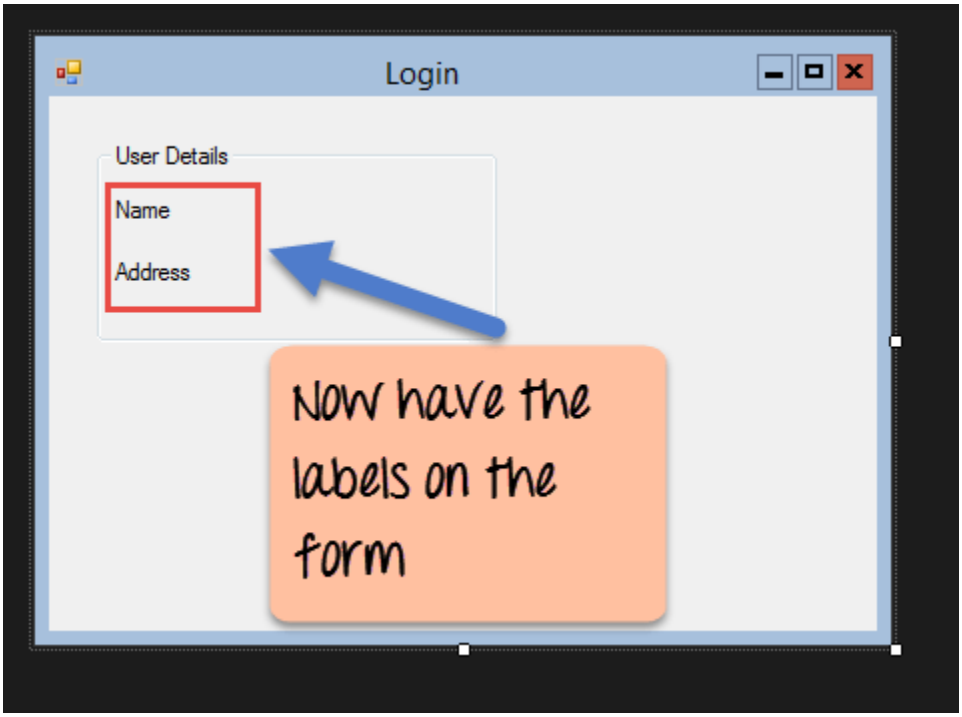


Step 2) Once the label has been added, go to the properties window by clicking on the label control. In the properties window, go to the Text property of each label control.



Once we make the above changes, we will see the following output

Output:-

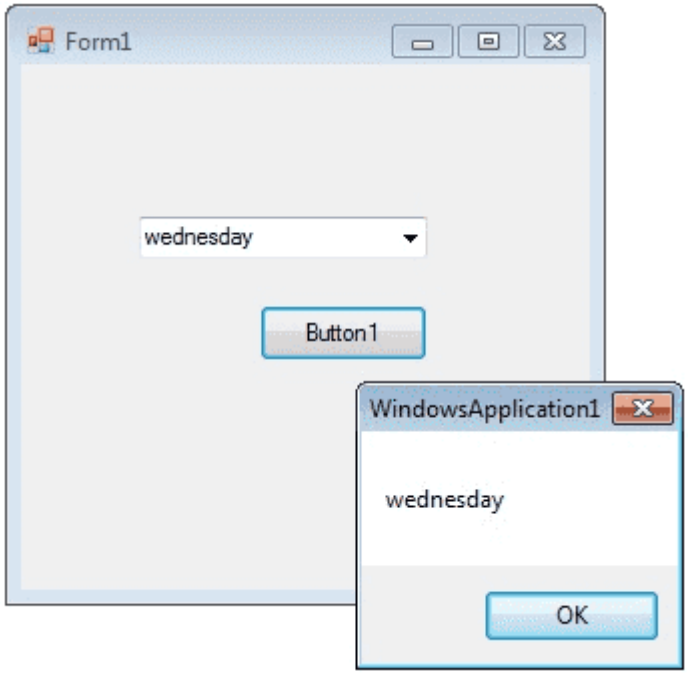


we can see the label controls added to the form.

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

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            label1.Text = "This is my first Lable";
            label1.BorderStyle = BorderStyle.FixedSingle;
            label1.TextAlign = ContentAlignment.MiddleCenter;
        }
    }
}
```

ComboBox Control

C# controls are located in the Toolbox of the development environment, and we use them to create objects on a form with a simple series of mouse clicks and dragging motions. A ComboBox displays a text box combined with a ListBox, which enables the user to select items from the list or enter a new value.



The user can type a value in the text field or click the button to display a drop down list. We can add individual objects with the Add method. We can delete items with the Remove method or clear the entire list with the Clear method.

Add a item to combobox

```
comboBox1.Items.Add("Sunday");
comboBox1.Items.Add("Monday");
comboBox1.Items.Add("Tuesday");
```

ComboBox SelectedItem

Retrieve value from ComboBox

If we want to retrieve the displayed item to a string variable , we can code like this

```
string var;

var = comboBox1.Text;

                    Or

var item = this.comboBox1.GetItemText(this.comboBox1.SelectedItem);

MessageBox.Show(item);
```

How to remove an item from ComboBox

We can remove items from a combobox in two ways. We can remove item at a the specified index or giving a specified item by name.

```
comboBox1.Items.RemoveAt(1);
```

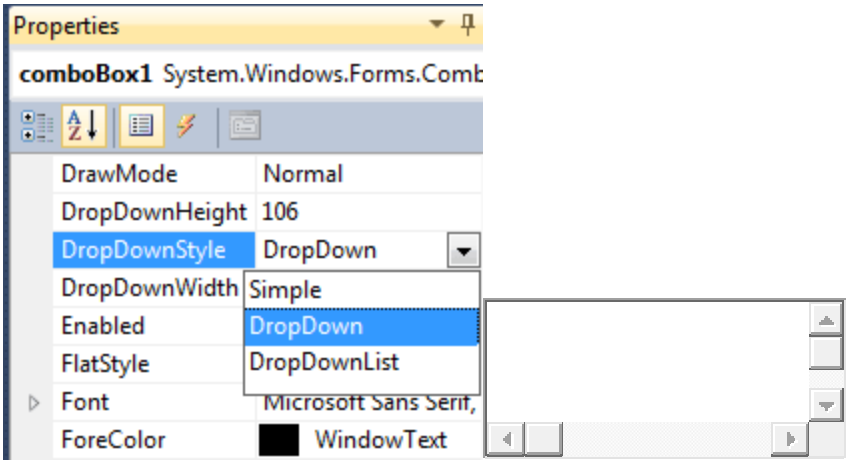
The above code will remove the second item from the combobox.

```
comboBox1.Items.Remove("Friday");
```

The above code will remove the item "Friday" from the combobox.

DropDownStyle

The DropDownStyle property specifies whether the list is always displayed or whether the list is displayed in a drop-down. The DropDownStyle property also specifies whether the text portion can be edited.



ComboBox Selected Value

Set the selected item in a comboBox

We can display selected item in a combobox in two ways.

```
comboBox1.Items.Add("test1");

comboBox1.Items.Add("test2");

comboBox1.Items.Add("test3");

comboBox1.SelectedItem = "test3";
```

or

```
comboBox1.SelectedIndex = comboBox1.FindStringExact("test3");
```

ComboBox DataSource Property

Populate a combo box with a DataSet

We can Programmatically Binding DataSource to ComboBox in a simple way..

Consider an sql string like...."select au_id,au_lname from authors";

Make a datasource and bind it like the following.

```
comboBox1.DataSource = ds.Tables[0];
comboBox1.ValueMember = "au_id";
comboBox1.DisplayMember = "au_lname";
```

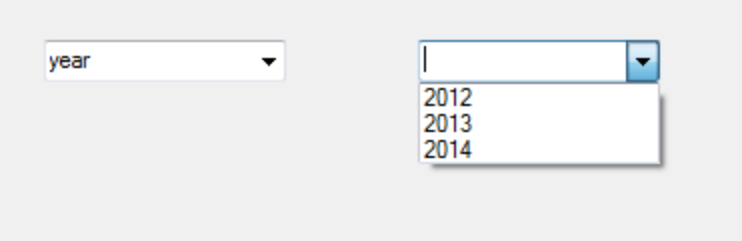
Combobox SelectedIndexChanged event

The SelectedIndexChanged event of a combobox fire when we change the slected item in a combobox. If we want to do something when we change the selection, we can write the program on SelectedIndexChanged event. From the following code we can understand how to set values in the SelectedIndexChanged event of a combobox. Drag and drop two combobox on the Form and copy and paste the following source code.

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

```
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {
            comboBox1.Items.Add("weekdays");
            comboBox1.Items.Add("year");
        }
        private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            comboBox2.Items.Clear();
            if (comboBox1.SelectedItem == "weekdays")
            {
                comboBox2.Items.Add("Sunday");
                comboBox2.Items.Add("Monday");
                comboBox2.Items.Add("Tuesday");
            }
            else if (comboBox1.SelectedItem == "year")
            {
                comboBox2.Items.Add("2012");
                comboBox2.Items.Add("2013");
                comboBox2.Items.Add("2014");
            }
        }
    }
}
```

Output



ComboBox Databinding

We can bind data to a Combobox from various resources like Dataset, List, Enum, Dictionary etc. From the following link we can study more about ... ComboBox Databinding

ComboBox Default Value

How to set a default value for a Combo Box

We can set combobox default value by using SelectedIndex property

```
comboBox1.SelectedIndex = 6;
```

Above code set 6th item as combobox default value

ComboBox readonly

How to make a combobox read only

We can make a ComboBox readonly, that means a user cannot write in a combo box but he can select the given items, in two ways. By default, DropDownStyle property of a Combobox is DropDown. In this case user can enter values to combobox. When we change the DropDownStyle property to DropDownList, the Combobox will become read only and user can not enter values to combobox. Second method, if we want the combobox completely read only, we can set comboBox1.Enabled = false.

ComboBox Example

The following C# source code add seven days in a week to a combo box while load event of a Windows Form and int Button click event it displays the selected text in the Combo Box.

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

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
```

```
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            comboBox1.Items.Add("Sunday");
            comboBox1.Items.Add("Monday");
            comboBox1.Items.Add("Tuesday");
            comboBox1.Items.Add("wednesday");
            comboBox1.Items.Add("Thursday");
            comboBox1.Items.Add("Friday");
            comboBox1.Items.Add("Saturday");
            comboBox1.SelectedIndex = comboBox1.FindStringExact("Sunday");

        }

        private void button1_Click(object sender, EventArgs e)
        {
            string var;
            var = comboBox1.Text;
            MessageBox.Show(var);
        }
    }
}
```

Textbox

A TextBox control is used to display, or accept as input, a single line of text. This control has additional functionality that is not found in the standard Windows text box control, including multiline editing and password character masking.

A text box object is used to display text on a form or to get user input while a C# program is running. In a text box, a user can type data or paste it into the control from the clipboard.

For displaying a text in a TextBox control

```
textBox1.Text = "http://csharp.net-informations.com";
```

For getting a text from a TextBox control

```
string var;

var = textBox1.Text;
```

C# TextBox Properties

We can set TextBox properties through Property window or through program. We can open Properties window by pressing F4 or right click on a control and select Properties menu item.

Textbox property window

The below code set a textbox width as 250 and height as 50 through source code.

```
textBox1.Width = 250;
textBox1.Height = 50;
```

Background Color and Foreground Color

We can set background color and foreground color through property window and programmatically.

```
textBox1.BackColor = Color.Blue;
textBox1.ForeColor = Color.White;
```

Textbox BorderStyle

We can set 3 different types of border style for textbox, they are None, FixedSingle and fixed3d.

textBox1.BorderStyle = BorderStyle.Fixed3D;

TextBox Events

- **Keydown event**

We can capture which key is pressed by the user using KeyDown event

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        MessageBox.Show("You press Enter Key");

    }
    if (e.KeyCode == Keys.CapsLock)
    {
        MessageBox.Show("You press Caps Lock Key");
    }
}
```

- **TextChanged Event**

When user input or setting the Text property to a new value raises the TextChanged event

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
    label1.Text = textBox1.Text;
}
```

Textbox Maximum Length

Sets the maximum number of characters or words the user can input into the text box control.

textBox1.MaxLength = 40;

Textbox ReadOnly

When a program wants to prevent a user from changing the text that appears in a text box, the program can set the controls Read-only property is to True.

textBox1.ReadOnly = true;

Multiline TextBox

We can use the Multiline and ScrollBars properties to enable multiple lines of text to be displayed or entered.

```
textBox1.Multiline = true;
```

Textbox password character

TextBox controls can also be used to accept passwords and other sensitive information. We can use the PasswordChar property to mask characters entered in a single line version of the control

```
textBox1.PasswordChar = '*';
```

The above code set the PasswordChar to * , so when the user enter password then it display only * instead of typed characters.

Newline in a TextBox

We can add new line in a textbox using many ways.

```
textBox1.Text += "our text" + "\r\n";
```

or

```
textBox1.Text += "our text" + Environment.NewLine;
```

Retrieve integer values from textbox

```
int i;
i = int.Parse (textBox1.Text);
```

Parse method Converts the string representation of a number to its integer equivalent.

String to Float conversion

```
float  i;
i = float.Parse (textBox1.Text);
```

String to Double conversion

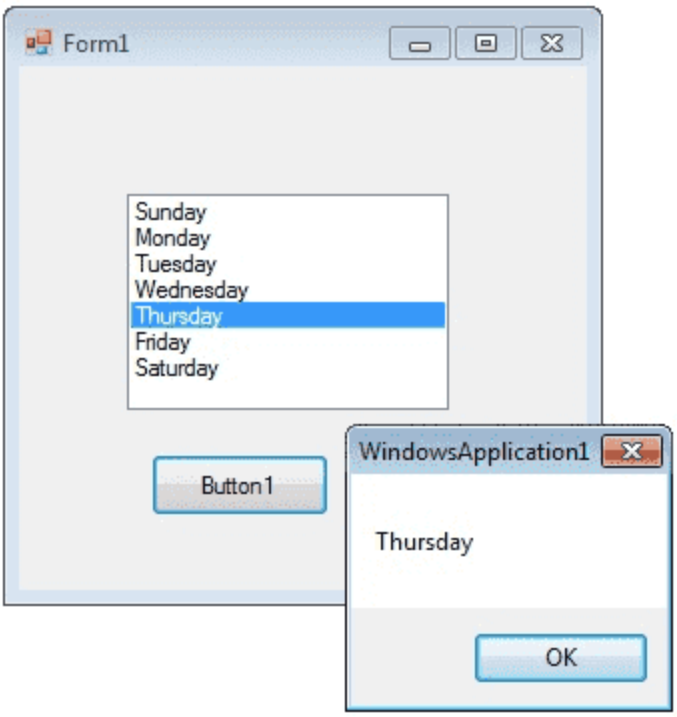
```
double   i;
i = float.Parse (textBox1.Text);
```

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

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
```

```
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            textBox1.Width = 250;
            textBox1.Height = 50;
            textBox1.Multiline = true;
            textBox1.BackColor = Color.Blue;
            textBox1.ForeColor = Color.White;
            textBox1.BorderStyle = BorderStyle.Fixed3D;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            string var;
            var = textBox1.Text;
            MessageBox.Show(var);
        }
    }
}
```

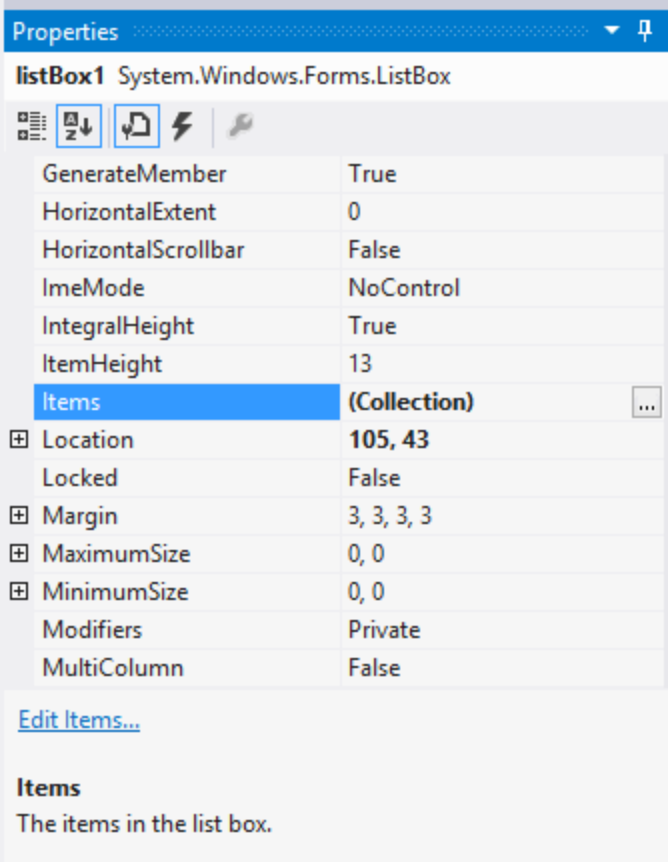
List box

The **ListBox control** enables we to display a list of items to the user that the user can select by clicking.



Setting ListBox Properties

We can set ListBox properties by using Properties Window. In order to get Properties window we can Press F4 or by right-clicking on a control to get the "Properties" menu item.



We can set property values in the right side column of the property window.

C# Listbox example

Add Items in a Listbox

Syntax

public int Add (object item);

In addition to display and selection functionality, the **ListBox** also provides features that enable we to efficiently add items to the ListBox and to find text within the items of the list. We can use the Add or Insert method to add items to a list box. The **Add method** adds new items at the end of an unsorted list box.

```
listBox1.Items.Add("Sunday");
```

If the **Sorted property** of the C# ListBox is set to true, the item is inserted into the list alphabetically. Otherwise, the item is inserted at the end of the ListBox.

Insert Items in a Listbox

Syntax

public void Insert (int index, object item);

we can **inserts** an item into the list box at the specified index.

```
listBox1.Items.Insert(0, "First");

listBox1.Items.Insert(1, "Second");

listBox1.Items.Insert(2, "Third");

listBox1.Items.Insert(3, "Forth");
```

Listbox Selected Item

If we want to retrieve a **single selected** item to a variable , use the following code.

Or we can use

```
string item = listBox1.GetItemText(listBox1.SelectedItem);
```

Or

```
string item= listBox1.Text;
```

Selecting Multiple Items from Listbox

The **SelectionMode property** determines how many items in the list can be selected at a time. A ListBox control can provide single or **multiple selections** using the SelectionMode property. If we change the selection mode property to multiple select, then we will retrieve a collection of items from ListBox1.SelectedItems property.

```
listBox1.SelectionMode = SelectionMode.MultiSimple;
```

The ListBox class has two SelectionMode. **Multiple** or **Extended** .

In **Multiple mode** , we can select or deselect any item in a ListBox by clicking it. In Extended mode, we need to hold down the Ctrl key to select additional items or the Shift key to select a range of items.

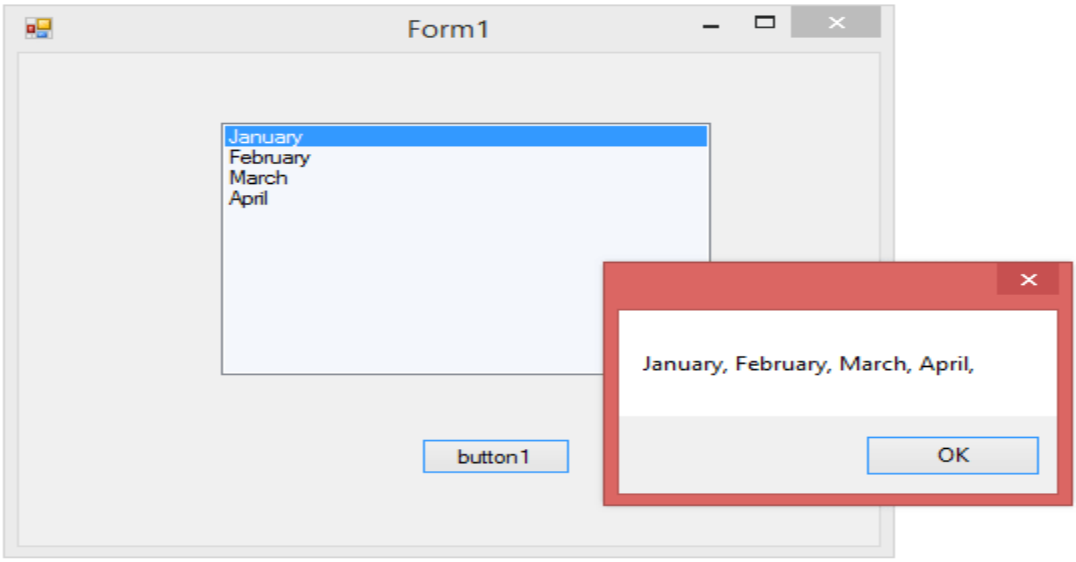
The following C# program initially fill seven days in a week while in the form load event and set the selection mode property to **MultiSimple** . At the Button click event it will display the selected items.

```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
   InitializeComponent();
  }
  private void Form1_Load(object sender, EventArgs e)
  {
   listBox1.Items.Add("Sunday");
   listBox1.Items.Add("Monday");
   listBox1.Items.Add("Tuesday");
   listBox1.Items.Add("Wednesday");
   listBox1.Items.Add("Thursday");
   listBox1.Items.Add("Friday");
   listBox1.Items.Add("Saturday");
   listBox1.SelectionMode = SelectionMode.MultiSimple;
  }
```

```
    private void button1_Click(object sender, EventArgs e)
    {
      foreach (Object obj in listBox1.SelectedItems )
      {
        MessageBox.Show(obj.ToString ());
      }
    }
  }
}
```

Getting All Items from List box

The Items collection of C# Winforms Listbox returns a **Collection type** of Object so we can use ToString() on each item to display its text value as below:




```
private void button1_Click_1(object sender, EventArgs e)
{
  string items = "";
  foreach (var item in listBox1.Items)
  {
    items += item.ToString() + ", ";
  }
  MessageBox.Show(items);
}
```

Bind a ListBox to a List

We can bind a List to a ListBox control by create a **fresh List Object** and add items to the List.

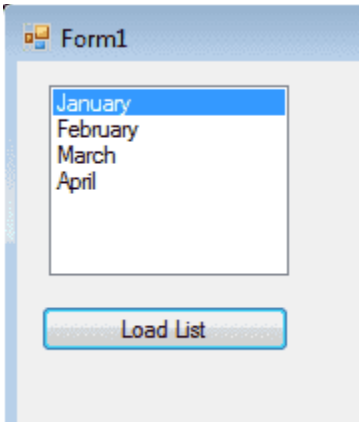
Creating a List

```
List<string> nList = new List<string>();
nList.Add("January");
nList.Add("February");
nList.Add("March");
nList.Add("April");
```

Binding to List

The next step is to bind this List to the Listbox. In order to do that we should **set datasource** of the Listbox.

```
listBox1.DataSource = nList;
```




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

{

  List<string> nList = new List<string>();
```

```
 nList.Add("January");

 nList.Add("February");

 nList.Add("March");

 nList.Add("April");

 listBox1.DataSource = nList;

}
```

bind a listbox to database values

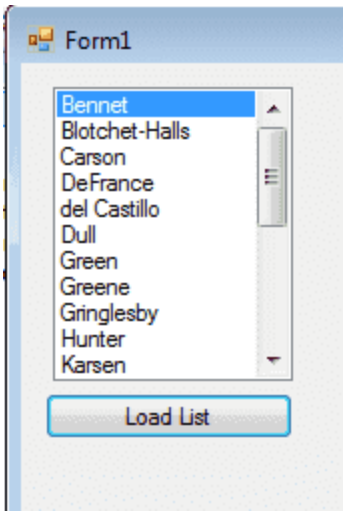
First of all, we could create a **connection string** and fetch data from database to a Dataset.

connetionString = "Data Source=ServerName;Initial Catalog=databasename;User ID=userid; Password=ourpassword"; sql = "select au_id,au_lname from authors";

Set Datasource for ListBox

Next step is that we have set Listbox **datasoure** as Dataset.

```
listBox1.DataSource = ds.Tables[0];

listBox1.ValueMember = "au_id";

listBox1.DisplayMember = "au_lname";
```



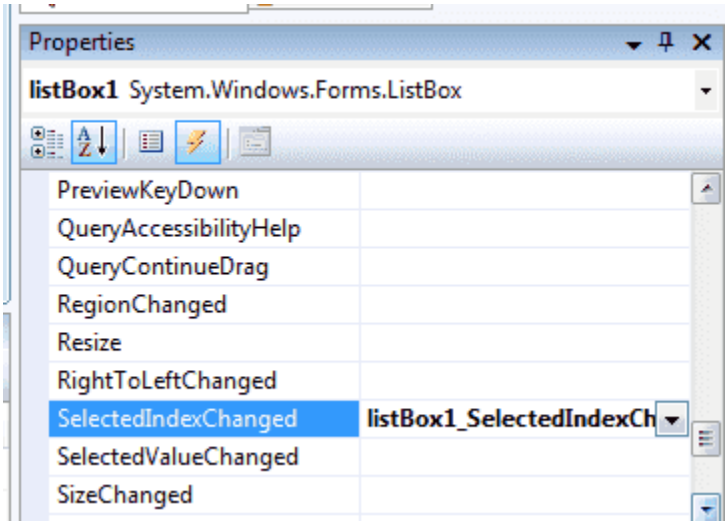

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e)
    {
      string connetionString = null;
      SqlConnection connection;
      SqlCommand command;
      SqlDataAdapter adapter = new SqlDataAdapter();
      DataSet ds = new DataSet();
      int i = 0;
      string sql = null;
```

```
    //connetionString = "Data Source=ServerName;Initial Catalog=databasename;User ID=userid;Password=ourpassword";
    //sql = "select au_id,au_lname from authors";
    connection = new SqlConnection(connetionString);
    try
    {
      connection.Open();
      command = new SqlCommand(sql, connection);
      adapter.SelectCommand = command;
      adapter.Fill(ds);
      adapter.Dispose();
      command.Dispose();
      connection.Close();
      listBox1.DataSource = ds.Tables[0];
      listBox1.ValueMember = "au_id";
      listBox1.DisplayMember = "au_lname";
    }
    catch (Exception ex)
    {
      MessageBox.Show("Cannot open connection ! ");
    }
  }
 }
}
```

SelectedIndexChanged event in ListBox

This event is fired when the item selection is changed in a **ListBox**. We can use this event in a situation that we want select an item from our listbox and accodring to this selection we can perform other programming needs.

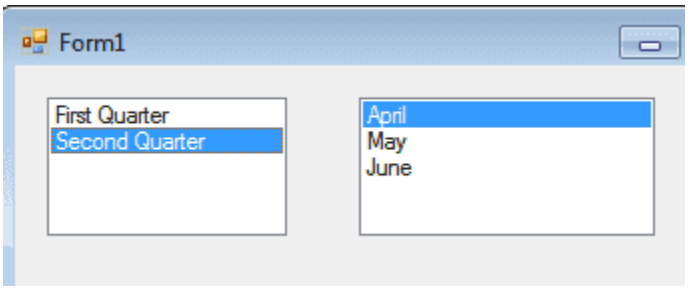
We can add the event handler using the **Properties Window** and selecting the Event icon and double-clicking on SelectedIndexChanged as we can see in following image.



The event will fire again when we select a new item. We can write we code within SelectedIndexChanged event . When we double click on ListBox the code will automatically come in we code editor like the following image.

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    //your code here
}
```

From the following example we can understand how to fire the SelectedIndexChanged event.



First we should drag two listboxes on our Form. First listbox we should set the List as **Datasource** , the List contents follows:

```
List<string> nList = new List<string>();

nList.Add("First Quarter");

nList.Add("Second Quarter");
```

When we load this form we can see the listbox is **populated** with List and displayed first quarter and second quarter. When we click the "First Quarter" the next listbox is populated with first quarter months and when we click "Second Quarter" we can see the second listbox is changed to second quarter months. From the following program we can understand how this happened.

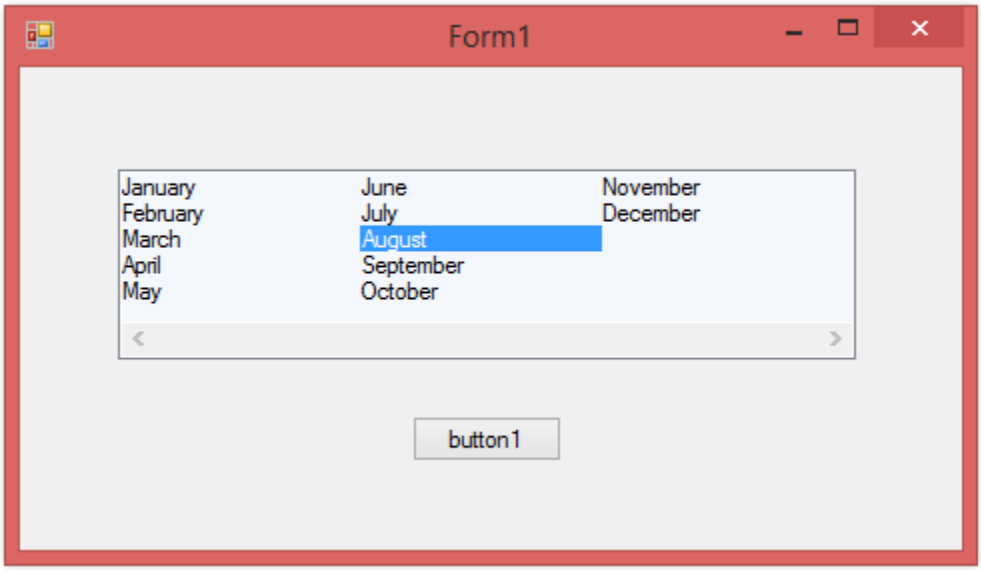
```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Windows.Forms;
using System.Collections.Generic;
namespace WindowsFormsApplication1
{
 public partial class Form1 : Form
 {
  public Form1()
  {
     InitializeComponent();
  }
  List  < string > fQ = new List  < string > ();
  List  < string > sQ = new List  < string > ();
  private void Form1_Load(object sender, EventArgs e)
  {
   fQ.Add("January");
   fQ.Add("February");
```

```
    fQ.Add("March");
    sQ.Add("April");
    sQ.Add("May");
    sQ.Add("June");
    List < string > nList = new List  < string > ();
    nList.Add("First Quarter");
    nList.Add("Second Quarter");
    listBox1.DataSource = nList;
  }
  private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
  {
    if (listBox1.SelectedIndex == 0)
    {
       listBox2.DataSource = null;
       listBox2.DataSource = fQ;
    }
    else if (listBox1.SelectedIndex == 1)
    {
       listBox2.DataSource = null;
       listBox2.DataSource = sQ;
    }
  }
 }
}
```

C# Listbox Column

A **multicolumn ListBox** places items into as many columns as are needed to make vertical scrolling unnecessary. The user can use the keyboard to navigate to columns that are not currently visible. First of all, we have Gets or sets a value indicating whether the ListBox supports **multiple columns**.

```
public bool MultiColumn { get; set; }
```

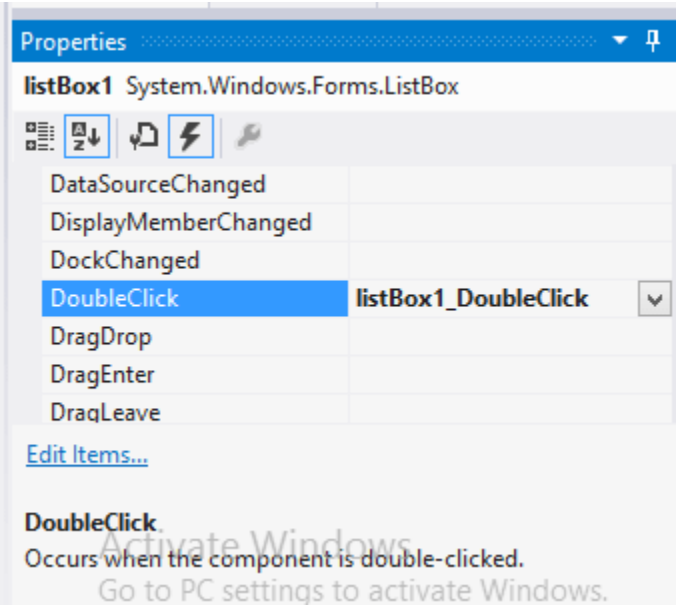


The following C# code example demonstrates a simple muliple column ListBox.

```
private void button1_Click_1(object sender, EventArgs e)
{
  listBox1.HorizontalScrollbar = true;
  listBox1.MultiColumn = true;
  listBox1.ScrollAlwaysVisible = true;
  listBox1.Items.AddRange(new object[] {  "January",  "February",  "March",  "April",  "May",
"June",  "July",  "August",  "September",  "October",  "November",  "December"});
}
```

Listbox Item Double Click Event

Add an event handler for the **Control.DoubleClick** event for our ListBox.



The following program shows when we double click the Listbox Item that event handler open up a MessageBox displaying the selected item.

```
 private void listBox1_DoubleClick(object sender, EventArgs e)
{
  if (listBox1.SelectedItem != null)
  {
    MessageBox.Show(listBox1.SelectedItem.ToString());
  }
}
```

Listbox vertical scrollbar

ListBox only shows a **vertical scroll bar** when there are more items then the displaying area. We can fix with ListBox.ScrollAlwaysVisible Property if we want the bar to be visible all the

time. If **ListBox.ScrollAlwaysVisible** Property is true if the vertical scroll bar should always be displayed; otherwise, false. The default is false.

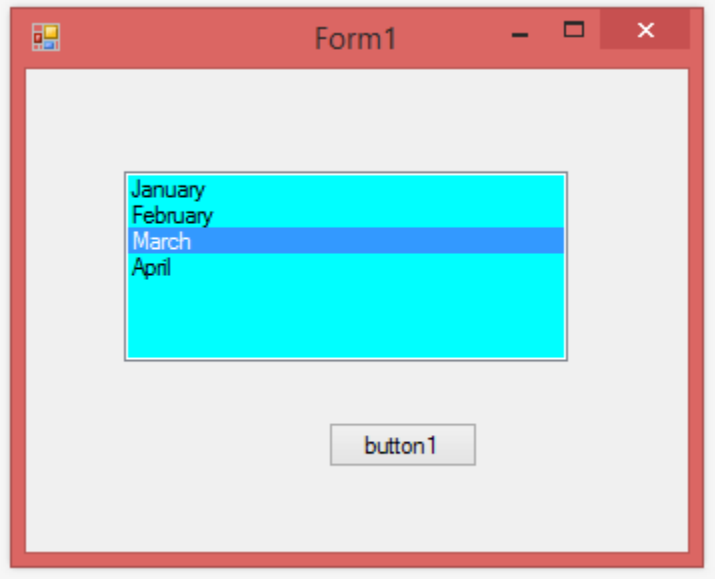
// Turn off the scrollbar.

```
ListBox1.ScrollAlwaysVisible = false;
```

Background and Foreground

The Listbox **BackColor** and **ForeColor properties** are used to set the background and foreground colors respectively. If we click on these properties in the Properties window, then the Color Dialog pops up and we can select the color we want.

Alternatively, we can set **background** and foreground colors from source code. The following code sets the BackColor and ForeColor properties:




```
listBox1.BackColor = System.Drawing.Color.Aqua;
listBox1.ForeColor = System.Drawing.Color.Black;
```

Clear all data in a listBox

Listbox **Items.Clear()** method clear all the items from ListBox.

```
private void button1_Click_1(object sender, EventArgs e)
{
   listBox1.Items.Clear();
}
```

ListBox.ClearSelected Method Unselects all items in the ListBox.

```
// Clear all selections in the ListBox.

listBox1.ClearSelected();
```

Remove item from Listbox

```
public void RemoveAt (int index);
```

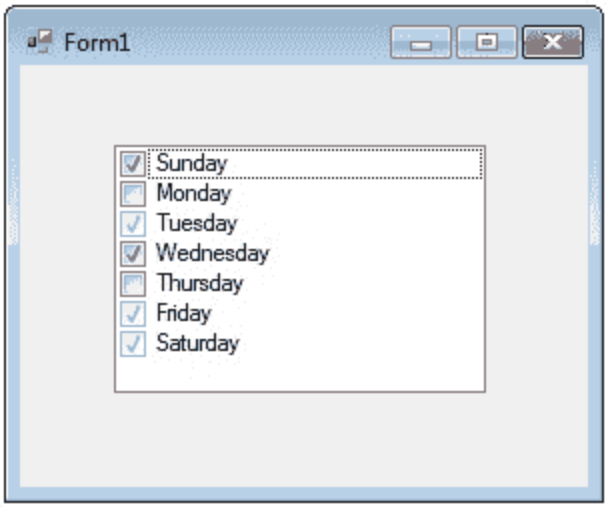
RemoveAt method removes the item at the **specified index** within the collection. When we remove an item from the list, the indexes change for subsequent items in the list. All information about the **removed item** is deleted.

Listbox Vs ListView Vs GridView

C# ListBox has many similarities with **ListView** or **GridView** (they share the parent class ItemsControl), but each control is oriented towards different situations. ListBox is best for **general UI** composition, particularly when the elements are always intended to be selectable, whereas ListView or GridView are best for **data binding** scenarios, particularly if virtualization or large data sets are involved. One most important difference is listview uses the extended selection mode by default .

Checked ListBox Control

The **CheckedListBox control** gives we all the capability of a list box and also allows we to display a **check mark** next to the items in the list box.



The user can place a check mark by one or more items and the checked items can be navigated with

the **CheckedListBox.CheckedItemCollection** and **CheckedListBox.CheckedIndexCollection**.

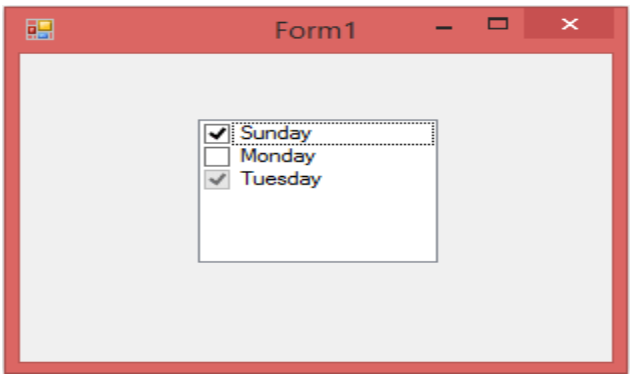
Checkedlistbox add items

Syntax

public int Add (object item, bool isChecked);

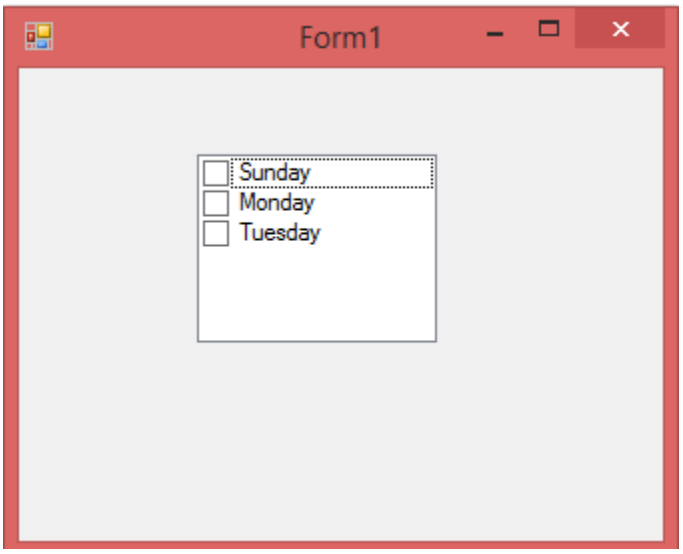
we can add individual items to the list with the **Add method** . The CheckedListBox object supports three states through the CheckState enumeration: Checked, Indeterminate, and Unchecked.

```
checkedListBox1.Items.Add("Sunday", CheckState.Checked);
checkedListBox1.Items.Add("Monday", CheckState.Unchecked);
checkedListBox1.Items.Add("Tuesday", CheckState.Indeterminate);
```



If we want to add objects to the list at run time, assign an array of object references with the **AddRange method** . The list then displays the default string value for each object.

```
string[] days = new[] { "Sunday", "Monday", "Tuesday" };
  checkedListBox1.Items.AddRange(days);
```



By default checkedlistbox items are **unchecked** .

Check all items in a Checkedlistbox

If we want to check an item in a Checkedlistbox, we need to call **SetItemChecked** with the relevant item.

```
public void SetItemChecked (int index, bool value);
```

Parameters

- index(Int32) - The index of the item to set the check state for.
- value(Boolean) - true to set the item as checked; otherwise, false.

If we want to set all items in a CheckedListBox to checked, change the value of **SetItemChecked** method to true.

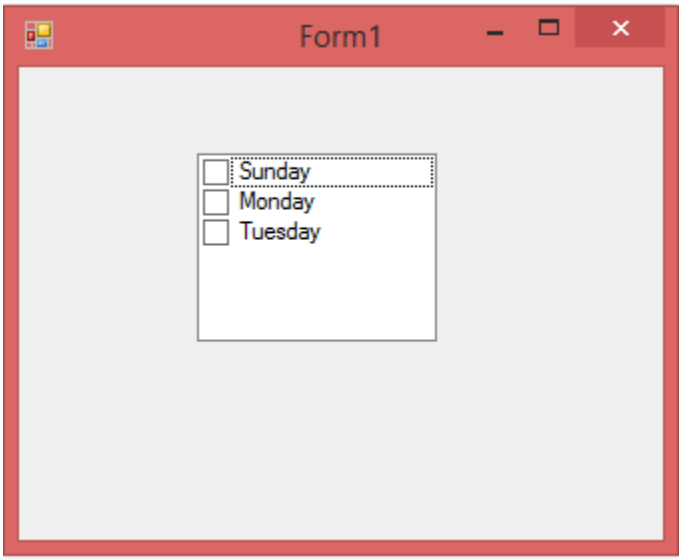
```
string[] days = new[] { "Sunday", "Monday", "Tuesday" };
checkedListBox1.Items.AddRange(days);
for (int i = 0; i < checkedListBox1.Items.Count; i++)
```

```
{
   checkedListBox1.SetItemChecked(i, true);
}
```

Uncheck all items in a Checkedlistbox

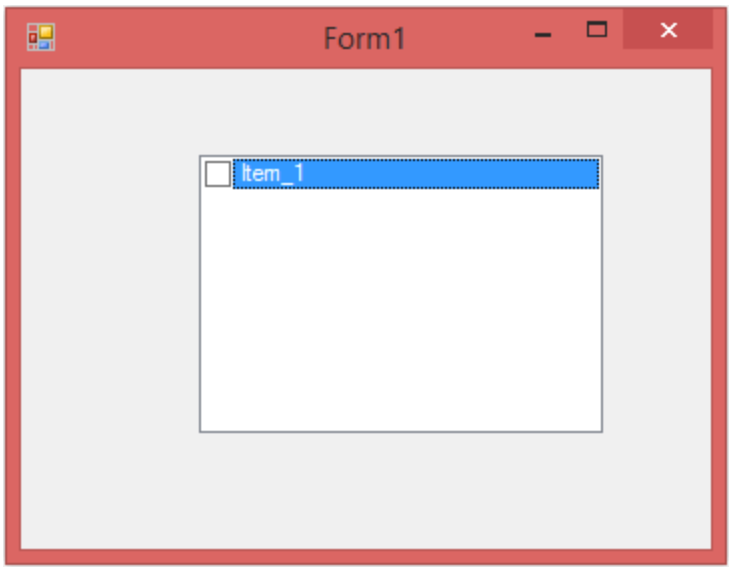
If we want to set all items in a CheckedListBox to unchecked, change the value of **SetItemChecked** method to false.

```
checkedListBox1.Items.Add("Sunday", CheckState.Checked);
checkedListBox1.Items.Add("Monday", CheckState.Checked);
checkedListBox1.Items.Add("Tuesday", CheckState.Checked);
for (int i = 0; i < checkedListBox1.Items.Count; i++)
{
   checkedListBox1.SetItemChecked(i, false);
}
```



CheckedListBox DataSource

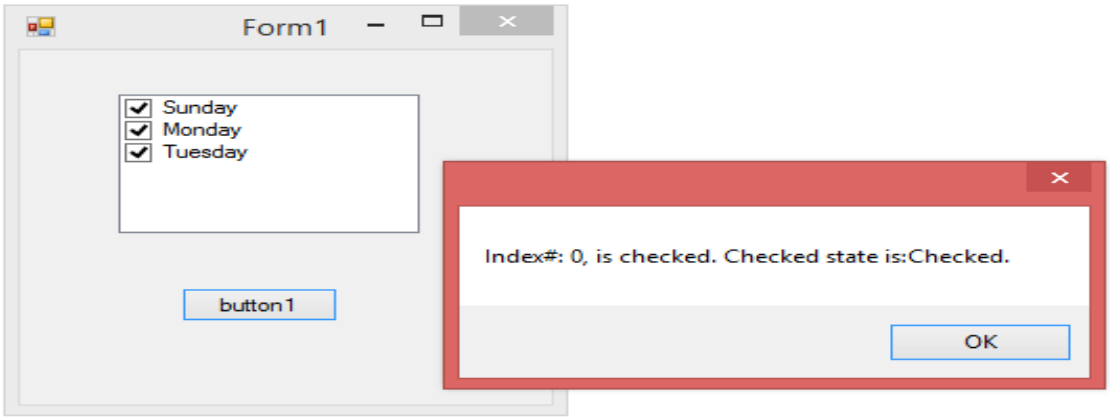
Following example shows how to bind a **DataSource** to CheckedListBox




```
DataTable dt = new DataTable();
DataColumn dc = new DataColumn("StringType", typeof(String));
dt.Columns.Add(dc);
DataRow dr = dt.NewRow();
dr[0] = "Item_1";
dt.Rows.Add(dr);
this.checkedListBox1.DataSource = dt;
this.checkedListBox1.DisplayMember = "StringType";
```

How to get value of checked item from CheckedListBox?

```
public System.Windows.Forms.CheckedListBox.ObjectCollection Items { get; }
```



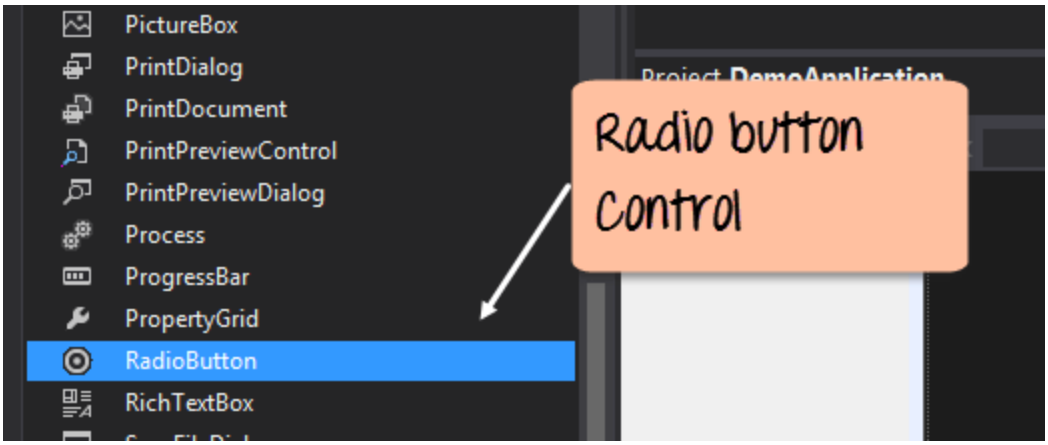
The following example uses the Items property to get the **CheckedListBox.ObjectCollection** to retrieve the index of an item using the ListBox.ObjectCollection.IndexOf method.

```
private void button1_Click_1(object sender, EventArgs e)
{
   foreach (int indexChecked in checkedListBox1.CheckedIndices)
   {
      // The indexChecked variable contains the index of the item.
  MessageBox.Show("Index: " + indexChecked.ToString() + ", is checked. Checked state is:" +
                  checkedListBox1.GetItemCheckState(indexChecked).ToString() + ".");
   }
}
```

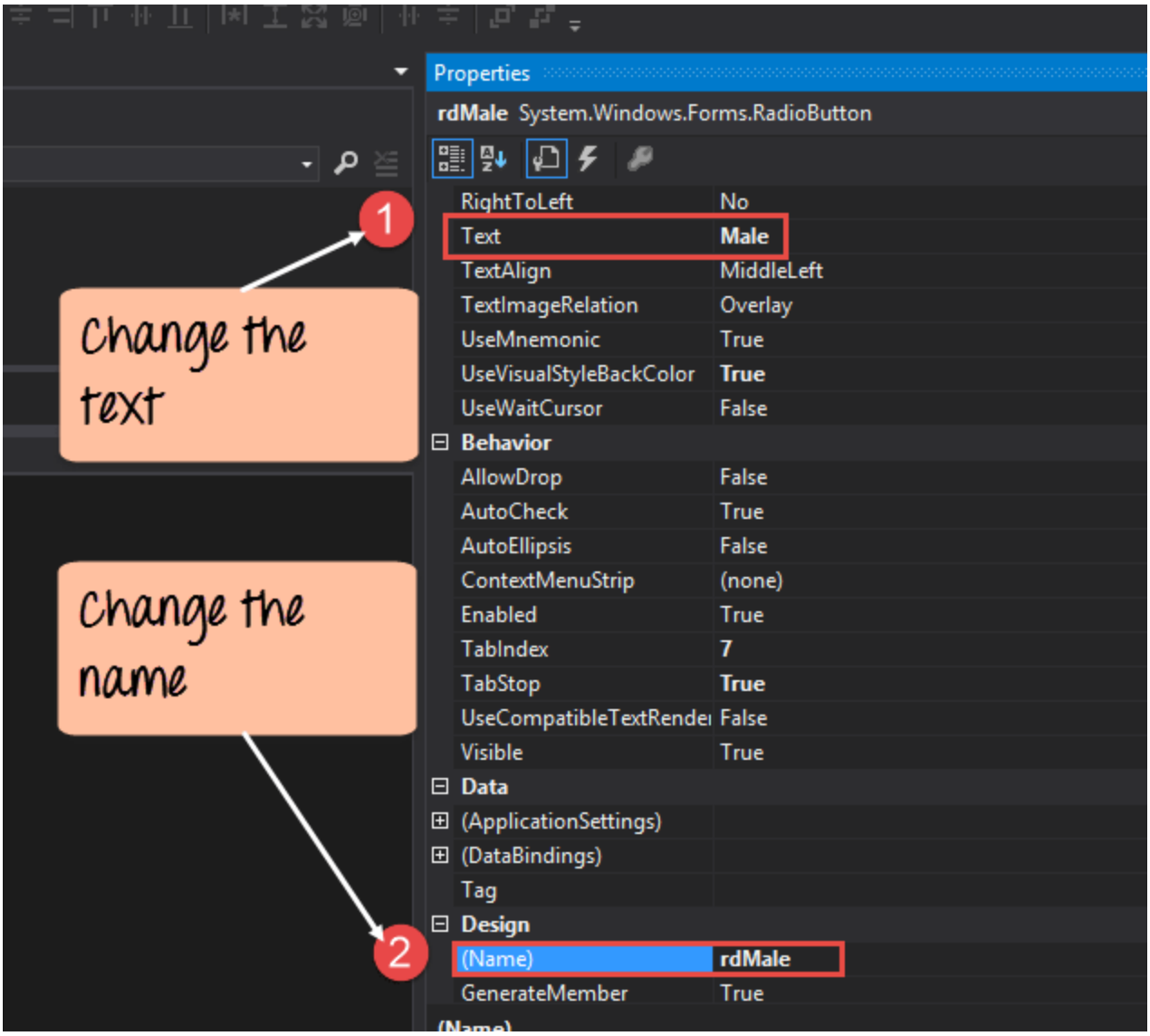
RadioButton

A Radiobutton is used to showcase a list of items out of which the user can choose one. Let's see how we can implement this with an example shown below. We will add a radio button for a male/female option.

Step 1) The first step is to drag the 'radiobutton' control onto the Windows Form from the toolbox as shown below.



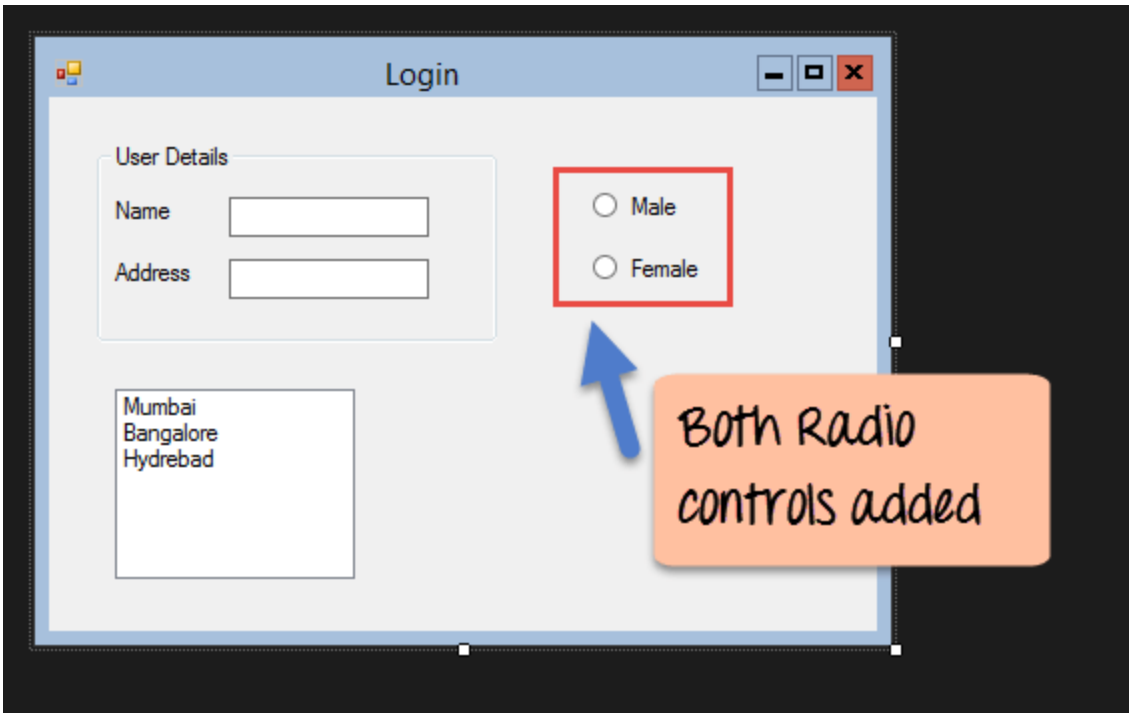
Step 2) Once the Radiobutton has been added, go to the properties window by clicking on the Radiobutton control.



1. First, we need to change the text property of both Radio controls. Go the properties windows and change the text to a male of one radiobutton and the text of the other to female.
2. Similarly, change the name property of both Radio controls. Go the properties windows and change the name to ‘rdMale’ of one radiobutton and to ‘rdfemale’ for the other one.

One we make the above changes, we will see the following output

Output:-

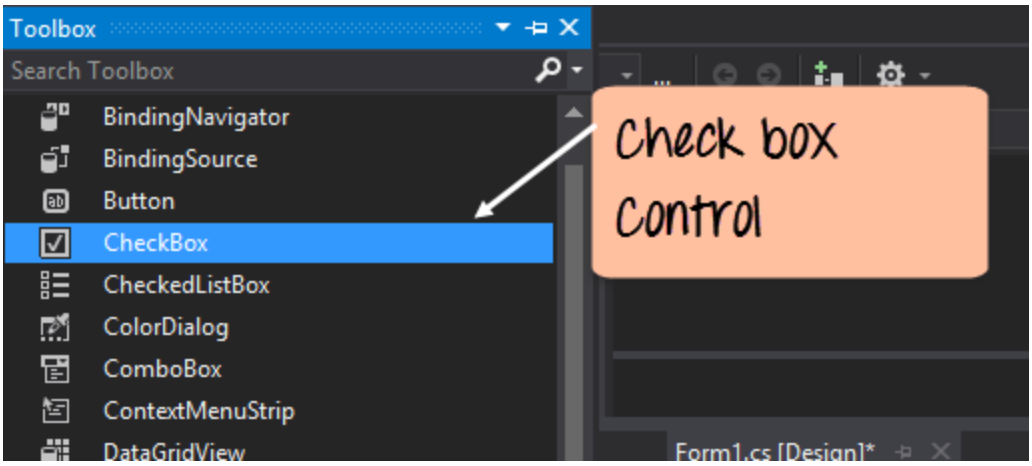


We will see the Radio buttons added to the Windows form.

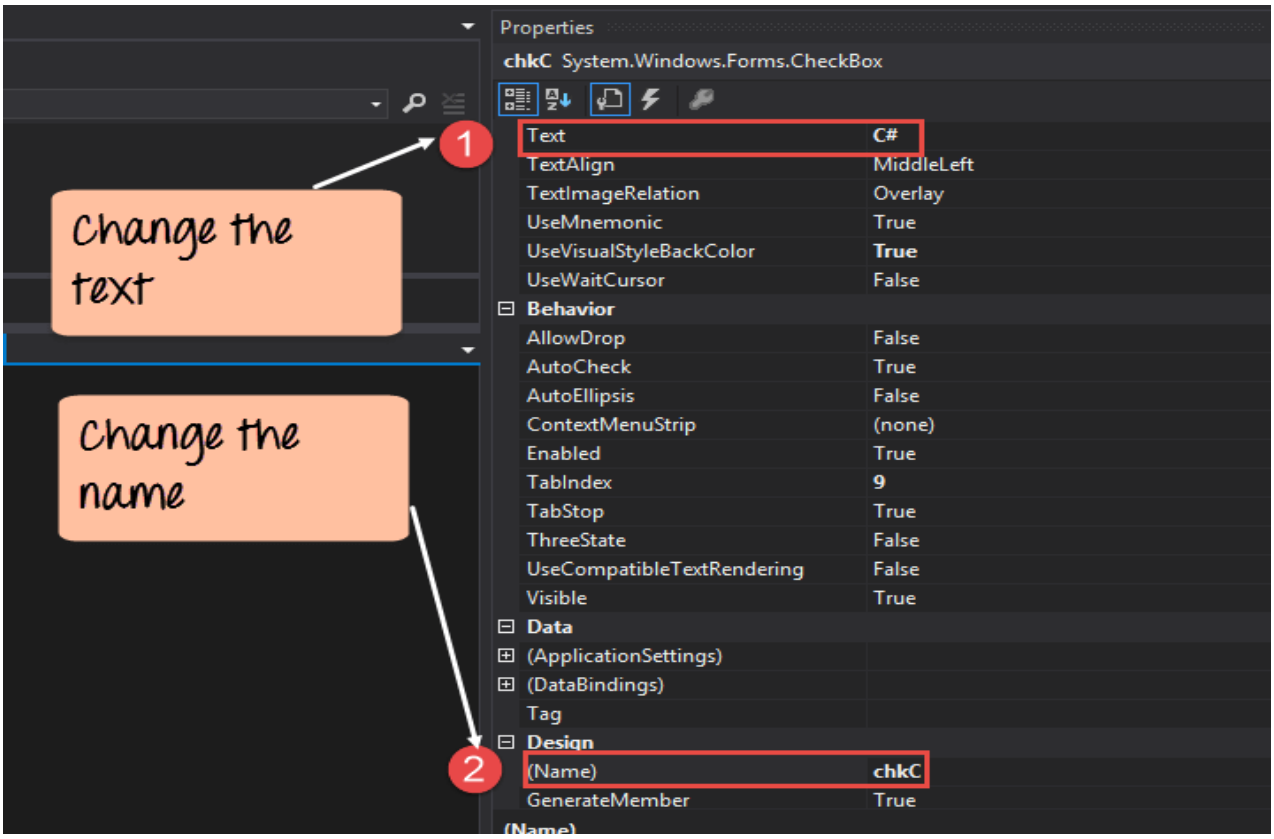
Checkbox

A checkbox is used to provide a list of options in which the user can choose multiple choices. Let's see how we can implement this with an example shown below. We will add 2 checkboxes to our Windows forms. These checkboxes will provide an option to the user on whether they want to learn C# or ASP.Net.

Step 1) The first step is to drag the checkbox control onto the Windows Form from the toolbox as shown below



Step 2) Once the checkbox has been added, go to the properties window by clicking on the Checkbox control.



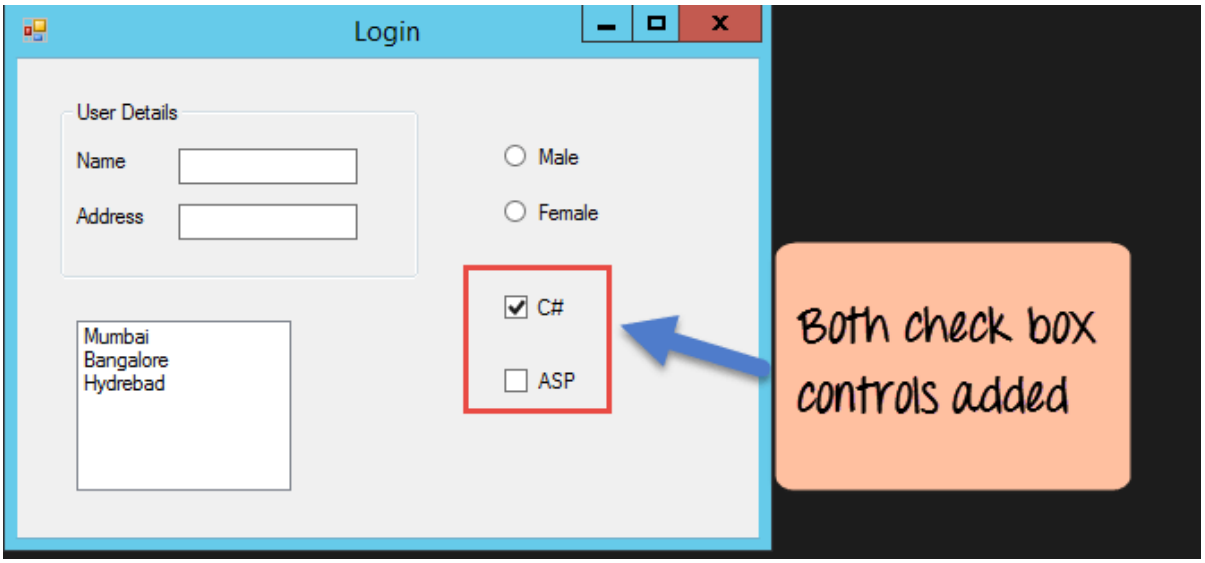
In the properties window,

1. First, we need to change the text property of both checkbox controls. Go the properties windows and change the text to C# and ASP.Net.

2. Similarly, change the name property of both Radio controls. Go the properties windows and change the name to chkC of one checkbox and to chkASP for the other one.

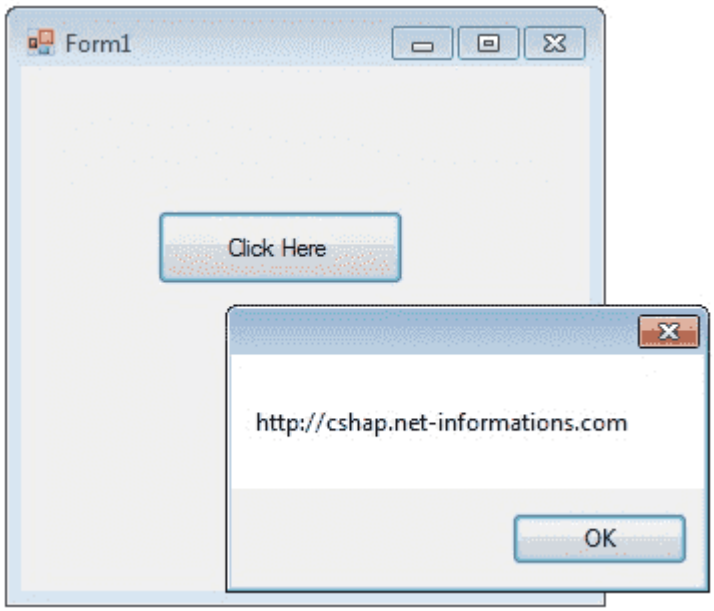
Once we make the above changes, we will see the following output

Output:-



Button

Windows Forms controls are reusable components that encapsulate user interface functionality and are used in client side Windows applications. A button is a control, which is an interactive component that enables users to communicate with an application. The Button class inherits directly from the ButtonBase class. A Button can be clicked by using the mouse, ENTER key, or SPACEBAR if the button has focus.



When we want to change display text of the Button, we can change the Text property of the button.

```
button1.Text = "Click Here";
```

Similarly if we want to load an Image to a Button control, we can code like this.

```
button1.Image = Image.FromFile ("C:\\testimage.jpg");
```

Call a Button's Click Event Programmatically

The Click event is raised when the Button control is clicked. This event is commonly used when no command name is associated with the Button control. Raising an event invokes the event handler through a delegate.

```
 private void Form1_Load(object sender, EventArgs e)
{
    Button b = new Button();
    b.Click += new EventHandler(ShowMessage);
    Controls.Add(b);
```

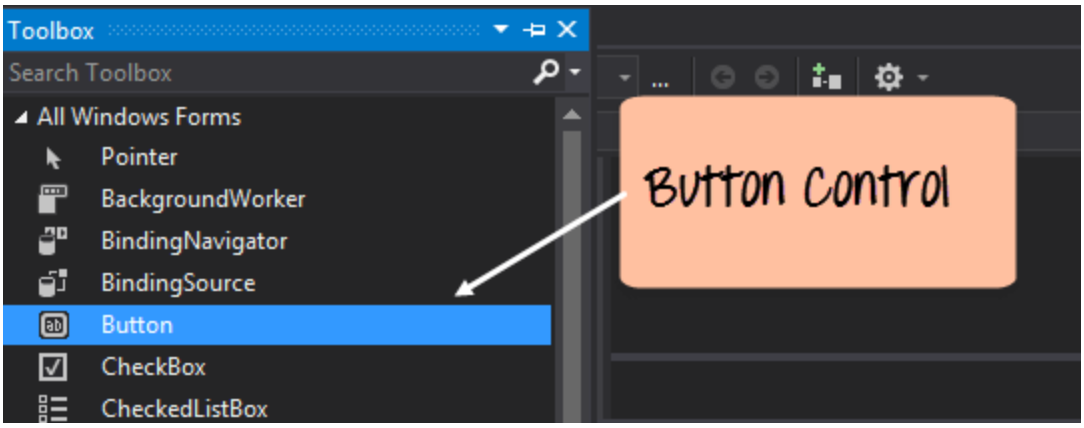
```
}
private void ShowMessage(object sender, EventArgs e)
{
   MessageBox.Show("Button Click");
}
```

The following C# source code shows how to change the button Text property while Form loading event and to display a message box when pressing a Button Control.

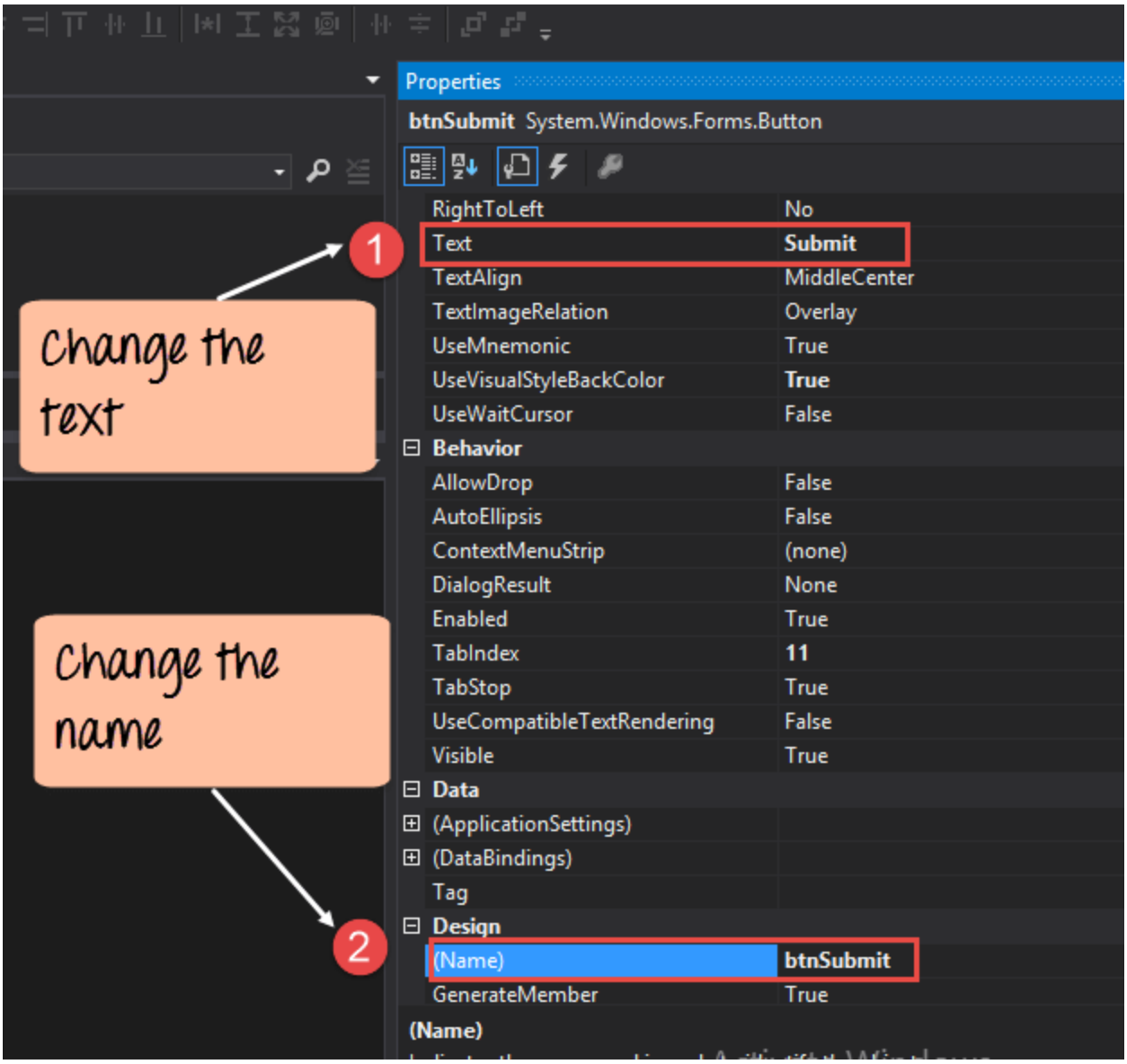
```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         button1.Text = "Click Here";
      }

      private void button1_Click(object sender, EventArgs e)
      {
         MessageBox.Show("http://cshap.net-informations.com");
      }
   }
}
```

A button is used to allow the user to click on a button which would then start the processing of the form. Let's see how we can implement this with an example shown below. We will add a simple button called 'Submit' which will be used to submit all the information on the form.

Step 1) The first step is to drag the button control onto the Windows Form from the toolbox as shown below



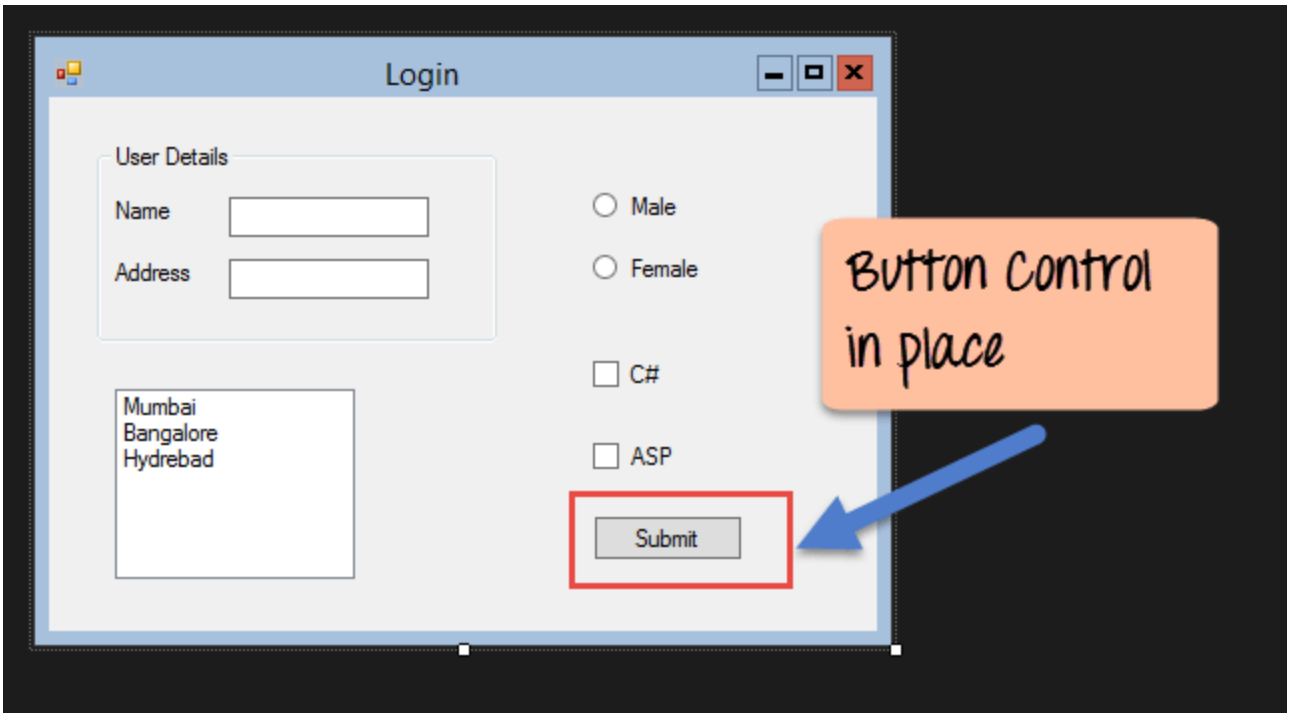
Step 2) Once the Button has been added, go to the properties window by clicking on the Button control.



1. First, we need to change the text property of the button control. Go the properties windows and change the text to ‘submit’.
2. Similarly, change the name property of the control. Go the properties windows and change the name to ‘btnSubmit’.

Once we make the above changes, we will see the following output

Output:-



Congrats, we now have our first basic Windows Form in place. Let's now go to the next topic to see how we can do Event handling for Controls.

C# Event Handling for Controls

When working with windows form, we can add events to controls. An event is something that happens when an action is performed. Probably the most common action is the clicking of a button on a form. In C# Windows Forms, we can add code which can be used to perform certain actions when a button is pressed on the form.

Normally when a button is pressed on a form, it means that some processing should take place.

Let's take a look at one of the event and how it can be handled before we go to the button event scenario.

The below example will showcase an event for the Listbox control. So whenever an item is selected in the listbox control, a message box should pop up which shows the item selected. Let's perform the following steps to achieve this.

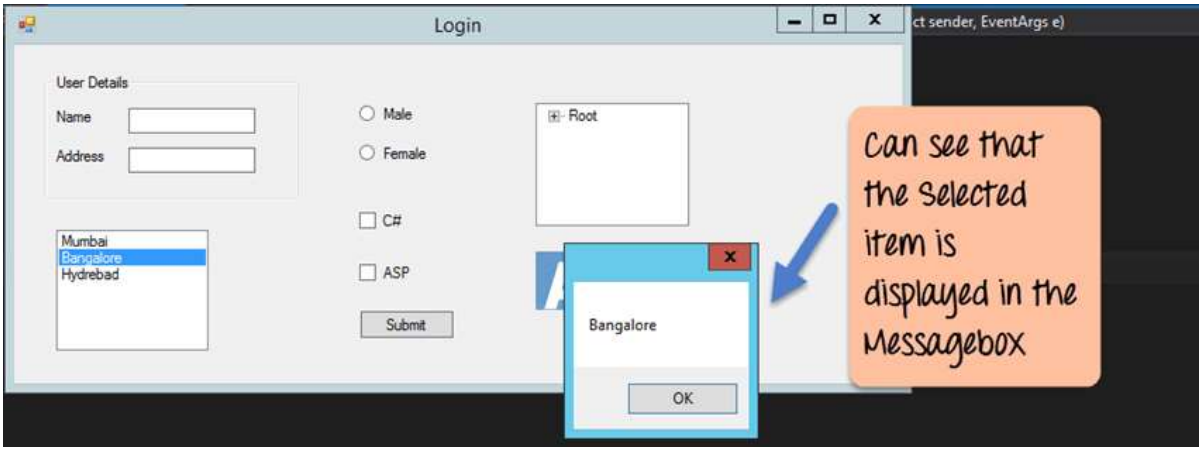
Step 1) Double click on the Listbox in the form designer**.** By doing this,Visual Studio will automatically open up the code file for the form. And it will automatically add an event method to the code. This event method will be triggered, whenever any item in the listbox is selected.

Above is the snippet of code which is automatically added by Visual Studio, when we double-click the List box control on the form. Now let's add the below section of code to this snippet of code, to add the required functionality to the listbox event.

1. This is the event handler method which is automatically created by Visual Studio when we double-click the List box control. We don't need to worry about the complexity of the method name or the parameters passed to the method.
2. Here we are getting the SelectedItem through the lstCity.SelectedItem property. Remember that lstCity is the name of our Listbox control. We then use the GetItemText method to get the actual value of the selected item. We then assign this value to the text variable.
3. Finally, we use the MessageBox method to display the text variable value to the user.

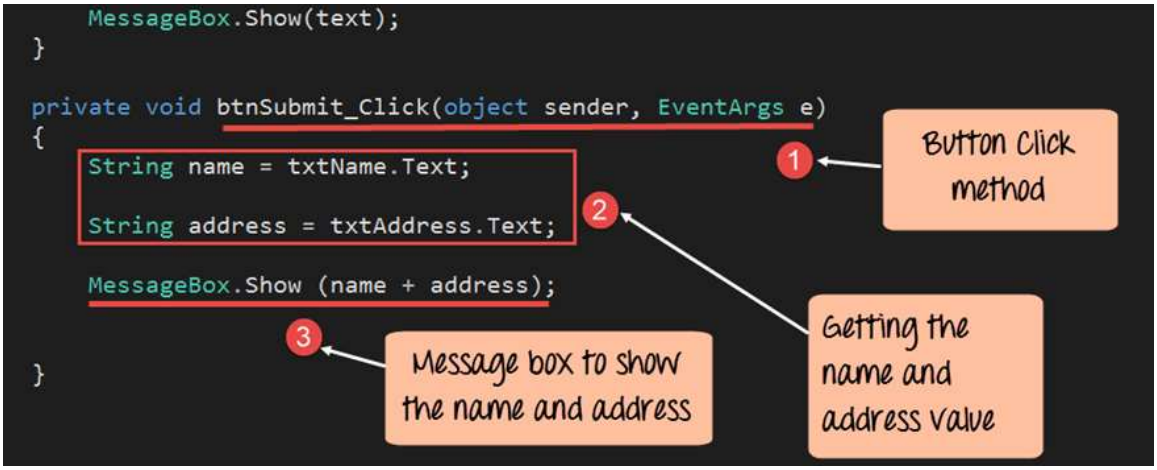
One we make the above changes, and run the program in Visual Studio we will see the following output

Output:-



From the output, we can see that when any item from the list box is selected, a message box will pops up. This will show the selected item from the listbox.

Now let's look at the final control which is the button click Method. Again this follows the same philosophy. Just double click the button in the Forms Designer and it will automatically add the method for the button event handler. Then we just need to add the below code.

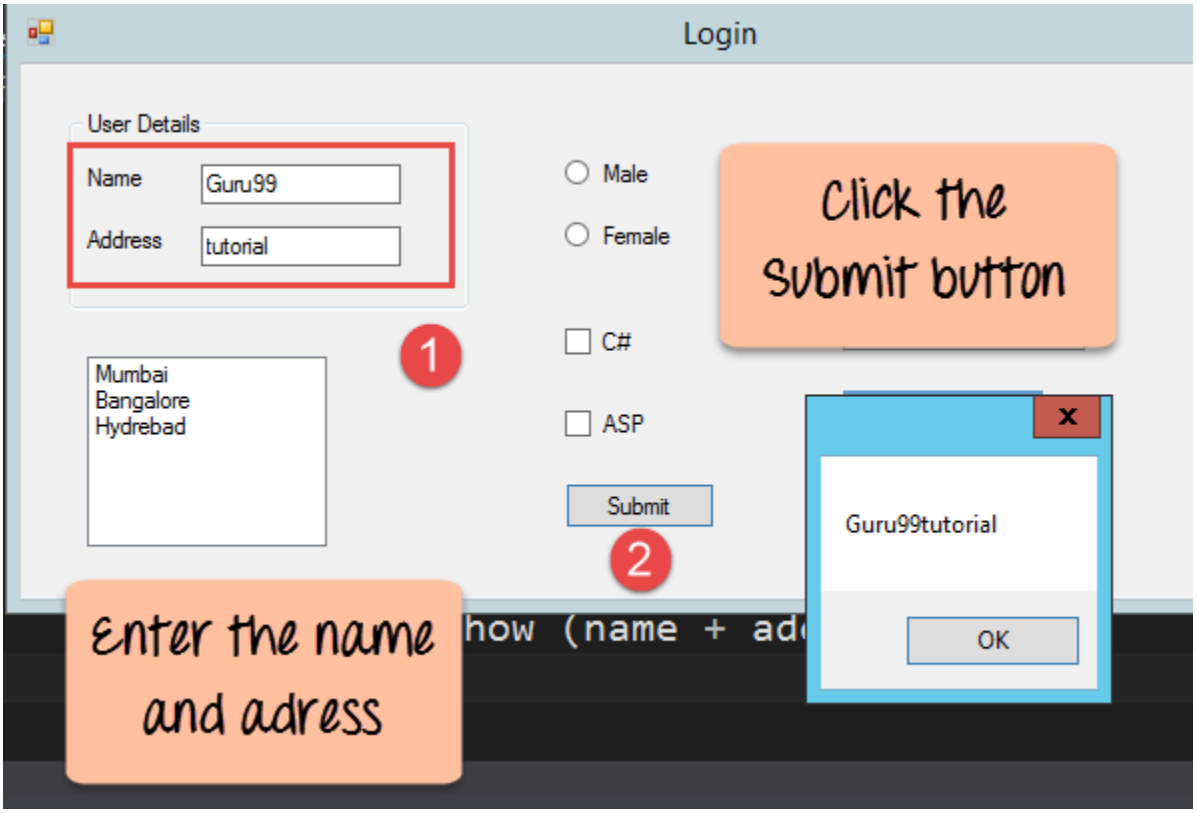


1. This is the event handler method which is automatically created by Visual Studio when we double click the button control. We don't need to worry on the complexity of the method name or the parameters passed to the method.

2. Here we are getting values entered in the name and address textbox. The values can be taken from the text property of the textbox. We then assign the values to 2 variables, name, and address accordingly.
3. Finally, we use the MessageBox method to display the name and address values to the user.

One we make the above changes, and run the program in Visual Studio we will see the following output

Output:-



1. First, enter a value in the name and address field.
2. Then click on the Submit button

Once we click the Submit button, a message box will pop, and it will correctly show we what we entered in the user details section.

Tree and PictureBox Control

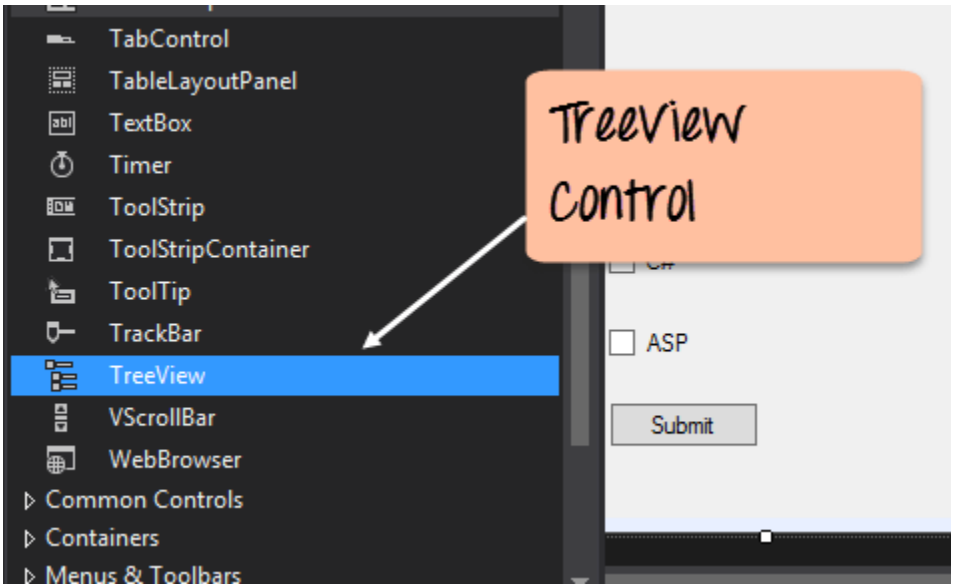
There are 2 further controls we can look at, one is the ‘Tree Control’ and the other is the ‘Image control’. Let’s look at examples of how we can implement these controls

Tree Control

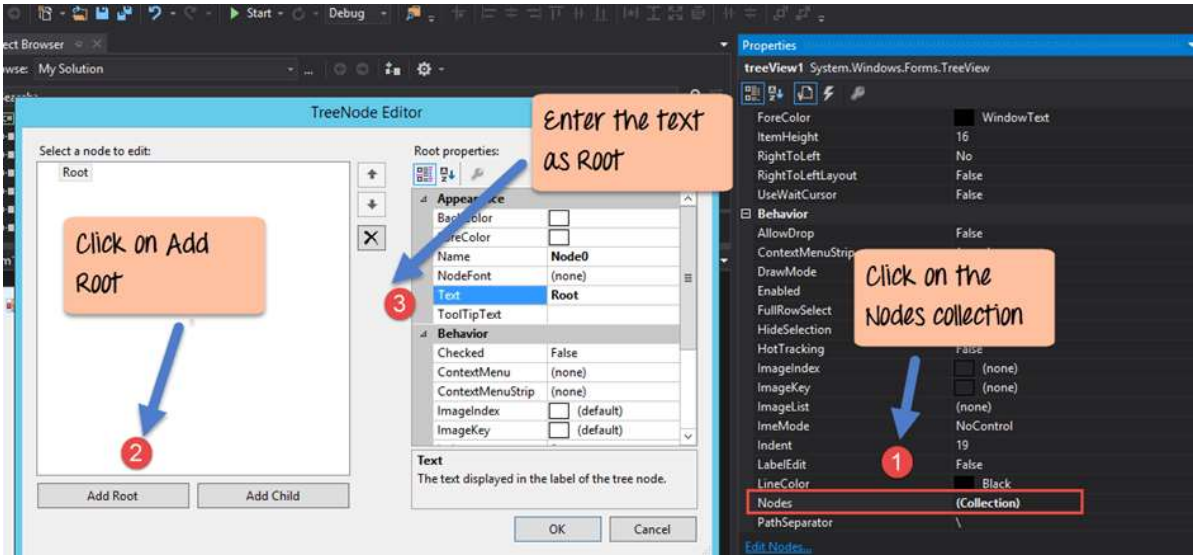
– The tree control is used to list down items in a tree like fashion. Probably the best example is when we see the Windows Explorer itself. The folder structure in Windows Explorer is like a tree-like structure.

Let’s see how we can implement this with an example shown below.

Step 1) The first step is to drag the Tree control onto the Windows Form from the toolbox as shown below

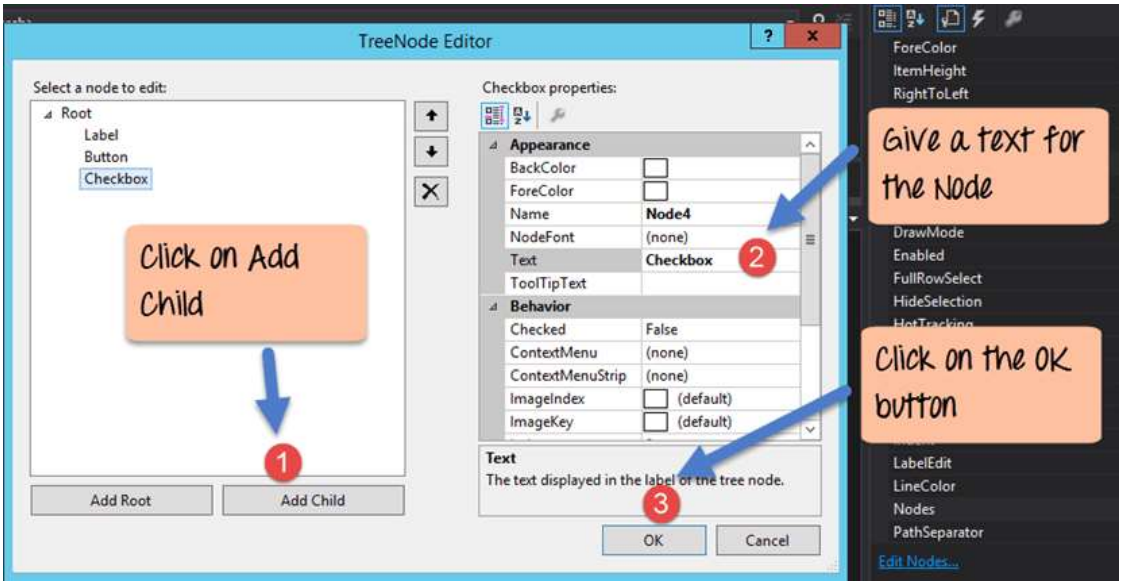


Step 2) The next step is to start adding nodes to the tree collection so that it can come up in the tree accordingly. First, let’s follow the below sub-steps to add a root node to the tree collection.



1. Go to the properties toolbox for the tree view control. Click on the Node's property. This will bring up the TreeNode Editor
2. In the TreeNode Editor click on the Add Root button to add a root node to the tree collection.
3. Next, change the text of the Root node and provide the text as Root and click 'OK' button. This will add Root node.

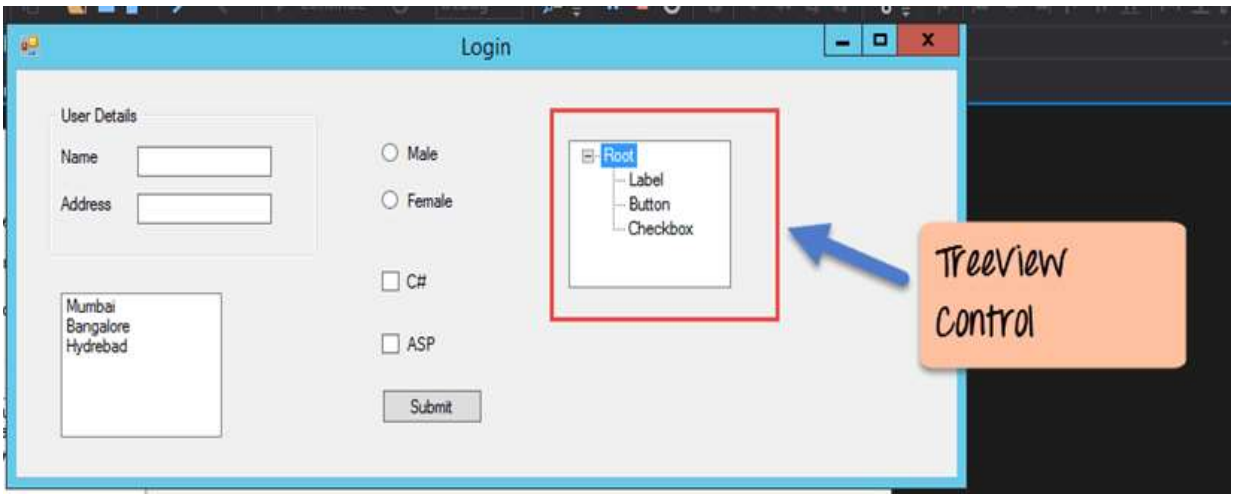
Step 3) The next step is to start adding the child nodes to the tree collection. Let's follow the below sub-steps to add child root node to the tree collection.



1. First, click on the Add child button. This will allow we to add child nodes to the Tree collection.
2. For each child node, change the text property. Keep on repeating the previous step and this step and add 2 additional nodes. In the end, we will have 3 nodes as shown above, with the text as Label, Button, and Checkbox respectively.
3. Click on the OK button

Once we have made the above changes, we will see the following output.

Output:-

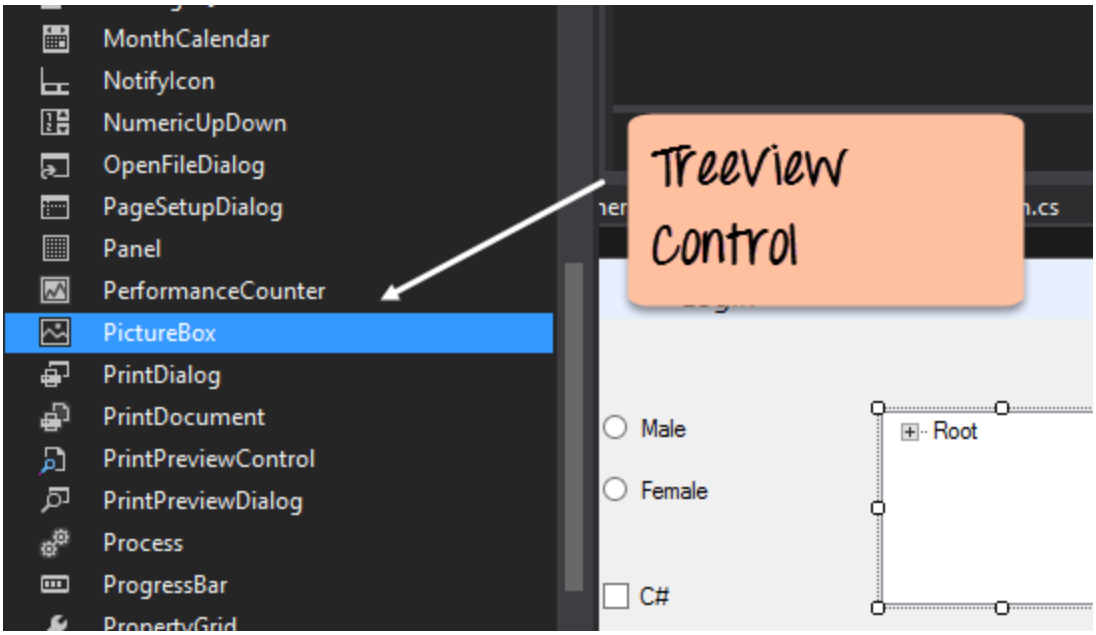


We will be able to see the Tree view added to the form. When we run the Windows form application, we can expand the root node and see the child nodes in the list.

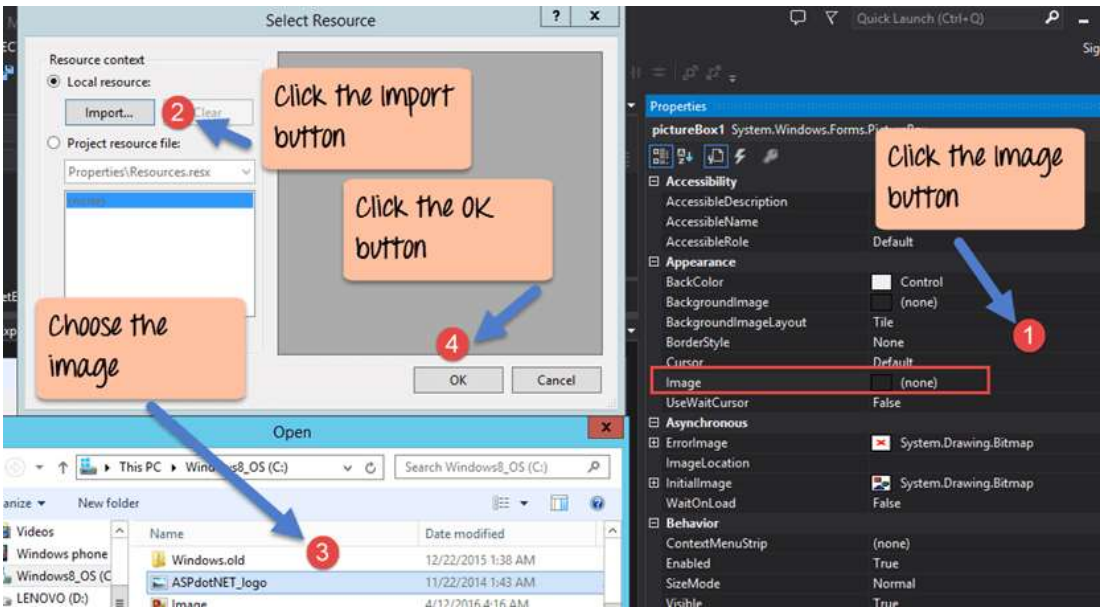
PictureBox Control

This control is used to add images to the Winforms C#. Let's see how we can implement this with an example shown below.

Step 1) The first step is to drag the PictureBox control onto the C# Windows Form from the toolbox as shown below



Step 2) The next step is to actually attach an image to the picture box control. This can be done by following the below steps.

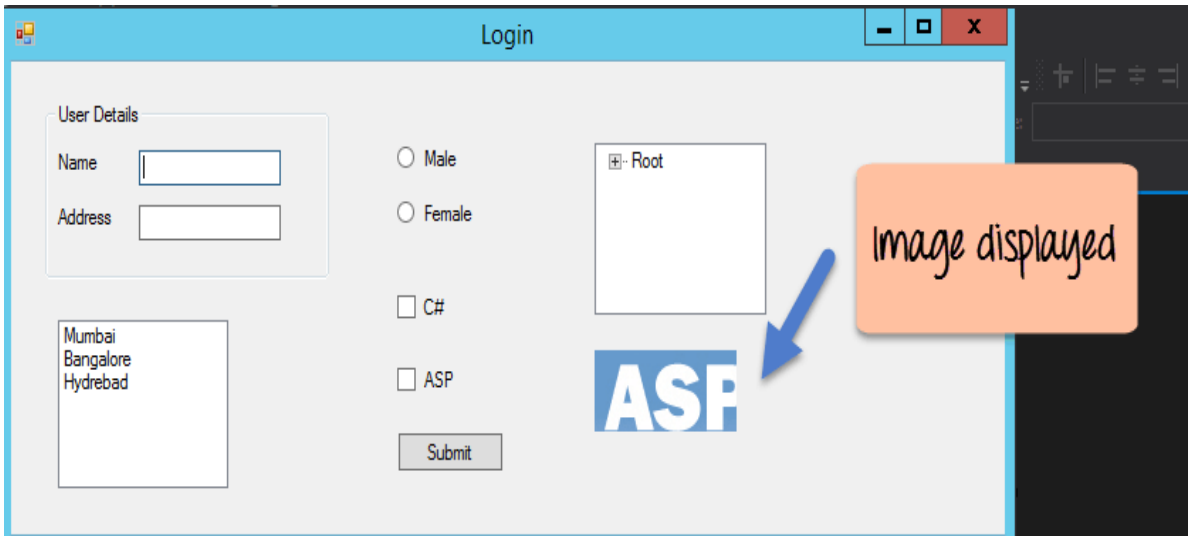


1. First, click on the Image property for the PictureBox control. A new window will pops out.
2. In this window, click on the Import button. This will be used to attach an image to the picturebox control.

3. A dialog box will pop up in which we will be able to choose the image to attach the picturebox
4. Click on the OK button

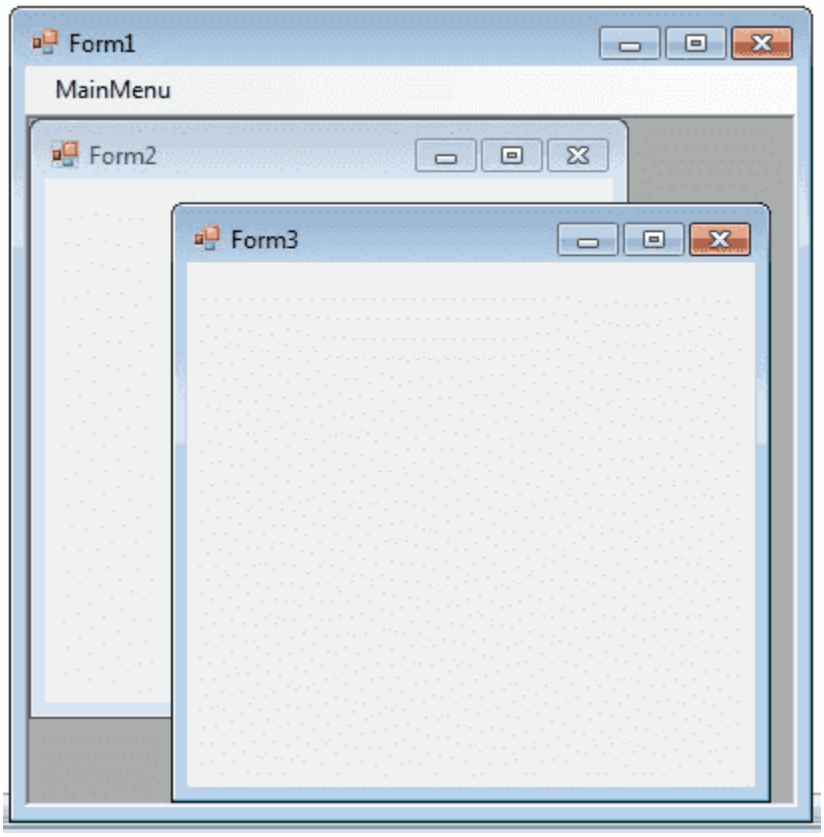
One we make the above changes, we will see the following output

Output:-



MDI Form

A Multiple Document Interface (MDI) programs can display multiple child windows inside them. This is in contrast to single document interface (SDI) applications, which can manipulate only one document at a time. Visual Studio Environment is an example of Multiple Document Interface (MDI) and notepad is an example of an SDI application. MDI applications often have a Window menu item with submenus for switching between windows or documents.



Any windows can become an MDI parent, if we set the IsMdiContainer property to True.

```
IsMdiContainer = true;
```

The following C# program shows a MDI form with two child forms. Create a new C# project, then we will get a default form Form1 . Then add two mnore forms in the project (Form2 , Form 3) . Create a Menu on our form and call these two forms on menu click event.

NOTE: If we want the MDI parent to auto-size the child form we can code like this.

```
form.MdiParent = this;
form.Dock=DockStyle.Fill;
form.Show();
```

```
using System;
using System.Drawing;
```

```
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         IsMdiContainer = true;
      }

      private void menu1ToolStripMenuItem_Click(object sender, EventArgs e)
      {
         Form2 frm2 = new Form2();
         frm2.Show();
         frm2.MdiParent = this;
      }

      private void menu2ToolStripMenuItem_Click(object sender, EventArgs e)
      {
         Form3 frm3 = new Form3();
         frm3.Show();
         frm3.MdiParent = this;
      }
   }
}
```

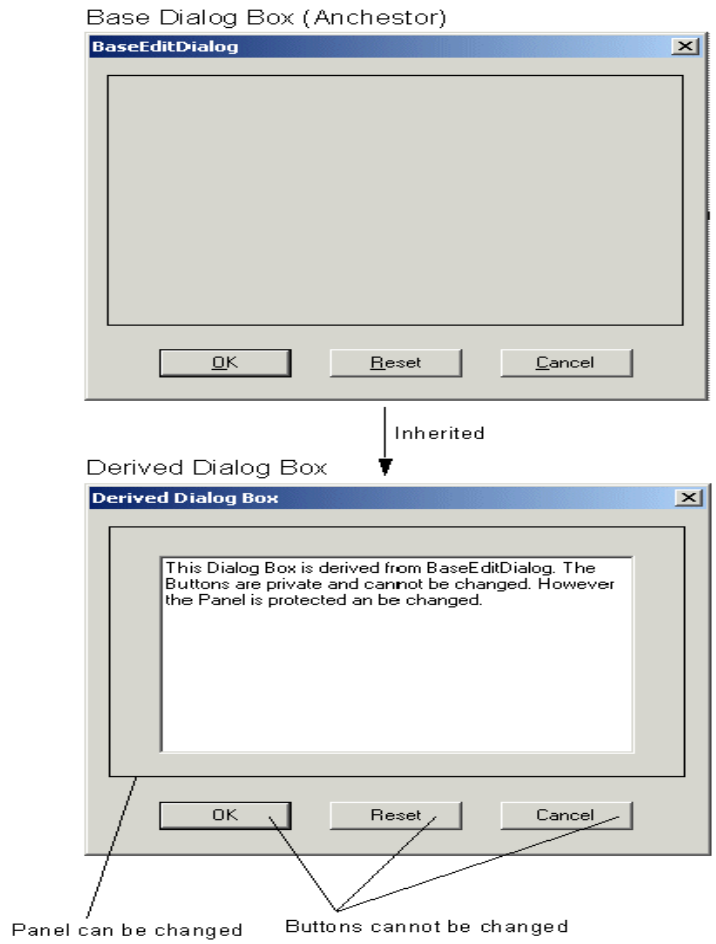
Form Inheritance

Form inheritance, a new feature of .NET that lets us create a base form that becomes the basis for creating more advanced forms. The new "derived" forms **automatically inherit all the functionality** contained in the base form. This design paradigm makes it easy to group common functionality and, in the process, reduce maintenance costs. When the base form is modified, the "derived" classes automatically follow suit and adopt the changes. The same concept applies to any type of object.

Example:

Visual inheritance allows us to see the controls on the base form and to add new controls. In this sample we will create a base dialog form and compile it into a class library. We will import this class library into another project and create a new form that inherits from the base dialog form. During this sample, we will see how to:

- Create a class library project containing a base dialog form.
- Add a Panel with properties that derived classes of the base form can modify.
- Add typical buttons that cannot be modified by inheritors of the base form.

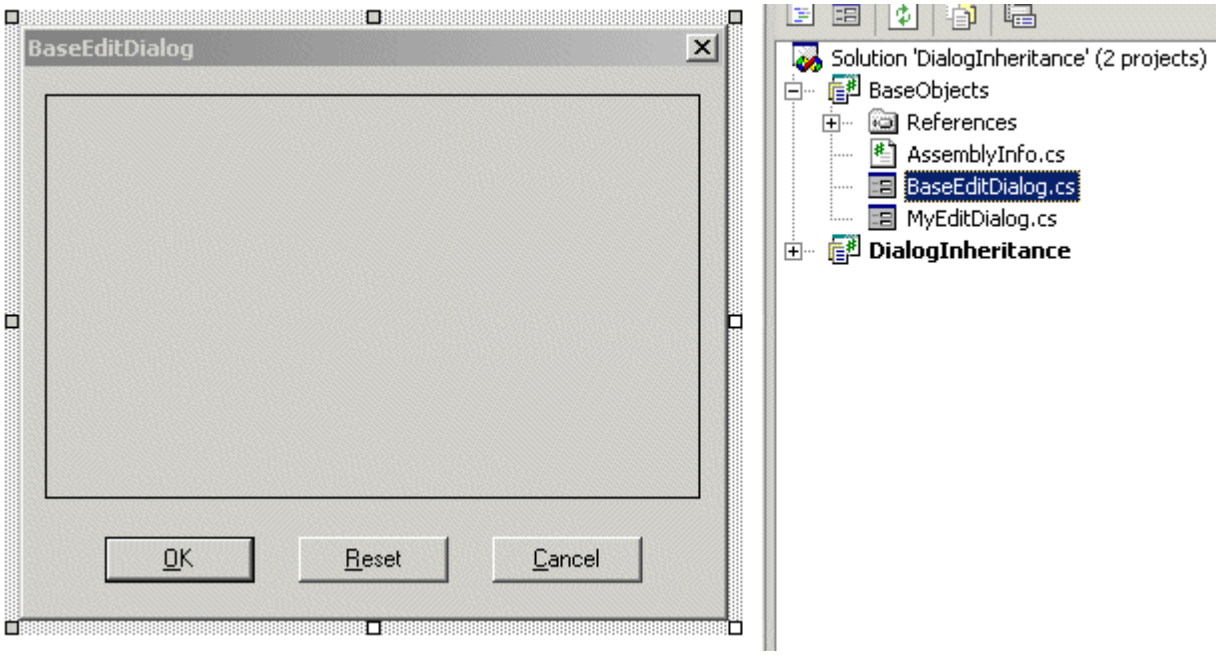


Create a class library project containing a base dialog form

In Visual C# .net perform the following steps

1. From the File menu, choose New and then Project to open the New Project dialog box.

2. Create a Windows Application named *BaseObjects*.
3. To create a class library instead of a standard Windows application, right-click the *BaseObjects* project node in Solution Explorer and select Properties.
4. In the properties for the project, change the output type from Windows Application to Class Library and click OK.
5. From the File menu, choose Save All to save the project and files to the default location.
6. Next add the buttons and the panel to the base form. To demonstrate visual inheritance, we will give the panel different access level by setting their **Modifiers properties**.
7. The OK and Cancel buttons have assigned actions due to their *DialogResult* setting. When either button is clicked, the dialog is deactivated and the appropriate value returned. We will require a way to save our modified settings when the OK button is clicked, and we need a way to perform an action when the Reset button is clicked. As a solution, let's add two protected methods that child classes can implement to handle these situations. We will cretae a *SaveSettings()* method to store the modified values and a *ResetSettings()* method to handle a click of the Reset button.
8. The Form class provides a Closing event that occurs whenever the form is about to close. The protected OnClosing method is invoked whenever the Close method is called, and it in turn raises the Closing event by invoking any registered event handlers. If Cancel is set to TRUE, then the close operation is cancelled and the application will continue to run. If Cancel is set to FALSE, then the close operation is not cancelled and the application will exit.



Here is the Code for BaseEditDialog.cs

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

namespace Akadia
{
   namespace BaseObjects
   {
      public class BaseEditDialog : System.Windows.Forms.Form
      {
         // These Objects cannot be changed in the derived Forms
         private System.Windows.Forms.Button btnOK;
         private System.Windows.Forms.Button btnReset;
         private System.Windows.Forms.Button btnCancel;
```

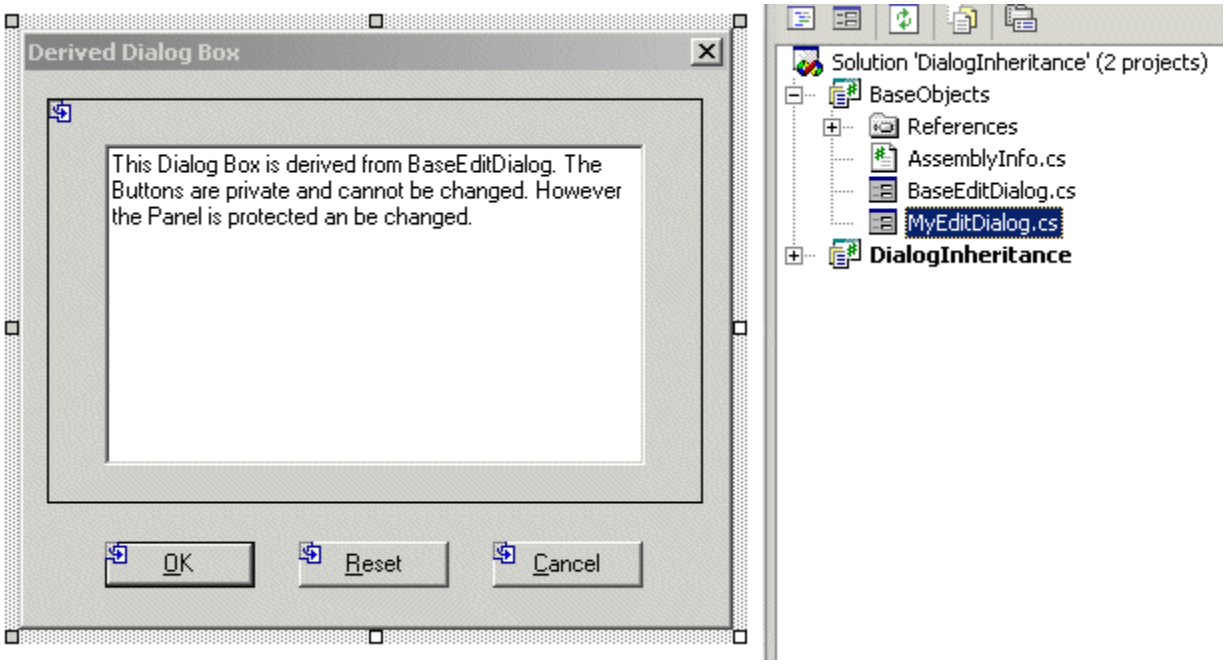
```
// These Objects can be changed in the derived Forms
protected System.Windows.Forms.Panel panel1;

.....
.....

// Derived Forms MUST override this method
protected virtual void ResetSettings()
{
   // Code logic in derived Form
}

// Derived Forms MUST override this method
protected virtual bool SaveSettings()
{
   // Code logic in derived Form
   return true;
}

// Handle Closing of the Form.
protected override void OnClosing(CancelEventArgs e)
{
   // If user clicked the OK button, make sure to
   // save the content. This must be done in the
   // SaveSettings() method in the derived Form.
   if (!e.Cancel && (this.DialogResult == DialogResult.OK))
   {
      // If SaveSettings() is OK (TRUE), then e.Cancel
      // will be FALSE, therefore the application will be exit.
      e.Cancel = !SaveSettings();
   }
```

```
            // Make sure any Closing event handler for the
            // form are called before the application exits.
            base.OnClosing(e);
        }

        // Event for Reset Button Click
        private void btnReset_Click(object sender, System.EventArgs e)
        {
            // Call ResetSettings()in the derived Form
            ResetSettings();
        }
    }
  }
}
```

To add an inherited form

1. Right-click the *BaseObjects* project and select Add and then Inherited Form.
2. In the Add New Item dialog box, verify that Inherited Form is selected, and click OK.
3. In the Inheritance Picker dialog box, enter *MyEditDialog*, this creates the derived form.
4. Open the inherited form in the Windows Forms Designer by double-clicking it, if it is not already open.

   In the Windows Forms Designer, the inherited buttons have a glyph in their upper corner, indicating they are inherited.
5. Add the methods *ResetSettings()* and *SaveSettings()* which **overrides the anchestor methods**.

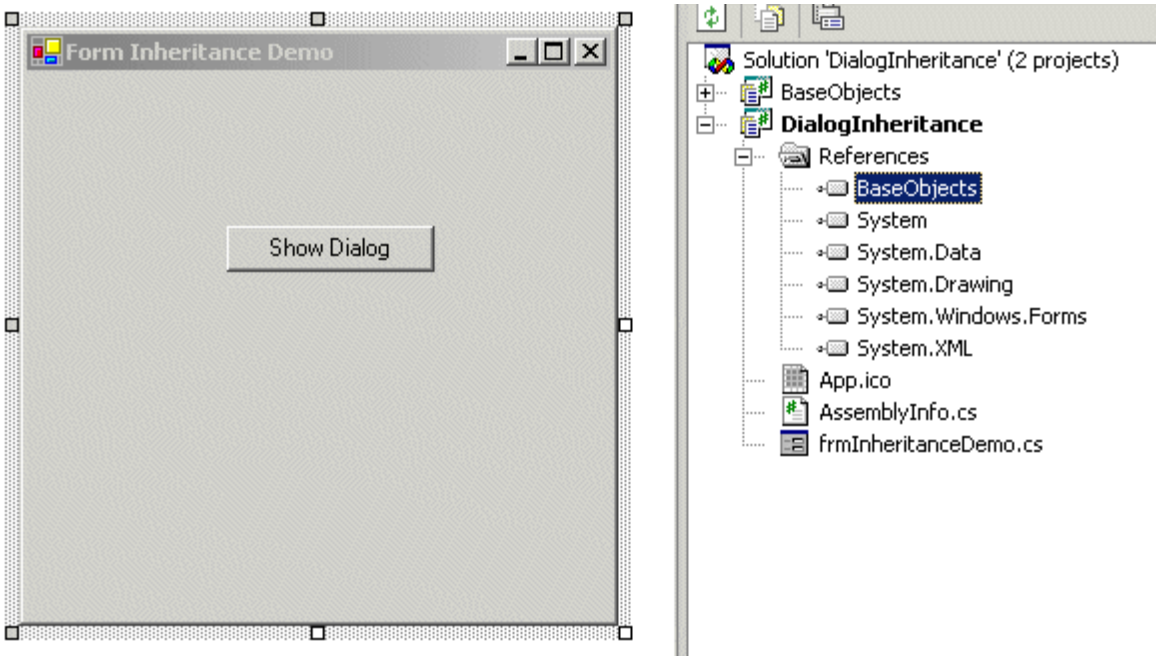



```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Windows.Forms;


namespace Akadia
{
   namespace BaseObjects
   {
      // This is a derived Dialog Box, to show how Inheritance
      // works in Visual C#
      public class MyEditDialog : Akadia.BaseObjects.BaseEditDialog
      {
         private System.Windows.Forms.TextBox txtBox;
         private System.ComponentModel.IContainer components = null;
```

```
        .....
        .....

        // The ResetSettings() method in the Anchestor
        // is overide. Therefore we can code the Logic
        // here in the derived Form.
        protected override void ResetSettings()
        {
            MessageBox.Show("Processing Reset Settings ...");
        }

        // The SaveSettings() method in the Anchestor
        // is overide. Therefore we can code the Logic
        // here in the derived Form.
        protected override bool SaveSettings()
        {
            MessageBox.Show("Processing Save Settings ...");
            return true;
        }
    }
  }
}
```

Create a project to test the inherited form application

1. From the File menu, choose Add Project and then New Project to open the New Project dialog box.
2. Create a Windows application named *DialogInheritance*.
3. Add a reference from this project to the *BaseObjects* project: Right-click references in the *DialogInheritance* project, choose Add Reference, Projects. Now we can select the appropriate *BaseObjects.dll*.

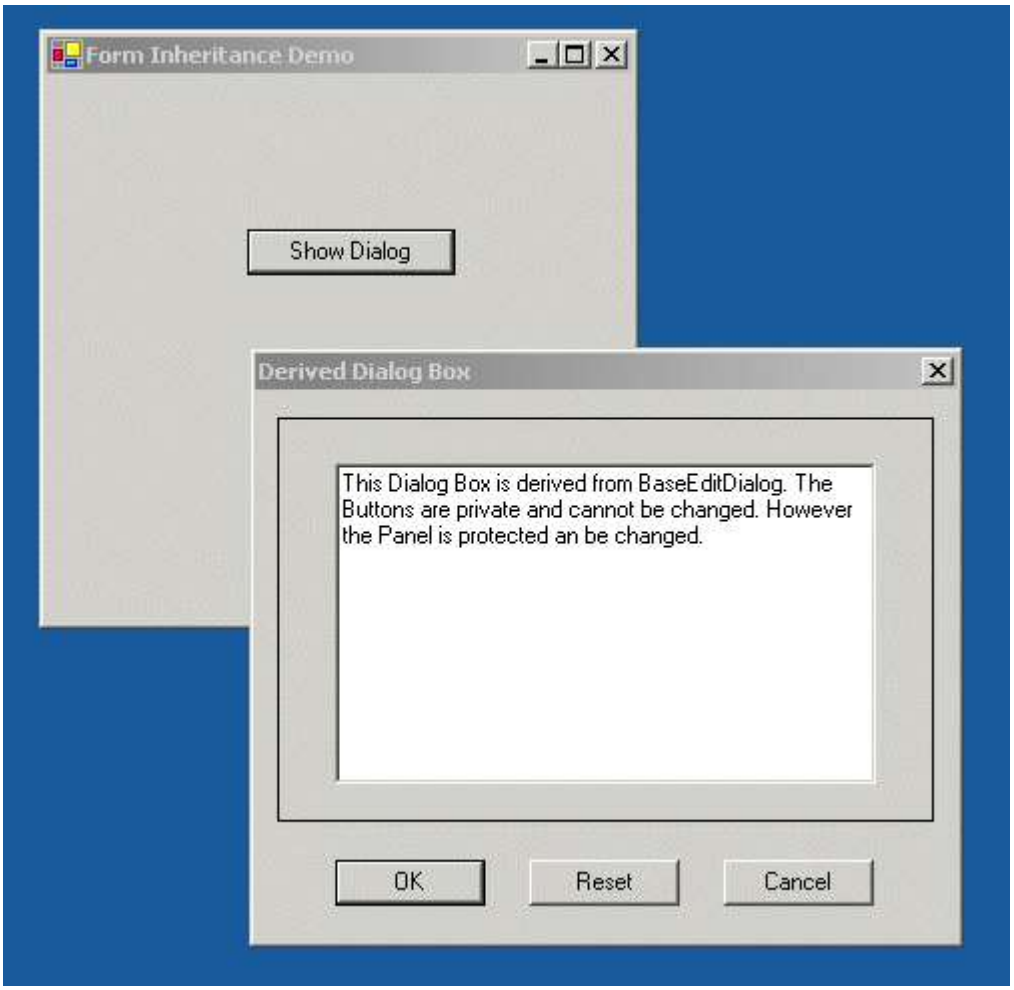



```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// Make our own base objects visible
using Akadia.BaseObjects;

namespace Akadia.DialogInheritance
{
   // This Form shows how Form Inheritance works in C#
   public class frmInheritanceDemo : System.Windows.Forms.Form
   {
      private System.Windows.Forms.Button btnShowDialog;
      private System.ComponentModel.Container components = null;
```

```
        public frmInheritanceDemo()
        {
            // Create visual Components
            InitializeComponent();
        }

        .....
        .....

        // The main entry point for the application.
        static void Main()
        {
            Application.Run(new frmInheritanceDemo());
        }

        // Call the derived Edit Dialog Form
        private void btnShowDialog_Click(object sender, System.EventArgs e)
        {
            MyEditDialog dlg;
            dlg = new MyEditDialog();

            if (dlg.ShowDialog() == DialogResult.OK)
            {
                MessageBox.Show("OK Button clicked in MyEditDialog()");
            }
        }
    }
}
```

Test the application

Start the application with [F5] and see how it works.



Dialogs

A dialog box in C# is a type of window, which is used to enable common communication or dialog between a computer and its user. A dialog box is most often used to provide the user with the means for specifying how to implement a command or to respond to a question. Windows.Form is a base class for a dialog box.

Sometimes, in a graphical user interface, a window is used to communicate with the user or establish a dialog between the user and the application. This additional window is called a dialog box. It may communicate information to the user; prompt the user for a response or both.

The simplest type of dialog box is the warning which displays a message and may require the user to acknowledge that the message has been read, usually by clicking “OK” or a decision as to whether or not an action should continue by clicking “OK” or “Cancel”.

Some dialog boxes are standard like the warning message or error message. Save the file and enter the password. These are called standard dialog boxes.

A dialog box can also be customized. Such a dialog box is called a custom dialog box.

Dialog boxes are special forms that are non-resizable. They are also used to display the messages to the user. The messages can be error messages, confirmation of the password, confirmation for the deletion of a particular record, Find-Replace utility of the word etc. There are standard dialog boxes to open and save a file, select a folder, print the documents, set the font or color for the text, etc.

Color dialog and Font dialog Example

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace dialogboxDemo
{
   class Program :Form
   {
      Button btn1 = new Button();

      Button btn2 = new Button();
```

```
TextBox txt = new TextBox();

ColorDialog cd = new ColorDialog();
FontDialog fd = new FontDialog();


public Program()
{
   this.Size = new Size(400,300);
   this.Location = new Point(300,200);
   this.Text = "Dialog Example Form";

   txt.Size = new Size(200, 75);
   txt.Location = new Point(20,20);

   btn1.Location = new Point(20,100);
   btn1.Text = "Font";
   btn1.Click += new EventHandler(btn1_click);

   btn2.Location = new Point(120, 100);
   btn2.Click += new EventHandler(btn2_click);
   btn2.Text = "Color";

   this.Controls.Add(txt);
   this.Controls.Add(btn1);
   this.Controls.Add(btn2);
}

private void btn1_click(Object sender , EventArgs e)
{
   fd.ShowDialog();
```

```
        txt.Font = fd.Font;


    }


    private void btn2_click(Object sender, EventArgs e)
    {
        cd.ShowDialog();
        this.BackColor = cd.Color;


    }



    static void Main(string[] args)
    {
        Application.Run(new Program());
    }
  }
}
```

Tooltips

A tooltip is a small rectangular pop-up window that displays a brief description of a control's purpose when the user rests the pointer on the control.

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

namespace Tooltips
{
    public class MyForm : Form
    {
        private FlowLawetPanel flowPanel;

        public MyForm()
        {
            InitComponents();
        }
        private void InitComponents()
```

```
        {
            Text = "Tooltips";
            ClientSize = new Size(800, 450);

            var button = new Button();
            button.Text = "Button";
            button.AutoSize = true;

            var btip = new ToolTip();
            btip.SetToolTip(button, "This is a Button Control");

           Controls.Add(button);

            CenterToScreen();
        }


        static void Main()
        {

            Application.EnableVisualStyles();
            Application.Run(new MyForm());
        }
    }

}
```

The code example creates a tooltip for two controls: one `Button` control and the `Form` control.

```
flowPanel = new FlowLawetPanel();
```

We place two buttons on the `FlowLawetPanel`. It dynamically lays out its contents horizontally or vertically. (The default dimension is vertical.)

```
var ftip = new ToolTip();
ftip.SetToolTip(flowPanel, "This is a FlowLawetPanel");
```

We create a new tooltip. With the `SetToolTip`, we assign the tooltip to the `FlowLawetPanel` control.

```
flowPanel.Dock = DockStyle.Fill;
```

The `FlowLawetPanel` fills the entire area of the form control.

```
var button = new Button();
button.Text = "Button";
button.AutoSize = true;
```

A new `Button` control is created. We set its text with the `Text` property and size it automatically to fit the text size.

```
var btip = new ToolTip();
btip.SetToolTip(button, "This is a Button Control");
```

A tooltip is added to the first `Button` control.

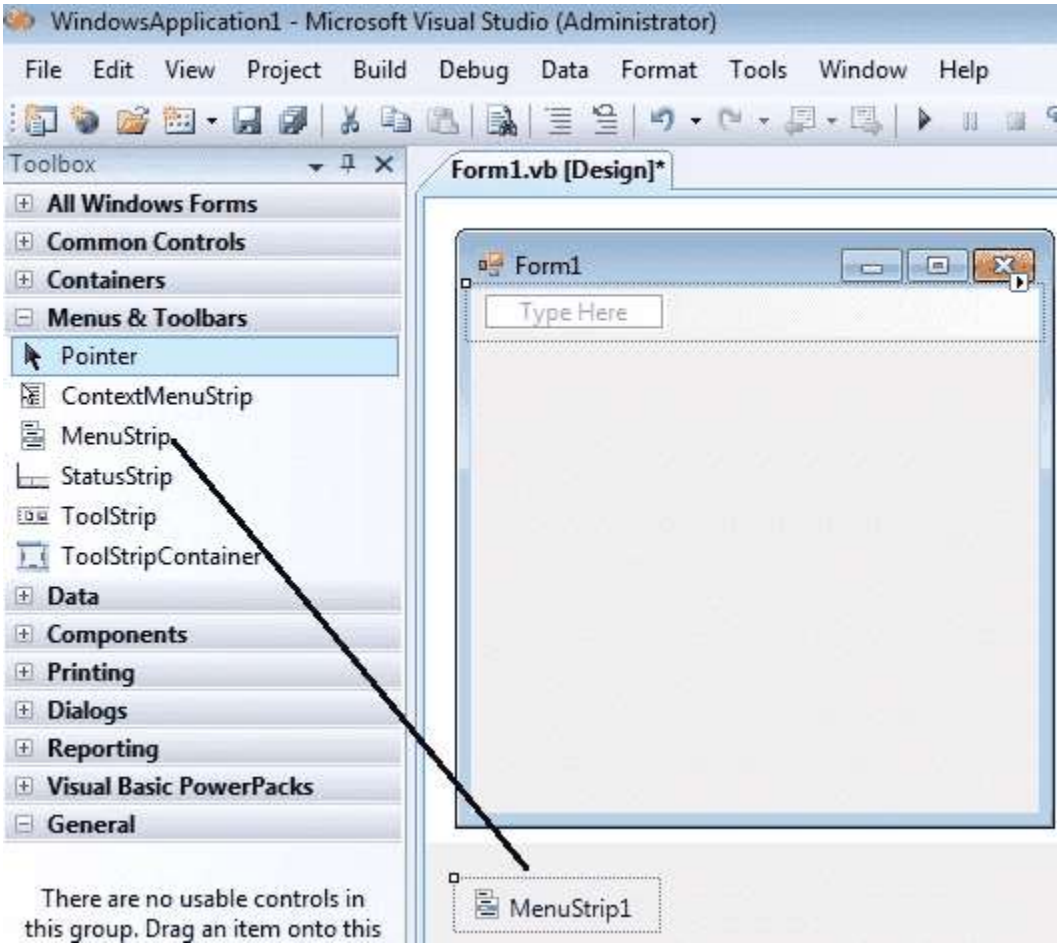
```
flowPanel.Controls.Add(button);
flowPanel.Controls.Add(button2);
Controls.Add(flowPanel);
```

The buttons are added to the flow panel and the flow panel is added to the form.

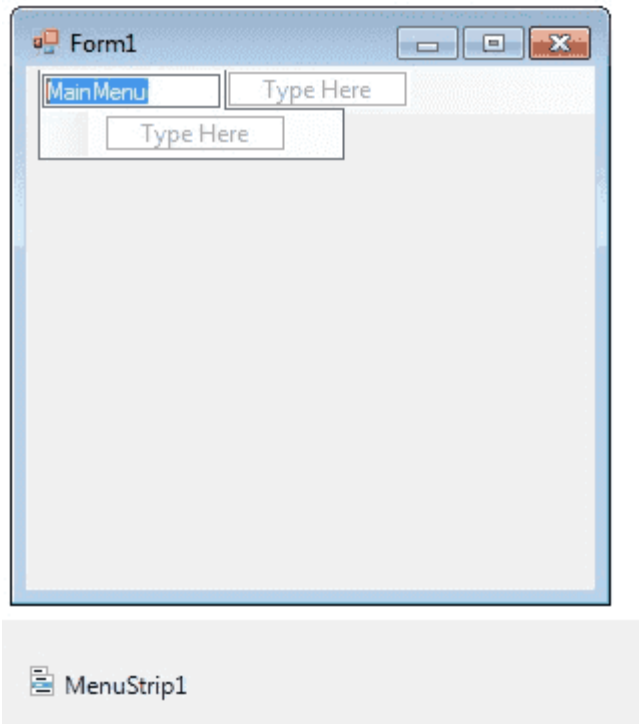
Menus and Context Menu

A Menu on a Windows Form is created with a MainMenu object, which is a collection of MenuItem objects. MainMenu is the container for the Menu structure of the form and menus are made of MenuItem objects that represent individual parts of a menu.

We can add menus to Windows Forms at design time by adding the MainMenu component and then appending menu items to it using the Menu Designer.



After drag the Menustrip on our form you can directly create the menu items by type a value into the "Type Here" box on the menubar part of your form. From the following picture we can understand how to create each menu items on mainmenu Object.



If we need a seperator bar , right click on our menu then go to insert->Seperator.

After creating the Menu on the form , you have to double click on each menu item and write the programs there depends on your requirements. The following C# program shows how to show a messagebox when clicking a Menu item.

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

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void menu1ToolStripMenuItem_Click(object sender, EventArgs
e)
        {
            MessageBox.Show("You are selected MenuItem_1");
```

```
            }
        }
}
```

Graphics and GDI +

GDI+ is next evolution of GDI. Using GDI objects in earlier versions of Visual Studio was a pain. In Visual Studio .NET, Microsoft has taken care of most of the GDI problems and have made it easy to use.

GDI+ resides in **System.Drawing.dll** assembly. All GDI+ classes are reside in the **System.Drawing**, **System.Text**, **System.Printing**, **System.Internal** , **System.Imaging**, **System.Drawing2D** and **System.Design** namespaces.

The first class we must discuss is the Graphics class. After the Graphics class, I will discuss other useful GDI+ classes and structures such as Pen, Brush, and Rectangle. The final part of this tutorial are some examples in C#.

The Graphics Class

The **Graphics** class encapsulates GDI+ drawing surfaces. Before drawing any object (for example circle, or rectangle) we have to create a surface using Graphics class. Generally we use Paint event of a Form to get the reference of the graphics. Another way is to override **OnPaint** method.

Here is how you get a reference of the Graphics object:

```
1. private void form1_Paint(object sender, PaintEventArgs e)
2. {
3.     Graphics g = e.Graphics;
4. }
```

OR:

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
}
```

Once you have the Graphics reference, you can call any of this class's members to draw various objects. Here are some of Graphics class's methods:

DrawArc	Draws an arc from the specified ellipse.
DrawBezier	Draws a cubic bezier curve.
DrawBeziers	Draws a series of cubic Bezier curves.
DrawClosedCurve	Draws a closed curve defined by an array of points.
DrawCurve	Draws a curve defined by an array of points.
DrawEllipse	Draws an ellipse.
DrawImage	Draws an image.
DrawLine	Draws a line.
DrawPath	Draws the lines and curves defined by a GraphicsPath.
DrawPie	Draws the outline of a pie section.
DrawPolygon	Draws the outline of a polygon.
DrawRectangle	Draws the outline of a rectangle.
DrawString	Draws a string.
FillEllipse	Fills the interior of an ellipse defined by a bounding rectangle.
FillPath	Fills the interior of a path.
FillPie	Fills the interior of a pie section.
FillPolygon	Fills the interior of a polygon defined by an array of points.
FillRectangle	Fills the interior of a rectangle with a Brush.
FillRectangles	Fills the interiors of a series of rectangles with a Brush.

FillRegion	Fills the interior of a Region.

In .NET, GDI+ functionality resides in the System.Drawing.dll. Before you start using GDI+ classes, you must add reference to the System.Drawing.dll and import System.Drawing namespace. If you are using Visual Studio .NET for your development, you can add reference to the GDI+ library using following:

Creating this project is simple. Create a Windows application and add reference to the System.Drawing.dll using Project->Add Reference menu item. See Figure 1.

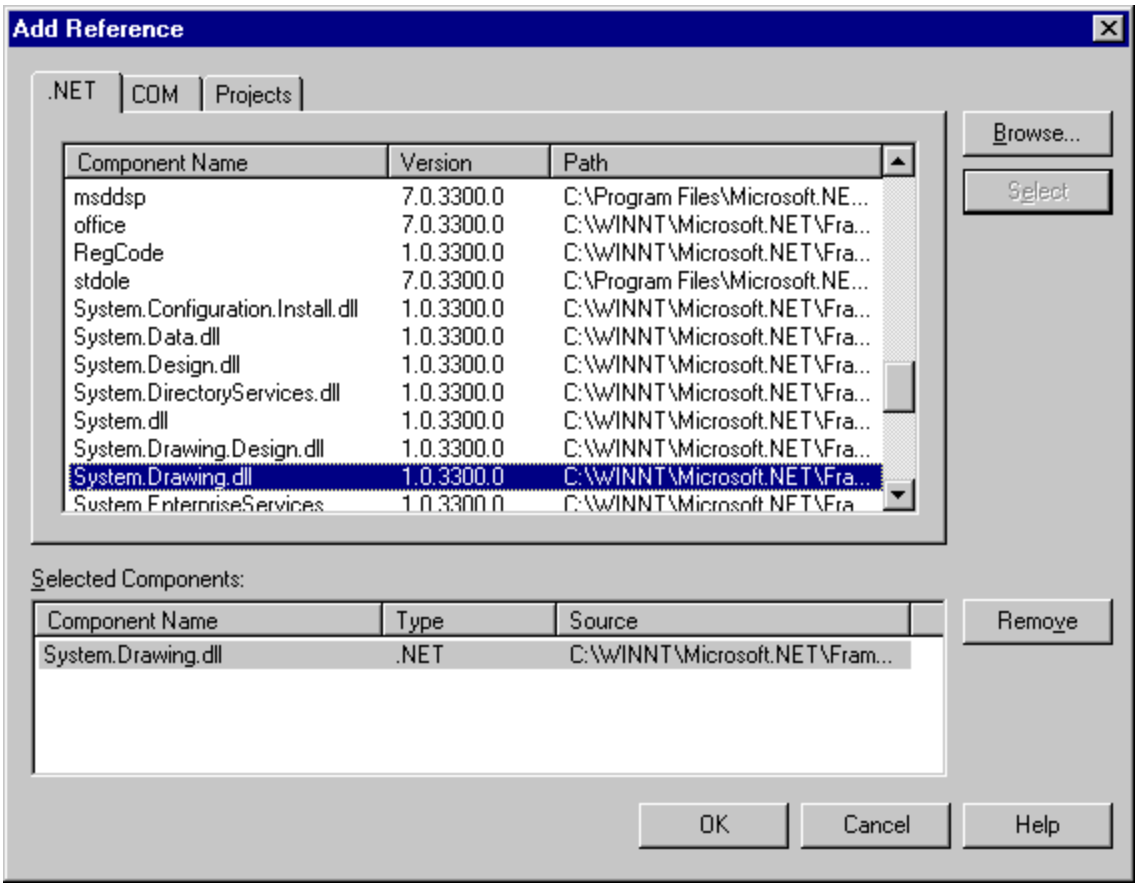


Figure 1. Adding reference to System.Drawing.dll

After that add these two lines.

```
1. using System.Drawing;
2. using System.Drawing.Drawing2D;
```

Note: If you create a Windows application using VS.NET, you only need write only one line.

```
1. using System.Drawing.Drawing2D;
```

After that add a Form_Paint event handler using the Properties Window. See Figure 2.

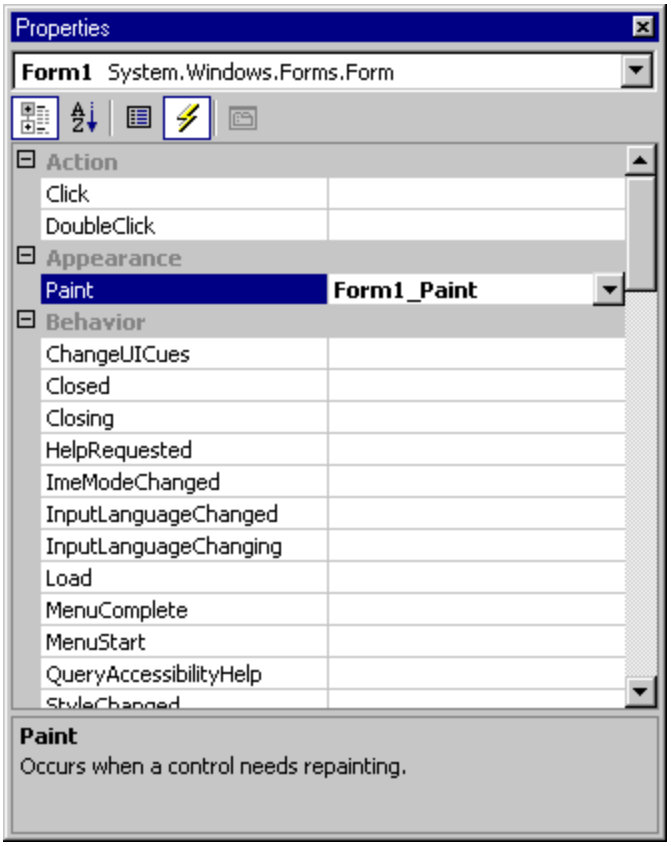


Figure 2. Adding Form_Paint event handler.

Note: You can also add Form paint event handler manually described above.

Graphics Objects

After creating a **Graphics** object, you can use it draw lines, fill shapes, draw text and so on. The major objects are:

Brush	Used to fill enclosed surfaces with patterns,colors, or bitmaps.
Pen	Used to draw lines and polygons, including rectangles, arcs, and pies

Font	Used to describe the font to be used to render text
Color	Used to describe the color used to render a particular object. In GDI+ color can be alpha blended

The Pen Class

A pen draws a line of specified width and style. You always use Pen constructor to create a pen. The constructor initializes a new instance of the **Pen class**. You can initialize it with a color or brush.

Initializes a new instance of the **Pen** class with the specified color.

```
1. public Pen(Color);
```

Initializes a new instance of the **Pen** class with the specified Brush.

```
1. public Pen(Brush);
```

Initializes a new instance of the **Pen** class with the specified Brush and width.

```
1. public Pen(Brush, float);
```

Initializes a new instance of the **Pen** class with the specified Color and Width.

```
1. public Pen(Color, float);
```

Here is one example:

```
1. Pen pn = new Pen( Color.Blue );
```

or

```
1. Pen pn = new Pen( Color.Blue, 100 );
```

Some of its most commonly used properties are:

Alignment	Gets or sets the alignment for objects drawn with this Pen.

Brush	Gets or sets the Brush that determines attributes of this Pen.
Color	Gets or sets the color of this Pen.
Width	Gets or sets the width of this Pen.

The Color Structure

A Color structure represents an ARGB color. Here are ARGB properties of it:

A	Gets the alpha component value for this **Color**.
B	Gets the blue component value for this **Color**.
G	Gets the green component value for this **Color**.
R	Gets the red component value for this **Color**.

You can call the Color members. Each color name (say Blue) is a member of the Color structure. Here is how to use a Color structure:

```
1. Pen pn = new Pen( Color.Blue );
```

The Font Class

The **Font class** defines a particular format for text such as font type, size, and style attributes. You use font constructor to create a font.

Initializes a new instance of the **Font** class with the specified attributes.

```
1. public Font(string, float);
```

Initializes a new instance of the **Font class** from the specified existing **Font** and **FontStyle**.

```
1. public Font(Font, FontStyle);
```

Where FontStyle is an enumeration and here are its members:

Member Name	Description
Bold	Bold text.
Italic	Italic text.

Regular	Normal text.
Strikeout	Text with a line through the middle.
Underline	Underlined text.

Here is one example:

```
1. Graphics g ;
2. Font font = new Font("Times New Roman", 26);
```

Some of its most commonly used properties are:

Bold	Gets a value indicating whether this **Font** is bold.
FontFamily	Gets the FontFamily of this **Font**.
Height	Gets the height of this **Font**.
Italic	Gets a value indicating whether this **Font** is Italic.
Name	Gets the face name of this **Font**.
Size	Gets the size of this **Font**.
SizeInPoints	Gets the size, in points, of this **Font**.
Strikeout	Gets a value indicating whether this **Font** is strikeout (has a line through it).
Style	Gets style information for this **Font**.
Underline	Gets a value indicating whether this **Font** is underlined.
Unit	Gets the unit of measure for this **Font**.

The Brush Class

The **Brush** class is an abstract base class and cannot be instantiated. We always use its derived classes to instantiate a brush object, such as SolidBrush, TextureBrush, RectangleGradientBrush, and LinearGradientBrush.

Here is one example:

```
1. LinearGradientBrush lBrush = new LinearGradientBrush(rect, Color.Red, Color.Yellow, Li
   nearGradientMode.BackwardDiagonal);
```

OR

```
1.  Brush brsh = new SolidBrush(Color.Red), 40, 40, 140, 140);
```

The SolidBrush class defines a brush made up of a single color. Brushes are used to fill graphics shapes such as rectangles, ellipses, pies, polygons, and paths.

The TextureBrush encapsulates a Brush that uses an fills the interior of a shape with an image.

The *LinearGradiantBrush* encapsulates both two-color gradients and custom multi-color gradients.

The Rectangle Structure

```
1.  public Rectangle(Point, Size); or public Rectangle(int, int, int, int);
```

The Rectangle structure is used to draw a rectangle on WinForms. Besides its constructor, the Rectangle structure has following members:

Bottom	Gets the y-coordinate of the lower-right corner of the rectangular region defined by this **Rectangle**.
Height	Gets or sets the width of the rectangular region defined by this **Rectangle**.
IsEmpty	Tests whether this **Rectangle** has a Width or a Height of 0.
Left	Gets the x-coordinate of the upper-left corner of the rectangular region defined by this **Rectangle**.
Location	Gets or sets the coordinates of the upper-left corner of the rectangular region represented by this **Rectangle**.
Right	Gets the x-coordinate of the lower-right corner of the rectangular region defined by this **Rectangle**.
Size	Gets or sets the size of this **Rectangle**.
Top	Gets the y-coordinate of the upper-left corner of the rectangular region defined by this **Rectangle**.
Width	Gets or sets the width of the rectangular region defined by

	this **Rectangle**.
X	Gets or sets the x-coordinate of the upper-left corner of the rectangular region defined by this **Rectangle**.
Y	Gets or sets the y-coordinate of the upper-left corner of the rectangular region defined by this **Rectangle**.

Its constructor initializes a new instance of the Rectangle class. Here is the definition:

```
1. public Rectangle(Point, Size); or public Rectangle(int, int, int, int);
```

The Point Structure

This structure is similar to the POINT structure in C++. It represents an ordered pair of x and y coordinates that define a point in a two-dimensional plane. The member x represents the x coordinates and y represents the y coordinates of the plane.

Here is how to instantiate a point structure:

```
1. Point pt1 = new Point( 30, 30);
2. Point pt2 = new Point( 110, 100);
```

Some sample Examples:

Drawing a rectangle

You can override OnPaint event of your form to draw an rectangle. The LinearGradientBrush encapsulates a brush and linear gradient.

```
1. protected override void OnPaint(PaintEventArgs pe)
2. {
3.     Graphics g = pe.Graphics ;
4.     Rectangle rect = new Rectangle(50, 30, 100, 100);
5.     LinearGradientBrush lBrush = new LinearGradientBrush(rect, Color.Red, Color.Yellow,
    LinearGradientMode.BackwardDiagonal);
6.     g.FillRectangle(lBrush, rect);
```

```
7.  }
```

The output of the above code looks like Figure 3.

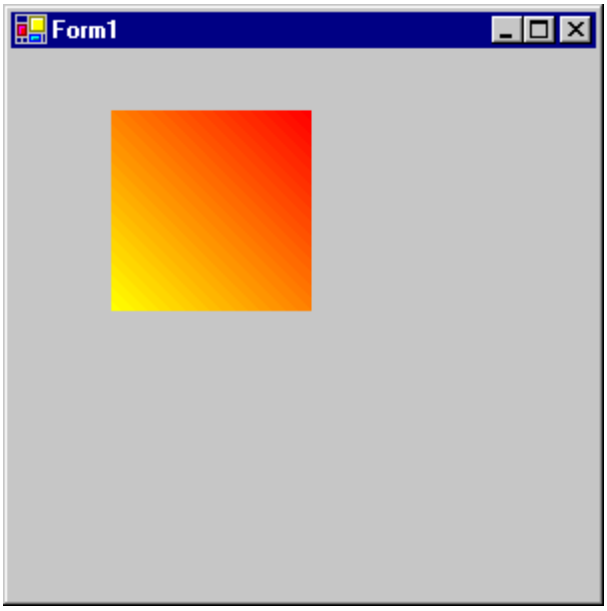


Figure 3. Drawing a rectangle.

Drawing an Arc

DrawArc function draws an arc. This function takes four arguments.

First is the Pen. You create a pen by using the **Pen** class. The Pen constructor takes at least one argument, the color or the brush of the pen. Second argument width of the pen or brush is optional.

```
1.  Pen pn = new Pen( Color.Blue );
```

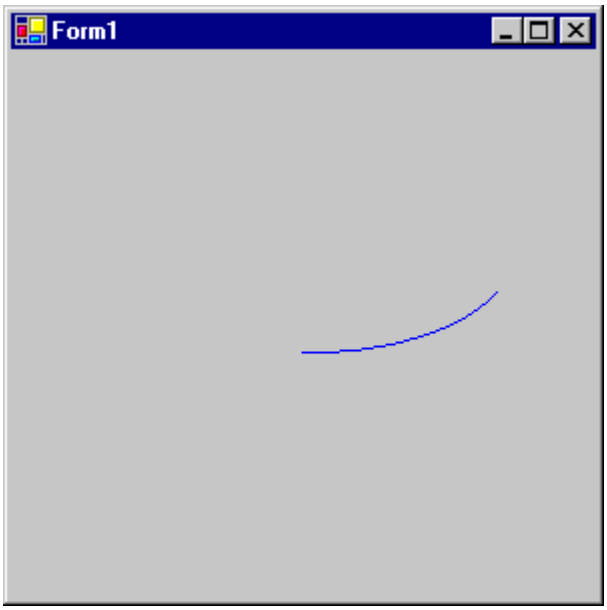
or

```
1.  Pen pn = new Pen( Color.Blue, 100 );
```

The second argument is a rectangle. You can create a rectangle by using Rectangle structure. The Rectangle constructor takes four int type arguments and they are left and right corners of the rectangle.

```
1.  Rectangle rect = new Rectangle(50, 50, 200, 100);
2.  protected override void OnPaint(PaintEventArgs pe)
3.  {
4.      Graphics g = pe.Graphics ;
5.      Pen pn = new Pen( Color.Blue );
6.      Rectangle rect = new Rectangle(50, 50, 200, 100);
7.      g.DrawArc( pn, rect, 12, 84 );
8.  }
```

The output looks like this:



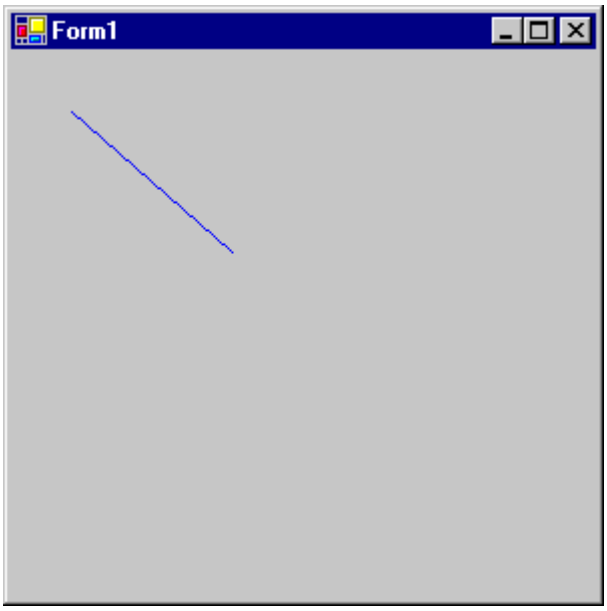
Drawing a Line

DrawLine function of the Graphics class draws a line. It takes three parameters, a pen, and two Point class parameters, starting and ending points. Point class constructor takes x, y arguments.

```
1.  protected override void OnPaint(PaintEventArgs pe)
2.  {
3.      Graphics g = pe.Graphics ;
4.      Pen pn = new Pen( Color.Blue );
5.      // Rectangle rect = new Rectangle(50, 50, 200, 100);
```

```
6.      Point pt1 = new Point( 30, 30);
7.      Point pt2 = new Point( 110, 100);
8.      g.DrawLine( pn, pt1, pt2 );
9.  }
```

The output looks like this:

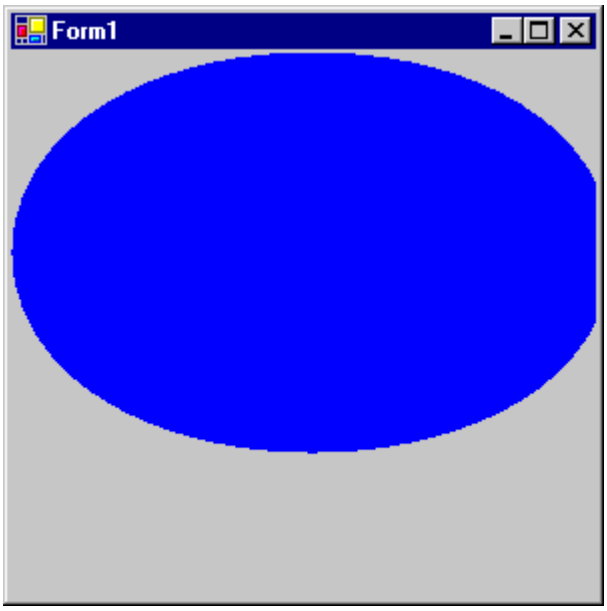


Drawing an Ellipse

An ellipse(or a circle) can be drawn by using DrawEllipse method. This method takes only two parameters, Pen and rectangle.

```
1.  protected override void OnPaint(PaintEventArgs pe)
2.  {
3.      Graphics g = pe.Graphics ;
4.      Pen pn = new Pen( Color.Blue, 100 );
5.      Rectangle rect = new Rectangle(50, 50, 200, 100);
6.      g.DrawEllipse( pn, rect );
7.  }
```

The output looks like this:



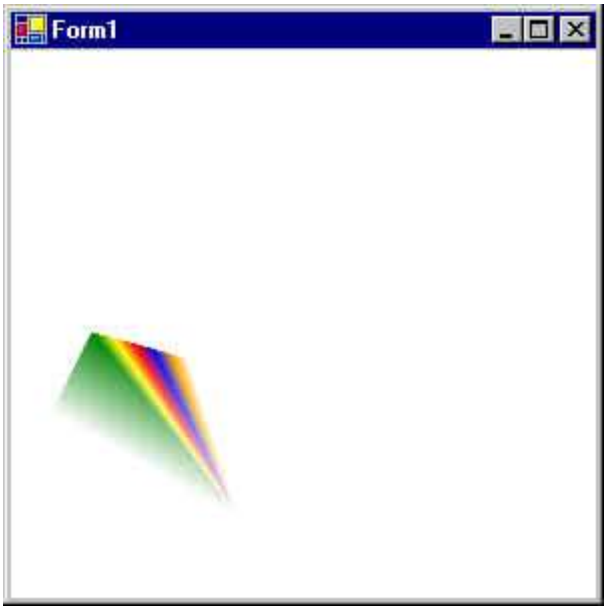
The FillPath

Drawing bazier curves is little more complex than other objects.

```
protected override void OnPaint(PaintEventArgs pe)
{
    Graphics g = pe.Graphics;
    g.FillRectangle(new SolidBrush(Color.White), ClientRectangle);
    GraphicsPath path = new GraphicsPath(new Point[] {
        new Point(40, 140), new Point(275, 200),
        new Point(105, 225), new Point(190, 300),
        new Point(50, 350), new Point(20, 180), },
    new byte[] {
        (byte)PathPointType.Start,
        (byte)PathPointType.Bezier,
        (byte)PathPointType.Bezier,
        (byte)PathPointType.Bezier,
        (byte)PathPointType.Line,
        (byte)PathPointType.Line,
    });
```

```
17.     PathGradientBrush pgb = new PathGradientBrush(path);
18.     pgb.SurroundColors = new Color[] { Color.Green,Color.Yellow,Color.Red, Color.Blue,
19. Color.Orange, Color.White, };
20.     g.FillPath(pgb, path);
21. }
```

The output looks like this:



Drawing Text and Strings

You can override OnPaint event of your form to draw an rectangle. The LinearGradientBrush encapsulates a brush and linear gradient.

```
1.  protected override void OnPaint(PaintEventArgs pe)
2.  {
3.      Font fnt = new Font("Verdana", 16);
4.      Graphics g = pe.Graphics;
5.      g.DrawString("GDI+ World", fnt, new SolidBrush(Color.Red), 14,10);
6.  }
```

The output looks like this: