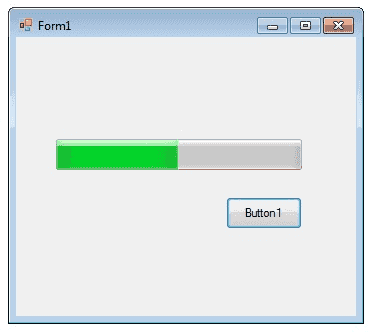
**C# ProgressBar Control**

A progress bar is a control that an application can use to indicate the progress of a lengthy operation such as calculating a complex result, downloading a large file from the Web etc.



ProgressBar controls are used whenever an operation takes more than a short period of time. The Maximum and Minimum properties define the range of values to represent the progress of a task.

**Minimum : Sets the lower value for the range of valid values for progress.**

**Maximum : Sets the upper value for the range of valid values for progress.**

**Value : This property obtains or sets the current level of progress.**

By default, Minimum and Maximum are set to 0 and 100. As the task proceeds, the ProgressBar fills in from the left to the right. To delay the program briefly so that you can view changes in the progress bar clearly.

The following C# program shows a simple operation in a progressbar .

private void button1\_Click(object sender, EventArgs e)

{

int i;

progressBar1.Minimum = 0;

progressBar1.Maximum = 200;

for (i = 0; i <= 200; i++)

{

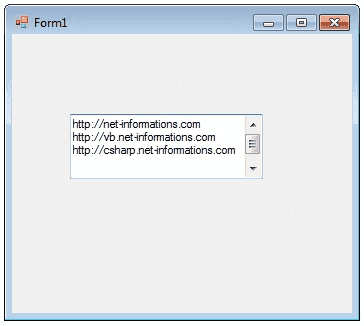
progressBar1.Value = i;

}

}

**C# ScrollBars Control**

A ScrollBar allows you to view content that is outside of the current viewing area by sliding the Thumb to make the content visible.



The ScrollBar control contains a Track control. The Track control consists of a Thumb control and two RepeatButton controls. You can increase and decrease the Value property of the ScrollBar control by pressing the RepeatButton controls or by moving the Thumb. You can set the Value property yourself in code, which moves the scroll box to match. The Minimum and Maximum properties determine the range of values that the control can display. The default range of values for the Value property is from 0 to 1.

The following C# program shows a TextBox control with scrollbars.

private void Form1\_Load(object sender, EventArgs e)

{

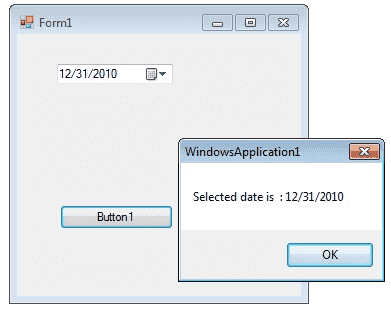
textBox1.Multiline = true;

textBox1.ScrollBars = ScrollBars.Both;

}

**C# DateTimePicker Control**

The DateTimePicker control allows you to display and collect date and time from the user with a specified format.



The DateTimePicker control has two parts, a label that displays the selected date and a popup calendar that allows users to select a new date. The most important property of the DateTimePicker is the Value property, which holds the selected date and time.

**dateTimePicker1.Value = DateTime.Today;**

The Value property contains the current date and time the control is set to. You can use the Text property or the appropriate member of Value to get the date and time value.

**DateTime iDate;**

**iDate = dateTimePicker1.Value;**

The control can display one of several styles, depending on its property values. The values can be displayed in four formats, which are set by the Format property: Long, Short, Time, or Custom.

**dateTimePicker1.Format = DateTimePickerFormat.Short;**

The following C# program shows how to set and get the value of a DateTimePicker1 control.

private void Form1\_Load(object sender, EventArgs e)

{

dateTimePicker1.Format = DateTimePickerFormat.Short;

dateTimePicker1.Value = DateTime.Today;

}

private void button1\_Click(object sender, EventArgs e)

{

DateTime iDate;

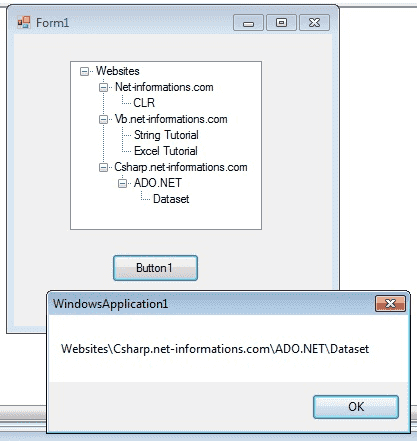
iDate = dateTimePicker1.Value;

MessageBox.Show("Selected date is " + iDate);

}

**C# Treeview Control**

The TreeView control contains a hierarchy of TreeViewItem controls. It provides a way to display information in a hierarchical structure by using collapsible nodes . The top level in a tree view are root nodes that can be expanded or collapsed if the nodes have child nodes.



You can explicitly define the TreeView content or a data source can provide the content. The user can expand the TreeNode by clicking the plus sign (+) button, if one is displayed next to the TreeNode, or you can expand the TreeNode by calling the TreeNode.Expand method. You can also navigate through tree views with various properties: FirstNode, LastNode, NextNode, PrevNode, NextVisibleNode, PrevVisibleNode.

The fullpath method of treeview control provides the path from root node to the selected node.

**treeView1.SelectedNode.FullPath.ToString ();**

Tree nodes can optionally display check boxes. To display the check boxes, set the CheckBoxes property of the TreeView to true.

**treeView1.CheckBoxes = true;**

The following C# program shows a simple demonstration of treeview control

private void Form1\_Load(object sender, EventArgs e)

{

TreeNode tNode ;

tNode = treeView1.Nodes.Add("Websites");

treeView1.Nodes[0].Nodes.Add("Net-informations.com");

treeView1.Nodes[0].Nodes[0].Nodes.Add("CLR");

treeView1.Nodes[0].Nodes.Add("Vb.net-informations.com");

treeView1.Nodes[0].Nodes[1].Nodes.Add("String Tutorial");

treeView1.Nodes[0].Nodes[1].Nodes.Add("Excel Tutorial");

treeView1.Nodes[0].Nodes.Add("Csharp.net-informations.com");

treeView1.Nodes[0].Nodes[2].Nodes.Add("ADO.NET");

treeView1.Nodes[0].Nodes[2].Nodes[0].Nodes.Add("Dataset");

}

private void button1\_Click(object sender, EventArgs e)

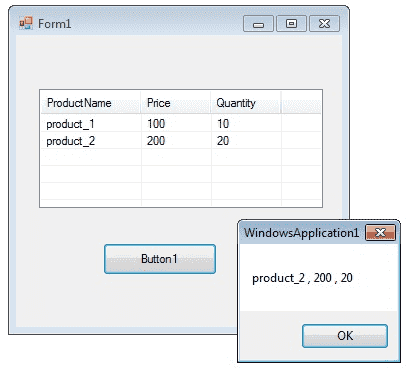
{

MessageBox.Show(treeView1.SelectedNode.FullPath.ToString ());

}

**C# ListView Control**

The ListView control is an ItemsControl that is derived from ListBox.



ListView provides a large number of properties that provide flexibility in appearance and behavior. The View property allows you to change the way in which items are displayed. The SelectionMode property for a ListView determines how many items a user can select at one time.

The following C# program first set its view property as Details and GridLines property as true and FullRowSelect as true.

**listView1.View = View.Details;**

**listView1.GridLines = true;**

**listView1.FullRowSelect = true;**

After that it fills column header and then the column values.

**listView1.Columns.Add("ProductName", 100);**

Finally in the button click event, it will display the selected row values in a message box.

private void Form1\_Load(object sender, EventArgs e)

{

listView1.View = View.Details;

listView1.GridLines = true;

listView1.FullRowSelect = true;

//Add column header

listView1.Columns.Add("ProductName", 100);

listView1.Columns.Add("Price", 70);

listView1.Columns.Add("Quantity", 70);

//Add items in the listview

string[] arr = new string[4];

ListViewItem itm ;

//Add first item

arr[0] = "product\_1";

arr[1] = "100";

arr[2] = "10";

itm = new ListViewItem(arr);

listView1.Items.Add(itm);

//Add second item

arr[0] = "product\_2";

arr[1] = "200";

arr[2] = "20";

itm = new ListViewItem(arr);

listView1.Items.Add(itm);

}

private void button1\_Click(object sender, EventArgs e)

{

string productName = null;

string price = null;

string quantity = null;

productName = listView1.SelectedItems[0].SubItems[0].Text;

price = listView1.SelectedItems[0].SubItems[1].Text;

quantity = listView1.SelectedItems[0].SubItems[2].Text;

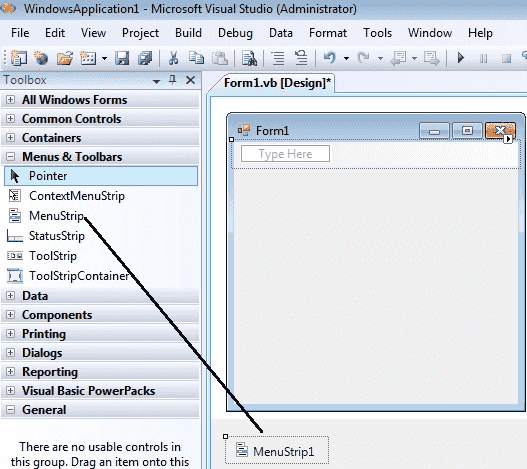
MessageBox.Show (productName + " , " + price + " , " + quantity);

}

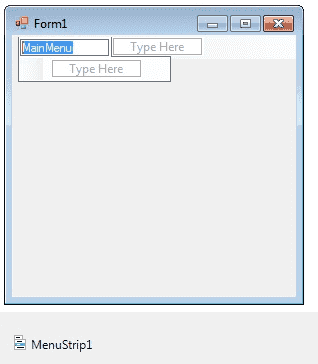
**C# Menu Control**

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.

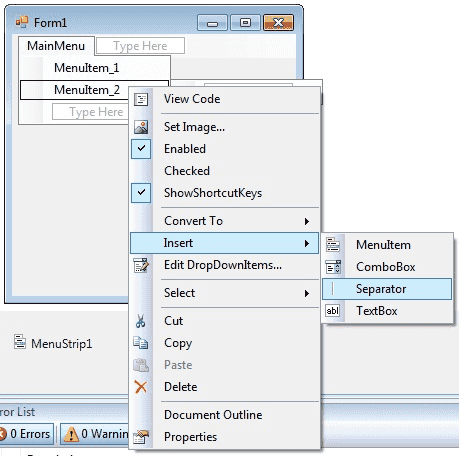
You can add menus to Windows Forms at design time by adding the MainMenu component and then appending menu items to it using the Menu Designer.



After drag the Menustrip on your 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 you can understand how to create each menu items on mainmenu Object.



If you need a seperator bar , right click on your 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.

private void menu1ToolStripMenuItem\_Click(object sender, EventArgs e)

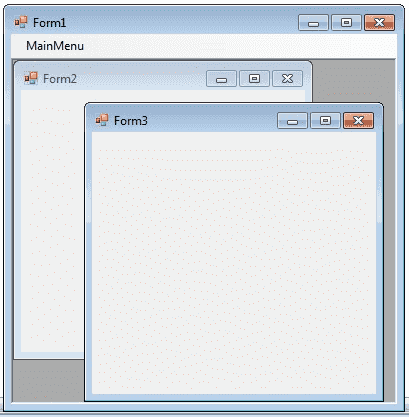
{

MessageBox.Show("You are selected MenuItem\_1");

}

**C# 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 you 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 you will get a default form Form1 . Then add two mnore forms in the project (Form2 , Form 3) . Create a Menu on your form and call these two forms on menu click event. Click here to see how to create a Menu on your form [How to Menu Control C#](http://csharp.net-informations.com/gui/cs-menu.htm).

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

**form.MdiParent = this;**

**form.Dock=DockStyle.Fill;**

**form.Show();**

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();

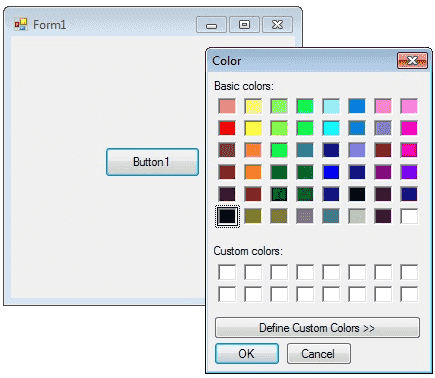
frm3.Show();

frm3.MdiParent = this;

}

**C# Color Dialog Box**

There are several classes that implement common dialog boxes, such as color selection , print setup etc.



A ColorDialog object is a dialog box with a list of colors that are defined for the display system. The user can select or create a particular color from the list, which is then reported back to the application when the dialog box exits. You can invite a color dialog box by calling ShowDialog() method.

**ColorDialog dlg = new ColorDialog();**

**dlg.ShowDialog();**

The following C# program invites a color dialog box and retrieve the selected color to a string.

private void button1\_Click(object sender, EventArgs e)

{

ColorDialog dlg = new ColorDialog();

dlg.ShowDialog();

if (dlg.ShowDialog() == DialogResult.OK)

{

string str = null;

str = dlg.Color.Name;

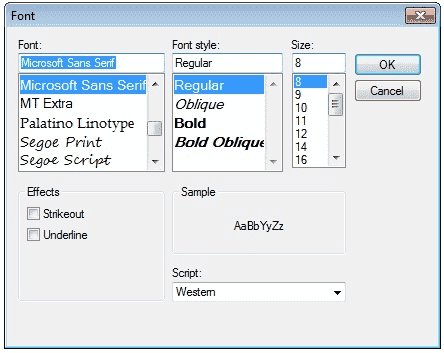
MessageBox.Show (str);

}

}

**C# Font Dialog Box**

Font dialog box represents a common dialog box that displays a list of fonts that are currently installed on the system. The Font dialog box lets the user choose attributes for a logical font, such as font family and associated font style, point size, effects , and a script .



The following C# program invites a Font Dialog Box and retrieve the selected Font Name and Font Size.

private void button1\_Click(object sender, EventArgs e)

{

FontDialog dlg = new FontDialog();

dlg.ShowDialog();

if (dlg.ShowDialog() == DialogResult.OK)

{

string fontName;

float fontSize;

fontName = dlg.Font.Name;

fontSize = dlg.Font.Size;

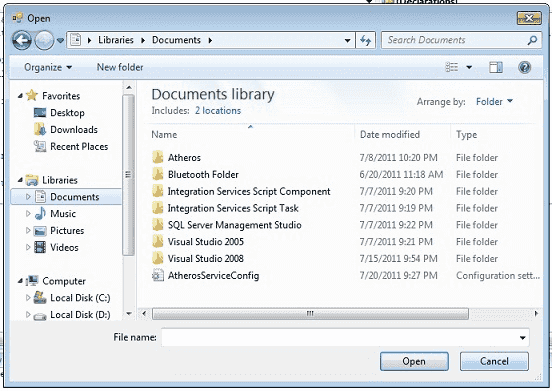
MessageBox.Show(fontName + " " + fontSize );

}

}

**C# OpenFile Dialog Box**

The OpenFileDialog component allows users to browse the folders of their computer or any computer on the network and select one or more files to open. The dialog box returns the path and name of the file the user selected in the dialog box.



The FileName property can be set prior to showing the dialog box. This causes the dialog box to initially display the given filename. In most cases, your applications should set the InitialDirectory, Filter, and FilterIndex properties prior to calling ShowDialog.

The following C# program invites an OpenFile Dialog Box and retrieve the selected filename to a string.

private void button1\_Click(object sender, EventArgs e)

{

OpenFileDialog dlg = new OpenFileDialog();

dlg.ShowDialog();

if (dlg.ShowDialog() == DialogResult.OK)

{

string fileName;

fileName = dlg.FileName;

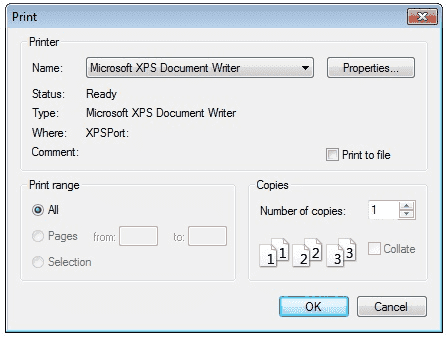
MessageBox.Show(fileName);

}

}

**C# Print Dialog Box**

A user can use the Print dialog box to select a printer, configure it, and perform a print job. Print dialog boxes provide an easy way to implement Print and Print Setup dialog boxes in a manner consistent with Windows standards.



The Print dialog box includes a Print Range group of radio buttons that indicate whether the user wants to print all pages, a range of pages, or only the selected text. The dialog box includes an edit control in which the user can type the number of copies to print. By default, the Print dialog box initially displays information about the current default printer.

private void button1\_Click(object sender, EventArgs e)

{

PrintDialog dlg = new PrintDialog();

dlg.ShowDialog();

}

}

}

**keyPress event in C#**

**Handle Keyboard Input at the Form Level in C#**

Windows Forms processes keyboard input by raising keyboard events in response to Windows messages. Most Windows Forms applications process keyboard input exclusively by handling the keyboard events.

**How do I detect keys pressed in C#**

You can detect most physical key presses by handling the KeyDown or KeyUp events. Key events occur in the following order:

KeyDown

KeyPress

KeyUp

**How to detect when the Enter Key Pressed in C#**

The following C# code behind creates the KeyDown event handler. If the key that is pressed is the Enter key, a MessegeBox will displayed .

**if (e.KeyCode == Keys.Enter)**

**{**

**MessageBox.Show("Enter Key Pressed ");**

**}**

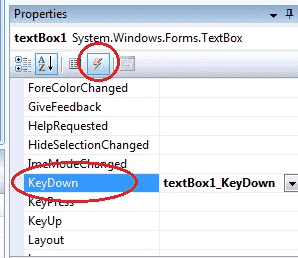
**How to get TextBox1\_KeyDown event in your C# source file ?**

Select your TextBox control on your Form and go to Properties window. Select Event icon on the properties window and scroll down and find the KeyDown event from the list and double click the Keydown Event. The you will get the KeyDown event in your source code editor.

**private void textBox1\_KeyDown(.....)**

**{**

**}**



**Difference between the KeyDown Event, KeyPress Event and KeyUp Event**

**KeyDown Event :** This event raised as soon as the user presses a key on the keyboard, it repeats while the user keeps the key depressed.

**KeyPress Event :** This event is raised for character keys while the key is pressed and then released. This event is not raised by noncharacter keys, unlike KeyDown and KeyUp, which are also raised for noncharacter keys

**KeyUp Event :** This event is raised after the user releases a key on the keyboard.

**KeyPress Event :**

using System;

using System.Windows.Forms;

private void textBox1\_KeyPress(object sender, KeyPressEventArgs e)

{

if (e.KeyChar == (char)Keys.Enter)

{

MessageBox.Show("Enter key pressed");

}

if (e.KeyChar == 13)

{

MessageBox.Show("Enter key pressed");

}

}

**KeyDown Event :**

private void textBox1\_KeyDown(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.Enter)

{

MessageBox.Show("Enter key pressed");

}

}

**KeyUp Event :**

The following C# source code shows how to capture Enter KeyDown event from a TextBox Control.

private void textBox1\_KeyUp(object sender, KeyEventArgs e)

{

if (e.KeyCode == Keys.Enter)

{

MessageBox.Show("Enter key pressed");

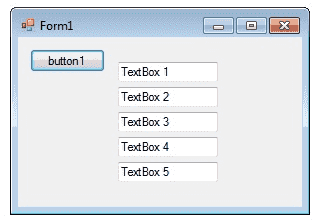
}

}

**How to create Dynamic Controls in C# ?**

**How to create Control Arrays in C# ?**

Visual Studio .NET does not have control arrays like Visual Basic 6.0 does. The good news is that you can still set things up to do similar things. The advantages of C# dynamic controls is that they can be created in response to how the user interacts with the application. Common controls that are added during run-time are the Button and TextBox controls. But of course, nearly every C# control can be created dynamically.



**How to create Dynamic Controls in C# ?**

The following program shows how to create a dynamic TextBox control in C# and setting the properties dynamically for each TextBox control. Drag a Button control in the form and copy and paste the following source code . Here each Button click the program create a new TextBox control dyanmically.

private void button1\_Click(object sender, EventArgs e)

{

AddNewTextBox();

}

public System.Windows.Forms.TextBox AddNewTextBox()

{

System.Windows.Forms.TextBox txt = new System.Windows.Forms.TextBox();

this.Controls.Add(txt);

txt.Top = cLeft \* 25;

txt.Left = 100;

txt.Text = "TextBox " + this.cLeft.ToString();

cLeft = cLeft + 1;

return txt;

}

# Keep Form on Top of All Other Windows

The System.Windows.Forms namespace contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system. You can bring a Form on top of application by simply setting the Form.topmost form property to true will force the form to the top layer of the screen, while leaving the user able to enter data in the forms below.

**Form2 frm = new Form2();**

**frm.TopMost = true;**

**frm.Show();**

Topmost forms are always displayed at the highest point in the z-order of the windows on the desktop. You can use this property to create a form that is always displayed in your application, such as a MessageBox window.

using System;

using System.Windows.Forms;

namespace WindowsFormsApplication1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private void button1\_Click(object sender, EventArgs e)

{

Form2 frm = new Form2();

frm.TopMost = true;

frm.Show();

}

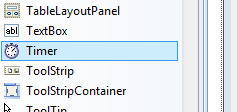
}

}

**C# Timer Control**

**What is Timer Control ?**

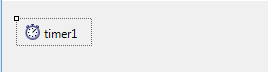
The Timer Control plays an important role in the development of programs both Client side and Server side development as well as in Windows Services. With the **Timer Control** we can raise events at a specific interval of time without the interaction of another thread.



**Use of Timer Control**

We require Timer Object in many situations on our development environment. We have to use Timer Object when we want to set an interval between events, periodic checking, to start a process at a fixed time schedule, to increase or decrease the speed in an animation graphics with time schedule etc.

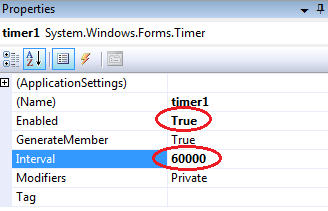
A Timer control does not have a visual representation and works as a component in the background.



**How to Timer Control ?**

We can control programs with Timer Control in millisecond, seconds, minutes and even in hours. The Timer Control allows us to set Intervel property in milliseconds. That is, one second is equal to 1000 milliseconds. For example, if we want to set an interval of 1 minute we set the value at Interval property as 60000, means 60x1000 .

By default the Enabled property of Timer Control is False. So before running the program we have to set the Enabled property is True , then only the Timer Control starts its function.



**Timer example**

In the following program we display the current time in a Label Control. In order to develop this program, we need a Timer Control and a Label Control. Here we set the timer interval as 1000 milliseconds, that means one second, for displaying current system time in Label control for the interval of one second.

private void timer1\_Tick(object sender, EventArgs e)

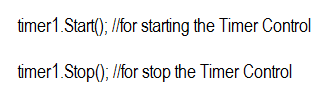
{

label1.Text = DateTime.Now.ToString();

}

**Start and Stop Timer Control**

The Timer control have included the Start and Stop methods for start and stop the Timer control functions.



Here we run this program only 10 seconds. In order to doing this ,in the following program we set Timer interval as 1000 (1 second) and check each seconds for stopping the Timer Control after 10 seconds.

namespace WindowsFormsApplication2

{

public partial class Form1 : Form

{

int second = 0;

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

timer1.Interval = 1000;

timer1.Start();

}

private void timer1\_Tick(object sender, EventArgs e)

{

label1.Text = DateTime.Now.ToString();

second = second + 1;

if (second >= 10)

{

timer1.Stop();

MessageBox.Show("Exiting from Timer....");

}

}

}

}