



**JAIN**  
DEEMED-TO-BE UNIVERSITY

SCHOOL OF COMPUTER  
APPLICATIONS  
&  
INFORMATION TECHNOLOGY

Department of Master of Computer Applications

Semester III

C# & .NET Technologies

**Unit IV**

Windows Presentation Foundation

- WPF, which stands for **Windows Presentation Foundation**, is Microsoft's latest approach to a GUI framework, used with the .NET framework.
- A GUI framework allows you to create an application with a wide range of GUI elements, like labels, textboxes and other well known elements.
- There are a lot of GUI frameworks out there, but for .NET developers, the most interesting ones are currently WinForms and WPF.
- WPF is the newest, but Microsoft is still maintaining and supporting WinForms.

Features	WinForms	WPF
<b>Advance</b>	Windows form is old concept for developing desktop applications.	WPF is advanced or latest concept for developing the applications.
<b>Simple</b>	Windows forms are simple to use as controls can be easily used	WPF is complex to use as compared to windows forms
<b>Scalable</b>	Windows forms are less scalable, if the UI element needs to be extend later on	WPF is extensively scalable for the UI elements in applications
<b>Secure</b>	Windows forms has less secure features	WPF has enhanced secure features
<b>Design</b>	Windows forms are not be used where designing is required	WPF is mainly used for designing the UI part of the application
<b>Performance</b>	In windows forms, things are achieved at slower rate	In WPF things are mainly achieved at very fast rate comparatively.

# The Motivation Behind WPF

**Table 1. .NET 2.0 Solutions to Desired Functionalities**

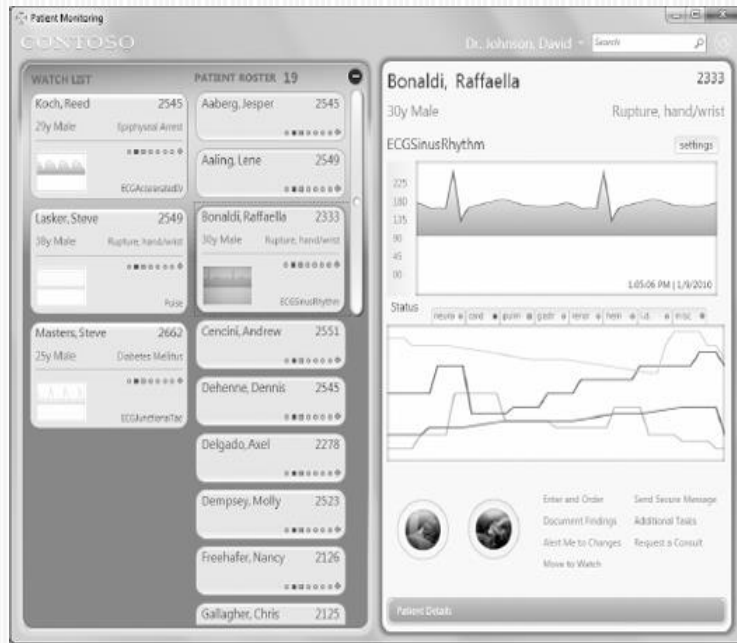
Desired Functionality	.NET 2.0 Solution
Building windows with controls	Windows Forms
2D graphics support	GDI+ (System.Drawing.dll)
3D graphics support	DirectX APIs
Support for streaming video	Windows Media Player APIs
Support for flow-style documents	Programmatic manipulation of PDF files

**Table 2. .NET 3.0 Solutions to Desired Functionalities**

Desired Functionality	.NET 3.0 Solution
Building windows with controls	WPF
2D graphics support	WPF
3D graphics support	WPF
Support for streaming video	WPF
Support for flow-style documents	WPF

- Unifying Diverse APIs
- Providing a Separation of Concerns via XAML
- Providing an Optimized Rendering Model
- Simplifying Complex UI Programming

- Traditional Desktop Applications

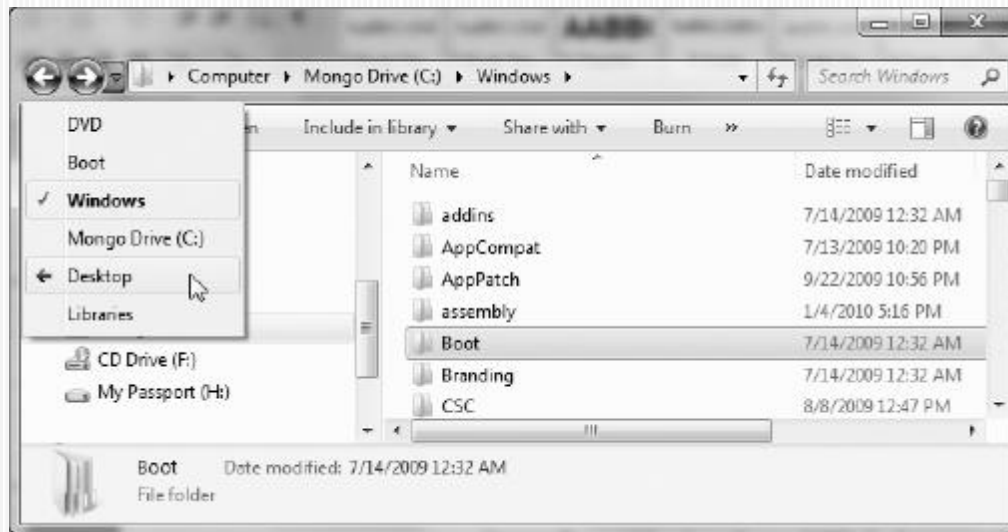


This WPF desktop application makes use of several WPF APIs



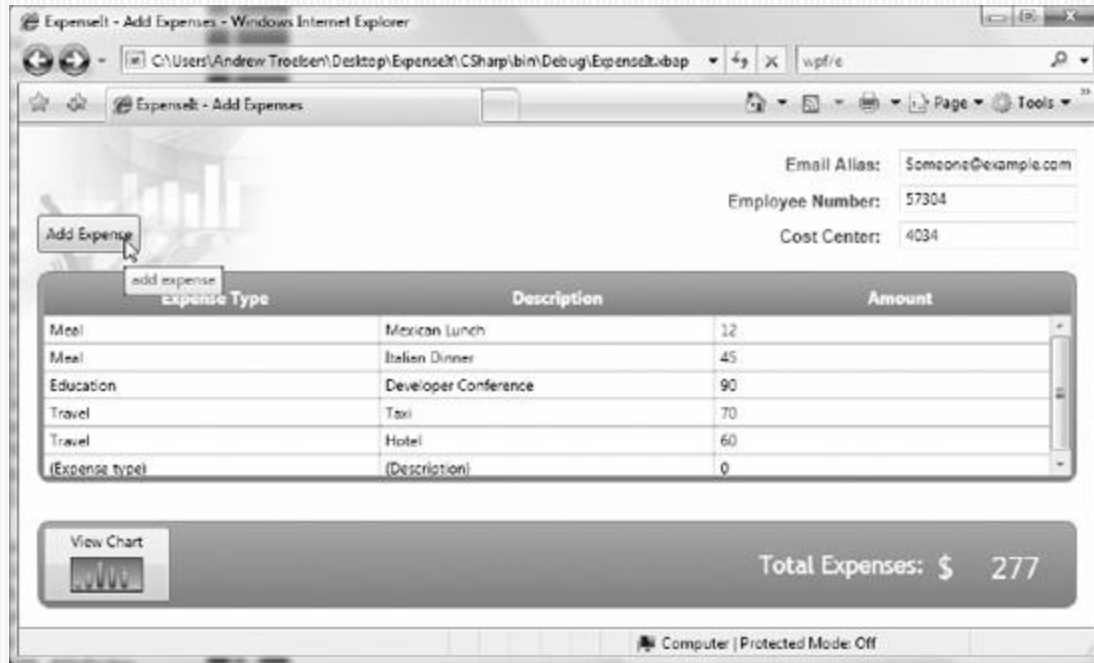
Transformations and animations are very simple under WPF

- **Navigation-Based WPF Applications**



**A navigation-based desktop program**

- **XBAP Applications**



**XBAP programs are downloaded to a local machine and hosted within a web browser**



- **Like any .NET technology, WPF is represented by several assemblies.**
- The Visual Studio WPF project templates automatically set references to the core assemblies.
- However, if you build WPF apps using other IDEs or via **msbuild.exe**, you will need to set assembly references manually.
- The following key libraries are managed .NET assemblies.

WPF Assembly	Meaning in Life
WindowsBase.dll	<p>Defines the base infrastructure of WPF, including dependency properties support.</p> <p>While this assembly contains types used within the WPF framework, the majority of these types can be used within other .NET applications.</p>
PresentationCore.dll	<p>This assembly defines numerous types that constitute the foundation of the WPF GUI layer.</p>
PresentationFramework.dll	<p>This assembly—the ‘meatiest’ of the three—defines the WPF controls types, animation and multimedia support, data binding support, and other WPF services.</p> <p>For all practical purposes, this is the assembly you will spend most of your time working with directly.</p>
System.Xaml.dll	<p>This library (which is new to .NET 4.0) provides types to process and manipulate XAML at runtime.</p>

- **Although these assemblies provide hundreds of types within numerous namespaces, consider this partial list of WPF namespaces:**
- You will encounter other namespaces during the remainder of this class.
- Again, consult the .NET Framework SDK documentation for full details.

WPF Namespace	Meaning in Life
System.Windows	Here you will find core types such as Application and Window that are required by any WPF desktop project.
System.Windows.Controls	Here you will find all of the expected WPF widgets, including types to build menu systems, tool tips, and numerous layout managers.
System.Windows.Markup	This namespace defines a number of types that allow XAML markup and the equivalent binary format, BAML, to be parsed.
System.Windows.Media	Within these namespaces you will find types to work with animations, 3D rendering, text rendering, and other multimedia primitives.
System.Windows.Navigation	This namespace provides types to account for the navigation logic employed by XAML browser applications / desktop navigation apps.
System.Windows.Shapes	This namespace defines basic geometric shapes (Rectangle, Polygon, etc.) used by various aspects of the WPF framework.

```
using System;  
using System.Windows;  
using System.Windows.Controls;
```

```
namespace WpfAppAllCode
```

```
{
```

```
// In this first example, you are defining a single class type to  
// represent the application itself and the main window.
```

```
class Program : Application
```

```
{
```

```
    [STAThread]
```

```
    static void Main(string[] args)
```

```
{
```

```
// Handle the Startup and Exit events, and then run the application.
```

```
    Program app = new Program();
```

```
    app.Startup += AppStartUp;
```

```
    app.Exit += AppExit;
```

```
    app.Run(); // Fires the Startup event.
```

```
}
```

```
static void AppExit(object sender, ExitEventArgs e)
{
    MessageBox.Show("App has exited");
}

static void AppStartup(object sender, StartupEventArgs e)
{
    // Create a Window object and set some basic properties.
    Window mainWindow = new Window();
    mainWindow.Title = "My First WPF App!";
    mainWindow.Height = 200;
    mainWindow.Width = 300;
    mainWindow.WindowStartupLocation =
WindowStartupLocation.CenterScreen;
    mainWindow.Show();
}
}
```

- How to get this code going:
- Create a Windows Presentation Foundation project named HelloWorldManual
- Remove the App.xaml and Window1.xaml from the project
- Add a new class file named MyWindow.cs and paste this into it:

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace HelloWorldManual
{
    public class MyWindow : Window
    {
        private Label label1;
        public MyWindow()
        {
            Width = 300;
            Height = 300;

            Grid grid = new Grid();
            Content = grid;

            Button button1 = new Button();
            button1.Content = "Say Hello!";
            button1.Height = 23;
            button1.Margin = new Thickness(96, 50, 107, 0);
            button1.VerticalAlignment = System.Windows.VerticalAlignment.Top;
            button1.Click += new RoutedEventHandler(button1_Click);
            grid.Children.Add(button1);
        }
    }
}
```

```
label1 = new Label();
    label1.Margin = new Thickness(84,115,74,119);
    grid.Children.Add(label1);

}

void button1_Click(object sender, RoutedEventArgs e)
{
    label1.Content = "Hello WPF!";
}

[STAThread]
public static void Main()
{
    Application app = new Application();

    app.Run(new MyWindow());
}
}
```

- How to get this code going:
- Create a Windows Presentation Foundation project named HelloWPF and paste the following code in the specified file.
- Paste this code into **Window1.xaml**:

```
<Window x:Class="HelloWPF.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="Window1"
Height="300" Width="300"> <Grid> <Button Height="23"
Margin="96,50,107,0" Name="button1" VerticalAlignment="Top"
Click="button1_Click">Say Hello!</Button> <Label Margin="84,115,74,119"
Name="label1"></Label> </Grid> </Window>
```



Paste this code into Window1.xaml.cs:

```
using System.Windows;
```

```
namespace HelloWPF
```

```
{  
    public partial class Window1 : Window  
    {  
        public Window1()  
        {  
            InitializeComponent();  
        }  
  
        private void button1_Click(object sender, RoutedEventArgs e)  
        {  
            label1.Content = "Hello WPF!";  
        } } }
```

## Basic Syntax

- When you create your new WPF project, you will encounter some of the XAML code by default in MainWindow.xaml as shown below.

```
<Window x:Class = "Resources.MainWindow"
    xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
    Title = "MainWindow" Height = "350" Width = "525">

    <Grid>

    </Grid>

</Window>
```

Information	Description
<code>&lt;Window</code>	It is the opening object element or container of the root.
<code>x:Class = "Resources.MainWindow"</code>	It is a partial class declaration which connects the markup to the partial class code defined behind.
<code>xmlns = "<u>http://schemas.microsoft.com/winfx/2006/xaml/presentation</u>"</code>	Maps the default XAML namespace for WPF client/framework
<code>xmlns:x = "<u>http://schemas.microsoft.com/winfx/2006/xaml</u>"</code>	XAML namespace for XAML language which maps it to x: prefix
<code>&gt;</code>	End of object element of the root
<code>&lt;Grid&gt;</code> <code>&lt;/Grid&gt;</code>	It is starting and closing tags of an empty grid object.
<code>&lt;/Window&gt;</code>	Closing the object element

- The syntax rules for XAML is almost similar to XML.
- If you look at an XAML document, then you will notice that it is actually a valid XML file, but an XML file is not necessarily an XAML file.
- It is because in XML, the value of the attributes must be a string while in XAML, it can be a different object which is known as Property element syntax.
- The syntax of an Object element starts with a left angle bracket (<) followed by the name of an object, e.g. Button.
- Define some Properties and attributes of that object element.
- The Object element must be closed by a forward slash (/) followed immediately by a right angle bracket (>).

- Example of simple object with no child element

```
<Button/>
```

- Example of object element with some attributes

```
<Button Content = "Click Me" Height = "30" Width = "60" />
```

- Example of an alternate syntax to define properties (Property element syntax)

```
<Button>  
  <Button.Content>Click Me</Button.Content>  
  <Button.Height>30</Button.Height>  
  <Button.Width>60</Button.Width>  
</Button>
```

- Example of Object with Child Element: StackPanel contains Textblock as child element

```
<StackPanel Orientation = "Horizontal">  
  <TextBlock Text = "Hello"/>  
</StackPanel>
```

# C# .NET simple example in which a button is created with some properties in XAML

```
<Window x:Class = "WPFXAMLOverview.MainWindow"  
  xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
  xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"  
  Title = "MainWindow" Height = "350" Width = "604">
```

```
<StackPanel>
```

```
<Button x:Name = "button" Content = "Click Me" HorizontalAlignment =  
"Left" Margin = "150" VerticalAlignment = "Top" Width = "75" />
```

```
</StackPanel>
```

```
</Window>
```

```
using System.Windows;
```

```
using System.Windows.Controls;
```

```
namespace WPFXAMLOverview
```

```
{    /// Interaction logic for MainWindow.xaml
```

```
public partial class MainWindow : Window
```

```
{
```

```
    public MainWindow()
```

```
{
```

```
        InitializeComponent();
```

```
        /// Create the StackPanel
```

```
        StackPanel stackPanel = new StackPanel();
```

```
        this.Content = stackPanel;
```

```
        /// Create the Button
```

```
        Button button = new Button();
```

```
        button.Content = "Click Me";
```

```
        button.HorizontalAlignment = HorizontalAlignment.Left;
```

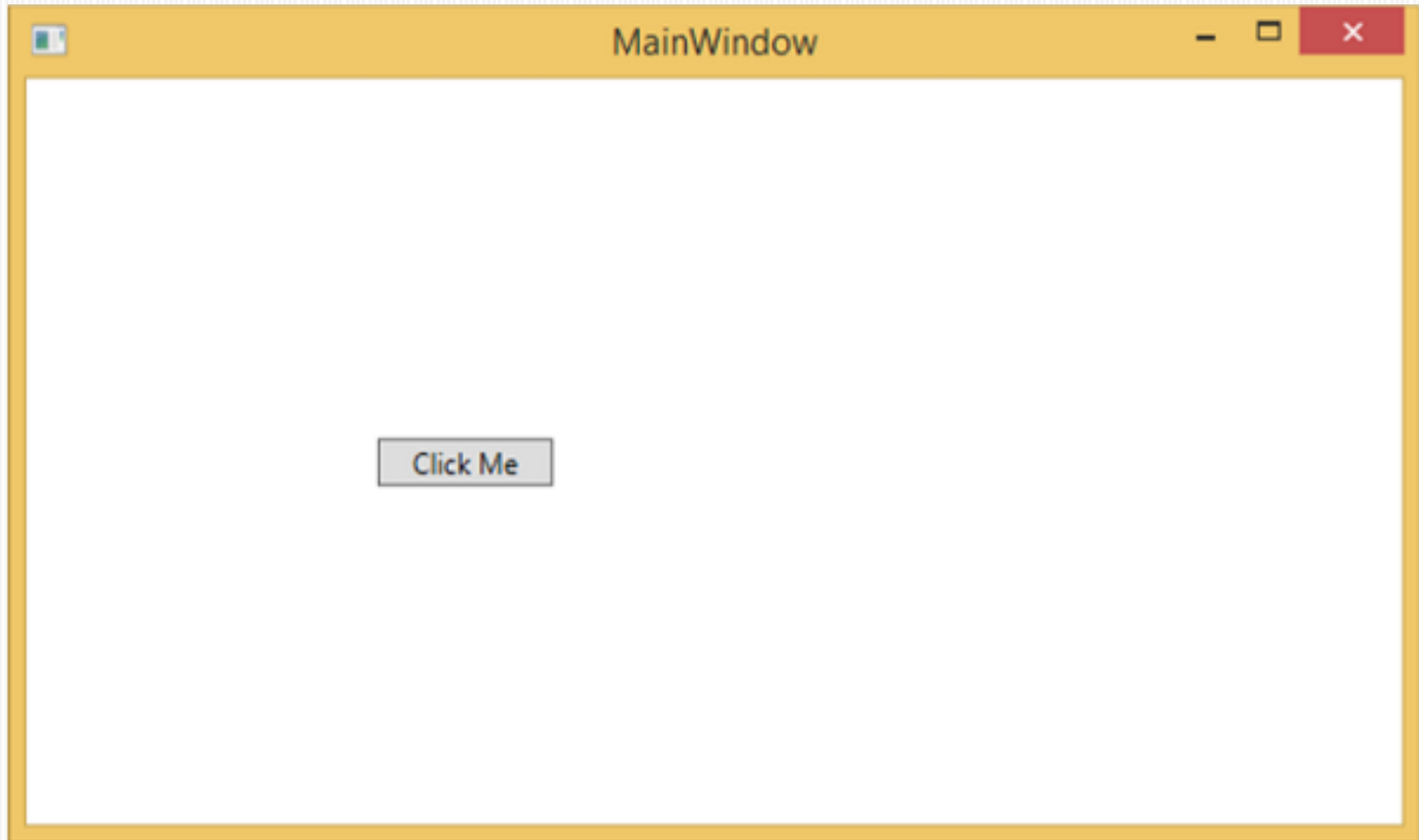
```
        button.Margin = new Thickness(150);
```

```
        button.VerticalAlignment = VerticalAlignment.Top;
```

```
        button.Width = 75;
```

```
        stackPanel.Children.Add(button);
```

```
    } } }
```





- From the above example, it is clear that what you can do in XAML to create, initialize, and set properties of objects, the same tasks can also be done using code.
- XAML is just another simple and easy way to design UI elements.
- With XAML, it doesn't mean that what you can do to design UI elements is the only way. You can either declare the objects in XAML or define them using code.
- XAML is optional, but despite this, it is at the heart of WPF design.
- The goal of XAML is to enable visual designers to create user interface elements directly.
- WPF aims to make it possible to control all visual aspects of the user interface from mark-up.

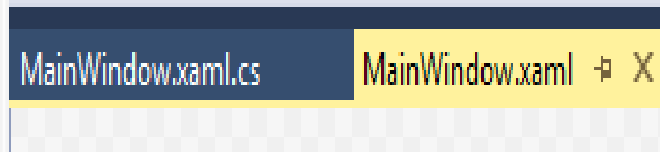
**Step 1)** In Visual Studio Go to File > Project

**Step 2)** In the new project window

- Select WPF App
- Enter Name as "MyWPF"
- Click OK

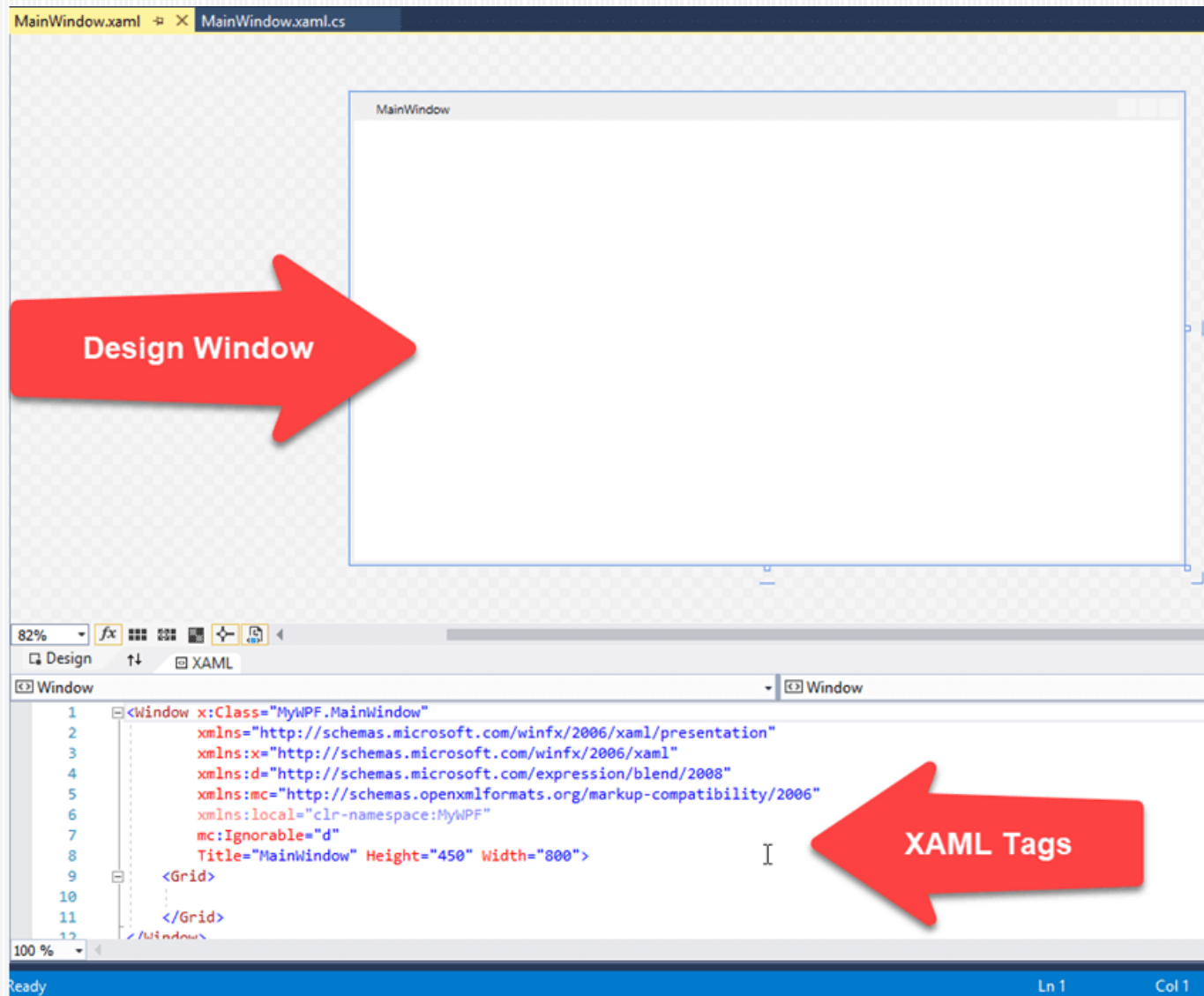
**Step 3)** Visual Studio creates two files by default

- XAML File (MainWindow.xaml)
- CS File (MainWindow.xaml.cs)



The MainWindow.xaml has

- A Design Window
- XAML File



In the XAML windows, the following tags are written by default

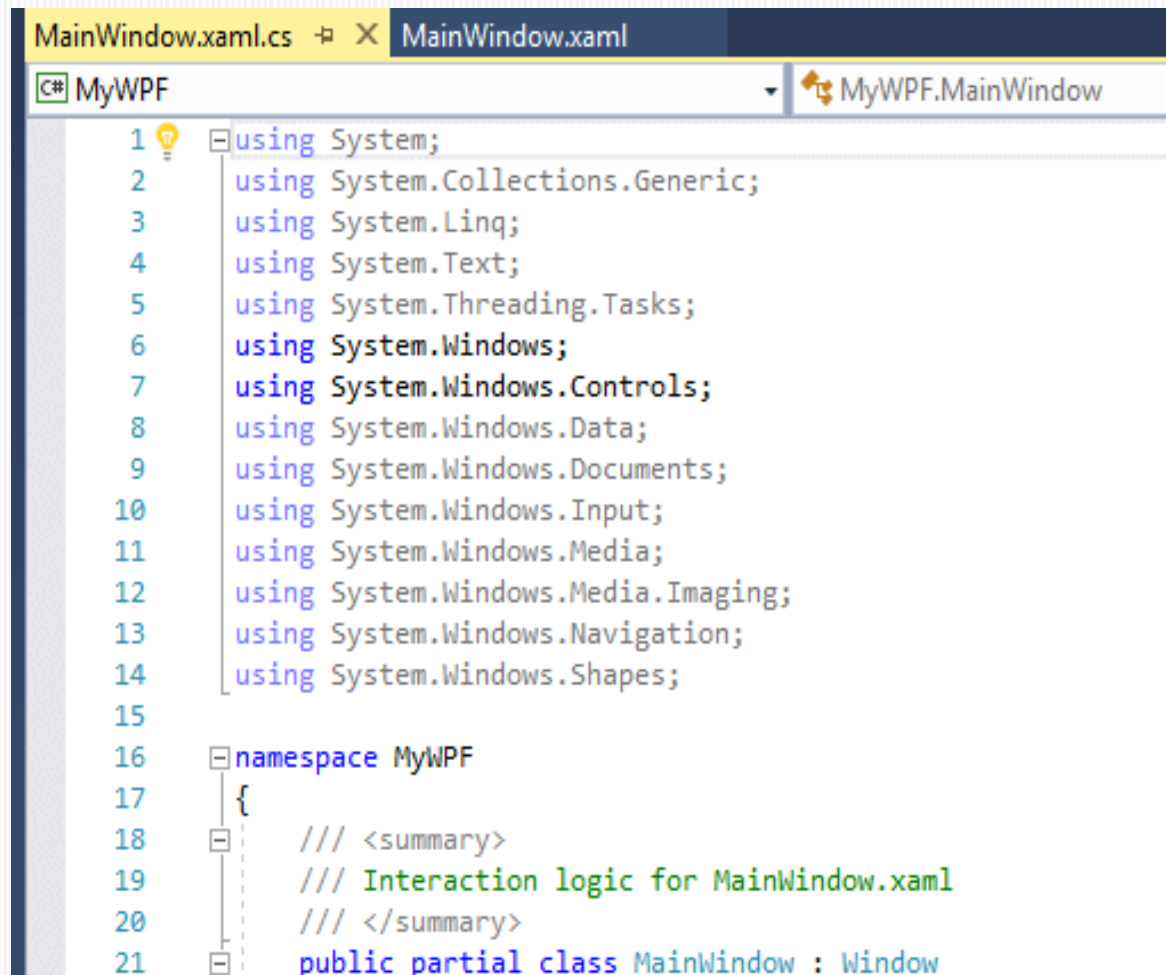


```

1  <Window x:Class="MyWPF.MainWindow"
2      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6      xmlns:local="clr-namespace:MyWPF"
7      mc:Ignorable="d"
8      Title="MainWindow" Height="450" Width="800">
9      <Grid>
10
11  </Grid>
12 </Window>
  
```

The Grid is the first element by default.

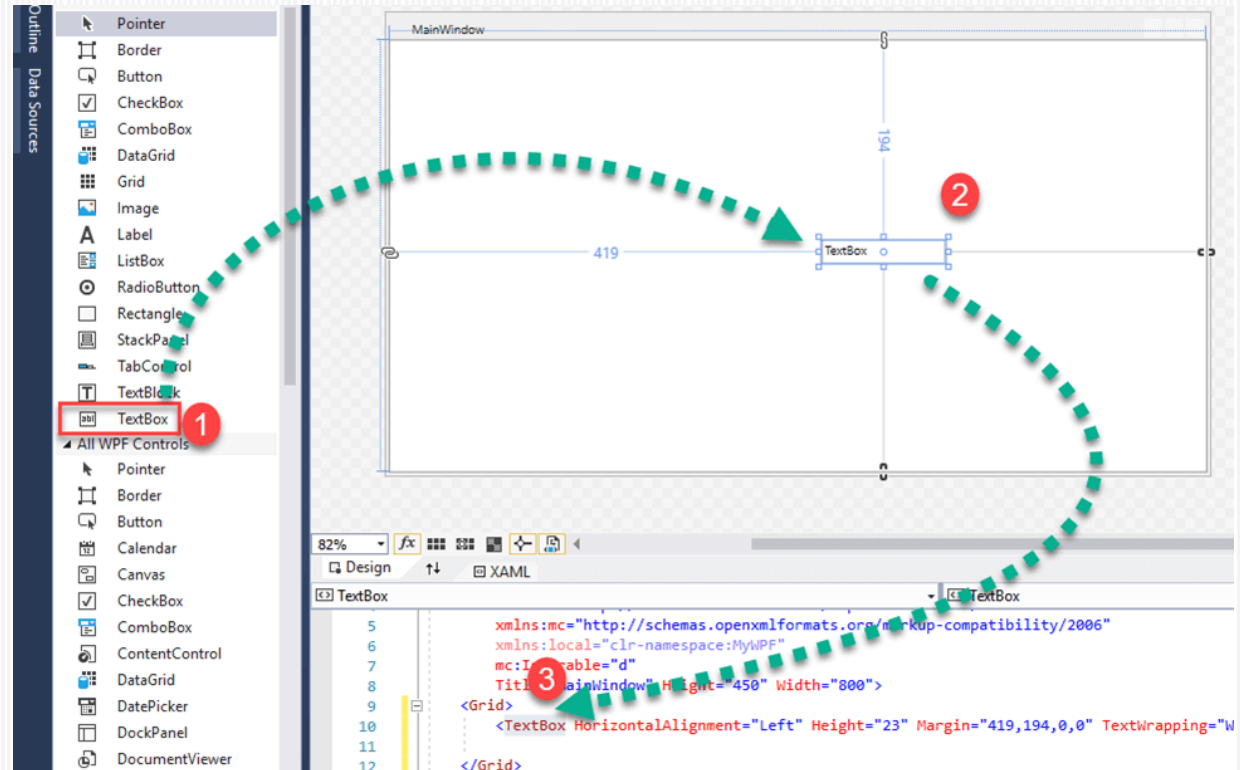
The MainWindow.xaml.cs contains the corresponding code behind the XAML design file



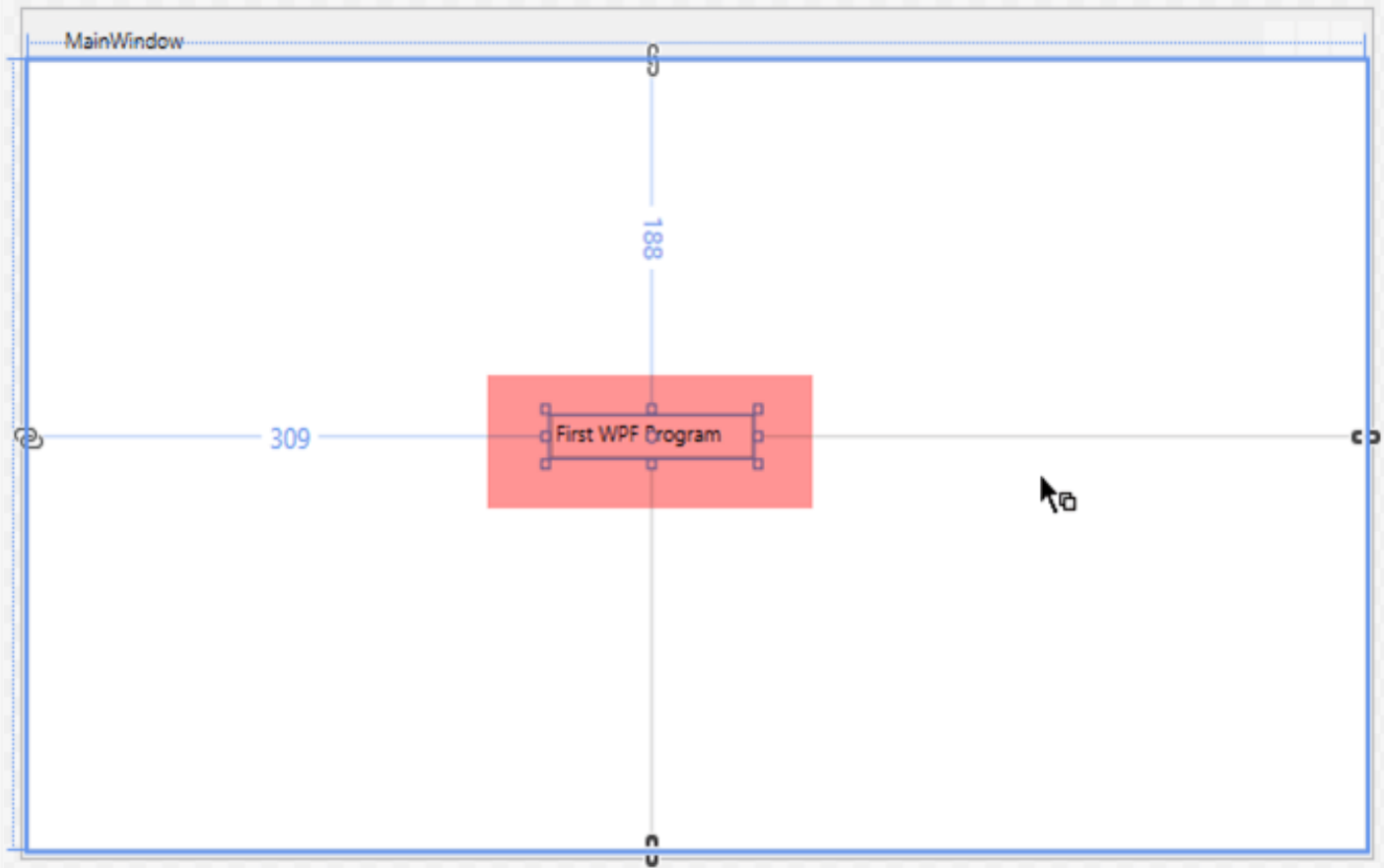
```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using System.Windows;
7 using System.Windows.Controls;
8 using System.Windows.Data;
9 using System.Windows.Documents;
10 using System.Windows.Input;
11 using System.Windows.Media;
12 using System.Windows.Media.Imaging;
13 using System.Windows.Navigation;
14 using System.Windows.Shapes;
15
16 namespace MyWPF
17 {
18     /// <summary>
19     /// Interaction logic for MainWindow.xaml
20     /// </summary>
21     public partial class MainWindow : Window
```

## Step 4) In toolbox,

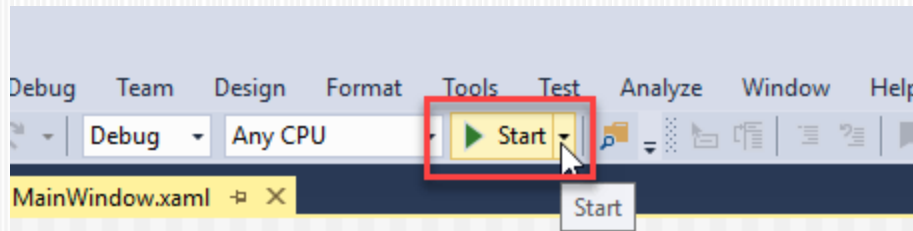
- Drag TextBox element to the design window
- A TextBox will appear in the design window
- You will see XAML code for TextBox added



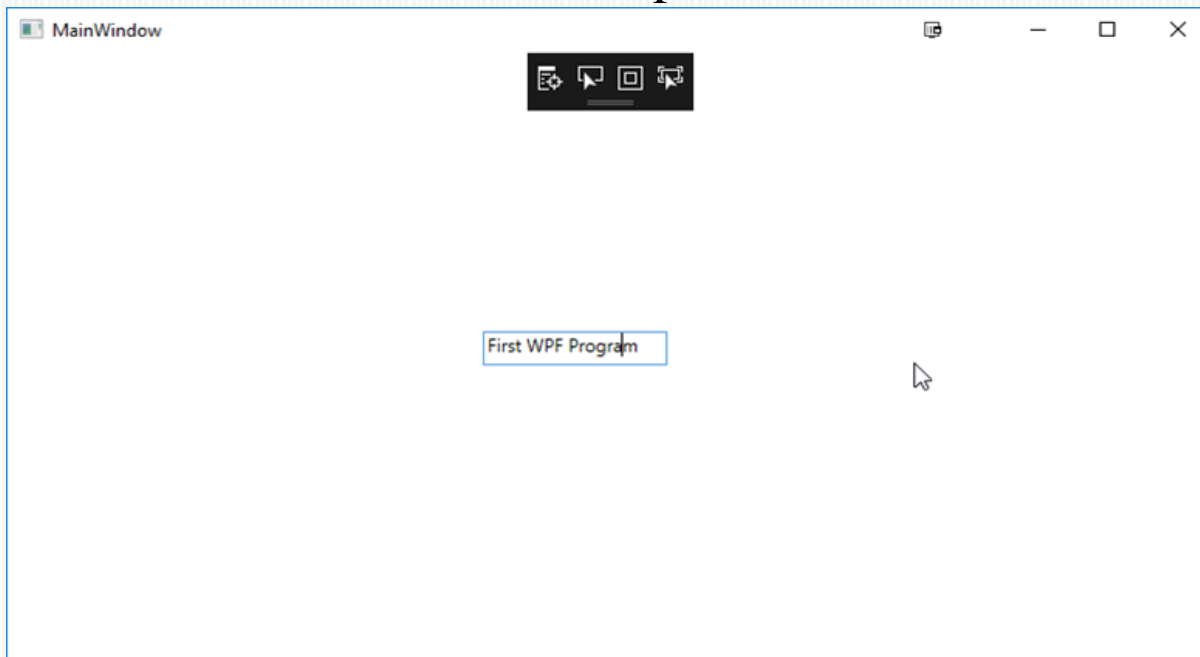
**Step 5)** Change text to "First WPF Program."



## Step 6) Click the Start Button



## Step 7) You will see a Window at Output





Adding a Code File for the **MainWindow Class**

```
// MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;
namespace WpfAppAllXaml
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            // Remember! This method is defined
            // within the generated MainWindow.g.cs file.
            InitializeComponent();
        }
        private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
        {
            this.Close();
        } } }
```

**MainWindow.xaml**

```
<Window x:Class="WpfAppAllXaml.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="A Window built using Code Files!"
Height="200" Width="300"
WindowStartupLocation ="CenterScreen">
<!-- The event handler is now in your code file -->
<Button x:Name="btnExitApp" Width="133" Height="24"
Content = "Close Window" Click ="btnExitApp_Clicked"/>
</Window>
```

**MyApp.xaml.cs**

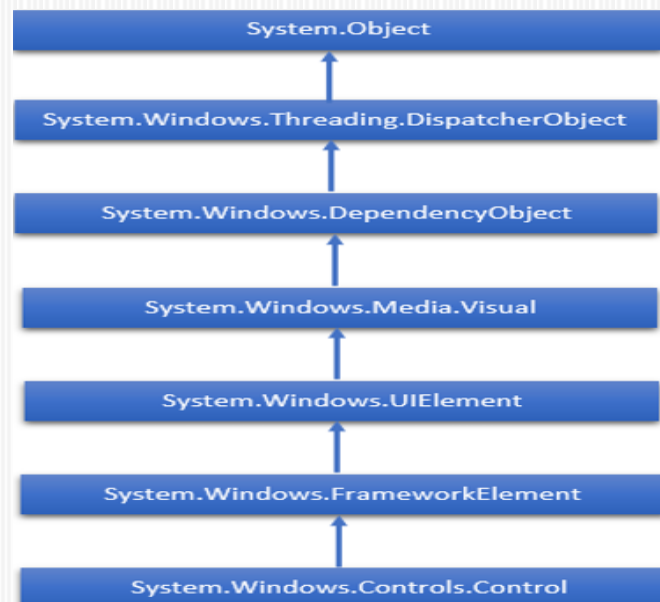
```
// MyApp.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;
namespace WpfAppAllXaml
{
    public partial class MyApp : Application
    {
        private void AppExit(object sender, ExitEventArgs e)
        {
            MessageBox.Show("App has exited");
        }
    }
}
```

## MyApp.xaml file

```
<Application x:Class="WpfAppAllXaml.MyApp"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml"
Exit="AppExit">
</Application>
```

# A Survey of the Core WPF Controls

- Windows Presentation Foundation (WPF) allows developers to easily build and create visually enriched UI based applications.
- The classical UI elements or controls in other UI frameworks are also enhanced in WPF applications.
- All of the standard WPF controls can be found in the Toolbox which is a part of the System.Windows.Controls.
- These controls can also be created in XAML markup language.



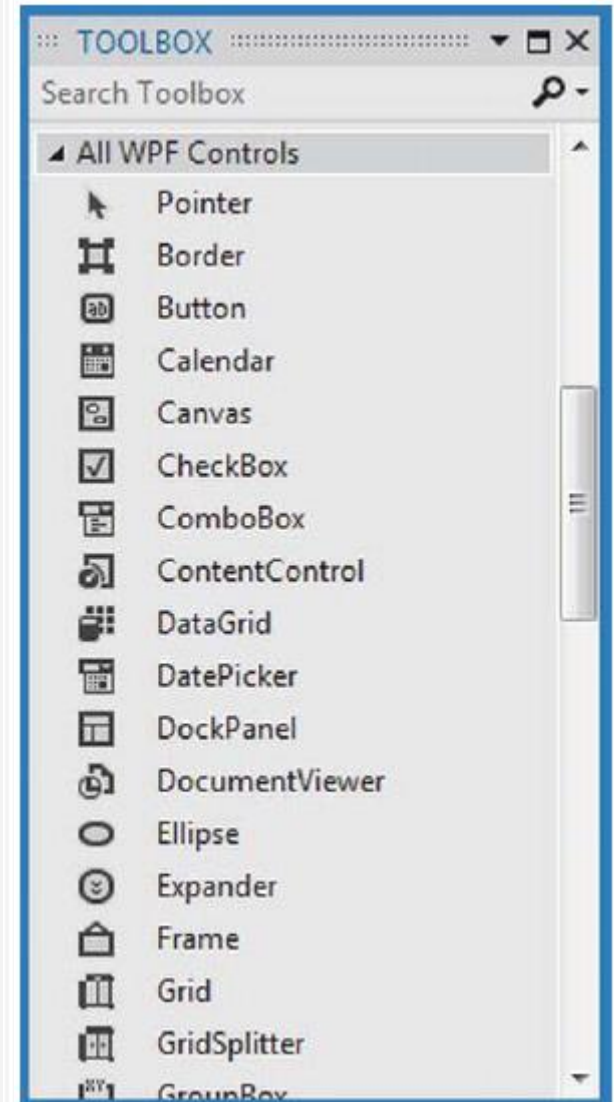
# A Survey of the Core WPF Controls

*Table 28-1. The Core WPF Controls*

WPF Control Category	Example Members	Meaning in Life
Core user input controls	Button, RadioButton, ComboBox, CheckBox, Calendar, DatePicker, Expander, DataGrid, ListBox, ListView, ToggleButton, TreeView, ContextMenu, ScrollBar, Slider, TabControl, TextBlock, TextBox, RepeatButton, RichTextBox, Label	WPF provides an entire family of controls you can use to build the crux of a user interface.
Window and control adornments	Menu, ToolBar, StatusBar, ToolTip, ProgressBar	You use these UI elements to decorate the frame of a Window object with input devices (such as the Menu) and user informational elements (e.g., StatusBar and ToolTip).
Media controls	Image, MediaElement, SoundPlayerAction	These controls provide support for audio/video playback and image display.
Layout controls	Border, Canvas, DockPanel, Grid, GridView, GridSplitter, GroupBox, Panel, TabControl, StackPanel, Viewbox, WrapPanel	WPF provides numerous controls that allow you to group and organize other controls for the purpose of layout management.

# C# .NET A Brief Review of the Visual Studio WPF Designer

- A majority of these standard WPF controls have been packaged up in the **System.Windows.Controls** namespace of the **PresentationFramework.dll** assembly.
- When you build a WPF application using Visual Studio, you will find most of these common controls contained in the Toolbox, provided you have a WPF designer open as the active window.



# C# .NET A Brief Review of the Visual Studio WPF Designer

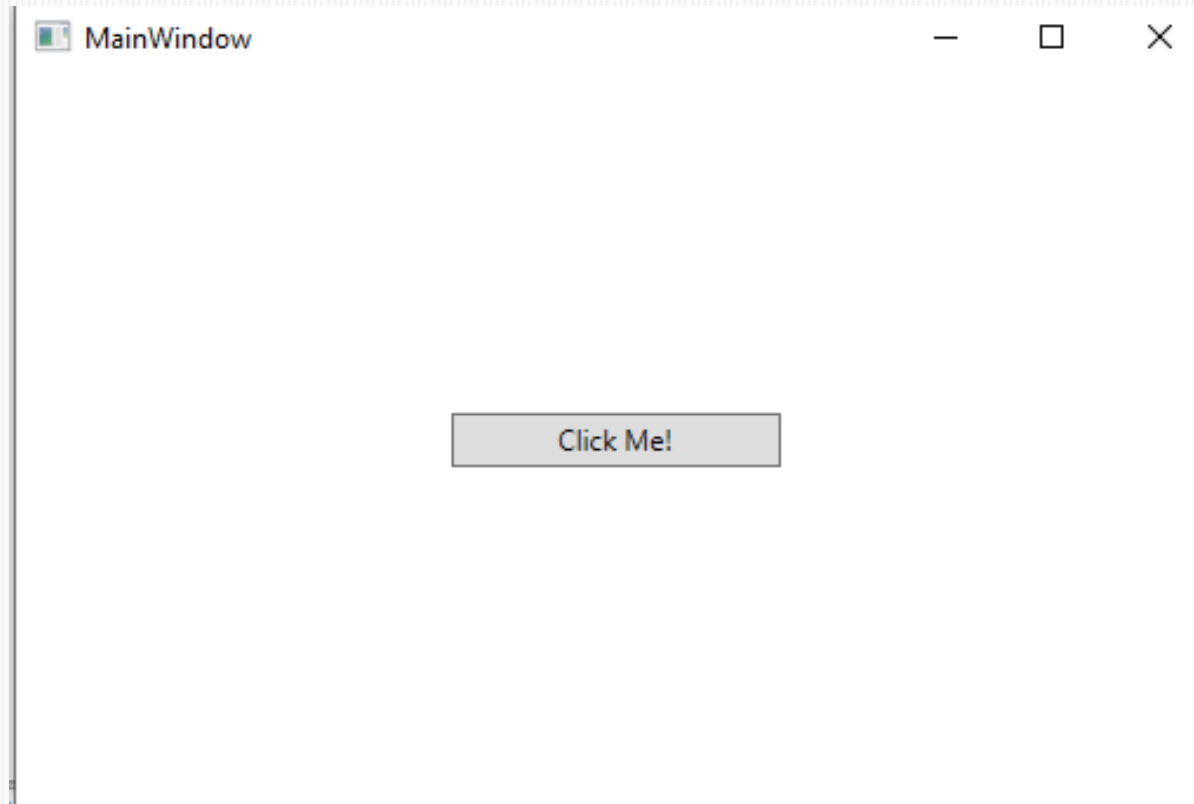
## Working with WPF Controls Using Visual Studio

- when you place a WPF control onto the Visual Studio designer, you want to set the x:Name property through the Properties window because this allows you to access the object in your related C# code file.
- You might also recall that you can use the Events tab of the Properties window to generate event handlers for a selected control.

```
<Button x:Name="btnMyButton" Content="Click Me!" Height="23" Width="140" Click="btnMyButton_Click" />
```

Here, you set the Content property of the Button to a simple string with the value "Click Me!".

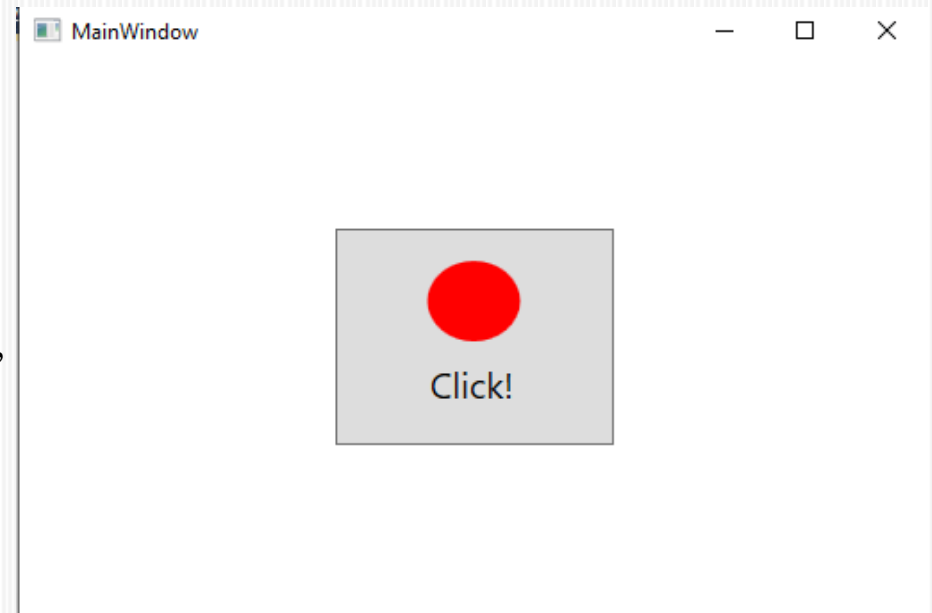




However, thanks to the WPF control content model, you could fashion a Button that contains the following complex content:

```
<Button x:Name="btnMyButton" Height="121" Width="156"  
Click="btnMyButton_Click">  
<Button.Content>  
<StackPanel Height="95" Width="128" Orientation="Vertical">  
<Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>  
<Label Width="59" FontSize="20" Content="Click!" Height="36" />  
</StackPanel>  
</Button.Content>  
</Button>
```

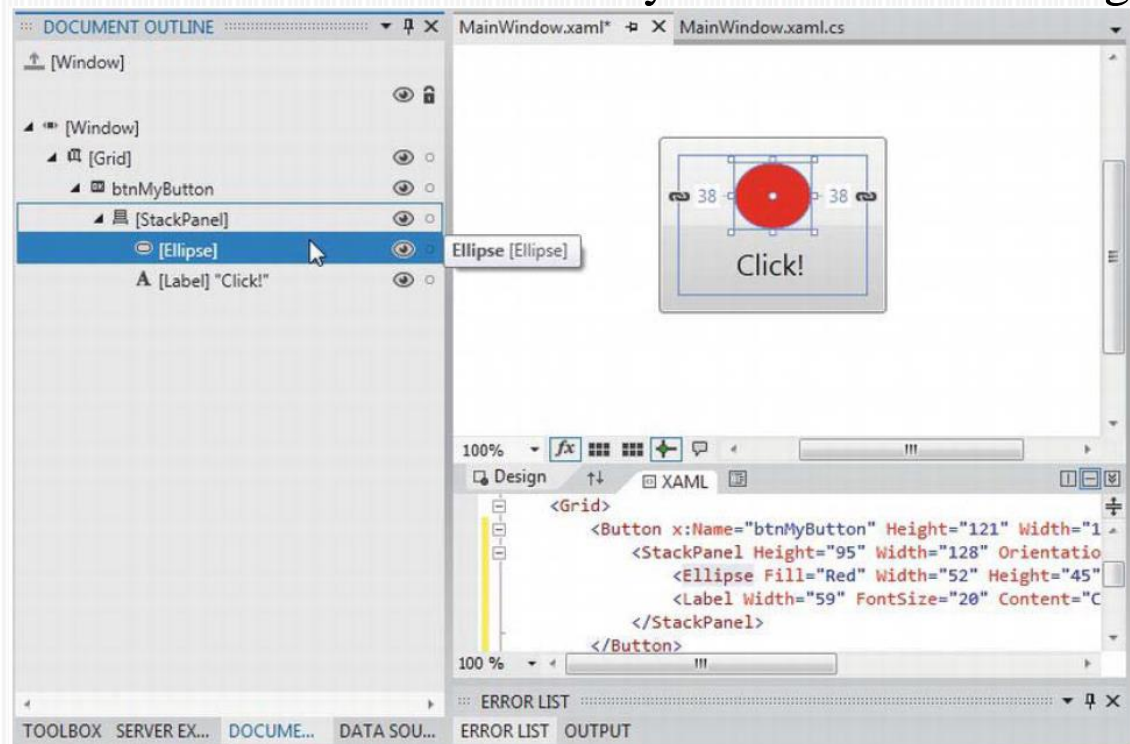
```
private void btnMyButton_Click(object sender,  
RoutedEventArgs e)  
{  
    MessageBox.Show("You clicked the button!");  
}
```



# C# .NET A Brief Review of the Visual Studio WPF Designer

## Working with the Document Outline Editor

- You should also be aware that the Document Outline window of Visual Studio (which you can open using the View □ Other Windows menu) is useful when designing a WPF control that has complex content.
- The logical tree of XAML is displayed for the Window you are building.
- If you click any of these nodes it is automatically selected in the designer for editing.



- A WPF application invariably contains a good number of UI elements (e.g., user input controls, graphical content, menu systems, and status bars) that need to be well organized within various windows.
- After you place the UI elements, you need to make sure they behave as intended when the end user resizes the window or possibly a portion of the window (as in the case of a splitter window).
- To ensure your WPF controls retain their position within the hosting window, you can take advantage of a good number of *panel types* (also known as *layout managers*).

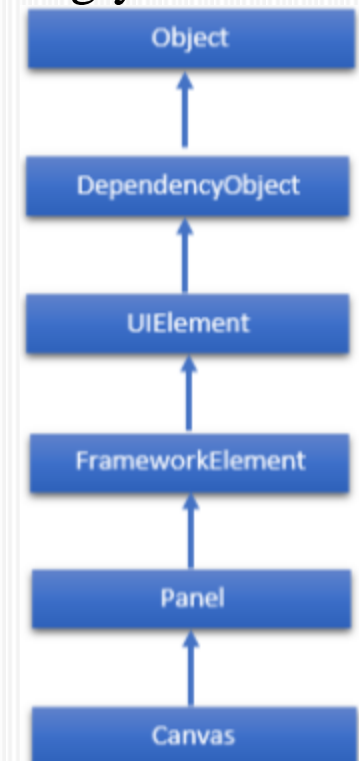
*Table 28-2. Core WPF Panel Controls*

Panel Control	Meaning in Life
Canvas	Provides a classic mode of content placement. Items stay exactly where you put them at design time.
DockPanel	Locks content to a specified side of the panel (Top, Bottom, Left, or Right).
Grid	Arranges content within a series of cells, maintained within a tabular grid.
StackPanel	Stacks content in a vertical or horizontal manner, as dictated by the <code>Orientation</code> property.
WrapPanel	Positions content from left-to-right, breaking the content to the next line at the edge of the containing box. Subsequent ordering happens sequentially from top-to-bottom or from right-to-left, depending on the value of the <code>Orientation</code> property.

## Controlling Content Layout Using Panels



- Canvas panel is the basic layout Panel in which the child elements can be positioned explicitly using coordinates that are relative to the **Canvas** any side such as left, right, top and bottom.
- Typically, a Canvas is used for 2D graphic elements (such as Ellipse, Rectangle etc.), but not for UI elements because specifying absolute coordinates create trouble while resizing, localizing or scaling your XAML application.



- Background
- Children
- Height
- ItemHeight
- ItemWidth
- LogicalChildren
- LogicalOrientation
- LeftProperty
- Margin
- Name
- Orientation
- Parent
- Resources
- Style
- TopProperty
- Width
- ZIndexProperty



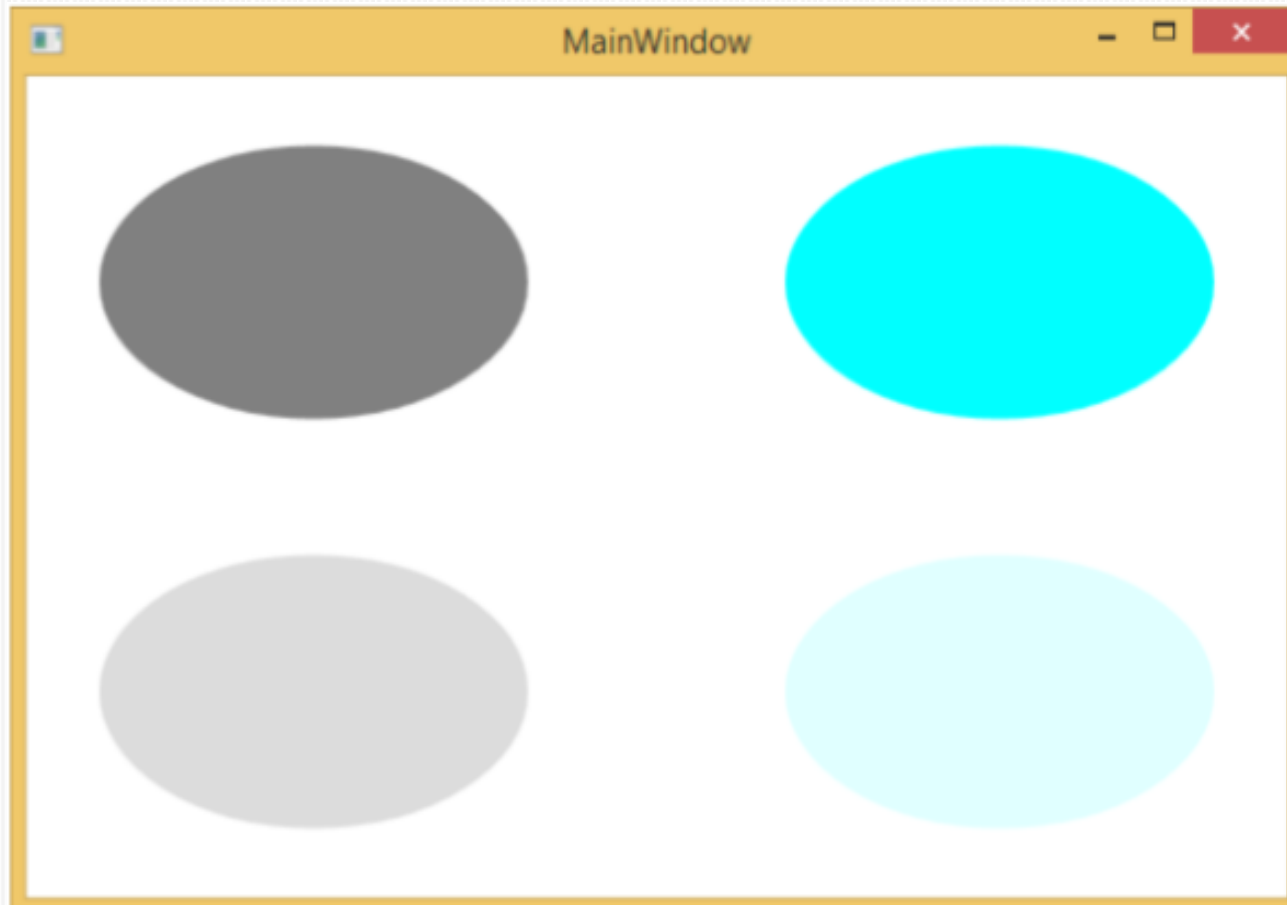
# Commonly Used Methods of Canvas

Sr. No.	Methods & Description
<b>GetLeft</b>	Gets the value of the Canvas.Left XAML attached property for the target element.
<b>GetTop</b>	Gets the value of the Canvas.Top XAML attached property for the target element.
<b>GetZIndex</b>	Gets the value of the Canvas.ZIndex XAML attached property for the target element.
<b>SetLeft</b>	Sets the value of the Canvas.Left XAML attached property for a target element.
<b>SetTop</b>	Sets the value of the Canvas.Top XAML attached property for a target element.
<b>SetZIndex</b>	Sets the value of the Canvas.ZIndex XAML attached property for a target element.

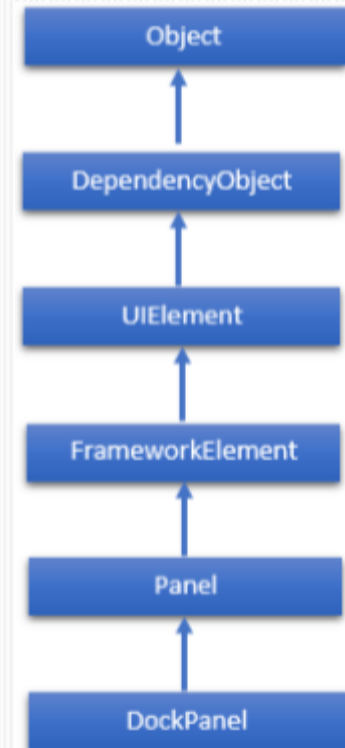
```
<Window x:Class = "WPFFConvas.MainWindow"
    xmlns =
"http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d = "http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc = "http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local = "clr-namespace:WPFFConvas"
    mc:Ignorable = "d" Title = "MainWindow" Height = "400" Width =
"604">

    <Grid>
        <Canvas Width = "580" Height = "360" >
            <Ellipse Canvas.Left = "30" Canvas.Top = "30"
                Fill = "Gray" Width = "200" Height = "120" />
            <Ellipse Canvas.Right = "30" Canvas.Top = "30"
                Fill = "Aqua" Width = "200" Height = "120" />
            <Ellipse Canvas.Left = "30" Canvas.Bottom = "30"
                Fill = "Gainsboro" Width = "200" Height = "120" />
            <Ellipse Canvas.Right = "30" Canvas.Bottom = "30"
                Fill = "LightCyan" Width = "200" Height = "120" />
        </Canvas>
    </Grid>

</Window>
```



- DockPanel defines an area to arrange child elements relative to each other, either horizontally or vertically. With DockPanel you can easily dock child elements to top, bottom, right, left and center using the **Dock** property.
- With **LastChildFill** property, the last child element fill the remaining space regardless of any other dock value when set for that element.



- Background
- Children
- Height
- Dock
- ItemHeight
- ItemWidth
- LogicalChildren
- LogicalOrientation
- LastChildFill
- Margin
- Name
- Orientation
- Parent
- Resources
- Style
- Width

# Commonly Used Methods of DockPanel

Sr. No.	Method & Description
<b>GetDock</b>	Gets the value of the Dock attached property for a specified UIElement.
<b>SetDock</b>	Sets the value of the Dock attached property to a specified element.

```
<Window x:Class = "WPFDockPanel.MainWindow"
    xmlns =
"http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d = "http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc = "http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local = "clr-namespace:WPFDockPanel"
    mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width =
"604">

    <Grid>
        <DockPanel LastChildFill = "True">
            <Button Content = "Top" DockPanel.Dock = "Top" Click =
"Click_Me" />
            <Button Content = "Bottom" DockPanel.Dock = "Bottom" Click
= "Click_Me" />
            <Button Content = "Left" Click = "Click_Me" />
            <Button Content = "Right" DockPanel.Dock = "Right" Click =
"Click_Me" />
            <Button Content = "Center" Click = "Click_Me" />
        </DockPanel>
    </Grid>

</Window>
```

```
using System.Windows;
using System.Windows.Controls;

namespace WPFDockPanel {
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>

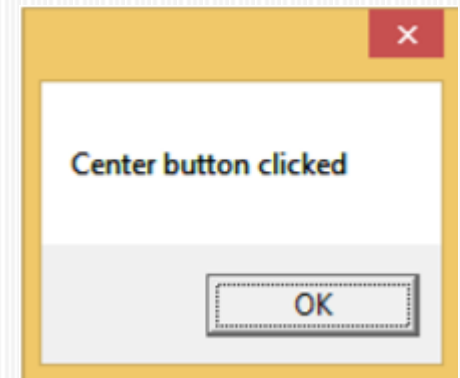
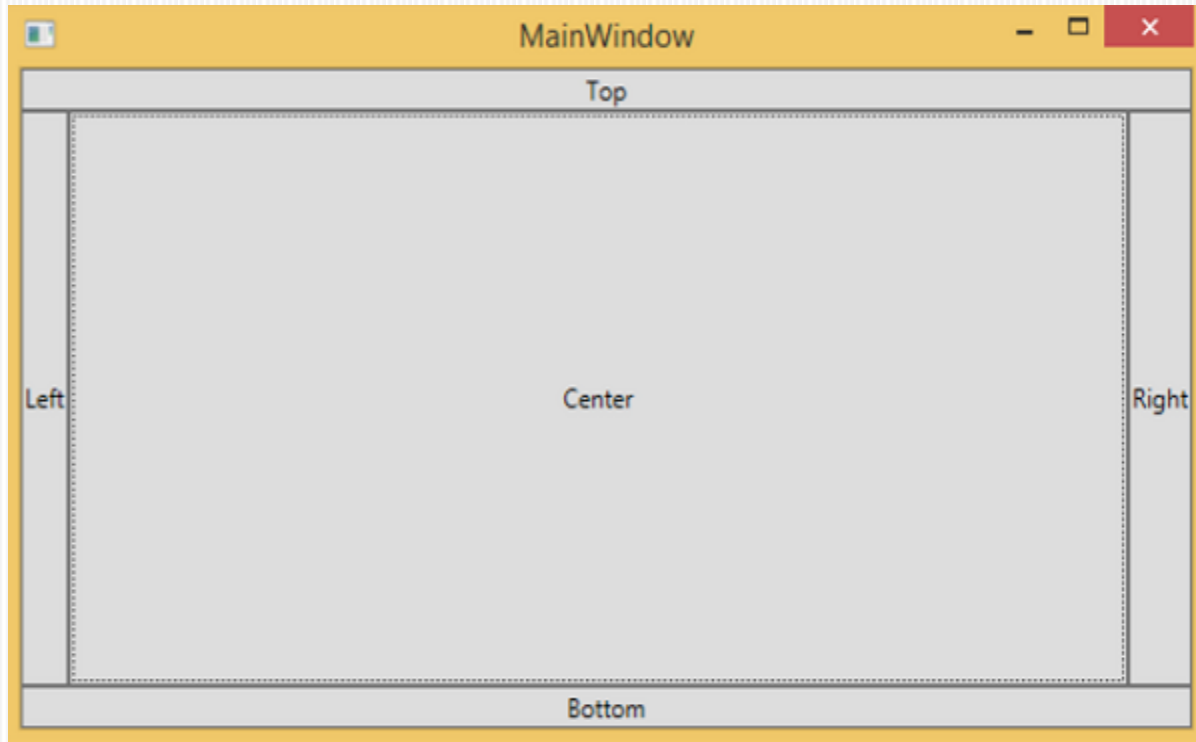
    public partial class MainWindow : Window {

        public MainWindow() {
            InitializeComponent();
        }

        private void Click_Me(object sender, RoutedEventArgs e) {
            Button btn = sender as Button;
            string str = btn.Content.ToString() + " button clicked";
            MessageBox.Show(str);
        }

    }
}
```





- A Grid Panel provides a flexible area which consists of rows and columns. In a Grid, child elements can be arranged in tabular form. Elements can be added to any specific row and column by using **Grid.Row** and **Grid.Column** properties
- By default, a Grid panel is created with one row and one column. Multiple rows and columns are created by RowDefinitions and ColumnDefinitions properties.
- The height of rows and the width of columns can be defined in the following three ways –
- **Fixed value** – To assign a fixed size of logical units (1/96 inch)
- **Auto** – It will take space which are required for the controls in that specific row/column.
- **Star (\*)** – It will take the remaining space when Auto and fixed sized are filled.

- Background
- Children
- ColumnDefinitions
- ItemHeight
- ItemWidth
- Margin
- Name
- Orientation
- Parent
- Resources
- Style
- Width
- RowDefinitions

## Commonly Used Methods of Grid Class

Sr. No.	Methods & Description
<b>GetColumn</b>	Gets the value of the Grid.Column XAML attached property from the specified FrameworkElement.
<b>GetColumnSpan</b>	Gets the value of the Grid.ColumnSpan XAML attached property from the specified FrameworkElement.
<b>GetRow</b>	Gets the value of the Grid.Row XAML attached property from the specified FrameworkElement.
<b>SetColumn</b>	Sets the value of the Grid.Column XAML attached property on the specified FrameworkElement.
<b>SetRow</b>	Sets the value of the Grid.Row XAML attached property on the specified FrameworkElement.
<b>SetRowSpan</b>	Sets the value of the Grid.RowSpan XAML attached property on the specified FrameworkElement.

```

<Window x:Class = "WPFGGrid.MainWindow"
    xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d = "http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc = "http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local = "clr-namespace:WPFGGrid"
mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width = "604">

    <Grid x:Name = "FormLayoutGrid" Background = "AliceBlue">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width = "Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height = "*" />
            <RowDefinition Height = "*" />
            <RowDefinition Height = "*" />
        </Grid.RowDefinitions>

        <TextBlock Grid.Row = "0" Grid.Column = "0" Text = "Name" Margin
= "10"
            HorizontalAlignment = "Left" VerticalAlignment = "Center"
Width = "100" />
        <TextBox Grid.Row = "0" Grid.Column = "1" Margin = "10" />
        <TextBlock Grid.Row = "1" Grid.Column = "0" Text = "ID" Margin =
"10"
            HorizontalAlignment = "Left" VerticalAlignment = "Center"
Width = "100" />
        <TextBox Grid.Row = "1" Grid.Column = "1" Margin = "10" />
        <TextBlock Grid.Row = "2" Grid.Column = "0" Text = "Age" Margin
= "10"
            HorizontalAlignment = "Left" VerticalAlignment = "Center"
Width = "100" />
        <TextBox Grid.Row = "2" Grid.Column = "1" Margin = "10" />
    </Grid> </Window>

```

MainWindow

Name

ID

Age

Stack panel is a simple and useful layout panel in XAML. In stack panel, child elements can be arranged in a single line, either horizontally or vertically, based on the orientation property. It is often used whenever any kind of list is to be created.

### Commonly Used Properties of StackPanel

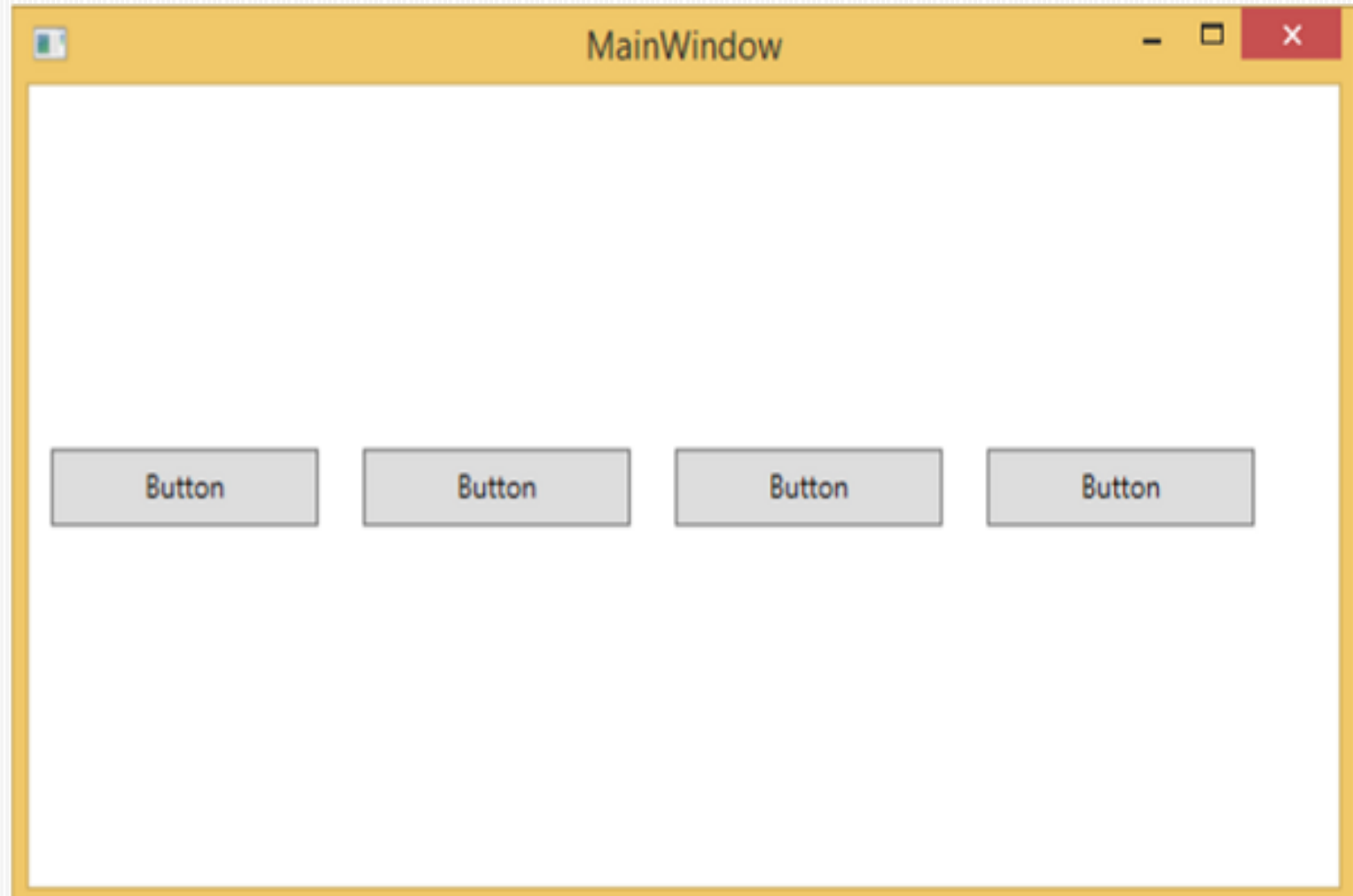
- Background
- Children
- Height
- ItemHeight
- ItemWidth
- LogicalChildren
- LogicalOrientation
- Margin
- Name
- Orientation
- Parent
- Resources
- Style
- Width

```
<Window x:Class = "WPFStackPanel.MainWindow"
    xmlns =
"http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d = "http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc = "http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local = "clr-namespace:WPFStackPanel"
    mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width =
"604">

    <Grid>
        <StackPanel Orientation = "Horizontal">
            <Button x:Name = "button" Content = "Button" Margin = "10"
Width = "120" Height = "30" />
            <Button x:Name = "button1" Content = "Button" Margin =
"10" Width = "120" Height = "30" />
            <Button x:Name = "button2" Content = "Button" Margin =
"10" Width = "120" Height = "30" />
            <Button x:Name = "button3" Content = "Button" Margin =
"10" Width = "120" Height = "30" />
        </StackPanel>
    </Grid>

</Window>
```





- In WrapPanel, child elements are positioned in sequential order, from left to right or from top to bottom based on the orientation property.
- The only difference between StackPanel and WrapPanel is that it doesn't stack all the child elements in a single line; it wraps the remaining elements to another line if there is no space left.
- WrapPanel is mostly used for tabs or menu items.
  - Background
  - Children
  - Height
  - ItemHeight
  - ItemWidth
  - LogicalChildren
  - LogicalOrientation
  - Margin
  - Name
  - Orientation
  - Parent
  - Resources
  - Style
  - Width

```
<Window x:Class = "WPFWrapPanel.MainWindow"
    xmlns =
"http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d = "http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc = "http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local = "clr-namespace:WPFWrapPanel"
    mc:Ignorable = "d" Title = "MainWindow" Height = "350" Width =
"604">

    <Grid>
        <WrapPanel Orientation = "Vertical">
            <TextBlock Text = "First Name" Width = "60" Height = "20"
Margin = "5" />
            <TextBox Width = "200" Height = "20" Margin = "5" />
            <TextBlock Text = "Last Name" Width = "60" Height = "20"
Margin = "5" />
            <TextBox Width = "200" Height = "20" Margin = "5"/>
            <TextBlock Text = "Age" Width = "60" Height = "20" Margin
= "5" />
            <TextBox Width = "60" Height = "20" Margin = "5" />
            <TextBlock Text = "Title" Width = "60" Height = "20"
Margin = "5" />
            <TextBox Width = "200" Height = "20" Margin = "5" />
        </WrapPanel>
    </Grid>

</Window>
```

MainWindow

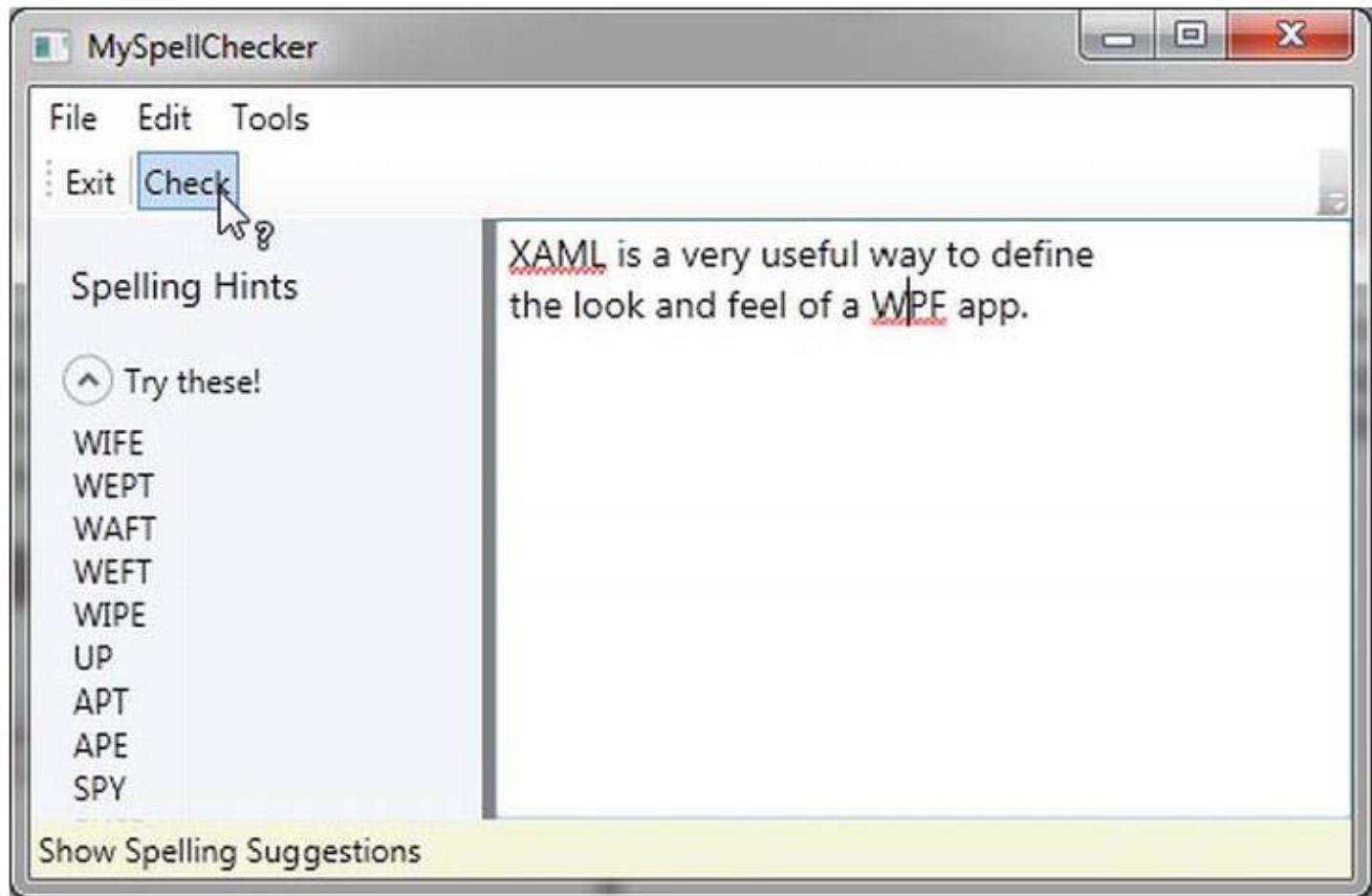
Fist Name

Last Name

Age

Title

AIM:



- To begin building this UI, update the initial XAML definition for your Window type so it uses a
- <DockPanel> child element, rather than the default <Grid>, as follows:
- 

```
<Window x:Class="MyWordPad.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MySpellChecker" Height="331" Width="508"
WindowStartupLocation="CenterScreen" >
<!-- This panel establishes the content for the window -->
<DockPanel>
</DockPanel>
</Window>
```

- **Building the Menu System**

```
<!-- Dock menu system on the top -->
```

```
<Menu DockPanel.Dock = "Top"
```

```
HorizontalAlignment="Left" Background="White" BorderBrush = "Black">
```

```
<MenuItem Header="_File">
```

```
<Separator/>
```

```
<MenuItem Header = "_Exit" MouseEnter = "MouseEnterExitArea"
```

```
MouseLeave = "MouseLeaveArea" Click = "FileExit_Click"/>
```

```
</MenuItem>
```

```
<MenuItem Header="_Tools">
```

```
<MenuItem Header = "_Spelling Hints"
```

```
MouseEnter = "MouseEnterToolsHintsArea"
```

```
MouseLeave = "MouseLeaveArea" Click = "ToolsSpellingHints_Click"/>
```

```
</MenuItem>
```

```
</Menu>
```

## Cs code

```
public partial class MainWindow : System.Windows.Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    protected void FileExit_Click(object sender, RoutedEventArgs args)
    {
        // Close this window.
        this.Close();
    }
    protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
    { }
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    { }
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    { }
    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    { } }
```



- **Building the ToolBar**

```
<!-- Put Toolbar under the Menu -->
```

```
<ToolBar DockPanel.Dock = "Top" >
```

```
<Button Content = "Exit" MouseEnter = "MouseEnterExitArea"
```

```
MouseLeave = "MouseLeaveArea" Click = "FileExit_Click"/>
```

```
<Separator/>
```

```
<Button Content = "Check" MouseEnter = "MouseEnterToolsHintsArea"
```

```
MouseLeave = "MouseLeaveArea" Click = "ToolsSpellingHints_Click"
```

```
Cursor = "Help" />
```

```
</ToolBar>
```

- **Building the StatusBar**

<!-- Put a StatusBar at the bottom -->

<StatusBar DockPanel.Dock ="Bottom" Background="Beige" >

<StatusBarItem>

<TextBlock Name="statBarText" Text="Ready"/>

</StatusBarItem>

</StatusBar>

## Finalizing the UI Design

```
<Grid DockPanel.Dock ="Left" Background ="AliceBlue">
<!-- Define the rows and columns -->
<Grid.ColumnDefinitions>
<ColumnDefinition />
<ColumnDefinition />
</Grid.ColumnDefinitions>
<GridSplitter Grid.Column ="0" Width ="5" Background ="Gray" />
<StackPanel Grid.Column ="0" VerticalAlignment ="Stretch" >
<Label Name ="lblSpellingInstructions" FontSize ="14" Margin ="10,10,0,0">
```

Spelling Hints

```
</Label>
<Expander Name ="expanderSpelling" Header ="Try these!"
Margin ="10,10,10,10">
<!-- This will be filled programmatically -->
<Label Name ="lblSpellingHints" FontSize ="12"/>
</Expander>
```

```
</StackPanel>
```

```
<!-- This will be the area to type within -->
```

```
<TextBox Grid.Column ="1"
SpellCheck.IsEnabled ="True"
AcceptsReturn ="True"
Name ="txtData" FontSize ="14"
BorderBrush ="Blue"
```

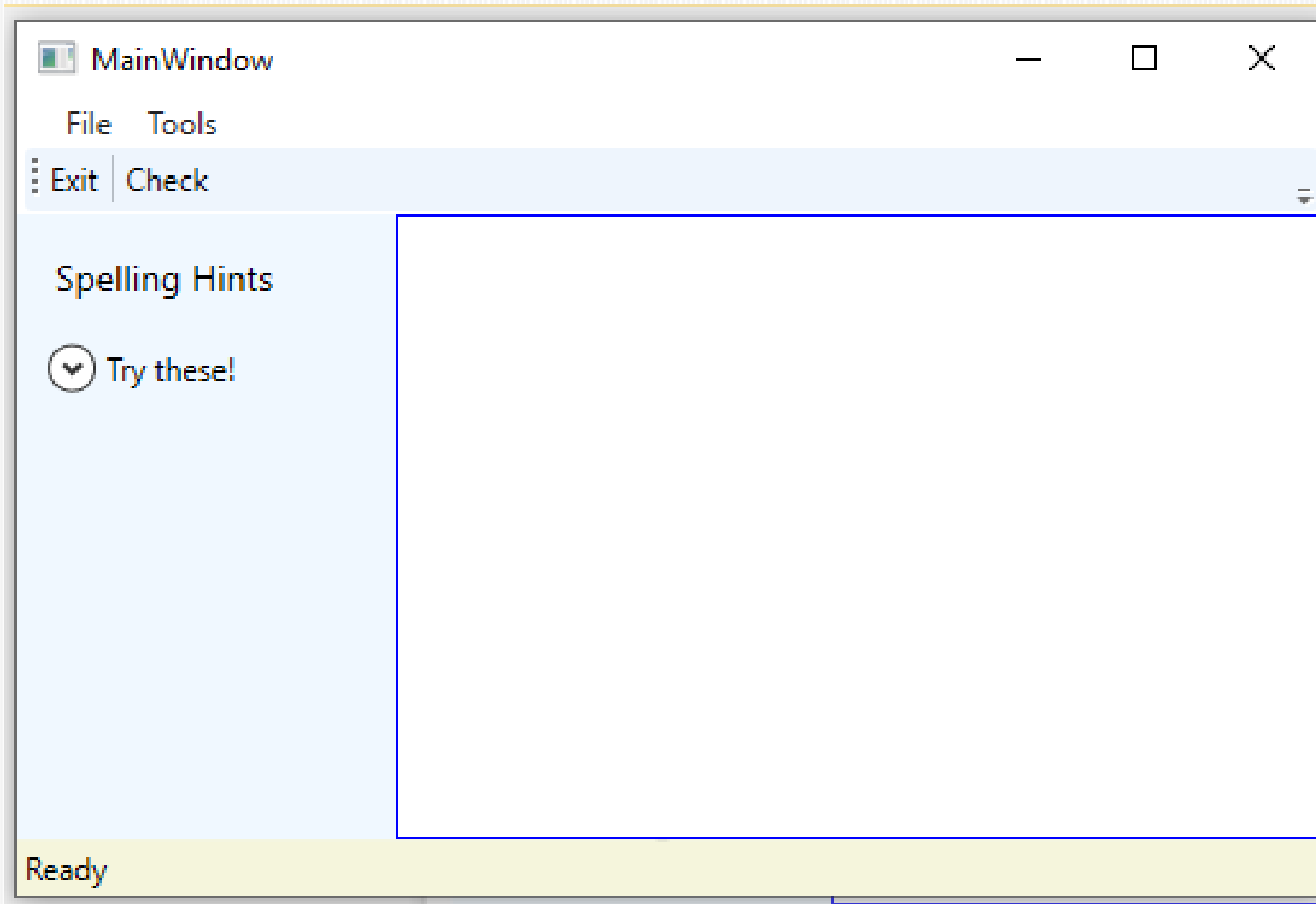
```
VerticalScrollBarVisibility ="Auto"
HorizontalScrollBarVisibility ="Auto">
```

- **Implementing the MouseEnter/MouseLeave Event Handlers**

- ```
public partial class MainWindow : System.Windows.Window
{
    ...
    protected void MouseEnterExitArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Exit the Application";
    }
    protected void MouseEnterToolsHintsArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Show Spelling Suggestions";
    }
    protected void MouseLeaveArea(object sender, RoutedEventArgs args)
    {
        statBarText.Text = "Ready";
    }
}
```

- Implementing the Spell Checking Logic

```
protected void ToolsSpellingHints_Click(object sender, RoutedEventArgs args)
{
    string spellingHints = string.Empty;
    // Try to get a spelling error at the current caret location.
    SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
    if (error != null)
    {
        // Build a string of spelling suggestions.
        foreach (string s in error.Suggestions)
        {
            spellingHints += string.Format("{0}\n", s);
        }
        // Show suggestions and expand the expander.
        lblSpellingHints.Content = spellingHints;
        expanderSpelling.IsExpanded = true;
    }
}
```



- It allows you to define actions in one place and then refer to them from all your user interface controls like menu items, toolbar buttons and so on.
- WPF will also listen for keyboard shortcuts and pass them along to the proper command, if any, making it the ideal way to offer keyboard shortcuts in an application.

## Command bindings

- Commands don't actually do anything by them self. At the root, they consist of the ICommand interface, which only defines an event and two methods: Execute() and CanExecute().
- The first one is for performing the actual action, while the second one is for determining whether the action is currently available.

- To perform the actual action of the command, you need a link between the command and your code and this is where the CommandBinding comes into play.
- A CommandBinding is usually defined on a Window or a UserControl, and holds a references to the Command that it handles, as well as the actual event handlers for dealing with the Execute() and CanExecute() events of the Command.

### Pre-defined commands

- The WPF team has defined over 100 commonly used commands that you can use.
- They have been divided into 5 categories, called **ApplicationCommands**, **NavigationCommands**, **MediaCommands**, **EditingCommands** and **ComponentCommands**.
- Especially ApplicationCommands contains commands for a lot of very frequently used actions like New, Open, Save and Cut, Copy and Paste.



```
<Window
x:Class="WpfTutorialSamples.Commands.UsingCommandsSample"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="UsingCommandsSample" Height="100" Width="200">
  <Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.New"
Executed="NewCommand_Executed"
CanExecute="NewCommand_CanExecute" />
  </Window.CommandBindings>

  <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
    <Button Command="ApplicationCommands.New">New</Button>
  </StackPanel>
</Window>
```

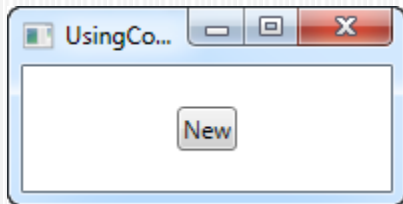
```
using System;  
using System.Collections.Generic;  
using System.Windows;  
using System.Windows.Input;
```

```
namespace WpfTutorialSamples.Commands  
{
```

```
    public partial class UsingCommandsSample : Window  
    {  
        public UsingCommandsSample()  
        {  
            InitializeComponent();  
        }  
    }
```

```
    private void NewCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)  
    {  
        e.CanExecute = true;  
    }
```

```
    private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)  
    {  
        MessageBox.Show("The New command was invoked");  
    } }
```



- In Code-behind, we handle the two events.
- The **CanExecute** handler, which WPF will call when the application is idle to see if the specific command is currently available, is very simple for this example, as we want this particular command to be available all the time.
- This is done by setting the **CanExecute** property of the event arguments to true.
- The **Executed** handler simply shows a message box when the command is invoked.
- If you run the sample and press the button, you will see this message.
- A thing to notice is that this command has a default keyboard shortcut defined, which you get as an added bonus.
- Instead of clicking the button, you can try to press Ctrl+N on your keyboard - the result is the same.

- A very common example of this is the toggling of buttons for using the Windows Clipboard, where you want the Cut and Copy buttons to be enabled only when text is selected, and the Paste button to only be enabled when text is present in the clipboard.
- This is exactly what we'll accomplish in this example:

```
<Window x:Class="WpfTutorialSamples.Commands.CommandCanExecuteSample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CommandCanExecuteSample" Height="200" Width="250">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Cut"
CanExecute="CutCommand_CanExecute" Executed="CutCommand_Executed" />
        <CommandBinding Command="ApplicationCommands.Paste"
CanExecute="PasteCommand_CanExecute" Executed="PasteCommand_Executed" />
    </Window.CommandBindings>
    <DockPanel>
        <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut" Width="60">_Cut</Button>
            <Button Command="ApplicationCommands.Paste" Width="60"
Margin="3,0">_Paste</Button>
        </WrapPanel>
        <TextBox AcceptsReturn="True"
Name="txtEditor" />
    </DockPanel>
</Window>
```

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;
```

```
namespace WpfTutorialSamples.Commands
```

```
{
```

```
public partial class CommandCanExecuteSample : Window
```

```
{
```

```
public CommandCanExecuteSample()
```

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

```
private void CutCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
```

```
{    e.CanExecute = (txtEditor != null) && (txtEditor.SelectionLength > 0);    }
```

```
private void CutCommand_Executed(object sender, ExecutedRoutedEventArgs e)
```

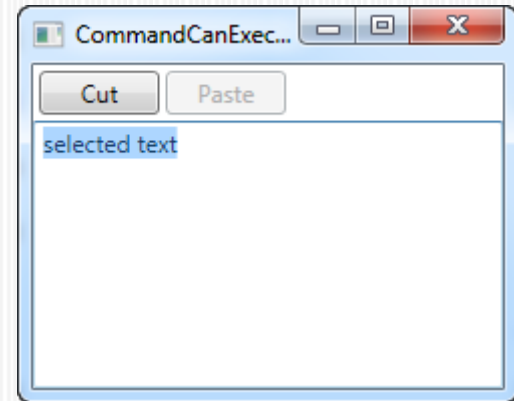
```
{        txtEditor.Cut();    }
```

```
private void PasteCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
```

```
{        e.CanExecute = Clipboard.ContainsText();    }
```

```
private void PasteCommand_Executed(object sender, ExecutedRoutedEventArgs e)
```

```
{        txtEditor.Paste();    }    }
```



```
<Window
x:Class="WpfTutorialSamples.Commands.CommandsWithCommandTargetSample"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="CommandsWithCommandTargetSample" Height="200" Width="250">
  <DockPanel>
    <WrapPanel DockPanel.Dock="Top" Margin="3">
      <Button Command="ApplicationCommands.Cut" CommandTarget="{Binding
ElementName=txtEditor}" Width="60">_Cut</Button>
      <Button Command="ApplicationCommands.Paste"
CommandTarget="{Binding ElementName=txtEditor}" Width="60"
Margin="3,0">_Paste</Button>
    </WrapPanel>
    <TextBox AcceptsReturn="True" Name="txtEditor" />
  </DockPanel>
</Window>
```

- Data binding is a mechanism in WPF Applications, which provides a simple and easy way for Windows Runtime apps to display and interact with the data.
- Data binding allows the flow of the data between UI elements and the data object on the user interface.
- When a binding is established and the data or your business model changes, then it reflects the updates automatically to the UI elements and vice versa.
- It is also possible to bind, not to a standard data source, but to another element on the page.
- Data binding is of two types- One-Way Data binding and Two-Way Data binding.

# One-Way Data binding

- In One-Way binding, the data is bound from its source (that is the object that holds the data) to its target (that is the object, which displays the data).
- Let's take a simple example to understand One-Way Data binding in detail.
  1. First of all, create a new WPF project with the name WPFDataBinding.
  2. The XAML code given below creates two labels, two textboxes, one button and initializes them with some properties.



```

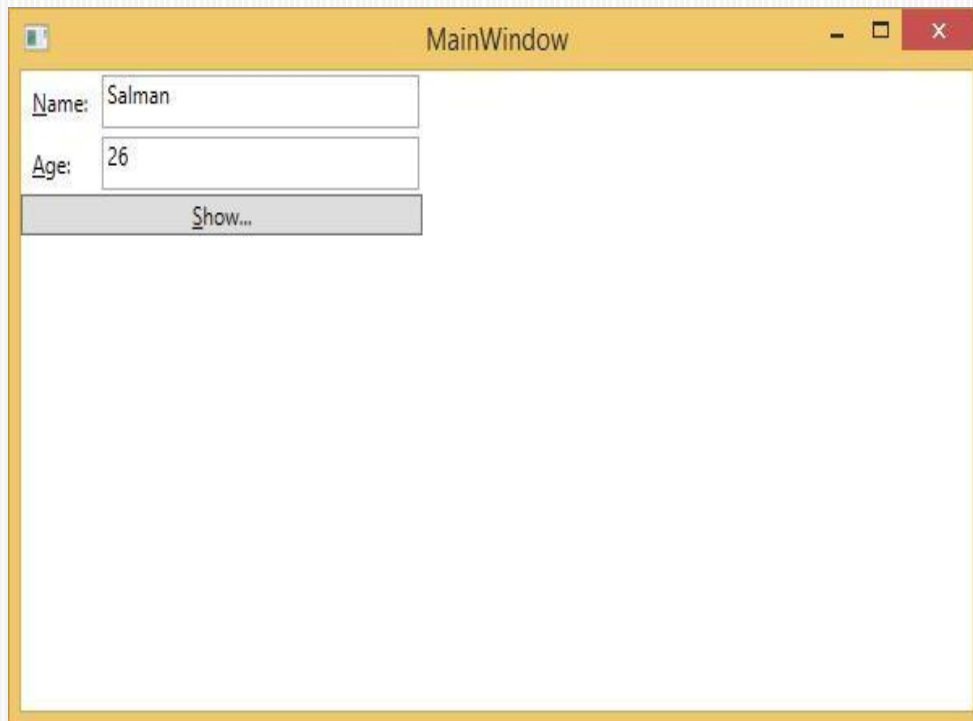
<Window x:Class="WPFDataBinding.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:local="clr-
namespace:WPFDataBinding" mc:Ignorable="d" Title="MainWindow" Height="350" Width="604">
<Grid> <Grid.RowDefinitions>
<RowDefinition Height="Auto" />
<RowDefinition Height="Auto" />
<RowDefinition Height="*" /> </Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto" />
<ColumnDefinition Width="200" /> </Grid.ColumnDefinitions>
<Label Name="nameLabel" Margin="2">_Name:</Label>
<TextBox Name="nameText" Grid.Column="1" Margin="2" Text="{Binding Name, Mode=OneWay}" />
<Label Name="ageLabel" Margin="2" Grid.Row="1">_Age:</Label>
<TextBox Name="ageText" Grid.Column="1" Grid.Row="1" Margin="2" Text="{Binding Age,
Mode=OneWay}" />
<StackPanel Grid.Row="2" Grid.ColumnSpan="2">
<Button Content="_Show..." Click="Button_Click" /> </StackPanel>
</Grid> </Window>

```

3. The text properties of both the textboxes binds to Name and Age, which are class variables of Person class, which is shown below.
4. In Person class, we have just two variables Name and Age and its object is initialized in MainWindow class.
5. In XAML code, we are binding to a property Name and Age, but we have not selected what object that property belongs to.
6. The easier way is to assign an object to DataContext whose properties; we are binding in C# code given below in MainWindowconstructor.

```
using System.Windows;
namespace WPFDataBinding {
    public partial class MainWindow: Window {
        Person person = new Person {
            Name = "Salman", Age = 26
        };
        public MainWindow() {
            InitializeComponent();
            this.DataContext = person;
        }
        private void Button_Click(object sender, RoutedEventArgs e) {
            string message = person.Name + " is " + person.Age;
            MessageBox.Show(message);
        }
    }
    public class Person {
        private string nameValue;
        public string Name {
            get {
                return nameValue;
            }
            set {
                nameValue = value;
            }
        }
        private double ageValue;
        public double Age {
            get {
                return ageValue;
            }
            set {
                if (value != ageValue) {
                    ageValue = value;
                }
            }
        }
    }
}
```

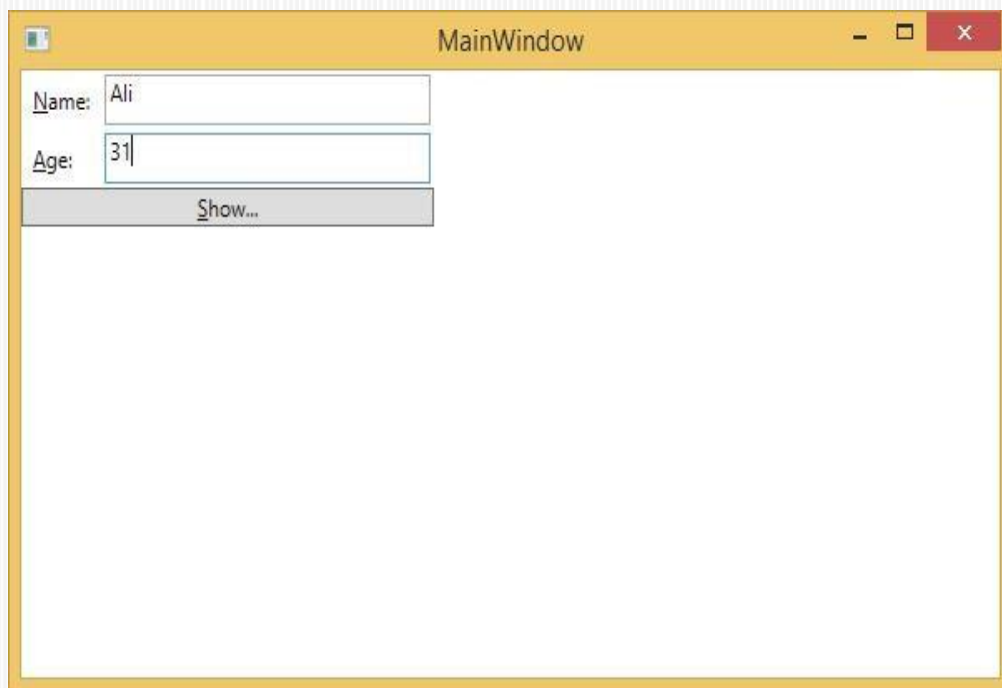
7. Let's run this Application and you can see immediately in our MainWindow, which we have successfully bound to the Name and Age of Person object.



When you click Show button, it will display the name and age on the message box.



Let's change the Name and Age in the dialog box.



If you click Show button now, it will again display the same message.



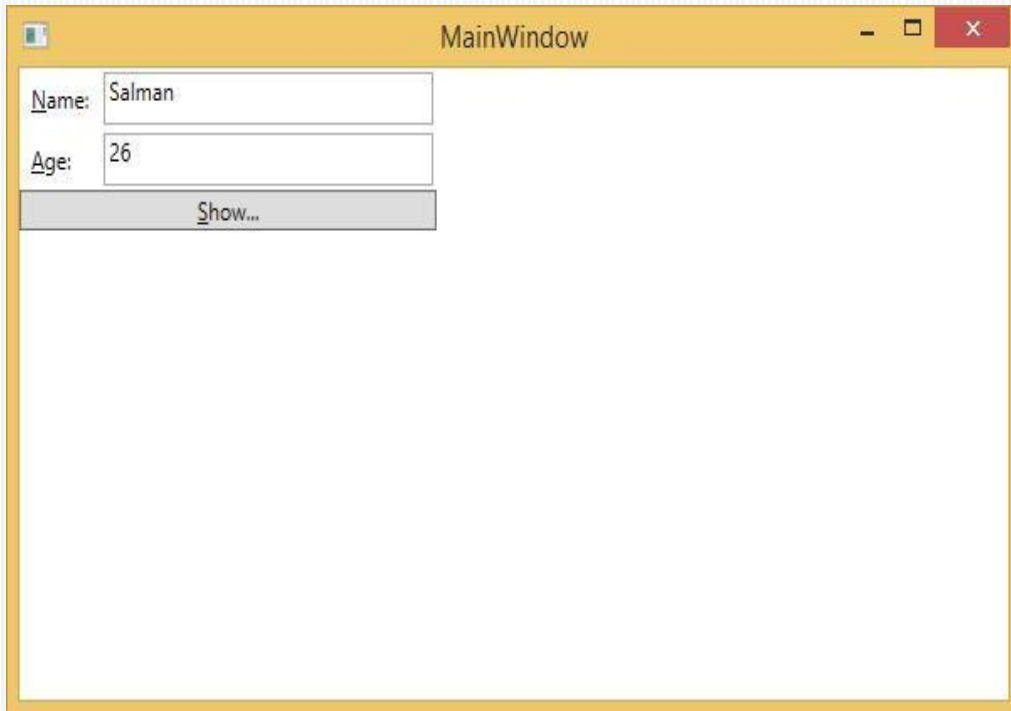
This is because Data binding mode is set to One-Way in XAML code. To show the updated data, you will need to understand Two-Way data binding.

## Two-Way Data binding

- In Two-Way binding, the user can modify the data through the user interface and have the data updated in the source.
- If the source changes while the user is looking at the View, you want the View to be updated.
- Let's take the same example but here, we will change the binding mode from One Way to Two Way in XAML code.

```
<Window x:Class="WPFDataBinding.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:local="clr-
namespace:WPFDataBinding" mc:Ignorable="d" Title="MainWindow" Height="350" Width="604">
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="Auto" />
<RowDefinition Height="Auto" />
<RowDefinition Height="*" /> </Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto" />
<ColumnDefinition Width="200" /> </Grid.ColumnDefinitions>
<Label Name="nameLabel" Margin="2">_Name:</Label>
<TextBox Name="nameText" Grid.Column="1" Margin="2" Text="{ Binding Name, Mode=TwoWay}" />
<Label Name="ageLabel" Margin="2" Grid.Row="1">_Age:</Label>
<TextBox Name="ageText" Grid.Column="1" Grid.Row="1" Margin="2" Text="{ Binding Age,
Mode=TwoWay}" />
<StackPanel Grid.Row="2" Grid.ColumnSpan="2">
<Button Content="_Show..." Click="Button_Click" /> </StackPanel>
</Grid>
</Window>
```

- Let's run this Application again.



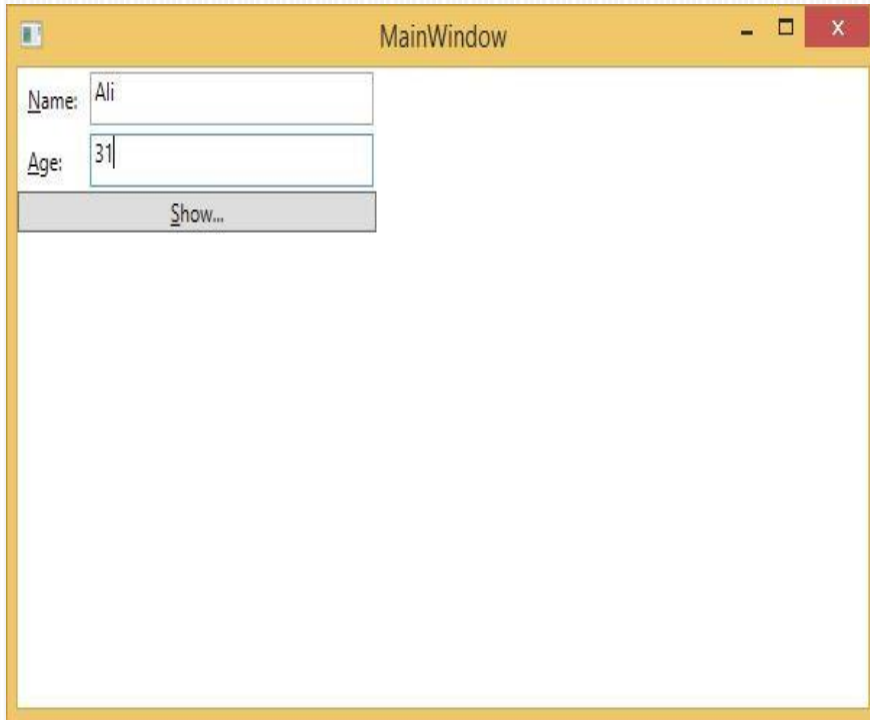
The screenshot shows a Windows application window titled "MainWindow". It contains two text input fields: the first is labeled "Name:" and contains the text "Salman"; the second is labeled "Age:" and contains the text "26". Below these fields is a button labeled "Show...".

It will produce the same output.





- Let's change the Name and Age values, as given below.



If you click Show button now, it will display the updated message.



We recommend that you execute the code given above with both the cases for a better understanding of the concept.

