tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

WPF stands for Windows Presentation Foundation. It is a powerful framework for building Windows applications. This tutorial explains the features that you need to understand to build WPF applications and how it brings a fundamental change in Windows applications.

Audience

This tutorial has been designed for all those readers who want to learn WPF and to apply it instantaneously in different type of applications.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of XML, Web Technologies and HTML.

Copyright & Disclaimer

Table of Contents

1. WPF – OVERVIEW

WPF stands for Windows Presentation Foundation. It is a powerful framework for building Windows applications. This tutorial explains the features that you need to understand to build WPF applications and how it brings a fundamental change in Windows applications.

WPF was first introduces in .NET framework 3.0 version, and then so many other features were added in the subsequent .NET framework versions.

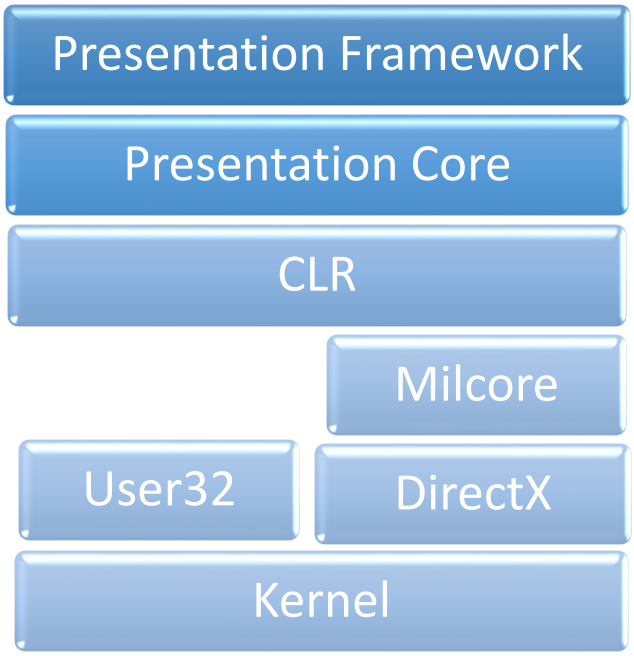
WPF Architecture

Before WPF, the other user interface frameworks offered by Microsoft such as MFC and Windows forms, were just wrappers around User32 and GDI32 DLLs, but WPF makes only minimal use of User32. So,

- WPF is more than just a wrapper.
- It is a part of the .NET framework.
- It contains a mixture of managed and unmanaged code.

The major components of WPF architecture are as shown in the figure below. The most important code part of WPF are:

- Presentation Framework
- Presentation Core
- Milcore



The **presentation framework** and the **presentation core** have been written in managed code. **Milcore** is a part of unmanaged code which allows tight integration with DirectX (responsible for display and rendering). **CLR** makes the development process more productive by offering many features such as memory management, error handling, etc.

WPF – Advantages

In the earlier GUI frameworks, there was no real separation between how an application looks like and how it behaved. Both GUI and behavior was created in the same language, e.g. C# or VB.Net which would require more effort from the developer to implement both UI and behavior associated with it.

In WPF, UI elements are designed in XAML while behaviors can be implemented in procedural languages such C# and VB.Net. So it very easy to separate behavior from the designer code.

With XAML, the programmers can work in parallel with the designers. The separation between a GUI and its behavior can allow us to easily change the look of a control by using styles and templates.

WPF – Features

WPF is a powerful framework to create Windows application. It supports many great features, some of which have been listed below:

Feature	Description
Control inside a Control	Allows to define a control inside another control as a content.
Data binding	Mechanism to display and interact with data between UI elements and data object on user interface.
Media services	Provides an integrated system for building user interfaces with common media elements like images, audio, and video.

Templates	In WPF you can define the look of an element directly with a Template
Animations	Building interactivity and movement on user Interface
Alternative input	Supports multi-touch input on Windows 7 and above.
Direct3D	Allows to display more complex graphics and custom themes

2. WPF – ENVIRONMENT SETUP

Microsoft provides two important tools for WPF application development.

- Visual Studio
- Expression Blend

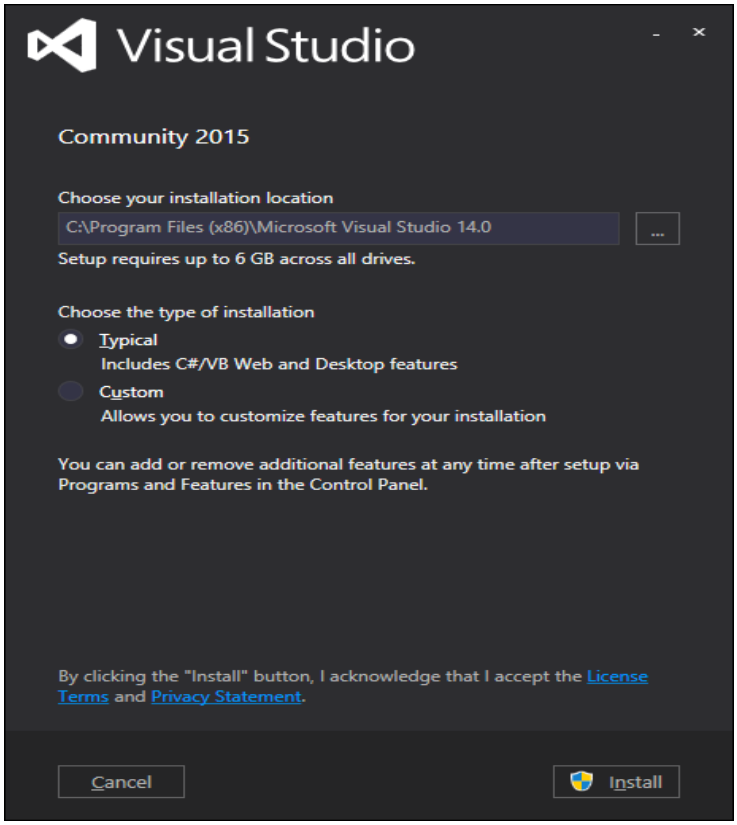
Both the tools can create WPF projects, but the fact is that Visual Studio is used more by developers, while Blend is used more often by designers. For this tutorial, we will mostly be using Visual Studio.

Installation

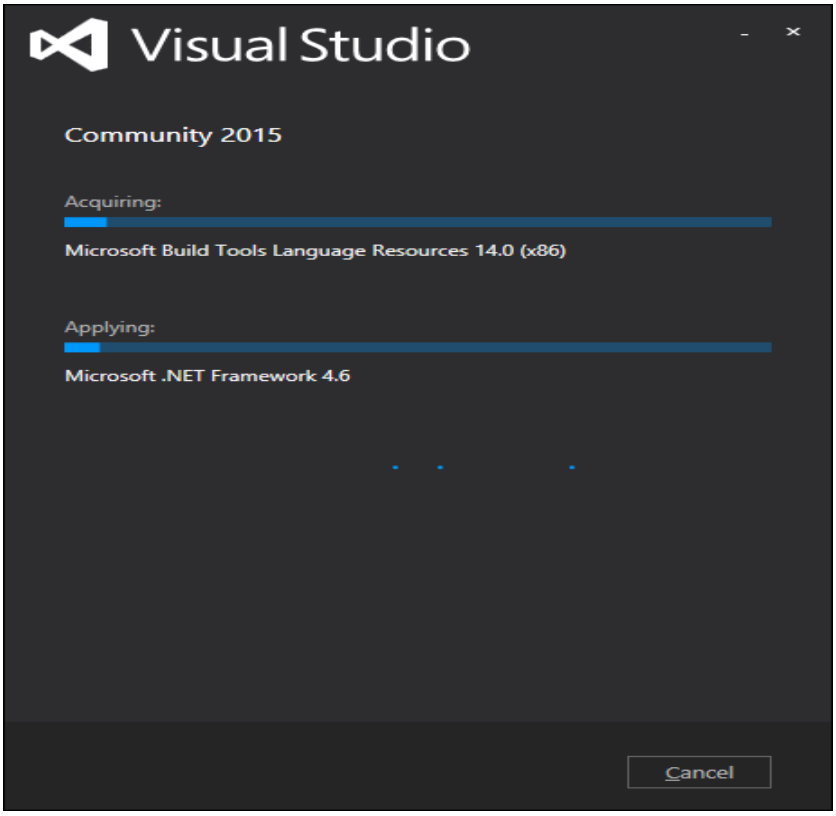
Microsoft provides a free version of Visual Studio which can be downloaded from https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx.

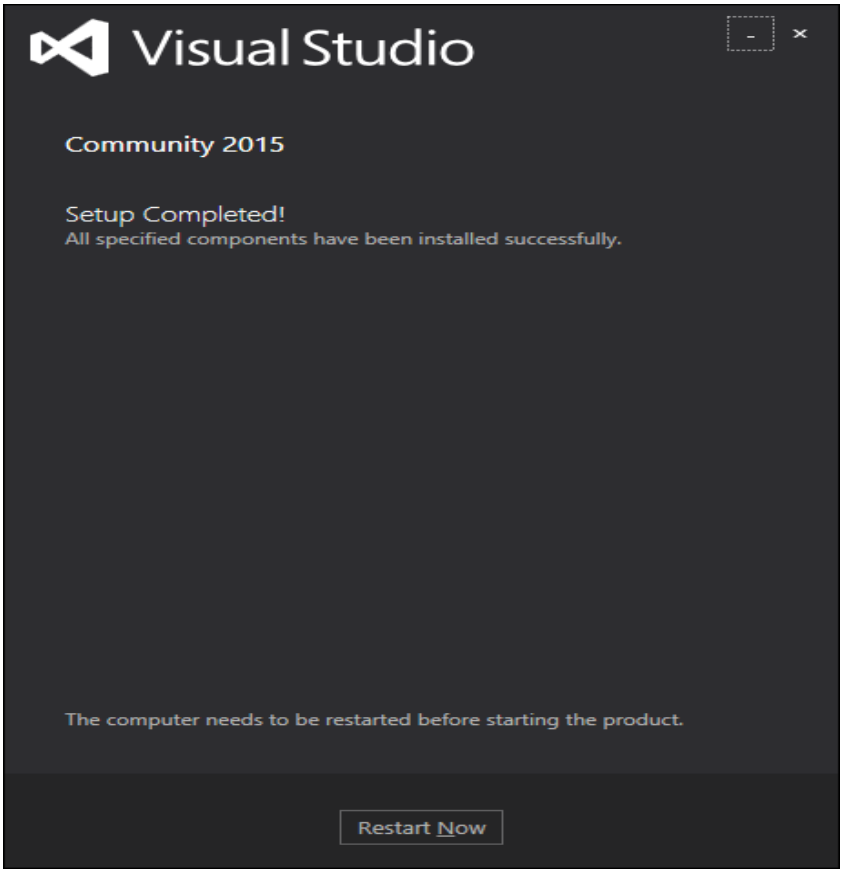
Download the files and follow the steps given below to set up WPF application development environment on your system.

1. After the download is complete, run the **installer**. The following dialog will be displayed.



2. Click the **Install** button and it will start the installation process.

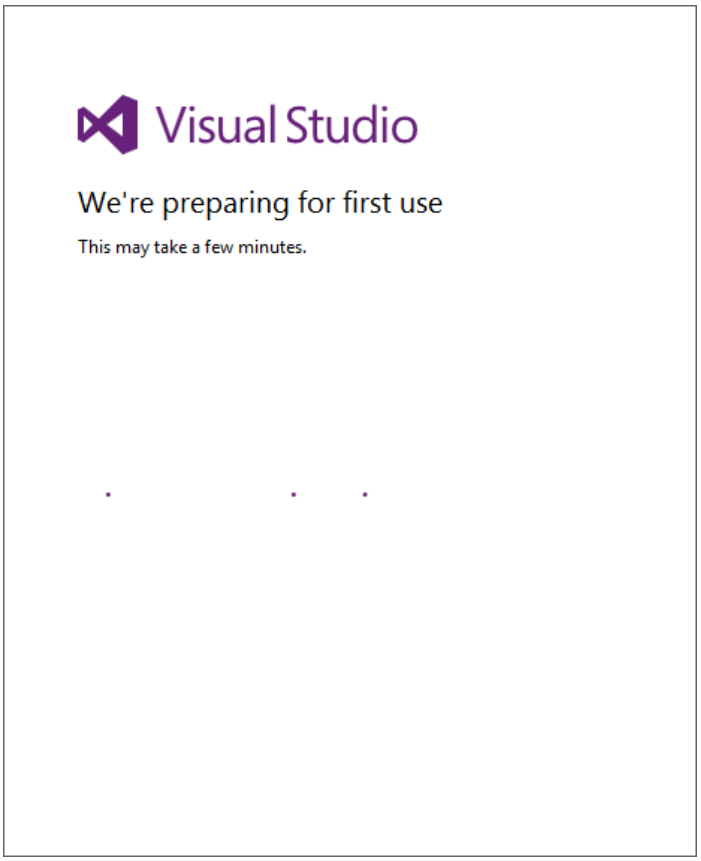




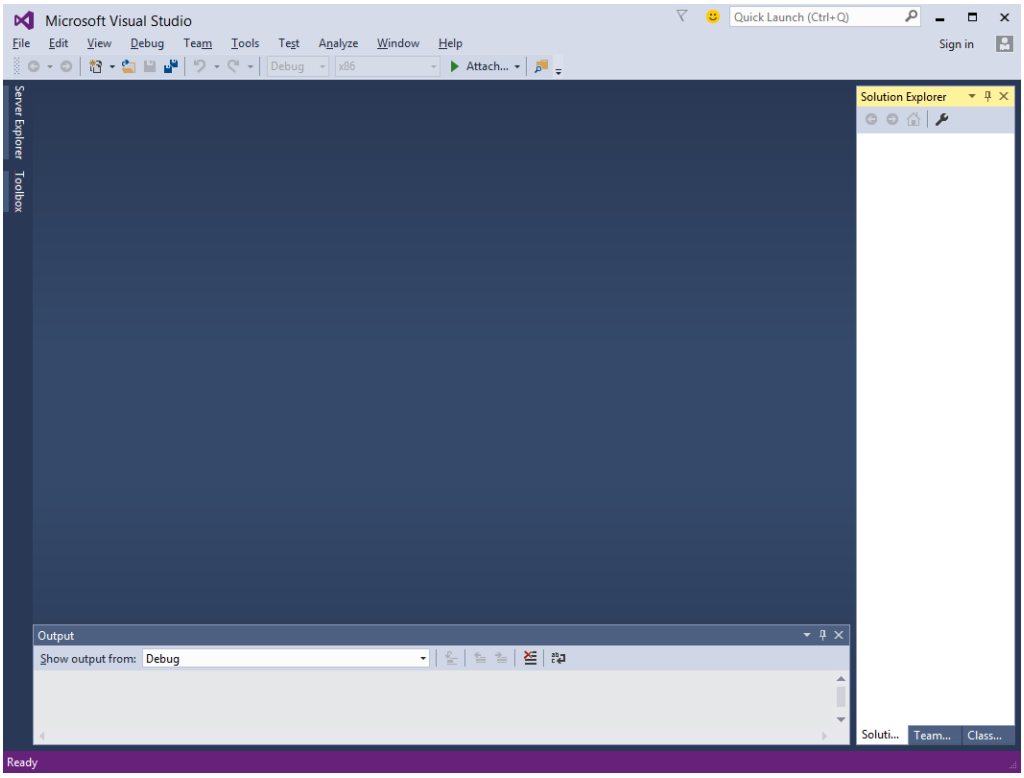
3. Once the installation process is completed successfully, you will get to see the following dialog box.

4. Close this dialog box and restart your computer if required.

5. Now open Visual Studio from the Start Menu which will open the following dialog box.



6. Once all is done, you will see the main window of Visual Studio.

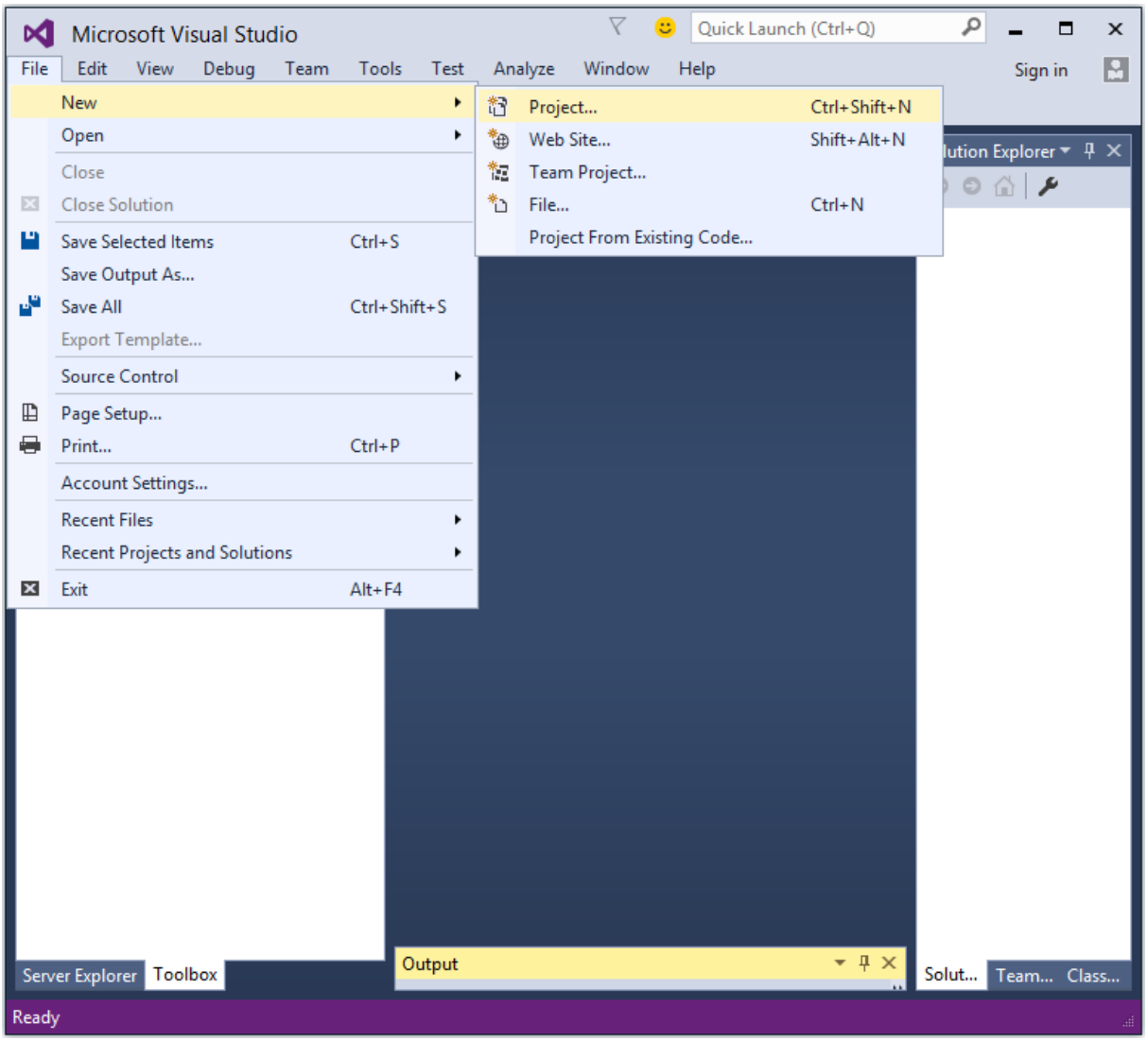


You are now ready to build your first WPF application.

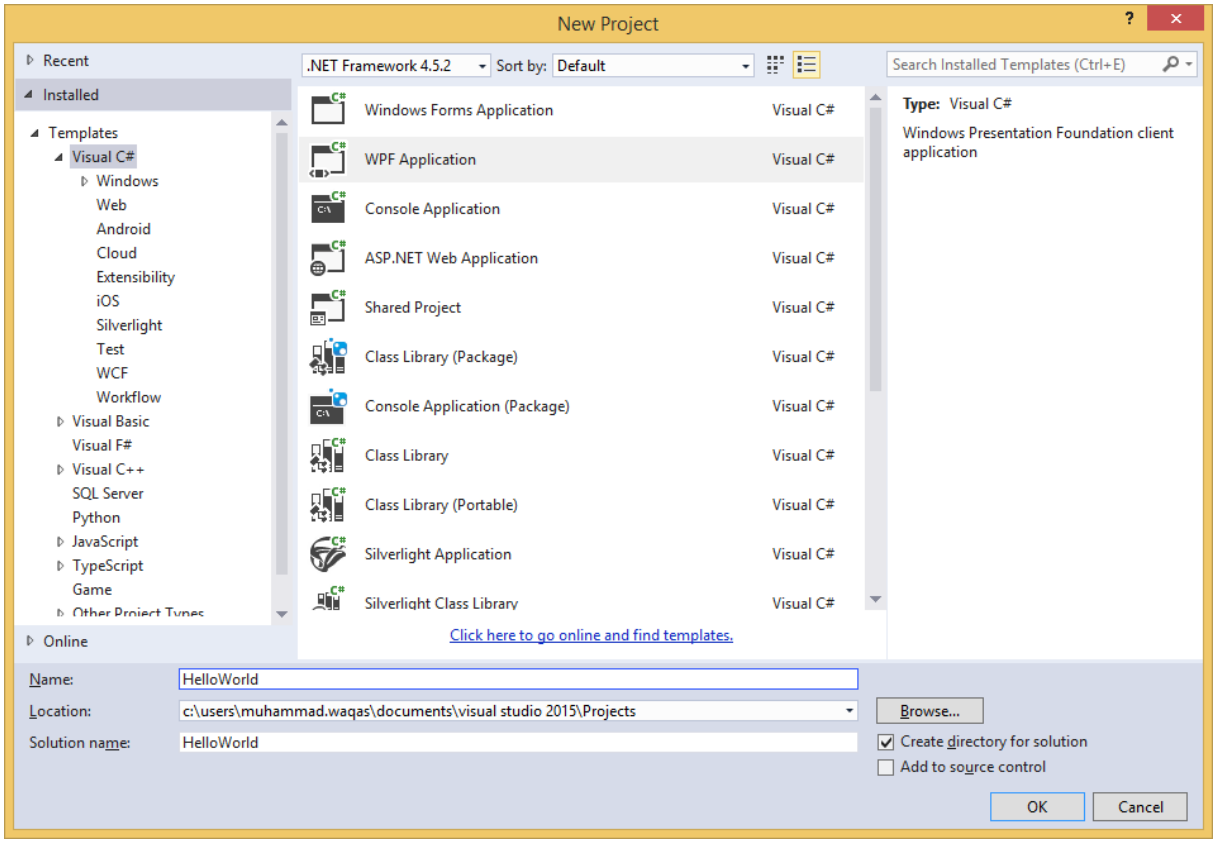
3. WPF – HELLO WORLD

In this chapter, we will develop a simple Hello World WPF application. So let's start the simple implementation by following the steps given below.

1. Click on File > New > Project menu option.



2. The following dialog box will be displayed.



3. Under Templates, select Visual C# and in the middle panel, select WPF Application.

4. Give the project a name. Type **HelloWorld** in the name field and click the OK button.

5. By default, two files are created, one is the **XAML** file (mainwindow.xaml) and the other one is the **CS** file (mainwindow.cs)

6. On mainwindow.xaml, you will see two sub-windows, one is the **design window** and the other one is the **source** (XAML) **window**.

7. In WPF application, there are two ways to design an UI for your application. One is to simply drag and drop UI elements from the toolbox to the Design Window. The second way is to design your UI by writing XAML tags for UI elements. Visual Studio handles XAML tags when drag and drop feature is used for UI designing.

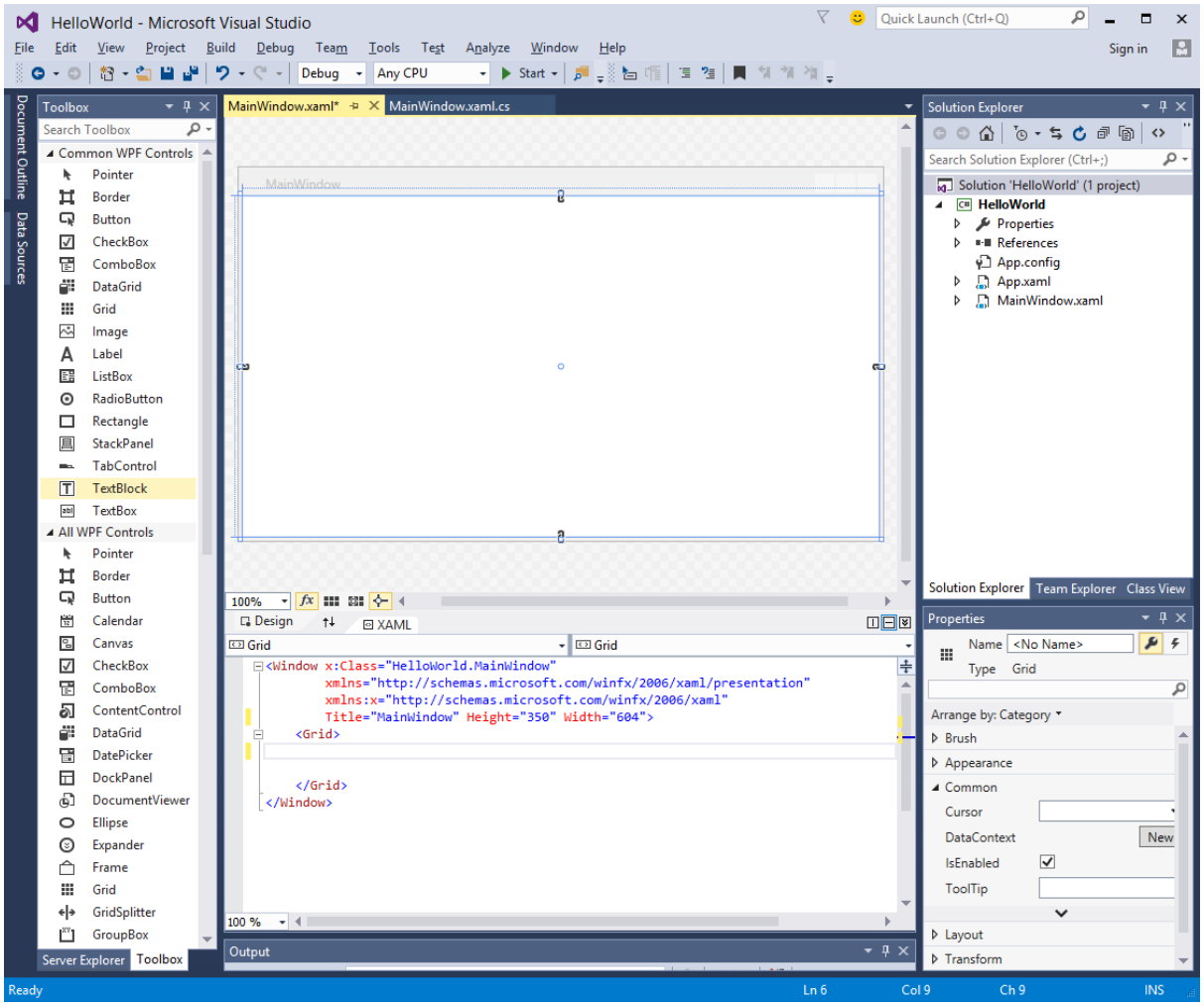
8. In mainwindow.xaml file, the following XAML tags are written by default.

```
<Window x:Class="HelloWorld.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

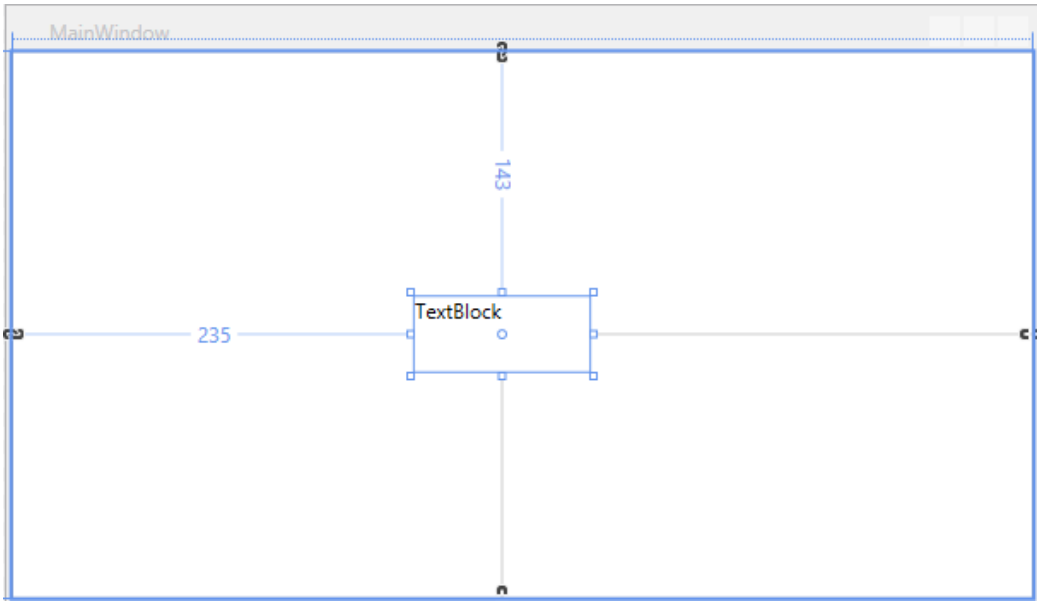
```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="604">
```

```
    <Grid>


    </Grid>
</Window>
```

9. By default, a Grid is set as the first element after page.
10. Let's go to the toolbox and drag a TextBlock to the design window.



11. You will see the TextBlock on the design window.



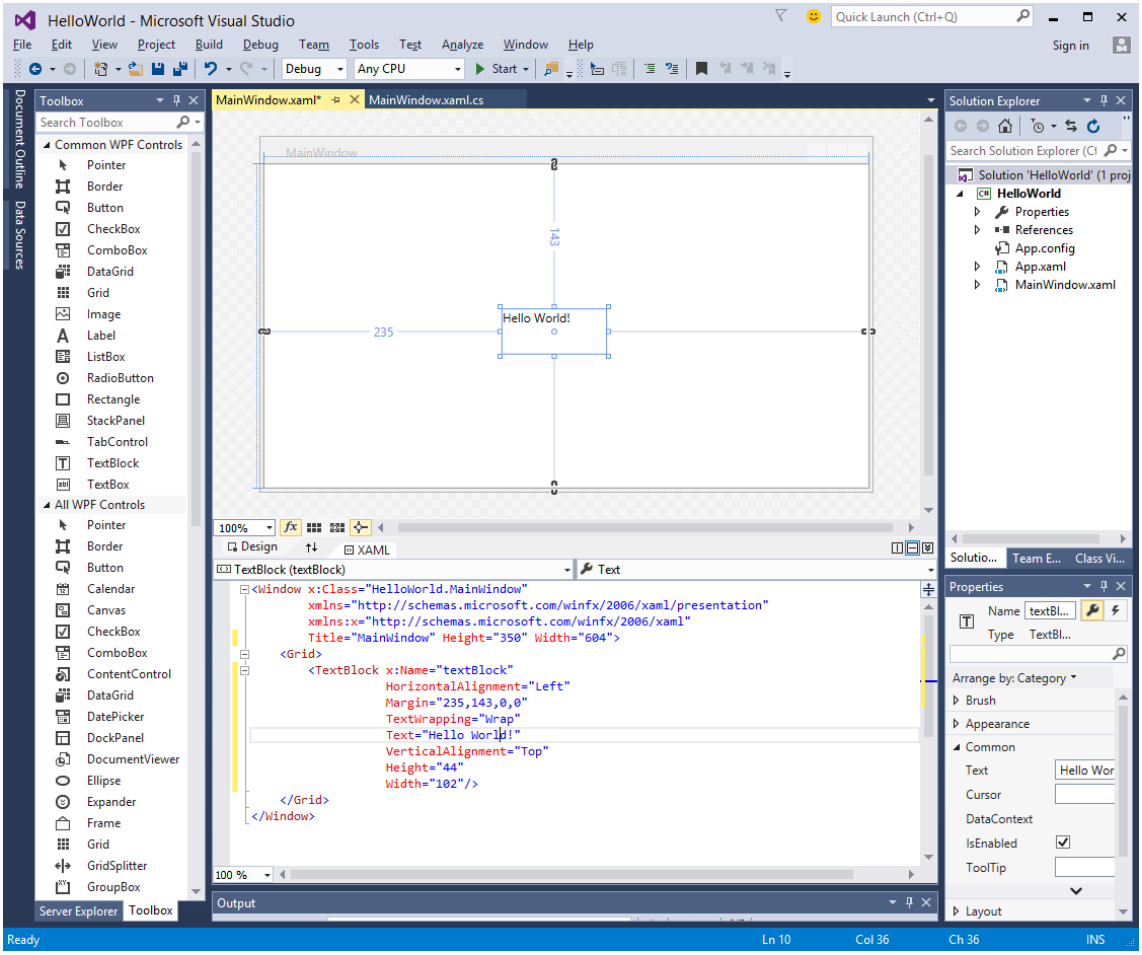
12. When you look at the source window, you will see that Visual Studio has generated the XAML code of the TextBlock for you.

13. Let's change the Text property of TextBlock in XAML code from TextBlock to Hello World.

```
<Window x:Class="HelloWorld.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">
    <Grid>
        <TextBlock x:Name="textBlock"
                   HorizontalAlignment="Left"
                   Margin="235,143,0,0"
                   TextWrapping="Wrap"
                   Text="Hello World!"
                   VerticalAlignment="Top"
                   Height="44"
                   Width="102"/>
```

```
    </Grid>
</Window>
```

14. Now, you will see the change on the Design Window as well.



When the above code is compiled and executed, you will see the following window.

Congratulations! You have designed and created your first WPF application.

4. WPF – XAML OVERVIEW

One of the first things you will encounter while working with WPF is XAML. XAML stands for Extensible Application Markup Language. It's a simple and declarative language based on XML.

- In XAML, it very easy to create, initialize, and set properties of objects with hierarchical relations.

- It is mainly used for designing GUIs, however it can be used for other purposes as well, e.g., to declare workflow in Workflow Foundation.

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>
```

The above XAML file contains different kinds of information. The following table briefly explains the role of each information.

Information	Description
<Window	It is the opening object element or container of the root.
x:Class="Resources.MainWindow"	It is a partial class declaration which connects the markup to the partial class code defined behind.
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"	Maps the default XAML namespace for WPF client/framework.
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"	XAML namespace for XAML language which maps it to x: prefix

>	End of object element of the root
<Grid> </Grid>	It is starting and closing tags of an empty grid object.
</Window>	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 do 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>
```

Why XAML in WPF

XAML is not only the most widely known feature of WPF, but it's also one of the most misunderstood features. If you have exposure to WPF, then you must have heard of XAML; but take a note of the following two less known facts about XAML:

- WPF doesn't need XAML
- XAML doesn't need WPF

They are in fact separable pieces of technology. To understand how that can be, let's look at a 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>
```

In case you choose not to use XAML in WPF, then you can achieve the same GUI result with procedural language as well. Let's have a look at the same example, but this time, we will create a button in C#.

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


namespace WPFXAMLOverview
```

```
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    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);

        }
    }
}
```

When you compile and execute either the XAML code or the C# code, you will see the same output as shown below.

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.

5. WPF – ELEMENTS TREE

There are many technologies where the elements and components are ordered in a tree structure so that the programmers can easily handle the object and change the behavior of an application. Windows Presentation Foundation (WPF) has a comprehensive tree structure in the form of objects. In WPF, there are two ways that a complete object tree is conceptualized:

- Logical Tree Structure
- Visual Tree Structure

With the help of these tree structures, you can easily create and identify the relationship between UI elements. Mostly, WPF developers and designers either use procedural language to create an application or design the UI part of the application in XAML keeping in mind the object tree structure.

Logical Tree Structure

In WPF applications, the structure of the UI elements in XAML represents the logical tree structure. In XAML, the basic elements of UI are declared by the developer. The logical tree in WPF defines the following:

- Dependency properties
- Static and dynamic resources
- Binding the elements on its name etc.

Let's have a look at the following example in which a button and a list box are created.

```
<Window x:Class="WPFElementsTree.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" Height="30" Width="70" Content="OK"
Margin="20"/>
        <ListBox x:Name="listBox" Height="100" Width="100" Margin="20">
            <ListBoxItem Content="Item 1"/>
            <ListBoxItem Content="Item 2"/>
            <ListBoxItem Content="Item 3"/>
        </ListBox>
    </StackPanel>
```

```
</Window>
```

If you look at the XAML code, you will observe a tree structure, i.e. the root node is the Window and inside the root node, there is only one child, that is StackPanel. But StackPanel contains two child elements, button and list box. List box has three more child list box items.

Visual Tree Structure

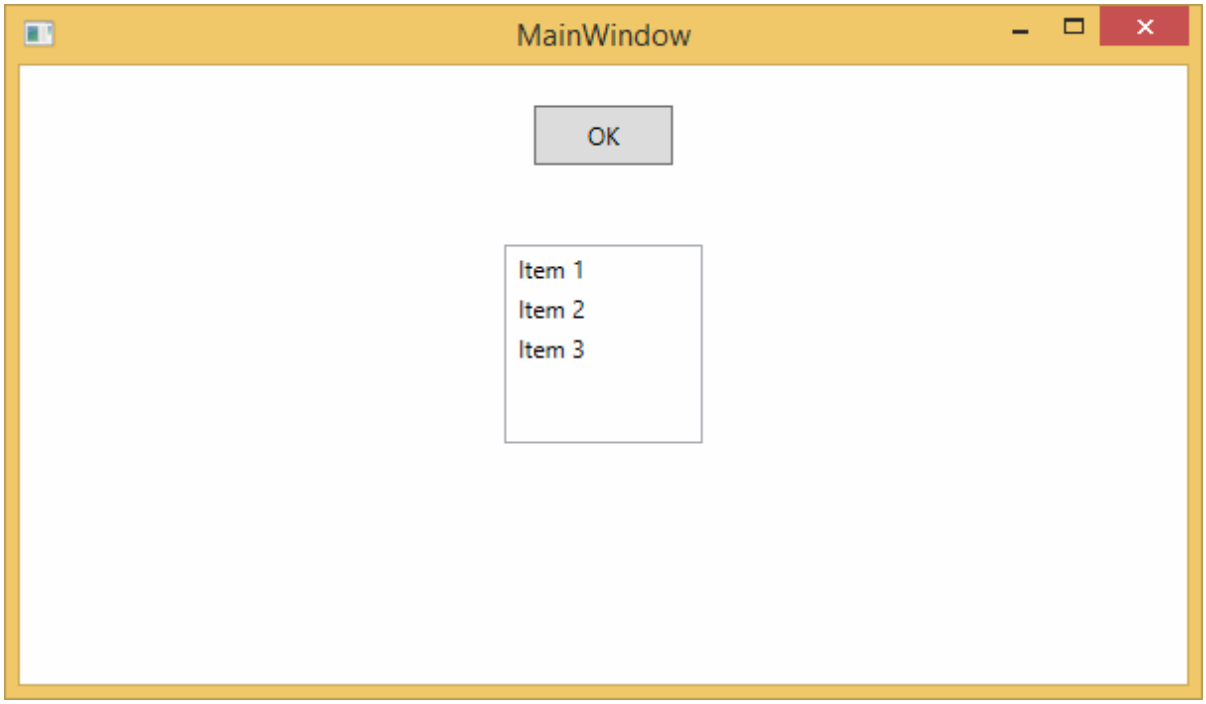
In WPF, the concept of the visual tree describes the structure of visual objects, as represented by the Visual Base Class. It signifies all the UI elements which are rendered to the output screen.

When a programmer wants to create a template for a particular control, he is actually rendering the visual tree of that control. The visual tree is also very useful for those who want to draw lower level controls for performance and optimization reasons.

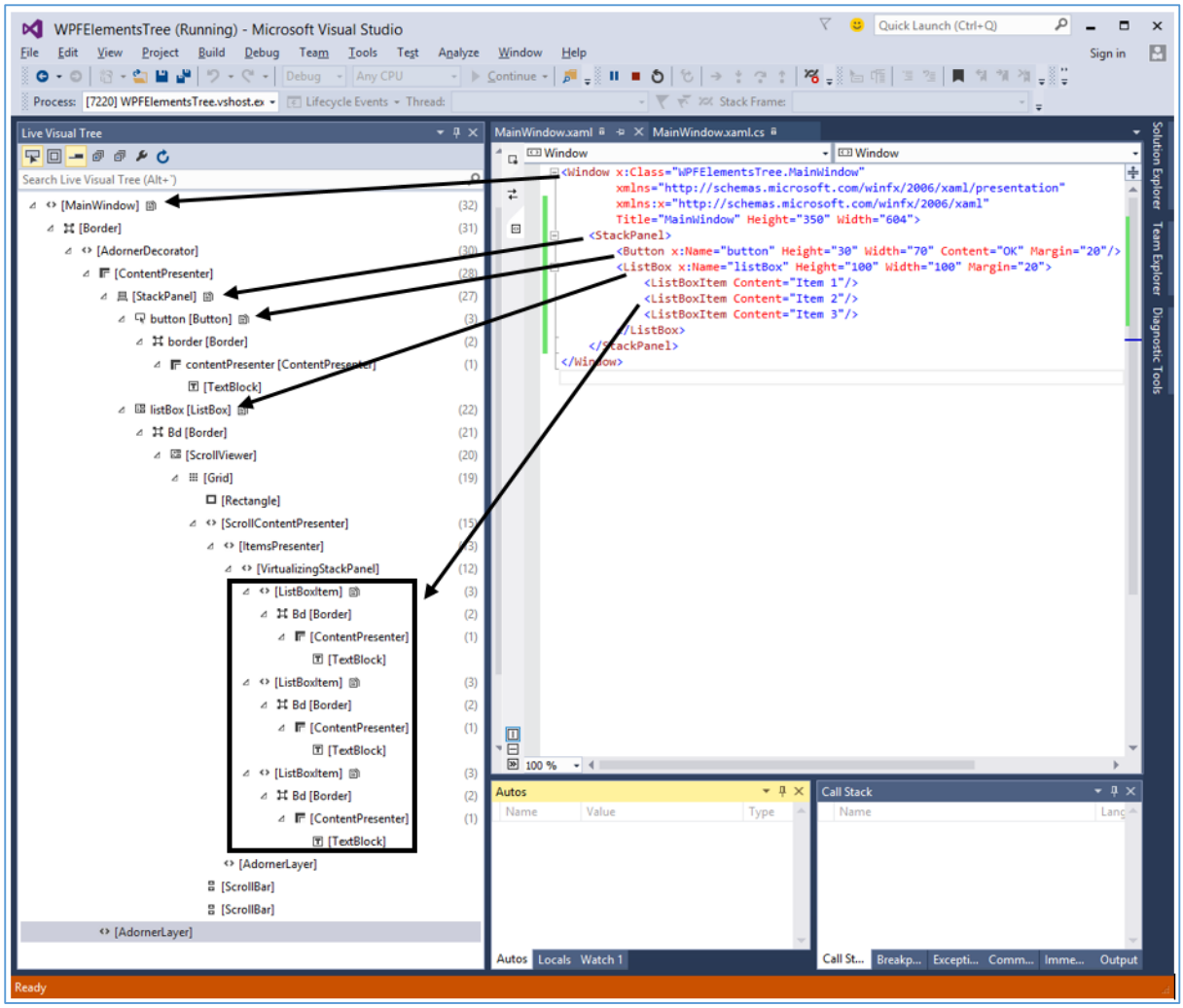
In WPF applications, visual tree is used for;

- Rendering the visual objects.
- Rendering the layouts.
- The routed events mostly travel along the visual tree, not the logical tree.

To see the visual tree of the above simple application which contains a button and a list box, let's compile and execute the XAML code and you will see the following window.



When the application is running, you can see the visual tree of the running application in Live Visual Tree window which shows the complete hierarchy of this application, as shown below.



The visual tree is typically a superset of the logical tree. You can see here that all the logical elements are also present in the visual tree. So these two trees are really just two different views of the same set of objects that make up the UI.

- The logical tree leaves out a lot of detail enabling you to focus on the core structure of the user interface and to ignore the details of exactly how it has been presented.

- The logical tree is what you use to create the basic structure of the user interface.

- The visual tree will be of interest if you're focusing on the presentation. For example, if you wish to customize the appearance of any UI element, you will need to use the visual tree.

6. WPF – DEPENDENCY PROPERTIES

In WPF applications, dependency property is a specific type of property which extends the CLR property. It takes the advantage of specific functionalities available in the WPF property system.

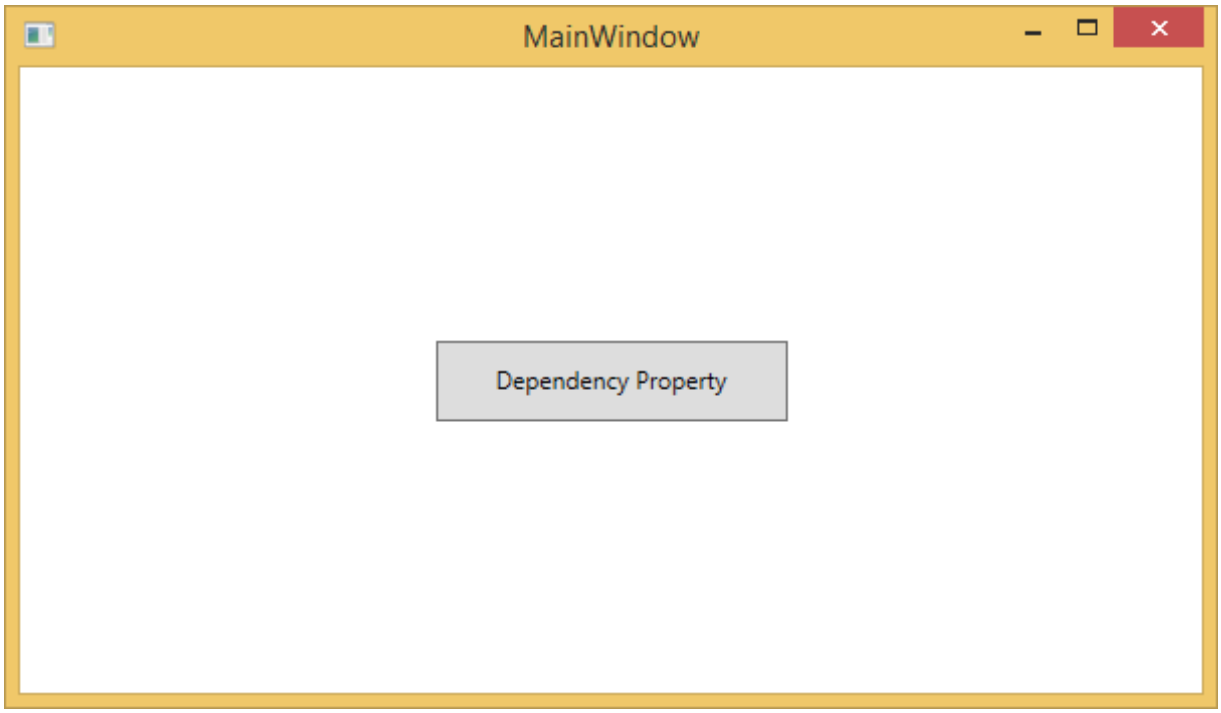
A class which defines a dependency property must be inherited from the **DependencyObject** class. Many of the UI controls class which are used in XAML are derived from the **DependencyObject** class and they support dependency properties, e.g. Button class supports the **IsMouseOver** dependency property.

The following XAML code creates a button with some properties.

```
<Window x:Class="WPFDependencyProperty.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WPFDependencyProperty"
        Title="MainWindow" Height="350" Width="604">
    <Grid>
        <Button  Height="40"
                 Width="175"
                 Margin="10"
                 Content="Dependency Property">
            <Button.Style>
                <Style TargetType="{x:Type Button}">
                    <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value="True">
                            <Setter Property="Foreground" Value="Red"/>
                        </Trigger>
                    </Style.Triggers>
                </Style>
            </Button.Style>
        </Button>
    </Grid>
</Window>
```

The x:Type markup extension in XAML has a similar functionality like typeof() in C#. It is used when attributes are specified which take the type of the object such as `<Style TargetType="{x:Type Button}">`

When the above code is compiled and executed, you would get the following **MainWindow**. When the mouse is over the button, it will change the foreground color of a button. When the mouse leaves the button, it changes back to its original color.



Why We Need Dependency Properties

Dependency property gives you all kinds of benefits when you use it in your application. Dependency Property can used over a CLR property in the following scenarios;

- If you want to set the style
- If you want data binding
- If you want to set with a resource (a static or a dynamic resource)
- If you want to support animation

Basically, Dependency Properties offer a lot of functionalities that you won't get by using a CLR property.

The main difference between **dependency properties** and other **CLR properties** are listed below:

- CLR properties can directly read/write from the private member of a class by using **getter** and **setter**. In contrast, dependency properties are not stored in local object.

- Dependency properties are stored in a dictionary of key/value pairs which is provided by the DependencyObject class. It also saves a lot of memory because it stores the property when changed. It can be bound in XAML as well.

Custom Dependency Properties

In .NET framework, custom dependency properties can also be defined. Follow the steps given below to define custom dependency property in C#.

- Declare and register your **dependency property** with system call register.
- Provide the **setter** and **getter** for the property.
- Define a **static handler** which will handle any changes that occur globally
- Define an **instance handler** which will handle any changes that occur to that particular instance.

The following C# code defines a dependency property to set the **SetText** property of the user control.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace WpfApplication3
{
    /// <summary>
    /// Interaction logic for UserControl1.xaml
    /// </summary>
    public partial class UserControl1 : UserControl
```

```
    {
        public UserControl1()
        {
            InitializeComponent();
        }

        public static readonly DependencyProperty SetTextProperty =
DependencyProperty.Register("SetText", typeof(string), typeof(UserControl1),
new PropertyMetadata("", new PropertyChangedCallback(OnSetTextChanged)));
        public string SetText
        {
            get { return (string)GetValue(SetTextProperty); }
            set { SetValue(SetTextProperty, value); }
        }
        private static void OnSetTextChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
        {
            UserControl1 UserControl1Control = d as UserControl1;
            UserControl1Control.OnSetTextChanged(e);
        }
        private void OnSetTextChanged(DependencyPropertyChangedEventArgs e)
        {
            tbTest.Text = e.NewValue.ToString();
        }

    }
}
```

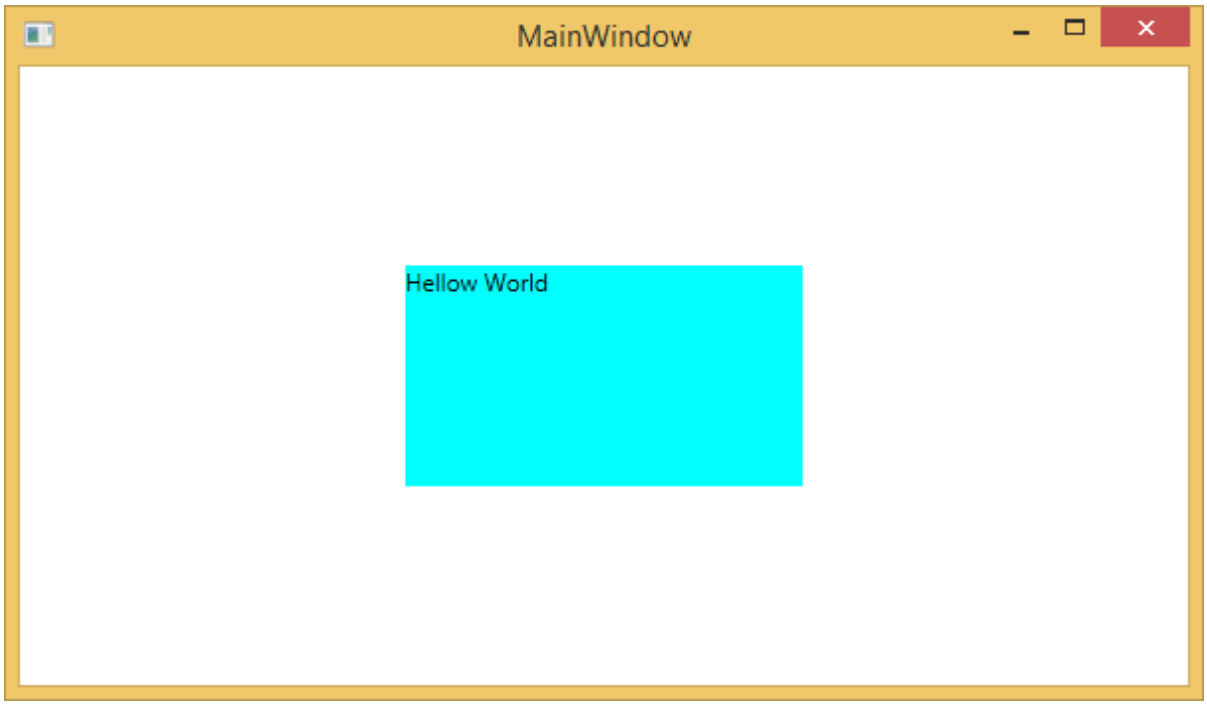
Here is the XAML file in which the TextBlock is defined as a user control and the Text property will be assigned to it by the SetText dependency property.

The following XAML code creates a user control and initializes its **SetText** dependency property.

```
<Window x:Class="WpfApplication3.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:views="clr-namespace:WpfApplication3"
```

```
        Title="MainWindow" Height="350" Width="604">
    <Grid>
        <views:UserControl1 SetText="Hellow World"/>
    </Grid>
</Window>
```

Let's run this application. You can immediately observe that in our MainWindow, the dependency property for user control has been successfully used as a Text.



7. WPF – ROUTED EVENTS

A **routed event** is a type of event that can invoke handlers on multiple listeners in an element tree rather than just the object that raised the event. It is basically a CLR event that is supported by an instance of the Routed Event class. It is registered with the WPF event system. RoutedEvents have three main routing strategies which are as follows;

- Direct Event
- Bubbling Event
- Tunnel Event

Direct Event

A direct event is similar to events in Windows forums which are raised by the element in which the event is originated.

Unlike a standard CLR event, direct routed events support class handling and they can be used in Event Setters and Event Triggers within your style of your Custom Control.

A good example of a direct event would be the MouseEnter event.

Bubbling Event

A bubbling event begins with the element where the event is originated. Then it travels up the visual tree to the topmost element in the visual tree. So, in WPF, the topmost element is most likely a window.

Tunnel Event

Event handlers on the element tree root are invoked and then the event travels down the visual tree to all the children nodes until it reaches the element in which the event originated.

The difference between a bubbling and a tunneling event is that a tunneling event will always start with a preview.

In a WPF application, events are often implemented as a tunneling/bubbling pair. So, you'll have a preview MouseDown and then a MouseDown event.

Given below is a simple example of a Routed event in which a button and three text blocks are created with some properties and events.

```
<Window x:Class="WPFRoutedEvents.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

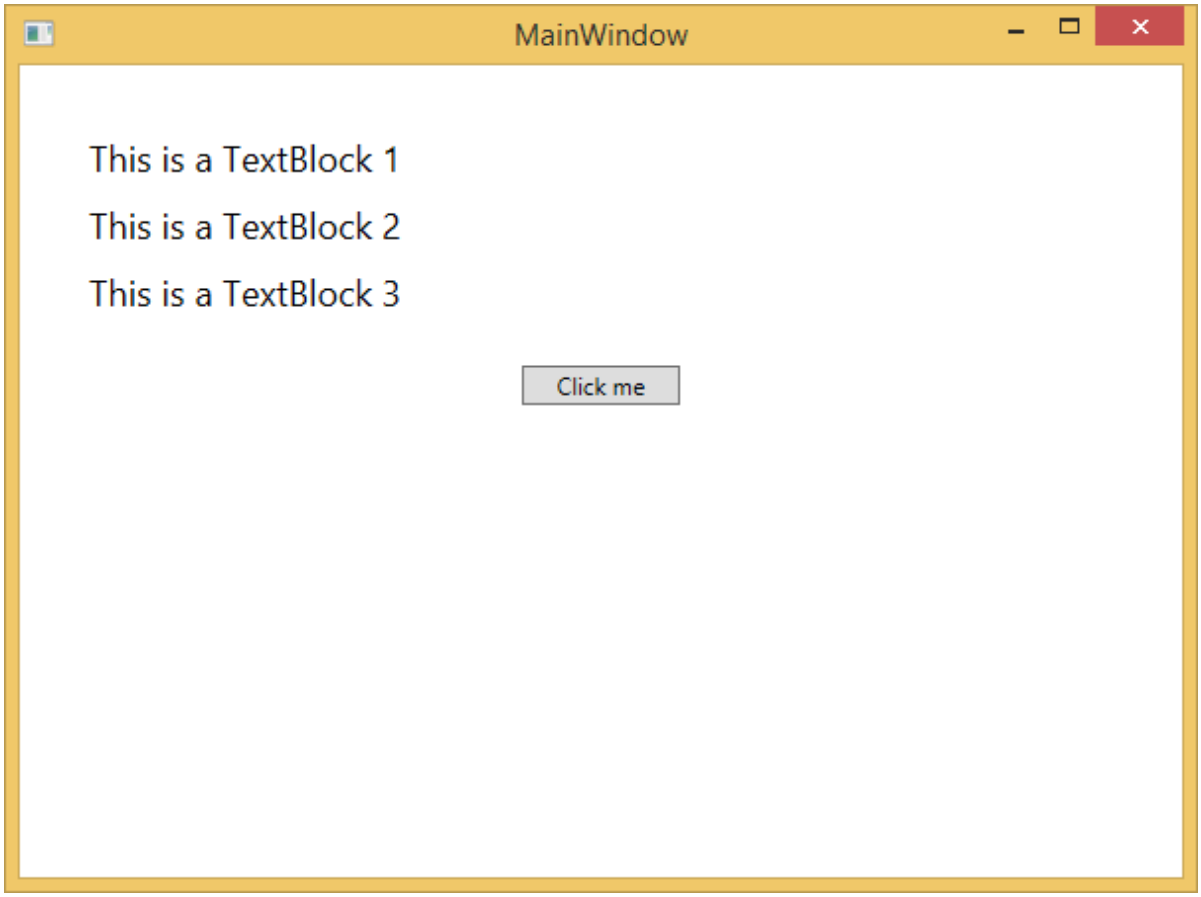
```
        Title="MainWindow" Height="450" Width="604"
        ButtonBase.Click ="Window_Click" >
    <Grid>
        <StackPanel Margin="20" ButtonBase.Click="StackPanel_Click">
            <StackPanel Margin="10">
                <TextBlock Name="txt1" FontSize="18" Margin="5" Text="This is a
TextBlock 1" />
                <TextBlock Name="txt2" FontSize="18" Margin="5" Text="This is a
TextBlock 2" />
                <TextBlock Name="txt3" FontSize="18" Margin="5" Text="This is a
TextBlock 3" />
            </StackPanel>
            <Button Margin="10" Content="Click me"
                    Click="Button_Click"
                    Width="80"/>
        </StackPanel>
    </Grid>
</Window>
```

Here is the C# code for the Click events implementation for Button, StackPanel, and Window.

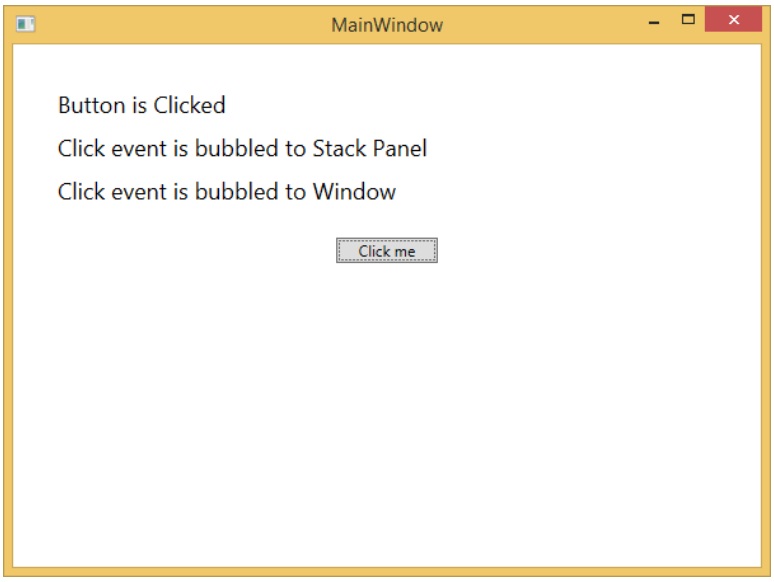
```
using System.Windows;

namespace WPFRoutedEvents
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
```

```
        {
            txt1.Text = "Button is Clicked";
        }
        private void StackPanel_Click(object sender, RoutedEventArgs e)
        {
            txt2.Text = "Click event is bubbled to Stack Panel";
        }
        private void Window_Click(object sender, RoutedEventArgs e)
        {
            txt3.Text = "Click event is bubbled to Window";
        }
    }
}
```

When you compile and execute the above code, it will produce the following window:



When you click on the button, the text blocks will get updated, as shown below.

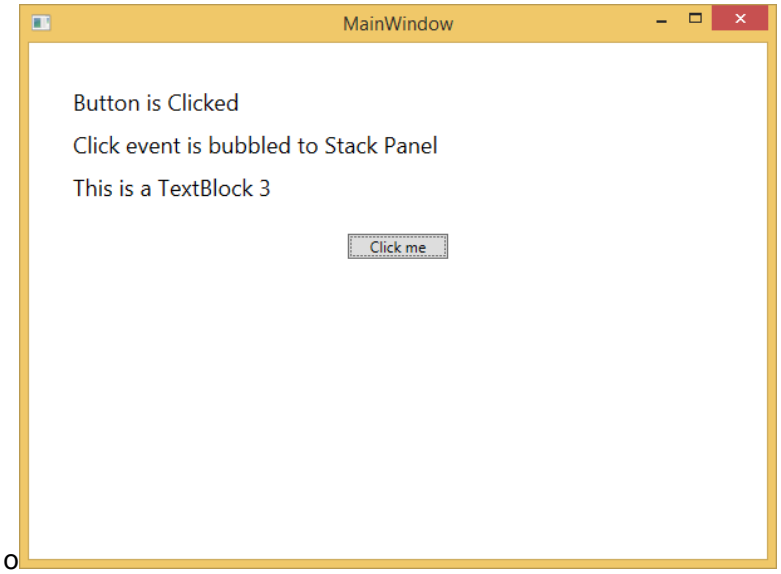


If you want to stop the routed event at any particular level, then you will need to set the **e.Handled = true;**

Let's change the **StackPanel_Click** event as shown below:

```
private void StackPanel_Click(object sender, RoutedEventArgs e)
        {
            txt2.Text = "Click event is bubbled to Stack Panel";
            e.Handled = true;
        }
```

When you click on the button, you will observe that the click event will not be routed to the window and will stop at the stackpanel and the 3rd text block will not be updated.



o

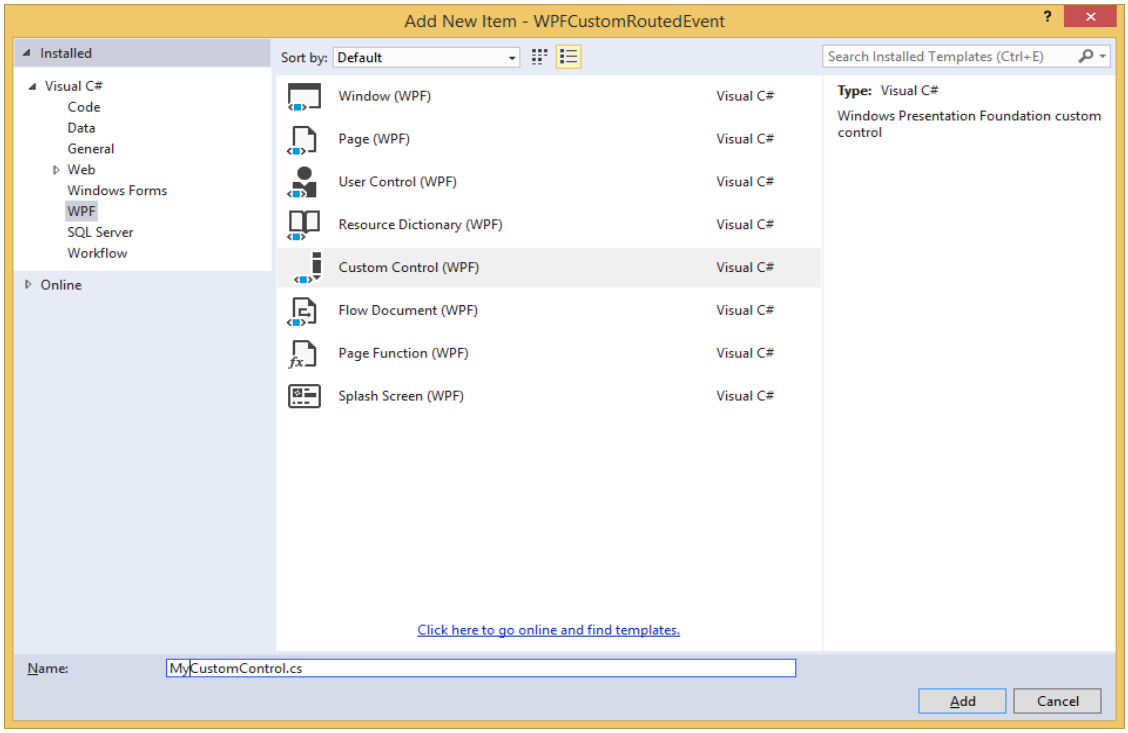
Custom Routed Events

In .NET framework, custom routed event can also be defined. You need to follow the steps given below to define a custom routed event in C#.

- Declare and register your routed event with system call RegisterRoutedEvent.
- Specify the Routing Strategy, i.e. Bubble, Tunnel, or Direct.
- Provide the event handler.

Let's take an example to understand more about custom routed events. Follow the steps given below:

1. Create a new WPF project with WPFCustomRoutedEvent
2. Right click on your solution and select Add > New Item...
3. The following dialog will open, now select **Custom Control (WPF)** and name it **MyCustomControl**.



4. Click the **Add** button and you will see that two new files (Themes/Generic.xaml and MyCustomControl.cs) will be added in your solution.

The following XAML code sets the style for the custom control in Generic.xaml file.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WPFCustomRoutedEvent">

    <Style TargetType="{x:Type local:MyCustomControl}">
        <Setter Property="Margin" Value="50"/>
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type local:MyCustomControl}">
                    <Border Background="{TemplateBinding Background}"
                            BorderBrush="{TemplateBinding BorderBrush}"
                            BorderThickness="{TemplateBinding BorderThickness}">
                        <Button x:Name="PART_Button" Content="Click Me" />
                    </Border>
                </ControlTemplate>
            </Setter.Value>
```

```
        </Setter>
    </Style>
</ResourceDictionary>
```

Given below is the C# code for the **MyCustomControl class** which inherits from the **Control class** in which a custom routed event Click is created for the custom control.

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

namespace WPFCustomRoutedEvent
{
    public class MyCustomControl : Control
    {
        static MyCustomControl()
        {
            DefaultStyleKeyProperty.OverrideMetadata(typeof(MyCustomControl),
new FrameworkPropertyMetadata(typeof(MyCustomControl)));
        }
        public override void OnApplyTemplate()
        {
            base.OnApplyTemplate();

            //demo purpose only, check for previous instances and remove the handler first
            var button = GetTemplateChild("PART_Button") as Button;
            if (button != null)
                button.Click += Button_Click;

        }

        void Button_Click(object sender, RoutedEventArgs e)
        {
            RaiseClickEvent();
        }

        public static readonly RoutedEvent ClickEvent =
```

```
            EventManager.RegisterRoutedEvent("Click", RoutingStrategy.Bubble,
            typeof(RoutedEventHandler), typeof(MyCustomControl));

        public event RoutedEventHandler Click
        {
            add { AddHandler(ClickEvent, value); }
            remove { RemoveHandler(ClickEvent, value); }
        }

        protected virtual void RaiseClickEvent()
        {
            RoutedEventArgs args = new
RoutedEventArgs(MyCustomControl.ClickEvent);
            RaiseEvent(args);
        }
    }
}
```

Here is the custom routed event implementation in C# which will display a message box when the user clicks it.

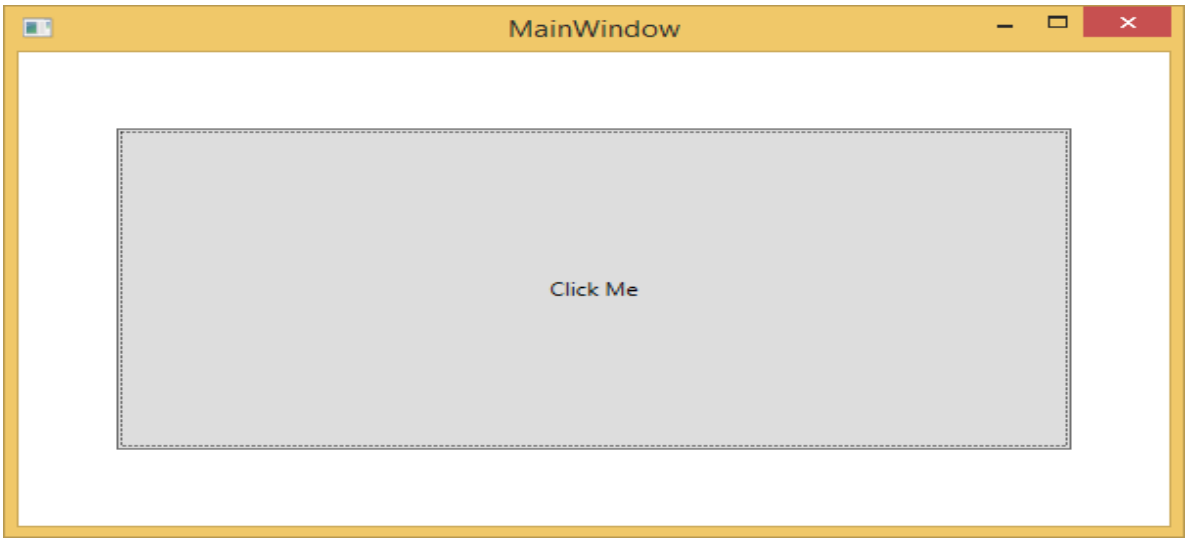
```
using System.Windows;

namespace WPFCustomRoutedEvent
{
    // <summary>
    // Interaction logic for MainWindow.xaml
    // </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void MyCustomControl_Click(object sender, RoutedEventArgs e)
```

```
        {
            MessageBox.Show("It is the custom routed event of your custom control");
        }
    }
}
```

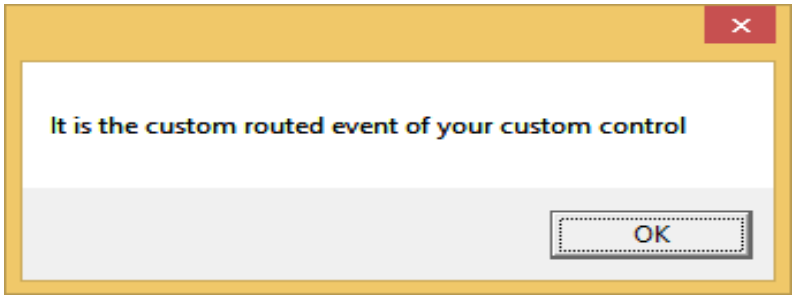
Here is the implementation in MainWindow.xaml to add the custom control with a routed event Click.

```
<Window x:Class="WPFCustomRoutedEvent.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WPFCustomRoutedEvent"
        Title="MainWindow" Height="350" Width="604">
    <Grid>
        <local:MyCustomControl Click="MyCustomControl_Click"/>
    </Grid>
</Window>
```

When the above code is compiled and executed, it will produce the following window which contains a custom control.



When you click on the custom control, it will produce the following message.



End of ebook preview

If you liked what you saw…

Buy it from our store @ https://store.tutorialspoint.com