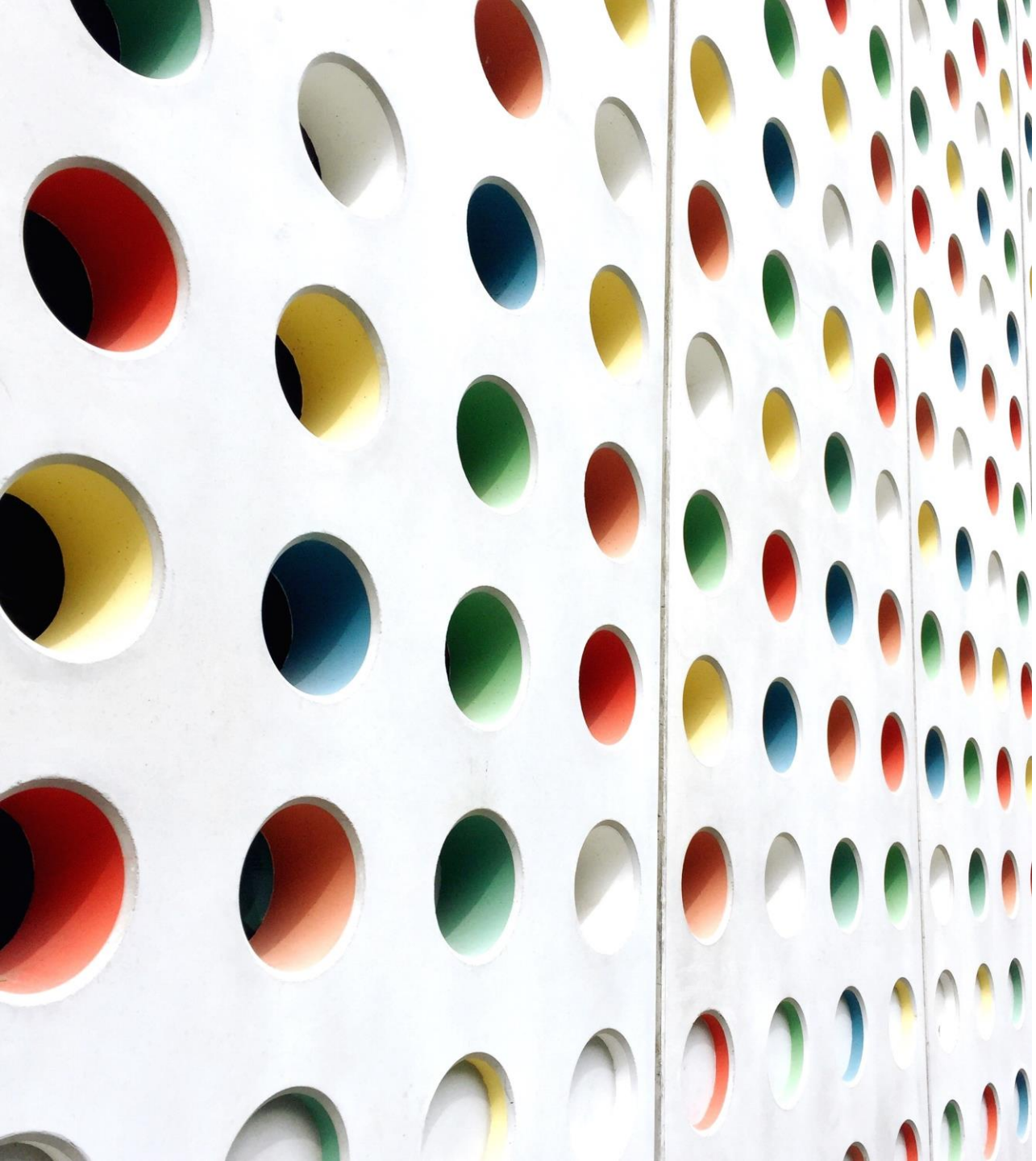
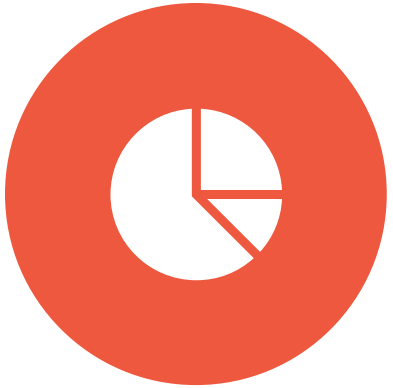


Windows Presentation Foundation (WPF)

ARCTECH INFO PRIVATE LIMITED

Title Lorem Ipsum



LOREM IPSUM DOLOR SIT AMET, CONSECTETUER ADIPISCING ELIT.

NUNC VIVERRA IMPERDIET ENIM. FUSCE EST. VIVAMUS A TELLUS.

PELLENTESQUE HABITANT MORBI TRISTIQUE SENECTUS ET NETUS.

What is WPF

Windows Presentation Foundation is a UI framework that creates desktop client applications.

WPF is part of .NET, so if you have previously built applications with .NET using ASP.NET or Windows Forms, the programming experience should be familiar.

WPF uses the Extensible Application Markup Language (XAML) to provide a declarative model for application programming.

Program with WPF

WPF types are mostly located in the System.Windows namespace.

If you have previously built applications with .NET with frameworks like ASP.NET and Windows Forms, the fundamental WPF programming experience should be familiar, you:

- Instantiate classes
- Set properties
- Call methods
- Handle events

WPF includes more programming constructs that enhance properties and events

- dependency properties and
- routed events.

Markup and code-behind

WPF lets you develop an application using both markup and code-behind, an experience with which ASP.NET developers should be familiar.

You generally use XAML markup to implement the appearance of an application while using managed programming languages (code-behind) to implement its behavior.

This separation of appearance and behavior has the following benefits:

- Development and maintenance costs are reduced.
- Development is more efficient
- Globalization and localization for WPF applications is simplified.

Markup - XAML

XAML – eXtensible Application Markup Language, is Microsoft's implementation of XML for describing a GUI

You typically use it to define windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics.

The following example uses XAML to implement the appearance of a window that contains a single button:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Week13Day01.MyFirstWindow“
    Title="Window with Button"
    Width="250" Height="100">

    <!-- Add button to window -->
    <Button Name="button">Click Me!</Button>
</Window>
```

Code-behind

The main behavior of an application is to respond to user interactions.

For example, when user clicks a menu or button, invoke some business logic and data access logic in response.

In WPF, this behavior is implemented in a code-behind file that is associated with the xaml file.

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="HelloWorldApp.MyWindow"
Title="Window with Button"
Width="250" Height="100">
    <!-- Add button to window -->
    <Button Name="Button1" Click="Button1_Click">Click Me!</Button>
</Window>
```

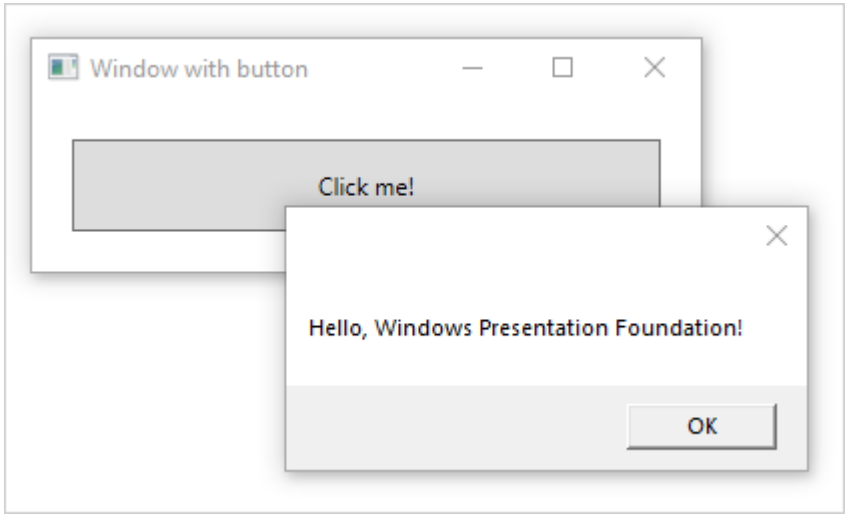
```
using System.Windows;
namespace HelloWorldApp
{
    public partial class MyWindow : Window
    {
        public MyWindow()
        {
            InitializeComponent();
        }

        void Button1_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}
```

Code Behind - InitalizeComponent

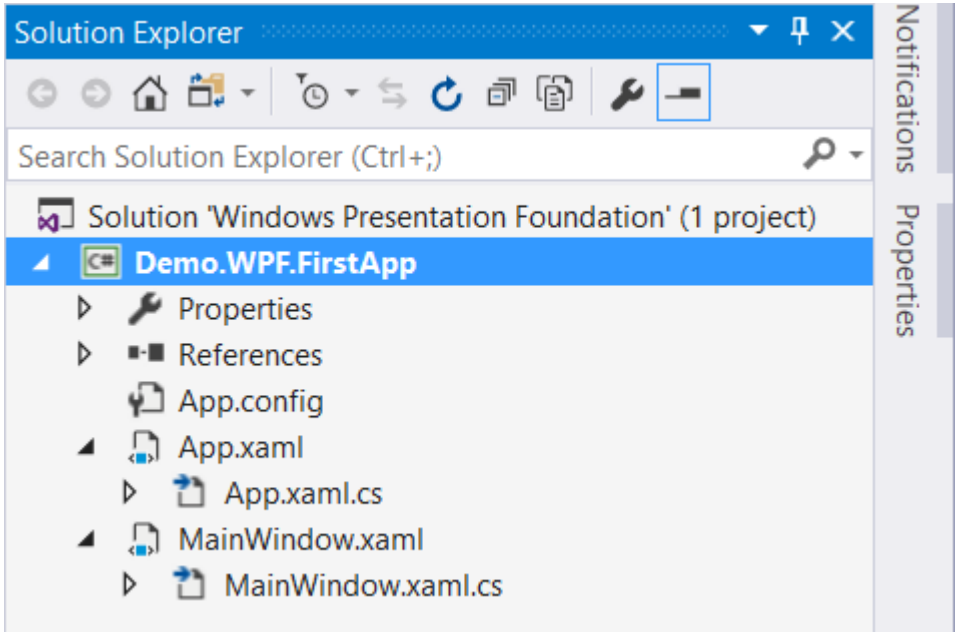
InitializeComponent is called from the code-behind class's constructor to merge the UI that is defined in markup with the code-behind class.

The combination of x:Class and InitializeComponent ensure that all elements include events are correctly implemented.

InitializeComponent has auto generated C# code based on the XAML.



WPF Project Structure



In a new WPF Project, 5 files are created automatically by Visual Studio

- App.config
  - Is used to store application settings like connectionstring etc.
- App.xaml & App.xaml.cs
  - Xaml is declarative start point of your application
  - Inherits Application class
  - Can subscribe to application-specific styles, events like `StartUp`, unhandled exceptions etc.
  - Xaml.cs is the code behind file
  - A project can have only one app.xaml file
- MainWindow.xaml & MainWindow.xaml.cs
  - Is the declarative window definition
  - Inherits Window class
  - Xaml.cs is the code behind file for the
  - A project can have multiple window xaml files

WPF Controls

The user experience is constructed using WPF controls. In WPF, control is an umbrella term that applies to a category of WPF classes that have the following characteristics:

- Hosted in either a window or a page.
- Have a user interface.
- Implement some behavior.

Built-in WPF controls

- **Buttons**: Button and RepeatButton.
- **Data Display**: DataGrid, ListView, and TreeView.
- **Date Display and Selection**: Calendar and DatePicker.
- **Dialog Boxes**: OpenFileDialog, PrintDialog, and SaveFileDialog.
- **Digital Ink**: InkCanvas and InkPresenter.
- **Documents**: DocumentViewer, FlowDocumentReader, FlowDocumentScrollViewer, and StickyNoteControl.
- **Input**: TextBox, RichTextBox, and PasswordBox.
- **Layout**: Border, BulletDecorator, Canvas, DockPanel, Expander, Grid, GridView, GridSplitter, GroupBox, Panel, ResizeGrip, Separator, ScrollBar, ScrollViewer, StackPanel, Thumb, Viewbox, VirtualizingStackPanel, Window, and WrapPanel.
- **Media**: Image, MediaElement, and SoundPlayerAction.
- **Menus**: ContextMenu, Menu, and ToolBar.
- **Navigation**: Frame, Hyperlink, Page, NavigationWindow, and TabControl.
- **Selection**: CheckBox, ComboBox, ListBox, RadioButton, and Slider.
- **User Information**: AccessText, Label, Popup, ProgressBar, StatusBar, TextBlock, and ToolTip.

Layout

To create a user interface, arrange your controls by location and size to form a layout.

A key requirement of any layout is to adapt to changes in window size and display settings.

Rather than writing code to adapt a layout in these circumstances, WPF provides a first-class, extensible layout system for you.

The WPF layout system has relative positioning, which increases the ability to adapt to changing window and display conditions.

The layout system also manages the negotiation between controls to determine the layout. The negotiation is a two-step process:

- first, a control tells its parent what location and size it requires.
- Second, the parent tells the control what space it can have.

Canvas: Child controls provide their own layout.

DockPanel: Child controls are aligned to the edges of the panel.

Grid: Child controls are positioned by rows and columns.

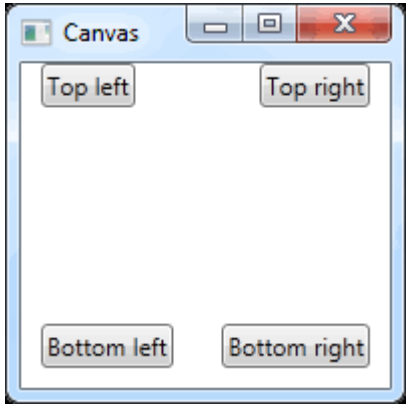
StackPanel: Child controls are stacked either vertically or horizontally.

VirtualizingStackPanel: Child controls are virtualized and arranged on a single line that is either horizontally or vertically oriented.

WrapPanel: Child controls are positioned in left-to-right order and wrapped to the next line when there isn't enough space. on the current line.

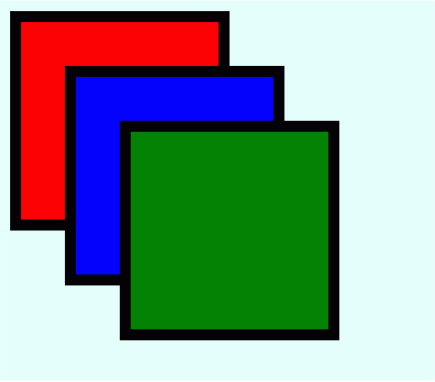
Layout - Canvas

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




```
<Canvas>
    <Button Canvas.Left="10">Top left</Button>
    <Button Canvas.Right="10">Top right</Button>
    <Button Canvas.Left="10" Canvas.Bottom="10">Bottom left</Button>
    <Button Canvas.Right="10" Canvas.Bottom="10">Bottom right</Button>
</Canvas>
```

```
<Canvas Background="LightCyan">
    <Rectangle Canvas.Left="10" Canvas.Top="10" Height="200" Width="200" Stroke="Black"
               StrokeThickness="10" Fill="Red" />
    <Rectangle Canvas.Left="60" Canvas.Top="60" Height="200" Width="200" Stroke="Black"
               StrokeThickness="10" Fill="Blue" />
    <Rectangle Canvas.Left="110" Canvas.Top="110" Height="200" Width="200" Stroke="Black"
               StrokeThickness="10" Fill="Green" />
</Canvas>
```



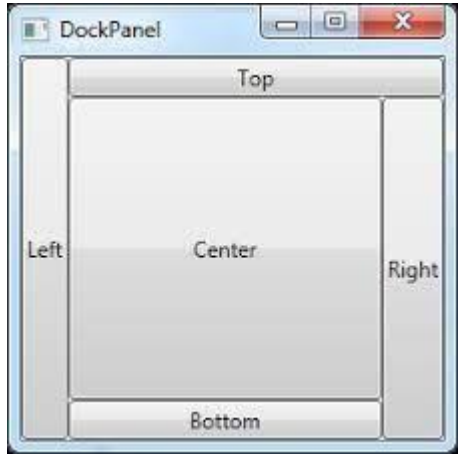
Layout - DockPanel

DockPanel defines an area to arrange child elements relative to each other

- horizontally or
- vertically.

With DockPanel you can easily dock child elements using the Dock property to

- top,
- bottom,
- right,
- left and
- center.




```
<DockPanel>
    <Button DockPanel.Dock="Left">Left</Button>
    <Button DockPanel.Dock="Top">Top</Button>
    <Button DockPanel.Dock="Right">Right</Button>
    <Button DockPanel.Dock="Bottom">Bottom</Button>
    <TextBox>Center</TextBox>
</DockPanel>
```

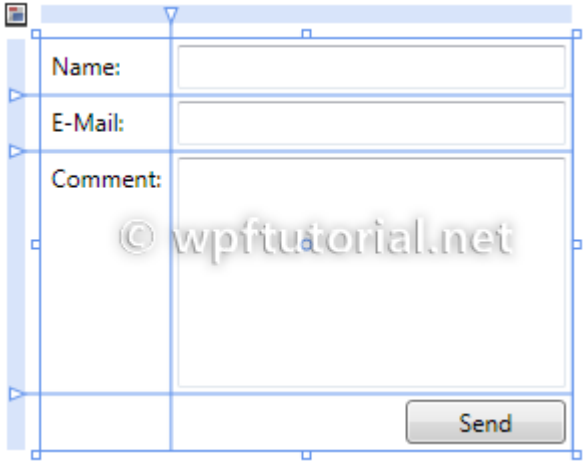
Layout - Grid

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="28" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="200" />
    </Grid.ColumnDefinitions>

    <Label Grid.Row="0" Grid.Column="0" Content="Name:"/>
    <Label Grid.Row="1" Grid.Column="0" Content="E-Mail:"/>
    <Label Grid.Row="2" Grid.Column="0" Content="Comment:"/>

    <TextBox Grid.Row="0" Grid.Column="1" Margin="3" />
    <TextBox Grid.Row="1" Grid.Column="1" Margin="3" />
    <TextBox Grid.Row="2" Grid.Column="1" Margin="3" />

    <Button Grid.Row="3" Grid.Column="1" HorizontalAlignment="Right"
            MinWidth="80" Margin="3" Content="Send" />
</Grid>
```

The grid is a layout panel that arranges its child controls in a tabular structure of rows and columns.

Its functionality is similar to the HTML table but more flexible.

A cell can contain multiple controls, they can span over multiple cells and even overlap themselves.

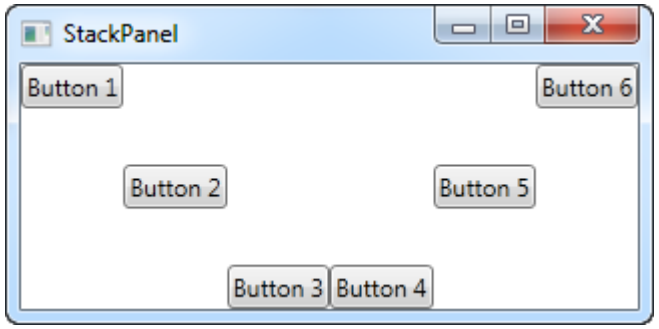


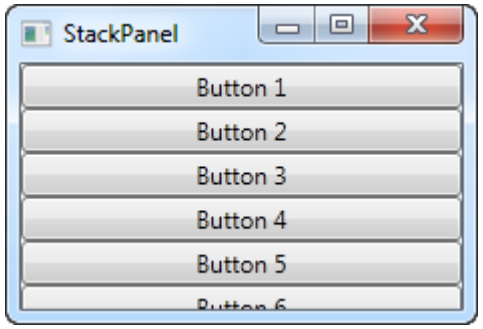
Layout - StackPanel

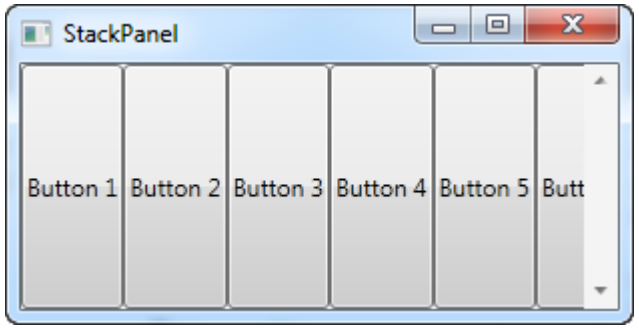
```
<StackPanel Orientation="Horizontal">
    <Button VerticalAlignment="Top">Button 1</Button>
    <Button VerticalAlignment="Center">Button 2</Button>
    <Button VerticalAlignment="Bottom">Button 3</Button>
    <Button VerticalAlignment="Bottom">Button 4</Button>
    <Button VerticalAlignment="Center">Button 5</Button>
    <Button VerticalAlignment="Top">Button 6</Button>
</StackPanel>
```

Arrange control either vertically or horizontally.

At the time of stack layout creation, you have to set the property Orientation either as Vertical or as Horizontal.

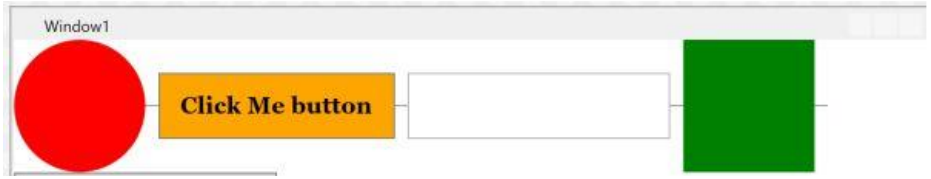






Layout - WrapPanel

```
<WrapPanel>
    <Ellipse Width="100" Height="100" Fill="Red" />
    <Separator Width="10" />
    <Button Background="Orange" Width="180" Height="50“
            FontFamily="Georgia" FontSize="18“
            FontWeight="Bold" Name="ClickMeButton">
            Click Me button
    </Button>
    <Separator Width="10" />
    <TextBox Height="50" Width="200" Name="Button1" />
    <Rectangle Width="100" Height="100" Fill="Green" />
</WrapPanel>
```



WPF WrapPanel control is a panel that positions child elements in sequential position from left to right by default.

If child elements that are stacked don’t fit in the row or column they are in, the remaining elements will wrap around in the same sequence.

See example of horizontal wrap panel

Namespaces in XAML

At the beginning of every XAML file you need to include two namespaces.

http://schemas.microsoft.com/winfx/2006/xaml/presentation.

- It is mapped to all wpf controls in System.Windows.Controls.

http://schemas.microsoft.com/winfx/2006/xaml

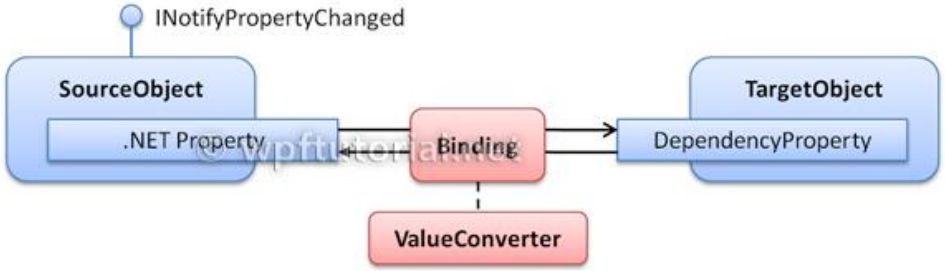
- it is mapped to System.Windows.Markup that defines the XAML keywords.

You can also directly include a CLR namespace in XAML by using the clr-namespace: prefix.

Example

```
<Window x:Class="Week13Day3Test.MainWindow“
        xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
        xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
        xmlns:local="clr-namespace:Week13Day3Test">
```

Data binding overview



Data Binding is a simple and powerful way to auto-update data between the business model and the user interface.

Every time when the data of your business model changes, it automatically reflects the updates to the user interface and vice versa.

This is the preferred method in WPF to bring data to the user interface.

Databinding can be unidirectional (source -> target or target <- source), or bidirectional (source <-> target).

Like in WinFoms, you set properties on a control manually like populate a ListBox by adding items to it from a loop

But the cleanest and purest WPF way is to add a binding between the source object and the destination UI element.

Data binding Hello World

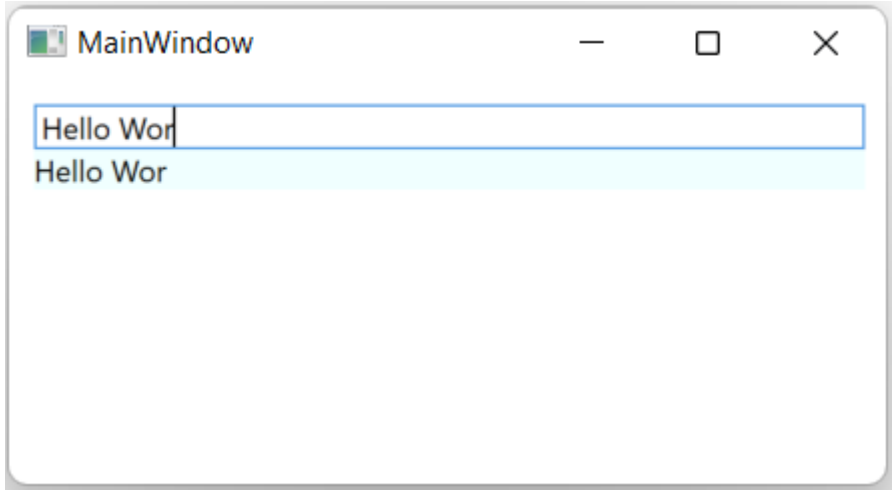
This example shows how we bind the value of the TextBlock to match the Text property of the TextBox.

The TextBlock is automatically updated when you enter text into the TextBox.

In WinForms, this would require us to listen to the KeyPressed event on the TextBox and then update the TextBlock each time the text changes

In WPF, with data binding, the connection between the two controls can be established automatically just by specifying it in xaml.

```
<StackPanel Margin="10">
   <TextBox Name="txtValue" />
   <TextBlock
      Text="{Binding Path=Text, ElementName=txtValue}" />
</StackPanel>
```



The syntax of a Binding

The binding magic happens between the curly braces

In XAML the curly braces contain a Markup Extension.

Binding is a type of Markup Extension, which allows us to describe the binding relationship for the Text property. In its most simple form, a binding can look like this:

{Binding Path=Text, ElementName=txtValue}

Using DataContext

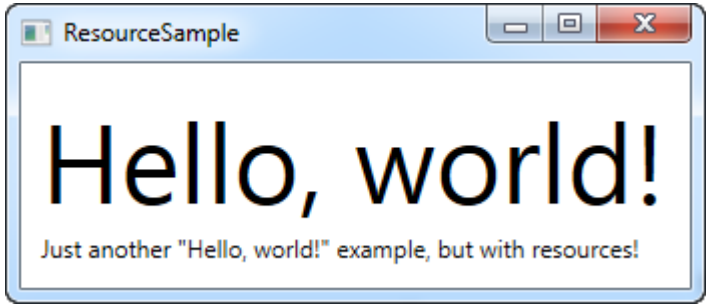
During binding you can specify ElementName on every control, when you want to bind to another element

- `<TextBlock Text="{Binding Path=Text, ElementName=txtValue}" />`

However, the DataContext property is the default source of your bindings when you want to bind to an object in your Model

- `<TextBlock Text="{Binding Path=Name}" />`
- Or
- `<TextBlock Text="{Binding Name}" />`
- In Code-Behind
- `this.DataContext=<object>;`

Static Resource Binding



WPF introduces a very handy concept: The ability to store data as a resource, either

- locally for a control,
- locally for the entire window or
- globally for the entire application.

The data can be whatever you want, from actual information to a hierarchy of WPF controls.

This allows you to place data in one place and then use it from or several other places, which is very useful.

```
<Window.Resources>
  <sys:String x:Key="strHelloWorld">Hello, world!</sys:String>
</Window.Resources>

<StackPanel Margin="10">
  <TextBlock Text="{StaticResource strHelloWorld}" FontSize="56" />
  <TextBlock>
    Just another
    "<TextBlock Text="{StaticResource strHelloWorld}" />"
    example, but with resources!
  </TextBlock>
</StackPanel>
```

Add the following attribute in the Window Tag

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Dynamic Resource Binding

Last example used StaticResource

A StaticResource is resolved only once, which is at the point where the XAML is loaded.

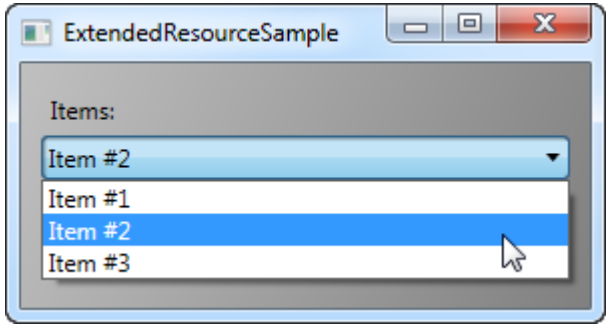
If the resource is then changed later on, this change will not be reflected where you have used the StaticResource.

A DynamicResource is resolved once it's actually needed, and then again if the resource changes.

It is like binding to a function which monitors this value and sends it to you each time it's changed.

Dynamic resources also allows you to use resources which are not even there during design time, e.g., if you add them from Code-behind during the startup of the application.

More Resource Types




```
<Window.Resources>
  <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
  <x:Array x:Key="ComboBoxItems" Type="sys:String">
    <sys:String>Item #1</sys:String>
    <sys:String>Item #2</sys:String>
    <sys:String>Item #3</sys:String>
  </x:Array>

  <LinearGradientBrush x:Key="WindowBackgroundBrush">
    <GradientStop Offset="0" Color="Silver"/>
    <GradientStopOffset="1" Color="Gray"/>
  </LinearGradientBrush>
</Window.Resources>
```

Local and Application wide Resources

LOCAL RESOURCES

```
<StackPanel Margin="10">
   <StackPanel.Resources>
      <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
   </StackPanel.Resources>
   <Label Content="{StaticResource ComboBoxTitle}" />
</StackPanel>
```

APPLICATION RESOURCES

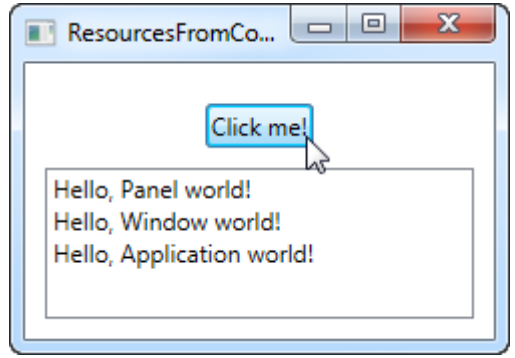
```
<Application x:Class="WpfTutorialSamples.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:sys="clr-namespace:System;assembly=mscorlib"
   StartupUri="WPF application/ExtendedResourceSample.xaml">
   <Application.Resources>
      <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
   </Application.Resources>
</Application>
```

Resources from Code Behind

```
<Application.Resources>
    <sys:String x:Key="strApp">Hello, Application world!</sys:String>
</Application.Resources>
```

```
<Window.Resources>
    <sys:String x:Key="strWindow">Hello, Window world!</sys:String>
</Window.Resources>
<DockPanel Margin="10" Name="pnlMain">
    <DockPanel.Resources>
        <sys:String x:Key="strPanel">Hello, Panel world!</sys:String>
    </DockPanel.Resources>
    …
</DockPanel>
```

```
lbResult.Items.Add(pnlMain.FindResource("strPanel").ToString());
lbResult.Items.Add(this.FindResource("strWindow").ToString());
lbResult.Items.Add(Application.Current.FindResource("strApp").ToString());
```



RelativeSource Binding

The relative source mode of the binding extension helps you to bind to an object with an relative relation to you.
You don't know your or its absolute position, but you know if

- it's the previous or next item,
- two levels above you or of a specific type.

{Binding Text, RelativeSource={RelativeSource Self}}	Binds Own Text property
{Binding Background, RelativeSource={RelativeSource Mode=FindAncestor,AncestorType={x:Type StackPanel}}}	Binds to the Background property of a parent element of type StackPanel

Exception handling in WPF

```
private void Application_DispatcherUnhandledException(
    object sender,
    DispatcherUnhandledExceptionEventArgs e)
{
    MessageBox.Show("An unhandled exception just occurred: ");
    e.Handled=true;
}
```

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string s=null;
    try
    {
        s.Trim();
    }
    catch(Exception ex)
    {
        MessageBox.Show(
            "A handled exception just occurred: ");
    }
}
```

```
<Application
    ...
DispatcherUnhandledException="Application_DispatcherUnhandledException">
```

Styles in WPF

WPF styles works just like CSS style

In CSS we define styles for a control, and we reuse the same wherever needed on the website

Similarly, using styles in WPF allows us to define the properties which can be reused

```
<Window.Resources>
    <Style x:Key="wpfStyle1"
      TargetType="{x:Type TextBlock}">
        <Setter Property="FontFamily" Value="Verdana"/>
        <Setter Property="FontSize" Value="10"/>
    </Style>
    <Style x:Key="wpfStyle2" TargetType="{x:Type TextBlock}">
        <Setter Property="FontFamily" Value="Arial"/>
        <Setter Property="FontSize" Value="16"/>
    </Style>
</Window.Resources>
```

```
<TextBlock Margin="26,41,39,0" Style="{StaticResource wpfStyle1}"
    Height="31" VerticalAlignment="Top">TextBlock with Style1
</TextBlock>
<TextBlock Margin="26,77,39,0" Height="32" VerticalAlignment="Top">
    TextBlock with no Style
</TextBlock>
<TextBlock Margin="26,113,67,88" Style="{StaticResource wpfStyle2}">
    TextBlock with Style2
</TextBlock>
```

Control Templates

Controls in WPF are separated into

- logic, that defines the states, events and properties and
- template, that defines the visual appearance of the control.

Each control has a default template.

- This gives the control a basic appearance.
- The default template is typically shipped together with the control

You can create your own control template and set it to the Template property of a control

- This will completely replace the appearance of the control.

The control template is often included in a style that contains other property settings.

Control Template Example

```
<Style x:Key="DialogButtonStyle" TargetType="Button">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Button}">
                <Grid>
                    <Ellipse Fill="{TemplateBinding Background}" Stroke="{TemplateBinding BorderBrush}"/>
                    <ContentPresenter HorizontalAlignment="Center“ VerticalAlignment="Center"/>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

ContentPresenter is used in a custom control template when you want to define a placeholder that renders the content of the original control

Triggers

WPF enables you to take actions based on some criteria called Triggers

It allows you to dynamically change the appearance and/or behavior of your control when some event occurs.

Triggers are used to change the value of any given property, when certain conditions are satisfied.

Triggers are usually defined in a style or in the root of a document which are applied to that specific control. There are three types of triggers –

- Property Triggers
- Data Triggers
- Event Triggers

Property Triggers

In property triggers, when a change occurs in one property, it will bring either an immediate or an animated change in another property.

For example, you can use a property trigger to change the appearance of a button when the mouse hovers over the button.

The following example code shows how to change the foreground color of a button when mouse hovers over the button.

```
<Window.Resources>
    <Style x:Key="TriggerStyle" TargetType="Button">
        <Setter Property="Foreground" Value="Blue" />
        <Style.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
                <Setter Property="Foreground"
                        Value="Green" />
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>

<Grid>
    <Button Width="100" Height="70“
        Style="{StaticResource TriggerStyle}"
        Content="Trigger"/>
</Grid>
```

Data Triggers

A data trigger performs some actions when the bound data satisfies some conditions.

See example:

When the checkbox is checked, it will change its foreground color to red.

```
<Window.Resources>
        <Style x:Key="TriggerStyle" TargetType="TextBlock">
            <Style.Triggers>
                <DataTrigger
                    Binding = "{Binding ElementName = redColorCheckBox, Path = IsChecked}"
                    Value = "true">
                    <Setter Property = "TextBlock.Foreground" Value = "Red"/>
                    <Setter Property = "TextBlock.Cursor" Value = "Hand" />
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>

    <StackPanel HorizontalAlignment = "Center">
        <CheckBox x:Name = "redColorCheckBox“
         Content = "Set red as foreground color" Margin = "20"/>

        <TextBlock Name = "txtblock" VerticalAlignment = "Center“
            Style="{StaticResource TriggerStyle}" Text = "Event Trigger"
            FontSize = "24" Margin = "20">
      </TextBlock>
    </StackPanel>
```

Event Triggers

An event trigger performs some actions when a specific event is fired.

It is usually used to accomplish some animation on control.

In the following example, when the button click event is fired, it will expand the button width and height.

```
<Grid>
    <Button Content = "Click Me" Width = "60" Height = "30">
        <Button.Triggers>
            <EventTrigger RoutedEvent = "Button.Click">
                <EventTrigger.Actions>
                    <BeginStoryboard>
                        <Storyboard>
                            <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty ="Width"
                                Duration = "0:0:4">
                                <LinearDoubleKeyFrame Value = "60" KeyTime = "0:0:0"/>
                                <LinearDoubleKeyFrame Value = "120" KeyTime = "0:0:1"/>
                                <LinearDoubleKeyFrame Value = "200" KeyTime = "0:0:2"/>
                                <LinearDoubleKeyFrame Value = "300" KeyTime = "0:0:3"/>
                            </DoubleAnimationUsingKeyFrames>

                            <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Height"
                                Duration = "0:0:4">
                                <LinearDoubleKeyFrame Value = "30" KeyTime = "0:0:0"/>
                                <LinearDoubleKeyFrame Value = "40" KeyTime = "0:0:1"/>
                                <LinearDoubleKeyFrame Value = "80" KeyTime = "0:0:2"/>
                                <LinearDoubleKeyFrame Value = "150" KeyTime = "0:0:3"/>
                            </DoubleAnimationUsingKeyFrames>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger.Actions>
            </EventTrigger>
        </Button.Triggers>
    </Button>
</Grid>
```

.NET Properties vs WPF Dependency Properties

.NET Properties are standard properties supported by the CLR

- `public string Title { get; set; }`

WPF Dependency properties extends the functionality provided by standard .NET properties

Dependency properties provide the following functionality

- Resource Binding
- Data Binding
- Styles
- Animation, etc.

For E.g., Text property of TextBox is a dependency property

When creating custom UserControls or Control Templates, you can create your own dependency properties

Attached Property

An attached property is a XAML concept.

Attached properties enable extra property/value pairs to be set on a XAML element even though the element doesn't define those extra properties in its object model.

The extra properties are globally accessible.

```
<DockPanel>
    <TextBox DockPanel.Dock="Top">Enter text</TextBox>
</DockPanel>
```

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

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Label Grid.Row="0" Grid.Column="0" Content="Name:"/>
    <Label Grid.Row="1" Grid.Column="0" Content="Age:"/>
    <Label Grid.Row="2" Grid.Column="0" Content="Comment:"/>
</Grid>
```

WPF Commands

Commands in WPF are the modern way to handle user interface events E.g., when the user clicks on a button or a menu item.

In a modern user interface, it's typical for a function to be reachable from several places though, invoked by different user actions. E.g., For copy, you can select the **Edit->Copy**, or **RightClick->Copy** or press **Ctrl+C** on the keyboard or click on the **copy icon** on toolbar

Each of these actions needs to perform exact same piece of code

In a WinForms application, you would have to

- define an event for each of them and
- then call a common function.

With the above example, that would lead to at least four event handlers and some code to handle the keyboard shortcut. This is not an ideal situation.

WPF Commands

Commands allow you to define actions in one place and then refer to them from all your user interface controls.

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.

Another problem with WinForms was that you would be responsible for writing code that could disable user interface elements when the action was not available.

- For instance, the Copy menu, copy icon, Ctrl+C, etc. are all disabled if you have not selected anything.
- As a programmer you have to manually enable and disable the main menu item, the toolbar button and the context menu item each time text selection changed.

With WPF commands, this is centralized.

- With one method you decide whether or not a given command can be executed,
- then WPF automatically toggles all the subscribing interface elements on or off automatically.
- This makes it so much easier to create a responsive and dynamic application!

WPF Data Controls

WPF has a wide range of controls for displaying a list of data.

They come in several shapes and forms and vary in how complex they are and how much work they perform for you.

The simplest variant is the ItemsControl, which is pretty much just a markup-based loop

You need to apply all the styling and templating yourself

ItemsControl Example

```
<Grid Margin="10">
  <ItemsControl Name="icTodoList">
    <ItemsControl.ItemTemplate>
      <DataTemplate>
        <Grid Margin="0,0,0,5">
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="100" />
          </Grid.ColumnDefinitions>
          <TextBlock Text="{Binding Title}" />
          <ProgressBar Grid.Column="1"
          Minimum="0" Maximum="100"
          Value="{Binding Completion}" />
        </Grid>
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ItemsControl>
</Grid>
```

```
{
…
List<TodoItem> items = new List<TodoItem>();

items.Add(new TodoItem() { Title = "Complete this WPF tutorial", Completion = 45 });

items.Add(new TodoItem() { Title = "Learn C#", Completion = 80 });

items.Add(new TodoItem() { Title = "Wash the car", Completion = 0 });

icTodoList.ItemsSource = items;
…
}

public class TodoItem
{
   public string Title { get; set; }
   public int Completion { get; set; }
}
```

WPF Threading Model - Topics

Single-threaded Apartment

Multi-threaded Apartment

WPF Dispatcher

WPF Threading Model – An Apartment

An apartment is a concurrency boundary; it's an imaginary box drawn around objects and client threads that separates clients and objects from different threads.

All objects in the process are grouped into Apartments.

There are two types of apartments in Threads:

- Single-Threaded Apartments
- Multi-Threaded Apartments

STA & MTA

Single-threaded apartments contains only one thread.

- All objects in this apartment can receive method calls from only this thread.
- Single-threaded apartment needs a message queue to handle calls from other threads.
- When other threads calls an object in STA thread then the method call are queued in the message queue
- The STA object will receive a call from that message queue.
- This means that other threads cannot directly call an object in the STA thread

Multi-threaded apartments contains one or more threads.

- All objects in this apartment can receive calls from any thread.
- All objects are self responsible for maintaining the synchronization of their data.

WPF Dispatcher

A WPF application must start in single-threaded apartment thread.

- STA have a message queue to synchronize method calls within the apartment.
- Other threads outside the apartment can't update the objects directly.
- They must place their method call into the message queue to update the objects in STA.

Dispatcher owns the message queue for the STA thread.

When you execute a WPF application, it automatically create a new Dispatcher object.

Whenever you change any element on the UI or any UI control event executes, this is handled in the UI thread

UI thread queues the called method into the Dispatcher queue.

Dispatcher execute its message queue into the synchronous order.

Why we need Dispatcher?

WPF works with Dispatcher object behind the scenes

We don't need to work with Dispatcher when we are working on the UI thread.

When we create a new thread for offloading the work and want to update the UI from the other thread then we need Dispatcher.

Only Dispatcher can update the objects in the UI

If non-UI thread has some code or method which updates objects in the UI,

- Non-UI thread must pass the code block/method to the Dispatcher

Dispatcher adds the method into the message queue.

UI Thread which is monitoring the message queue will execute the code block synchronously

Invoke & BeginInvoke

Dispatcher provides two methods for registering method to execute into the message queue.

Invoke

- Takes an Action or Delegate and execute the method synchronously.

BeginInvoke

- BeginInvoke method take an Action or Delegate and executes the method asynchronously.

```
private void InvokeMethodExample()
{
    Dispatcher.Invoke(() =>
        {
            btn1.Content = "By Invoke";
        });
}
```

```
private void BeginInvokeExample()
{
    DispatcherOperation op = Dispatcher.BeginInvoke((Action)(() => {
            btn1.Content = "By BeginInvoke";
        }));
}
```

UserControls in WPF

User controls, in WPF is represented by the UserControl class,

It is the concept of grouping markup and code into a reusable container

This helps if you want the same interface, with the same functionality, to be used in several different places and even across several applications.

A user control acts much like a WPF Window

- an area where you can place other controls, and
- then a Code-behind file where you can interact with these controls.
- The file that contains the user control also ends with .xaml,
- and the Code-behind ends with .xaml.cs - just like a Window.

The starting markup looks a bit different though

UserControl in WPF

```
<UserControl x:Class="WpfTutorialSamples.User_Controls.LimitedInputUserControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
   <Grid>

   </Grid>
</UserControl>
```

1. In Xaml, a root UserControl element instead of the Window element
2. DesignHeight and DesignWidth properties instead of Height and Width
   - controls the size of the user control in design-time
   - in runtime, the size will be decided by the container that holds the user control
3. In code behind, inherits *UserControl* instead of *Window*.