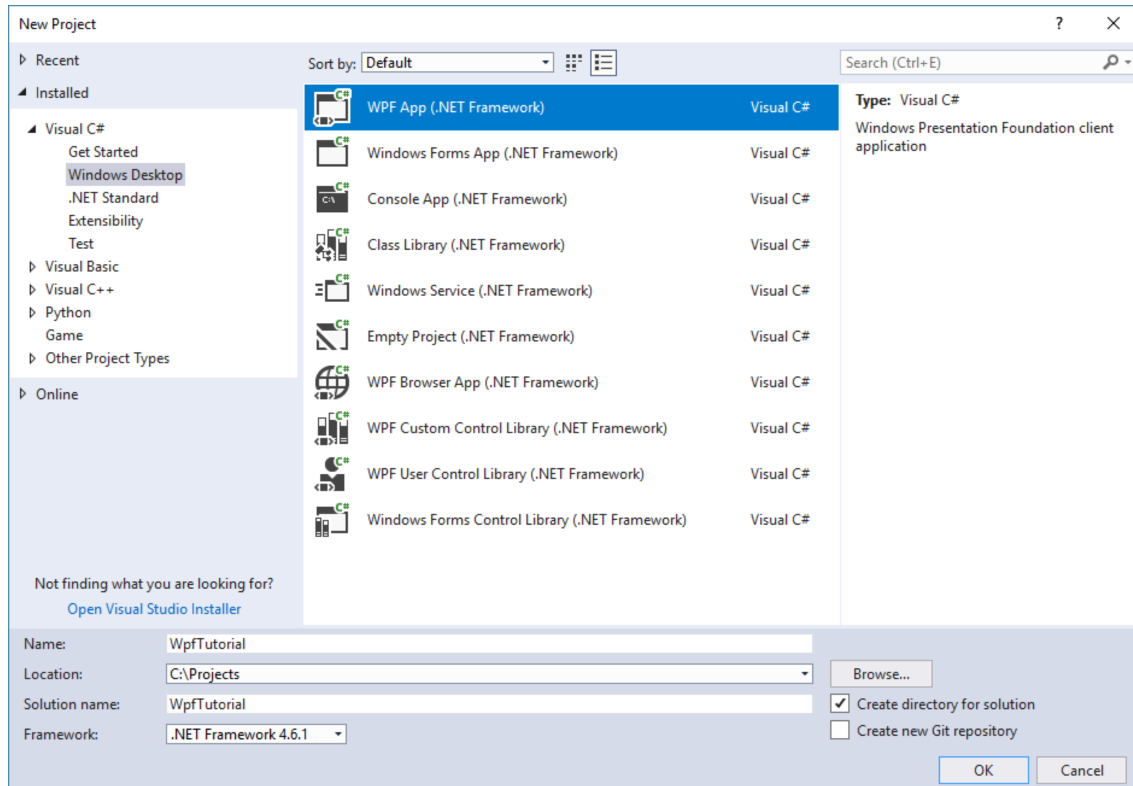
Windows Presentation Foundation (Microsoft Visual Studio 2017) - Tutorial

1 Introducing WPF

1.1 Creating project

Create new Visual Studio project with type: Visual C# → Windows Desktop → WPF App.



1.2 Getting familiar with WPF project

1.2.1 What is XAML ['zæməl]?

In WPF you create your Graphics User Interface using XML-based language called XAML. Purpose of that was making the collaboration between programmers team and graphics designers easier. Every element in XAML maps to an instance of a .NET class. For example the element `<Button>` instructs WPF to create a Button object. You can nest one element inside another and have flexibility to decide how to handle the situation. You can set the properties through attributes, but in more complex cases you will use nested tags with a special syntax. `Note:` It's important to understand that WPF doesn't require XAML. You can create your UI with C# code, using all WPF features (like bindings etc), but then you lose collaboration benefits.

1.2.2 App.xaml file

This is starting point for the application. Notice, that you can't find the Main() entry points. It's automatically generated from this file. You can specify your start window by changing the StartupUri property. You can also add attributes defining some global event handlers, try it out - Visual Studio will help:

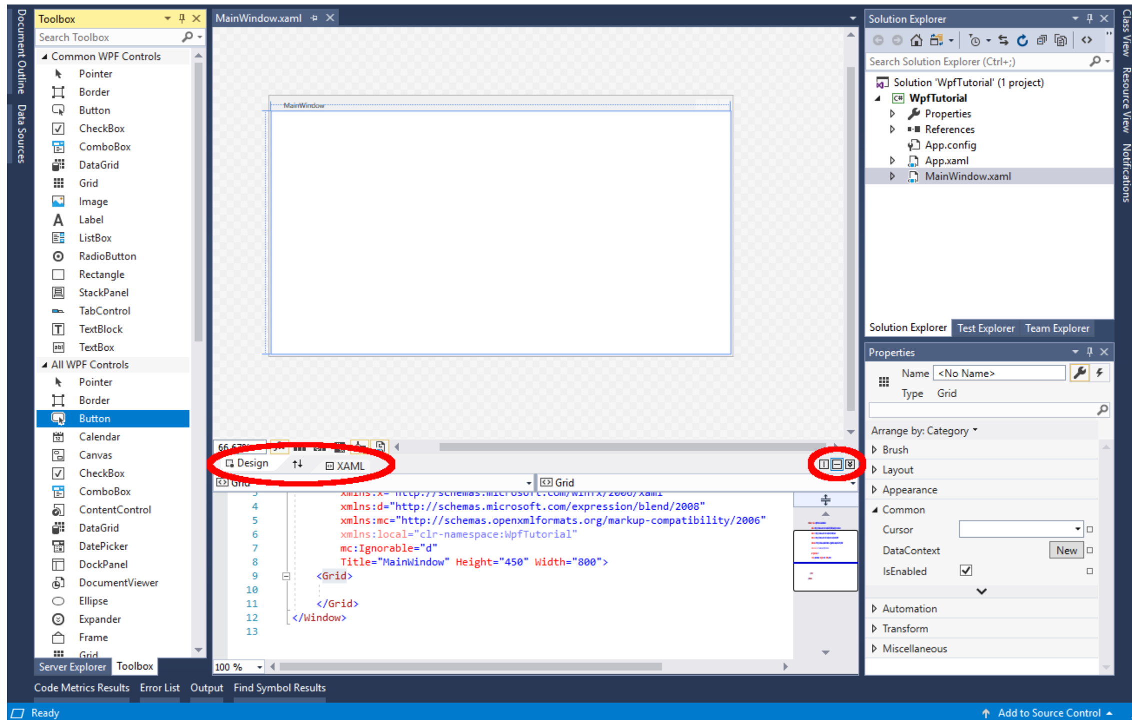
```
<Application x:Class="WpfTutorial.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfTutorial"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

1.2.3 MainWindow.xaml file

MainWindow.xaml file represents a new blank window.

1.2.4 Designer/XAML View



You can edit your UI by editing XAML code or using Designer. You can drag and drop items from the **Toolbox** tab onto the Designer and edit properties of selected controls in the **Properties** tab. Notice, that adding controls in Designer automatically changes your XAML.

1.2.5 XAML Code

Open and examine XAML Code from MainWindow.xaml file:

```
<Window x:Class="WpfTutorial.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:WpfTutorial"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

1.2.6 Window class

First line of the MainWindow.xaml document is:

```
<Window x:Class="WpfTutorial.MainWindow"
```

It tells which class contains implementation of your window.

1.2.7 XAML namespaces

Of course, it's not enough to just specify class name. You must also provide .NET namespace where the window class is located. By default few namespaces are defined:

```
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

The core WPF namespace. It contains all the WPF classes, including the controls. This namespace is declared without any prefix, so it becomes the default namespace for the entire document - every element is expected to be defined in this namespace unless you specify different one:

```
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

This namespace includes various utility features. This namespace is mapped to the prefix **x**. That means you can apply it by placing the namespace prefix before the element or attribute name (like `<x:Element>`):

```
4         xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

Namespace **d** enables designer-only attributes in your code. These designer attributes only affect the design aspects of how XAML behaves. The designer attributes are ignored when the same XAML is loaded by the XAML parser at application run-time:

```
5         xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

Namespace **mc** indicates and supports a markup compatibility mode for reading XAML. Typically, the d: prefix is associated with the attribute mc:Ignorable at line 7. This technique enables run time XAML parsers to ignore the design attributes, as described previously:

```
6         xmlns:local="clr-namespace:WpfTutorial"
```

Namespace **local** refers to the default namespace of your project. It enables you to reference controls created by you in **WpfTutorial** namespace. For example when you create new class (i.e. MyControl) and want to add it to the XAML document, you can place it in the XAML document by writing `<local:MyControl>`.

You can also add your own namespaces. You can do it by adding the following line:

```
          xmlns:my_namespace="clr-namespace:WpfTutorial.MyControlNamespace"
```

1.2.8 Properties

Properties of a window are specified by attributes:

```
8         Title="MainWindow" Height="450" Width="800"
```

1.2.9 Layout

A window control (like all content controls) can hold only one child control. To place more than one element you need to nest a container and then add other elements to that container.

```
9     <Grid>
10
11    </Grid>
```

As you can see, by default Visual Studio places **Grid** container inside a **Window**. Other containers are also available: **StackPanel**, **WrapPanel**, **DockPanel**, **UniformGrid** and **Canvas**.

1.2.10 MainWindow.xaml.cs file

This file contains event handles and code for defining window logic.

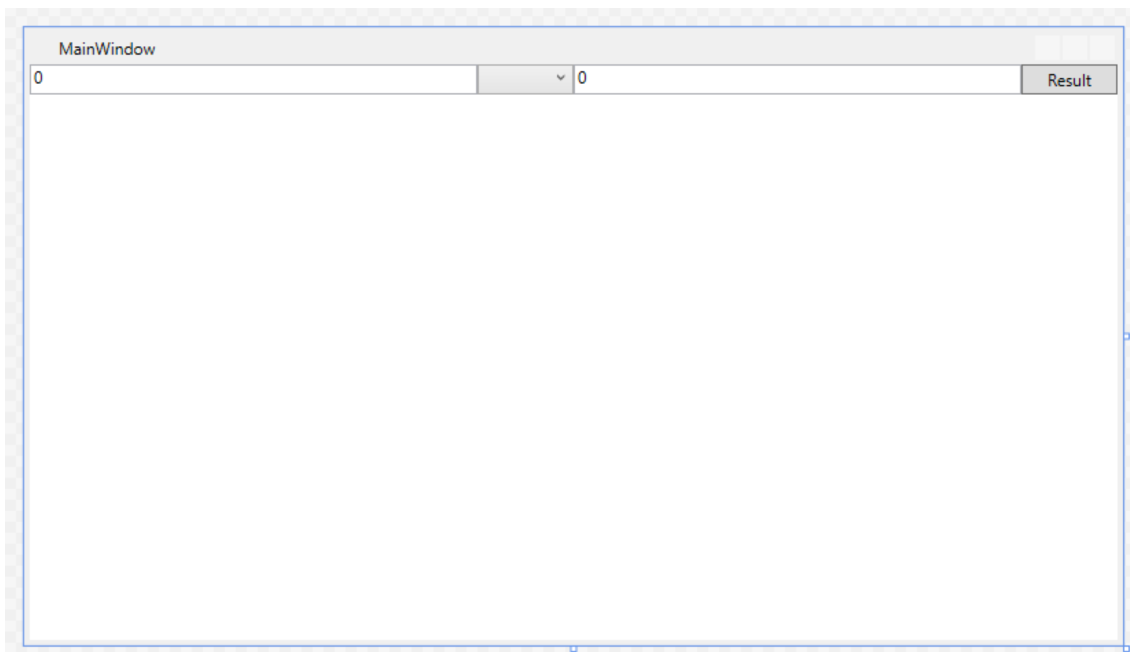
1.3 First WPF application - simple calculator

Place the following code in MainWindow.xaml between `<Window></Window>` tags:

```
9   <StackPanel Orientation="Vertical">
10    <Grid>
11      <Grid.ColumnDefinitions>
12        <ColumnDefinition/>
13        <ColumnDefinition Width="70" />
```

```
      <ColumnDefinition/>
      <ColumnDefinition Width="70" />
    </Grid.ColumnDefinitions>
    <TextBox Name="firstTextBox" Text="0" />
    <ComboBox Name="operationComboBox" Grid.Column="1">
      <ComboBox.Items>
        <ComboBoxItem>+</ComboBoxItem>
        <ComboBoxItem>-</ComboBoxItem>
        <ComboBoxItem>*</ComboBoxItem>
        <ComboBoxItem>/</ComboBoxItem>
      </ComboBox.Items>
    </ComboBox>
    <TextBox Name="secondTextBox" Grid.Column="2" Text="0" />
    <Button Name="resultButton" Grid.Column="3" Content="Result" />
  </Grid>
  <TextBlock Name="resultBlock" HorizontalAlignment="Center"
             FontSize="16" FontWeight="Bold" />
</StackPanel>
```

Control's Name property allows you to reference a control in C# code-behind. Open Designer and see what's changed.



1.3.1 Grid panel

This code is responsible for creating columns for Grid. Similarly, you can create Rows using RowDefinition:

```
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition Height="70" />
      <RowDefinition/>
      <RowDefinition Height="70" />
    </Grid.RowDefinitions>
```

Placing control in appropriate cell is done by Attached Properties - Grid.Row and Grid.Column:

```
    <ComboBox Name="operationComboBox" Grid.Column="1">
```

Columns and Rows indices start from 0, which is a default value, so every control without these properties is placed in the (0,0) cell.

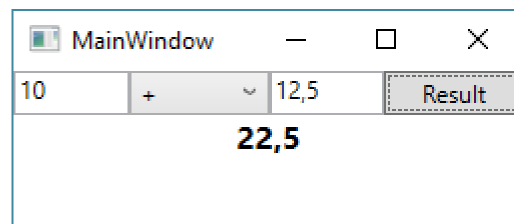
1.3.2 Event handlers

Double click **Result** button in the Designer. The Code-behind file is opened, and new method is generated. Put there the following code:

```
28 private void ResultButton_Click(object sender, RoutedEventArgs e)
29 {
30     double first, second;
31     if (!double.TryParse(firstTextBox.Text, out first) ||
32         !double.TryParse(secondTextBox.Text, out second))
33     {
34         MessageBox.Show("Incorrect input");
35         return;
36     }
37     switch (operationComboBox.Text)
38     {
39         case "+":
40         resultBlock.Text = String.Format("{0}", first + second);
41         break;
42         case "-":
43         resultBlock.Text = String.Format("{0}", first - second);
44         break;
45         case "*":
46         resultBlock.Text = String.Format("{0}", first * second);
47         break;
48         case "/":
49         if (second == 0)
50         {
51             MessageBox.Show("Division by zero");
52             return;
53         }
54         resultBlock.Text = String.Format("{0}", first / second);
55         break;
56     }
57 }
```

1.3.3 Running application

Run the application and see how it works. Notice how layout behaves when resizing the window.



2 WPF Color picker

We will create an application that displays color specified by RGB values.

2.1 New WPF project

Create new WPF Application Project. Name it "WpfColorPicker".

2.2 Application layout

To create application layout, place this code in MainWindow.xaml between `<Window></Window>` tags:

```
9     <Grid>
10      <Grid.RowDefinitions>
11        <RowDefinition Height="Auto" />
12        <RowDefinition Height="Auto" />
13        <RowDefinition Height="Auto" />
14        <RowDefinition Height="*" />
15      </Grid.RowDefinitions>
16      <Grid.ColumnDefinitions>
17        <ColumnDefinition Width="80" />
18        <ColumnDefinition Width="*" />
19        <ColumnDefinition Width="80" />
20      </Grid.ColumnDefinitions>
21
22    </Grid>
```

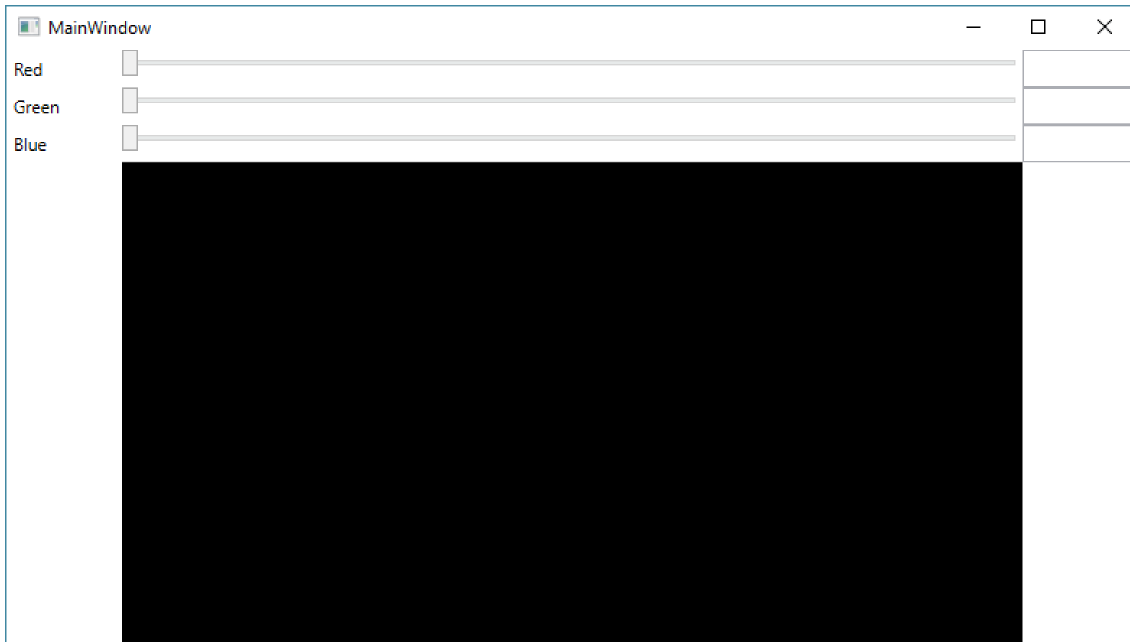
Our Grid has 4 rows and 3 columns.

2.3 Window Controls

Now we will add controls to the window. Add following code before `</Grid>`:

```
22      <Label>Red</Label>
23      <Slider Name="redSlider" Grid.Column="1" Value="0"/>
24      <TextBox Grid.Row="0" Grid.Column="2" />
25
26      <Label Grid.Row="1">Green</Label>
27      <Slider Name="greenSlider" Grid.Row="1" Grid.Column="1" Value="0"/>
28      <TextBox Grid.Row="1" Grid.Column="2" />
29
30      <Label Grid.Row="2">Blue</Label>
31      <Slider Name="blueSlider" Grid.Row="2" Grid.Column="1" Value="0" />
32      <TextBox Grid.Row="2" Grid.Column="2"/>
33
34      <Rectangle Grid.Row="3" Grid.Column="1" Fill="Black"/>
```

Compile and run your application.



2.4 Object Resources

Sometimes we want to create object (e.g. a brush) and use it on many controls in a window (or event an application). We can use "resources". WPF framework allows you to define resources in XAML code and then reuse it easily. What is more, it makes code much cleaner. To define new Brush as a resource add following code after `<Window>` tag and before `<Grid>`:

```
9   <Window.Resources>
10    <SolidColorBrush x:Key="windowBackgroundBrush" Color="LightBlue"/>
11  </Window.Resources>
```

Notice that resource has a key, which is used to reference it throughout the XAML file. Set Grid's background as newly created brush. To do so just change `<Grid>` line into:

```
13  <Grid Background="{StaticResource windowBackgroundBrush}">
```

2.5 Styles

Now, let's change the look of controls in our window. We can do it by changing the properties of each control. In most cases, all controls of the same type should look similarly and we'd like to avoid copying code. Here's where we can use styles. They allow us to define values for properties, and apply them to many controls. Styles are usually defined as resources.

Add these styles to Window Resources:

```
12    <Style x:Key="colorLabelStyle" TargetType="{x:Type Label}">
13      <Setter Property="FontSize" Value="16" />
14      <Setter Property="FontFamily" Value="Arial" />
15      <Setter Property="FontWeight" Value="Bold" />
16      <Setter Property="Margin" Value="0,3" />
17    </Style>
18
19    <Style x:Key="colorSliderStyle" TargetType="{x:Type Slider}">
20      <Setter Property="Minimum" Value="0" />
21      <Setter Property="Maximum" Value="255" />
22      <Setter Property="SmallChange" Value="1" />
23      <Setter Property="TickFrequency" Value="1" />
24      <Setter Property="IsSnapToTickEnabled" Value="True" />
25      <Setter Property="Margin" Value="5,0" />
26    </Style>
27
28    <Style TargetType="{x:Type TextBox}">
29      <Setter Property="BorderThickness" Value="2" />
```

```
30        <Setter Property="BorderBrush" Value="White" />
31        <Setter Property="Background">
32          <Setter.Value>
33            <LinearGradientBrush StartPoint="0.5, 0" EndPoint="0.5, 1">
34              <GradientStop Offset="0" Color="Blue"></GradientStop>
35              <GradientStop Offset="0.5" Color="DarkBlue"></GradientStop>
36              <GradientStop Offset="1" Color="Blue"></GradientStop>
37            </LinearGradientBrush>
38          </Setter.Value>
39        </Setter>
40        <Setter Property="Foreground" Value="White" />
41        <Setter Property="Margin" Value="3" />
42        <Setter Property="FontWeight" Value="Bold" />
43      </Style>
```

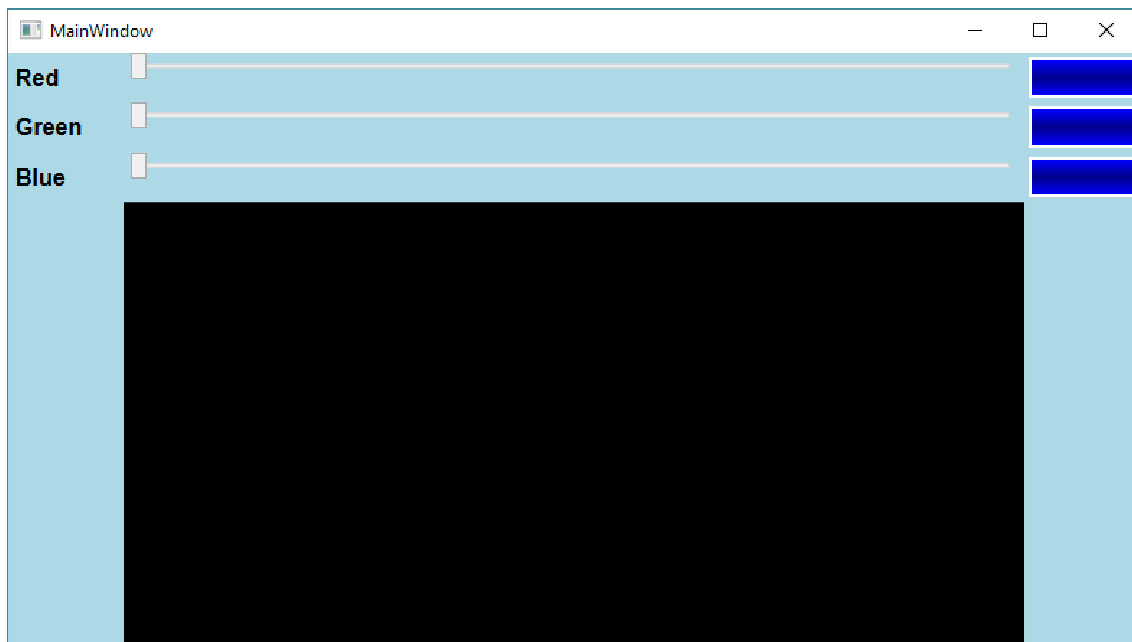
In order to apply styles to controls, just set Style's property value:

```
59      <Label Style="{StaticResource colorLabelStyle}" ... />
60      <Slider Style="{StaticResource colorSliderStyle}" ... />
```

Do it for each control of Label and Slider type.

Notice, that Style for a TextBox doesn't have a key. Styles without a key are automatically applied to all elements of a certain type.

Run your application to see the effects.



2.6 Binding

We'd like any changes in slider position to automatically update the text in the TextBox and vice-versa. WPF provides powerful tool for linking properties of two controls - **Data Binding**. It allows you to extract information from a source object and push it into target object - with little of code. It also allows two-way binding. Important matter is that the destination property must be **Dependency Property**, which is a special WPF property. Fortunately, almost all control properties are dependency properties.

In order to add binding to a TextBox change your first TextBox definition:

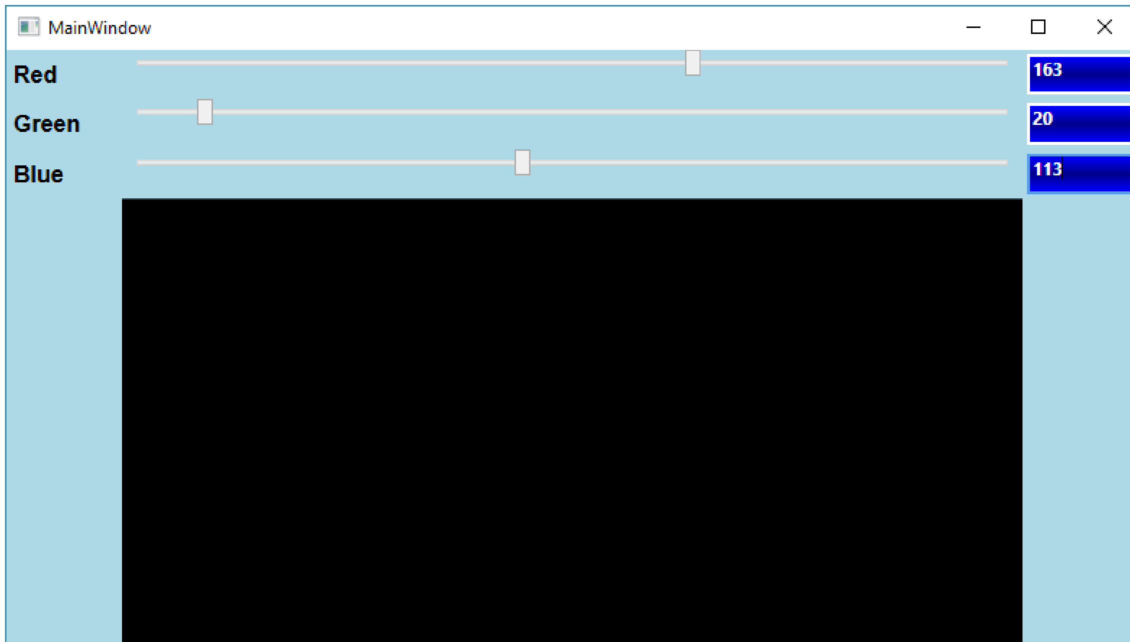
```
61      <TextBox Grid.Row="0" Grid.Column="2"
62           Text="{Binding ElementName=redSlider,Path=Value,Mode=TwoWay}" />
```

Do the same for second and third textbox fields:

```
66      <TextBox Grid.Row="1" Grid.Column="2"
67           Text="{Binding ElementName=greenSlider,Path=Value,Mode=TwoWay}" />
```

```
71      <TextBox Grid.Row="2" Grid.Column="2"
72           Text="{Binding ElementName=blueSlider,Path=Value,Mode=TwoWay}" />
```

And that's all. Run your application and move the sliders. Try to edit TextBox's text too.



You can also specify UpdateSourceTrigger property with PropertyChanged value set to see results immediately when changing the value - not when the focus is lost:

```
61      <TextBox Grid.Row="0" Grid.Column="2"
62           Text="{Binding ElementName=redSlider,Path=Value,Mode=TwoWay,
        UpdateSourceTrigger=PropertyChanged}" />
```

Try this out for all TextBox fields in the application.

2.7 Multibinding and Converters

The last thing is changing rectangle color according to the values of the sliders. We can do this using event handlers. We can also use binding, in this case: **Multibinding**. Before we can create multibinding, we must create a **Converter**. It's optional for single binding and required for multibinding. It allows you to convert data between inconsistent types. For example, you could bind control's Visibility to CheckBox. CheckBox's property "IsChecked" is of type bool, and Visibility is System.Windows.Visibility, so the only way to bind them is to write a converter that converts bool to Visibility type. Converters are classes that implement **IValueConverter** interface for single binding, and **IMultiValueConverter** for multibinding.

1) Add a new class to the project and name it "SlidersToColorConverter". Make it implement **IMultiValueConverter** and fill Convert method:

```
1  using System;
2  using System.Globalization;
3  using System.Windows.Data;
4  using System.Windows.Media;
5
6  namespace WpfColorPicker
7  {
8      class SlidersToColorConverter : IMultiValueConverter
9      {
```

```
10          public object Convert(object[] values, Type targetType,
11              object parameter, CultureInfo culture)
12          {
13              var red = System.Convert.ToByte(values[0]);
14              var green = System.Convert.ToByte(values[1]);
15              var blue = System.Convert.ToByte(values[2]);
16
17              return new SolidColorBrush(
18                  Color.FromArgb(255, red, green, blue));
19          }
20
21          public object[] ConvertBack(object value, Type[] targetTypes,
22              object parameter, CultureInfo culture)
23          {
24              throw new NotImplementedException();
25          }
26      }
27  }
```

As you see we gather 3 values (red, green and blue coordinates) and create a new Brush. Notice, we don't need to implement the ConvertBack method, as our binding will be one-way.

2) Open MainWindow.xaml.

3) Add converter to `<Window.Resources>`:

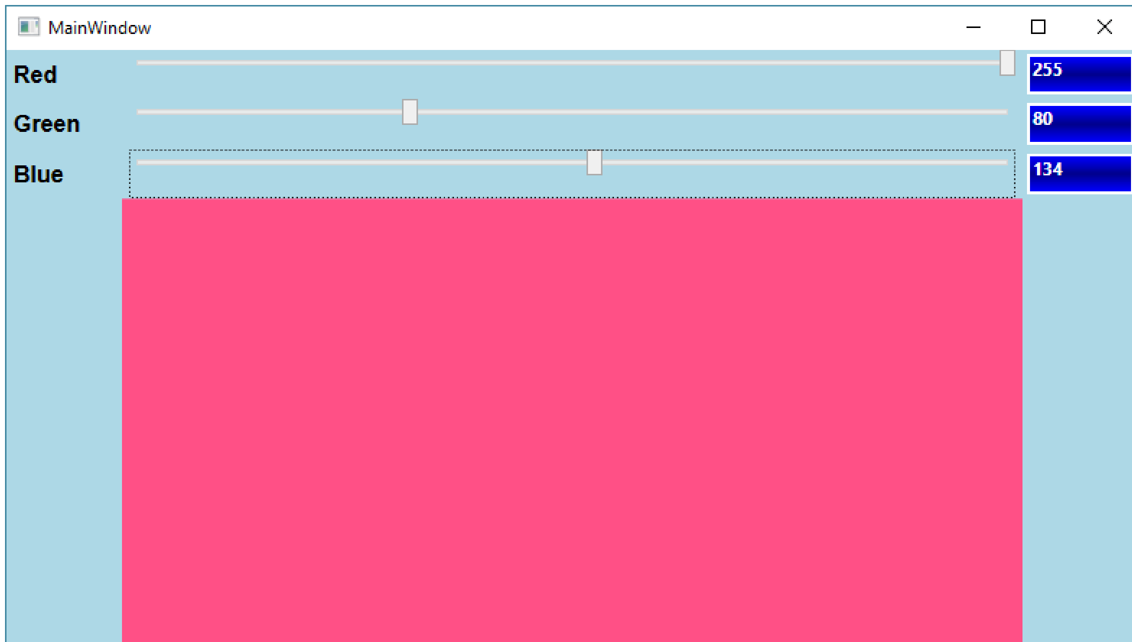
```
10      <local:SlidersToColorConverter x:Key="sliderToColorConverter" />
```

4) Create Multibinding.

   Rewrite your `<Rectangle>` element:

```
76      <Rectangle Grid.Row="3" Grid.Column="1">
77        <Rectangle.Fill>
78          <MultiBinding Converter="{StaticResource sliderToColorConverter}"
79                        Mode="OneWay">
80            <Binding ElementName="redSlider" Path="Value" />
81            <Binding ElementName="greenSlider" Path="Value" />
82            <Binding ElementName="blueSlider" Path="Value" />
83          </MultiBinding>
84        </Rectangle.Fill>
85      </Rectangle>
```

5) Compile and run the project. Try moving the sliders, see what happens.


MainWindow
Red
Green
Blue
255
80
134

3 WPF Color preview

Now, let's write an application that will display all colors from System.Windows.Media.Colors in a ListView. We want each item in the list to consist of a filled rectangle, color name and it's components.

3.1 New WPF project

Create new WPF project. Name it "WpfColorsPreview".

3.2 Color info class

Add new class to the project. Name it "ColorInfo":

```
1  using System.Windows.Media;
2
3  namespace WpfColorsPreview
4  {
5      class ColorInfo
6      {
7          public string Name { get; set; }
8          public string RgbInfo { get; set; }
9          public Color Rgb { get; set; }
10     }
11 }
```

This class will contain information about color. It holds its name, string with RGB components, and a color reference.

3.3 Binding to DataContext

Open MainWindow.xaml and add new event handler to the `<Window>` element:

```
<Window ...
        Loaded="Window_Loaded">
```

Write following code in the code-behind file:

```
29 private void Window_Loaded(object sender, RoutedEventArgs e)
30 {
31     var props = typeof(Colors).GetProperties(BindingFlags.Static |
32                                              BindingFlags.Public);
33     var colorInfos = props.Select(prop =>
34     {
35         var color = (Color)prop.GetValue(null, null);
36         return new ColorInfo()
37         {
38             Name = prop.Name,
39             Rgb = color,
40             RgbInfo = $"R:{color.R} G:{color.G}, B:{color.B}"
41         };
42     });
43     this.DataContext = colorInfos;
44 }
```

This code simply reads all public and static properties from Colors class and transforms each one into a new ColorInfo object. The most interesting is the last line. It assigns the result collection of ColorInfos to the DataContext property of a window. The DataContext property is used by bindings mechanism. When you don't specify source of your binding, WPF examines the DataContext property of each element in the tree starting at the current element and uses the first one that isn't null. This is extremely useful if you need to bind several properties of the same object to different elements because you can set the DataContext rather than referencing the object explicitly.

Now add ListView control between `<Grid></Grid>` tags:

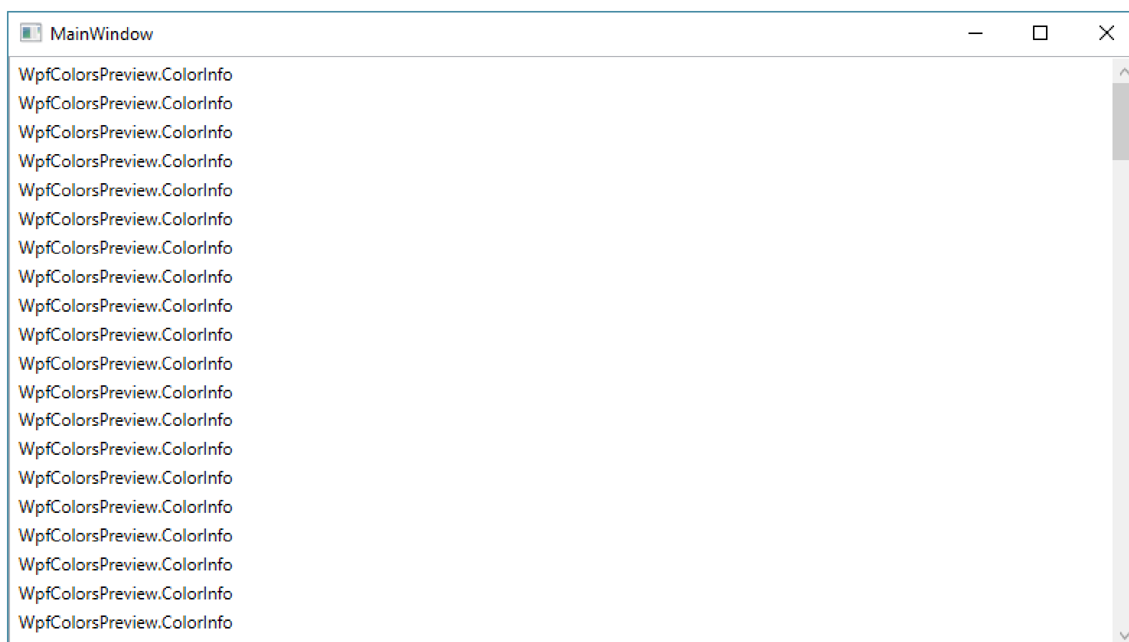
```
    <Grid>
        <ListView ItemsSource="{Binding}" />
    </Grid>
```

This binding doesn't have source specified, so it searches up the controls tree looking for non-null DataContext. It finds one in the Window control. Because we didn't specify any path, the whole DataContext object will be bound.

Note: We could have the same result by naming the ListView and assigning it's ItemSource to colorInfos list.

3.4 Running application

Now compile and run your application. You will notice that there are some strange elements in the list.



That's because WPF doesn't know how to display ColorInfo object, so it uses default ToString method instead throwing an exception. Thus, we need to create **DataTemplate** for the list.

3.5 Data Template

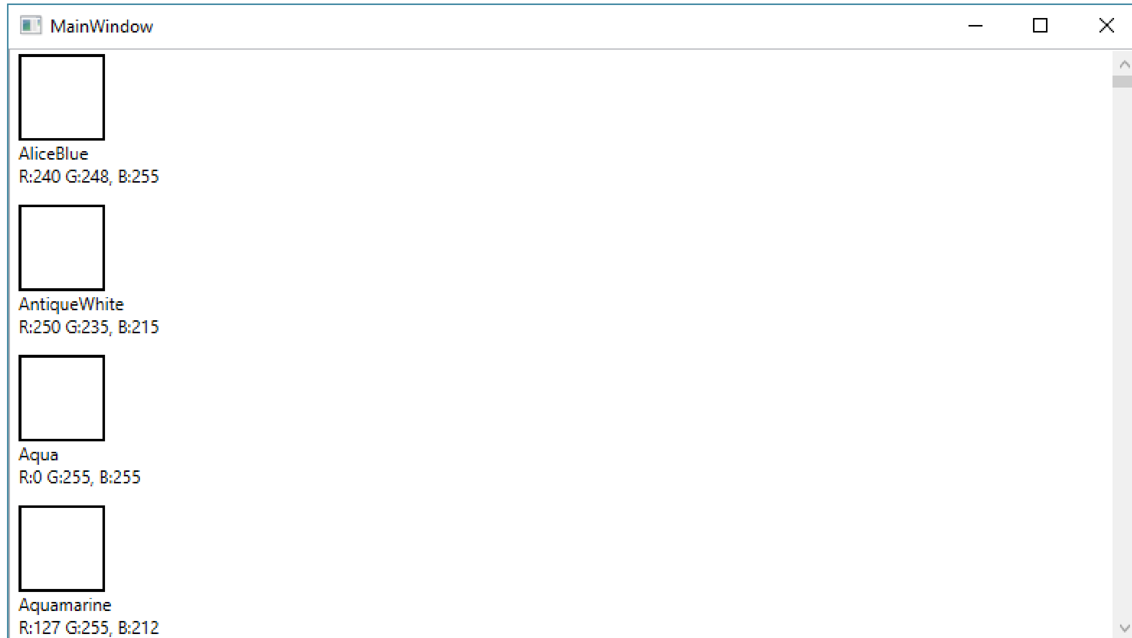
Data template defines how bounded objects should be displayed in a list. To define DataTemplate for a ListView replace its declaration with the following code:

```
    <ListView ItemsSource="{Binding}">
      <ListView.ItemTemplate>
        <DataTemplate>
          <StackPanel Orientation="Vertical" Width="120" Height="100">
            <Border Width="60" Height="60" HorizontalAlignment="Left"
                    BorderBrush="Black" BorderThickness="2">
              <Rectangle HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
                         Fill="{Binding Rgb}" />
            </Border>
            <TextBlock Text="{Binding Name}" />
            <TextBlock Text="{Binding RgbInfo}" />
          </StackPanel>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
```

We display 3 parts of ColorInfo: RGB color is displayed as a filled rectangle with a border, and color name with RgbInfo are presented in a TextBlocks. Note that in DataTemplate we just

simply bind to the properties of ColorInfo class.

When you start your application you will notice that Name and RgbInfo property are displayed properly, but all Rectangles are transparent. That's because we provided Color object for Rectangle.Fill, but it requires a Brush. To fix that we will need to create a converter.



3.6 Converter

1) Add new class to the project. Name it "ColorToBrushConverter":

```
using System;
using System.Globalization;
using System.Windows.Data;
using System.Windows.Media;

namespace WpfColorsPreview
{
    class ColorToBrushConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
            object parameter, CultureInfo culture)
        {
            if (!(value is Color))
                return null;

            return new SolidColorBrush((Color)value);
        }

        public object ConvertBack(object value, Type targetType,
            object parameter, CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

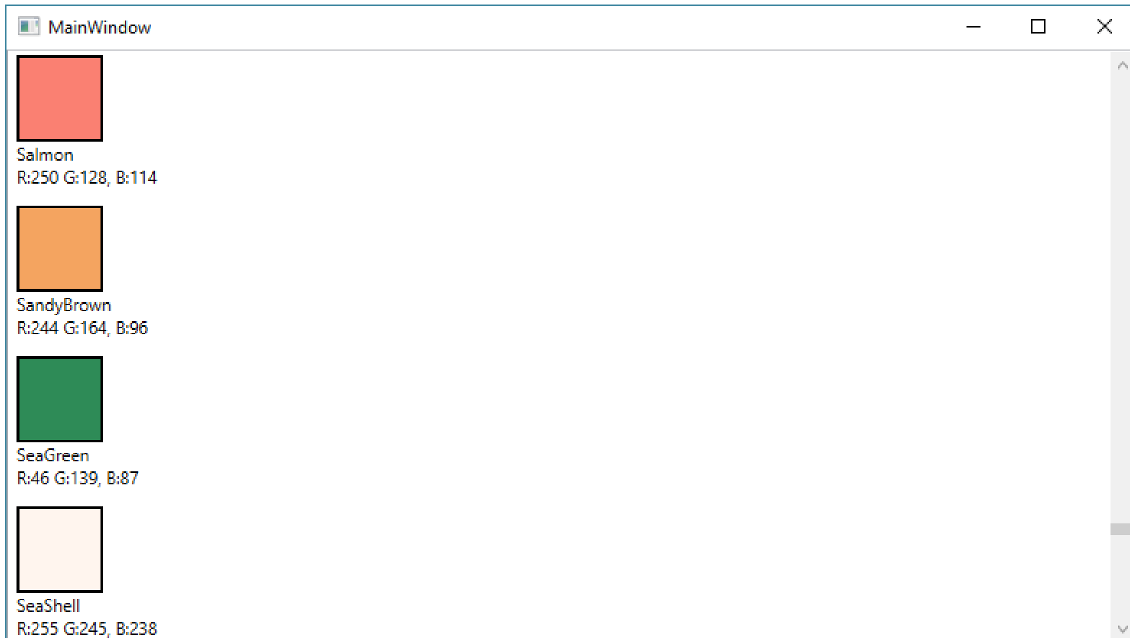
2) Add new converter to <Window.Resources>:

```
  <Window.Resources>
    <local:ColorToBrushConverter x:Key="colorToBrushConverter" />
  </Window.Resources>
```

3) Add converter as one of the binding properties:

```
22         Fill="{Binding Rgb, Converter={StaticResource colorToBrushConverter}}" />
```

4) Compile and run the application. You will see all colors displayed alongside with their names and RGB components.



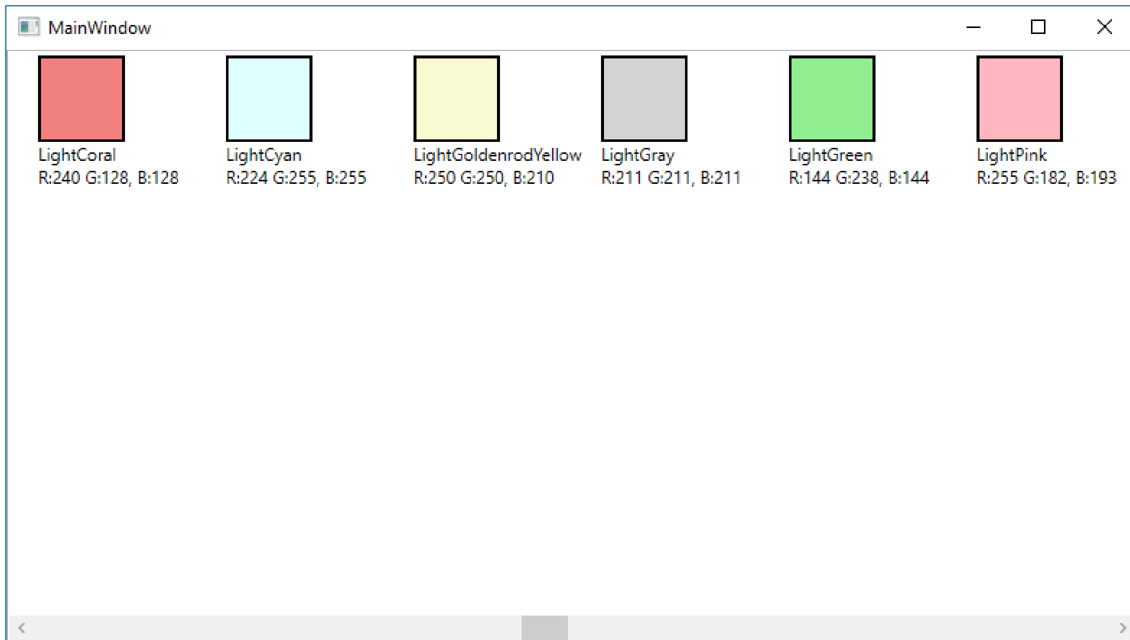
3.7 ListView PanelTemplate and Relative Binding

Items in our ListView are placed in one column. That's because default panel for ListView control is StackPanel which placing items vertically. Let's change it to WrapPanel.

1) Change ItemsPanelTemplate. Add code before `</ListView>` tag:

```
30         <ListView.ItemsPanel>
31           <ItemsPanelTemplate>
32             <WrapPanel />
33           </ItemsPanelTemplate>
34         </ListView.ItemsPanel>
```

2) Run the application. Unfortunately, now items are placed in one row. To adjust that we need to set WrapPanel's width to be equal to ListView control width.



3) Set panel width. Change `<WrapPanel/>` to the following:

```
32            <WrapPanel Width="{Binding ActualWidth,
33                   RelativeSource={RelativeSource AncestorType=ListView}}"/>
```

We create binding between ListView.ActualWidth and Width of a WrapPanel. We used ActualWidth because it contains rendered size of a control. More interesting part is the binding source type. It's third (beside Source and DataContext) kind of binding source: RelativeSource. It allows you to point to a source object based on its relation to the target object. You can bind an element to itself or to the parent element that meets some condition (in our case the condition is the ListView type).

4) Run application and try resizing window.

4 Bonus: 3D Moview Cube

4.1 New WPF project

Create new WPF project. Name it "Wpf3DCube".

4.2 Cube mesh

Add following code to the <Window.Resources>:

```
9   <Window.Resources>
10    <MeshGeometry3D x:Key="Cube"
11      Positions="-50,-50,-50  50,-50,-50  50, 50,-50
12               -50,-50,-50  50, 50,-50 -50, 50,-50
13                50,-50,-50  50,-50, 50  50, 50,-50
14                50, 50,-50, 50,-50, 50, 50, 50, 50
15               -50,-50,-50  50,-50,-50  50,-50, 50
16               -50,-50,-50 -50,-50, 50  50,-50, 50
17               -50, 50,-50  50, 50,-50 -50, 50, 50
18                50, 50,-50 -50, 50, 50  50, 50, 50
19               -50,-50,-50 -50,-50, 50 -50, 50,-50
20               -50,-50, 50 -50, 50,-50 -50, 50, 50
21               -50,-50, 50  50,-50, 50  50, 50, 50
22               -50,-50, 50  50, 50, 50 -50, 50, 50"
23      TriangleIndices="2 1 0 5 4 3 8 7 6 11 10 9 12 13 14 17 16 15 20 19 18
24                       21 22 23 24 25 26 29 28 27 30 31 32 33 34 35"
25      TextureCoordinates="1,1 0,1 0,0
26                          1,1 0,0 1,0
27                          1,1 0,1 1,0
28                          1,0 0,1 0,0
29                          1,1 0,1 0,0
30                          1,1 1,0 0,0
31                          1,1 0,1 1,0
32                          0,1 1,0 0,0
33                          0,1 1,1 0,0
34                          1,1 0,0 1,0
35                          0,1 1,1 1,0
36                          0,1 1,0 0,0" />
37  </Window.Resources>
```

This code defines a 3D cube mesh. Position properties are 3D coordinates of vertices. Triangle indices build triangles from vertices. Texture coordinates will be used in mapping our Media element control on the Cube.

4.3 Viewport3D

To display 3D data we use a **Viewport3D** control. It allows you to define 3D camera, lights and geometry. Place this code between <Grid></Grid> tags:

```
40    <Viewport3D>
41      <Viewport3D.Camera>
42        <PerspectiveCamera Position="100, 200, -300" LookDirection="-1, -2, 3"
43                           FieldOfView="60" />
44      </Viewport3D.Camera>
45    </Viewport3D>
```

We define camera as a perspective camera.

Next we must add some lights to the scene. Otherwise everything would be black.

Place following code (which adds 2 lights - ambient and directional) before </Viewport3D> tag:

```
46      <ModelVisual3D>
47        <ModelVisual3D.Content>
48          <Model3DGroup>
49            <AmbientLight Color="DarkGray" />
50            <DirectionalLight Color="White" Direction="0, -2, 1" />
51          </Model3DGroup>
52        </ModelVisual3D.Content>
53      </ModelVisual3D>
```

4.4 Using 3D Mesh Geometry

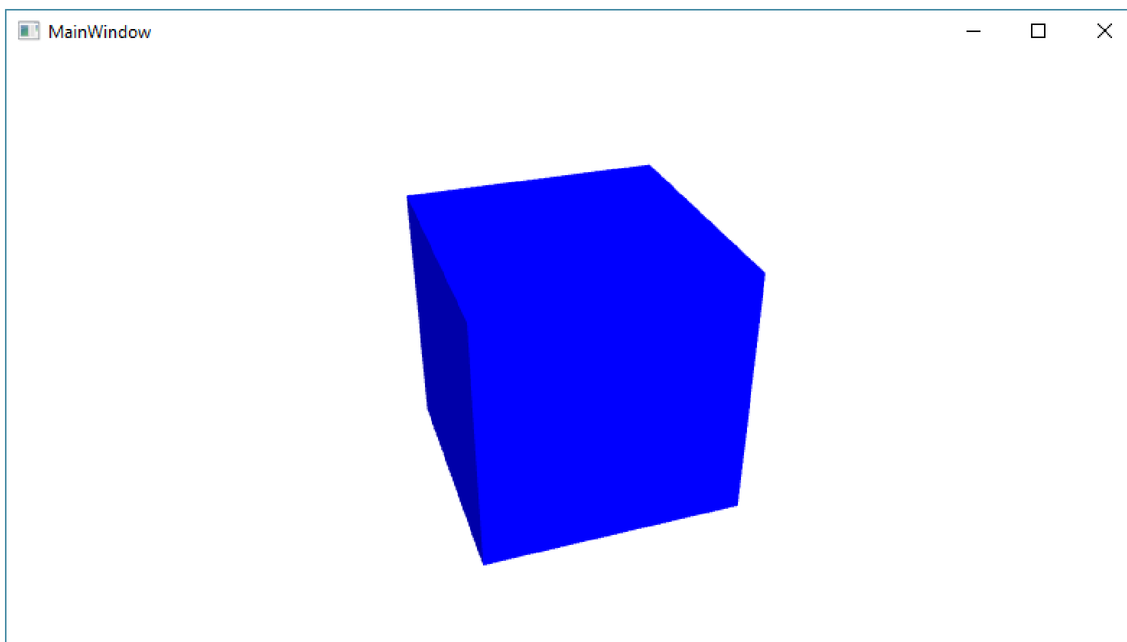
It's time to add our geometry to the scene. We can do it using new <ModelVisual3D> object, but instead we will use **Viewport2DVisual3D** as we want to display a movie on a cube. Viewport2DVisual3D allows placing WPF controls (like button) on a 3D object. Add following code before </Viewport3D> tag:

```
55        <Viewport2DVisual3D Geometry="{StaticResource Cube}">
56          <Viewport2DVisual3D.Material>
57            <DiffuseMaterial Brush="Blue" />
58          </Viewport2DVisual3D.Material>
59        </Viewport2DVisual3D>
```

As you can see we load our geometry from resource. We also have to define material for geometry: in our case we will use blue brush.

4.5 Running application

Run your application. You should see a blue cube.



4.6 Transformation, Animation and Trigger

Now, let's add animation to our cube. Animations allow us to change control properties over time.

Before we create animation we must add some transformation to the scene. Add this code after </Viewport2DVisual3D.Material>:

```
60          <Viewport2DVisual3D.Transform>
61            <Transform3DGroup>
62              <RotateTransform3D>
63                <RotateTransform3D.Rotation>
64                  <AxisAngleRotation3D x:Name="rotation" Axis="3 2 1" Angle="0" />
65                </RotateTransform3D.Rotation>
66              </RotateTransform3D>
67            </Transform3DGroup>
68          </Viewport2DVisual3D.Transform>
```

You can see that we added a rotate transformation. We are rotating our scene by an angle of 0 degrees, which doesn't change anything, but it's a good starting point for animation.

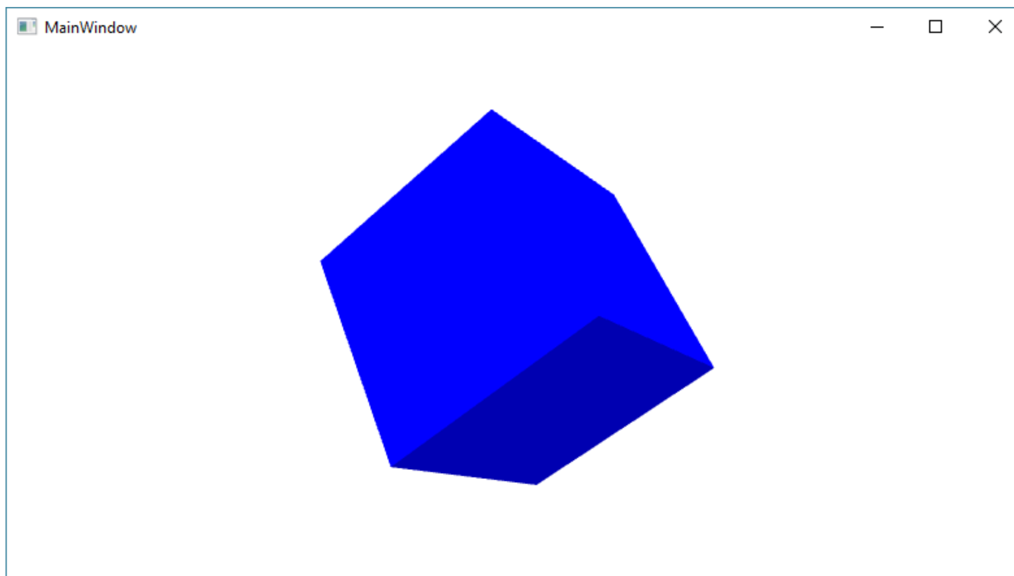
Add following code between </Viewport2DVisual3D> and </Viewport3D> tags:

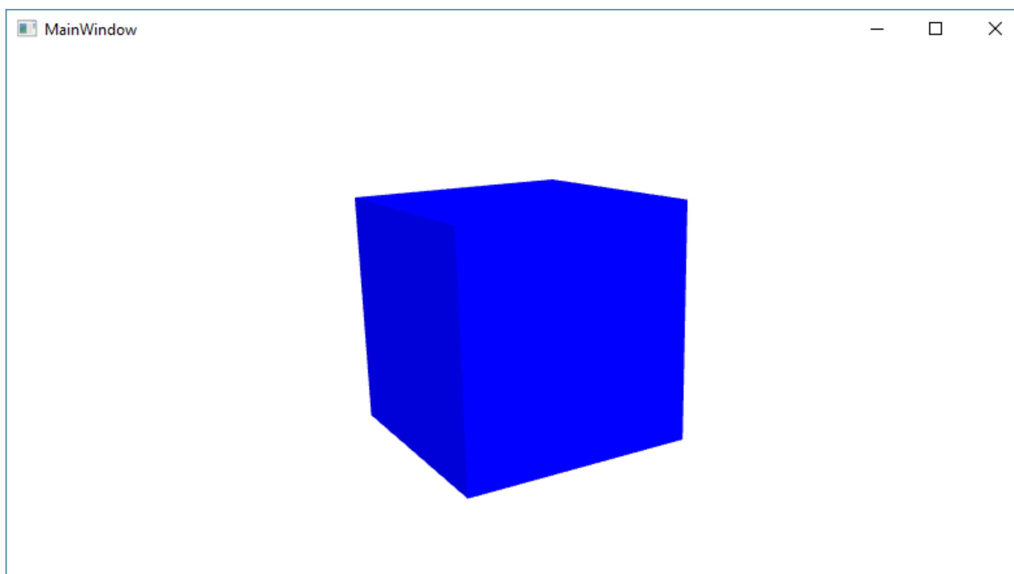
```
71        <Viewport3D.Triggers>
72          <EventTrigger RoutedEvent="Viewport3D.Loaded">
73            <BeginStoryboard>
74              <Storyboard>
75                <DoubleAnimation Storyboard.TargetName="rotation"
76                                 Storyboard.TargetProperty="Angle"
77                                 Duration="0:0:10" From="0" To="360"
78                                 RepeatBehavior="Forever" />
79              </Storyboard>
80            </BeginStoryboard>
81          </EventTrigger>
82        </Viewport3D.Triggers>
```

Our animation is controlled by a trigger. Trigger is started by Viewport3D.Loaded event. Animation lasts 10 seconds, it changes the angle of the rotation object from 0 to 360 degrees, then it is repeated endlessly.

4.7 Rotating cube

Start the application and watch the rotating cube.





4.8 Movie

To place a movie on the cube we have to inform Viewport2DVisual3D that we want to use controls instead of material, so change:

```
57              <DiffuseMaterial Brush="Blue" />
```

to:

```
57              <DiffuseMaterial Viewport2DVisual3D.IsVisualHostMaterial="True" />
```

Then add below code before </Viewport2DVisual3D> tag:

```
70          <Viewport2DVisual3D.Visual>
71            <MediaElement Source="PathToMovie.extension" LoadedBehavior="Play" />
72          </Viewport2DVisual3D.Visual>
```

Where **PathToMovie.extension** is a path to real file on your computer's disk.

4.9 Running application

Run the application. You can now see a movie on each side of the cube.