FREE eBook

LEARNING

wpf

Free unaffiliated eBook created from
Stack Overflow contributors.

#wpf

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: wpf

It is an unofficial and free wpf ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official wpf.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with wpf

Remarks

WPF (Windows Presentation Foundation) is Microsoft's recommended presentation technology for classic Windows desktop applications. WPF should not be confused with UWP (Universal Windows Platform) although similarities exist between the two.

WPF encourages data driven applications with a strong focus on multimedia, animation and data binding. Interfaces are created using a language called XAML (eXtensible Application Markup Language), a derivative of XML. XAML helps WPF programmers maintain separation of visual design and interface logic.

Unlike its predecessor Windows Forms, WPF uses a box model to layout all elements of the interface. Each element has a Height, Width and Margins and is arranged on screen relative to it's parent.

WPF stands for Windows Presentation Foundation and is also known under its Codename Avalon. It's a graphical Framework and part of Microsofts .NET Framework. WPF is pre-installed in Windows Vista, 7, 8 and 10 and can be installed on Windows XP and Server 2003.

Versions

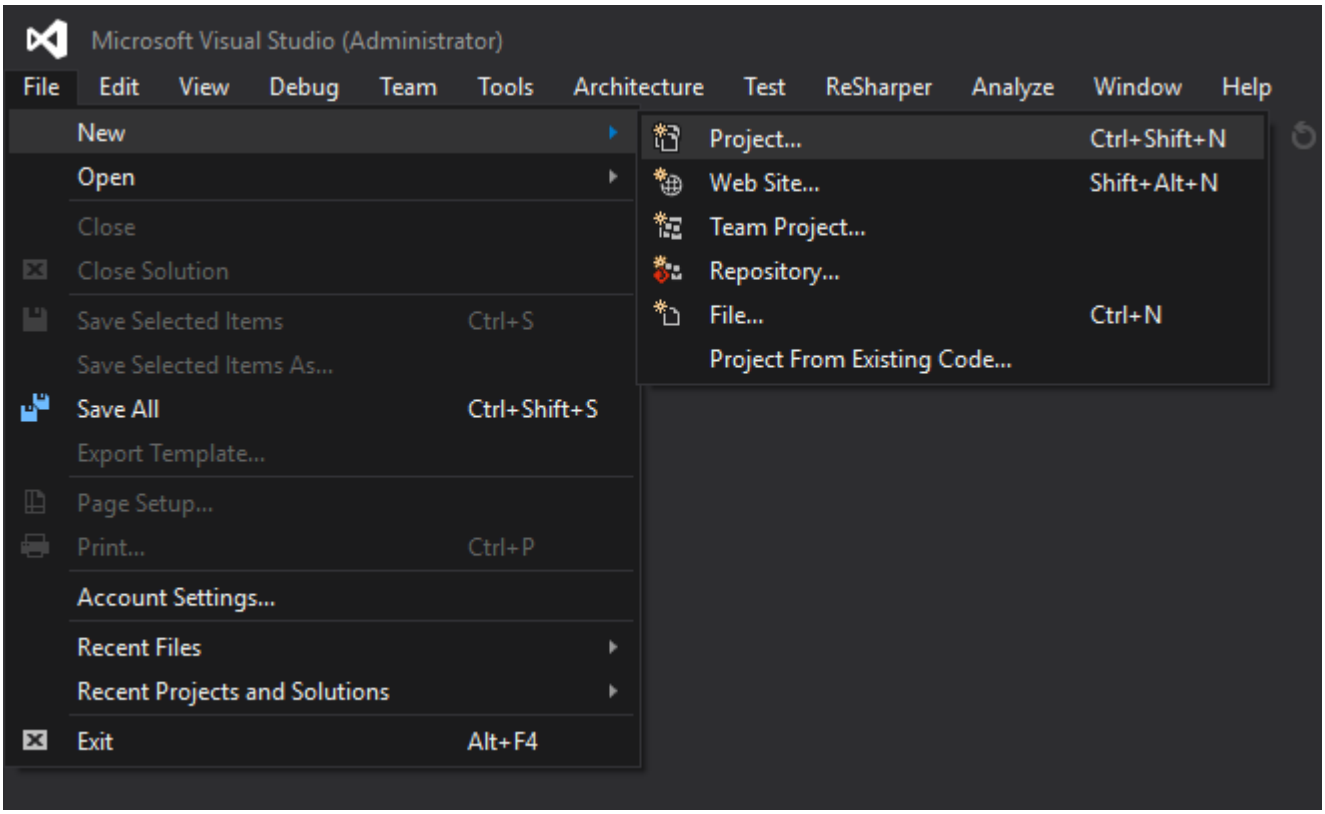
Version 4.6.1 - December 2015

Examples

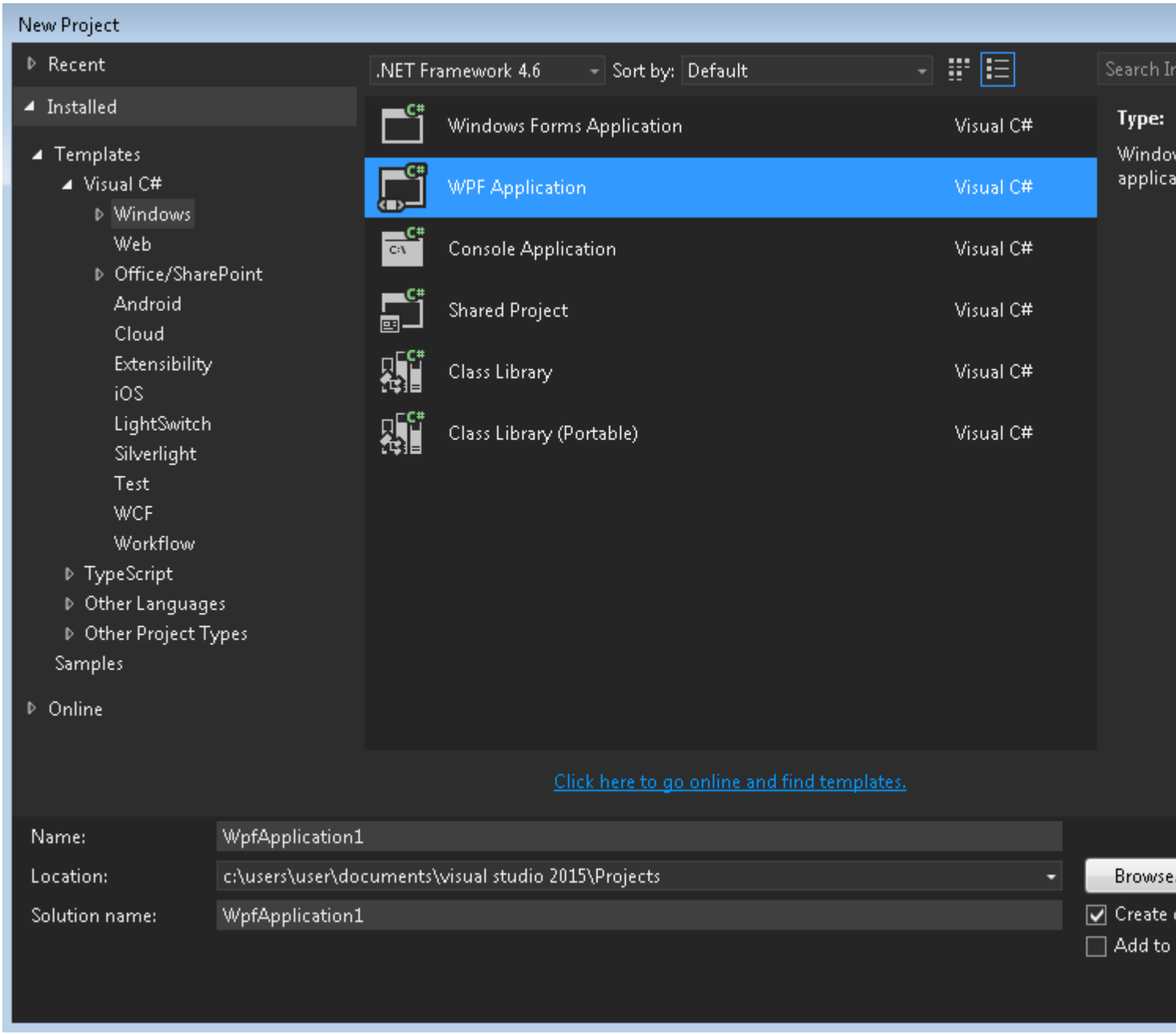
Hello World application

To create and run new WPF project in Visual Studio:

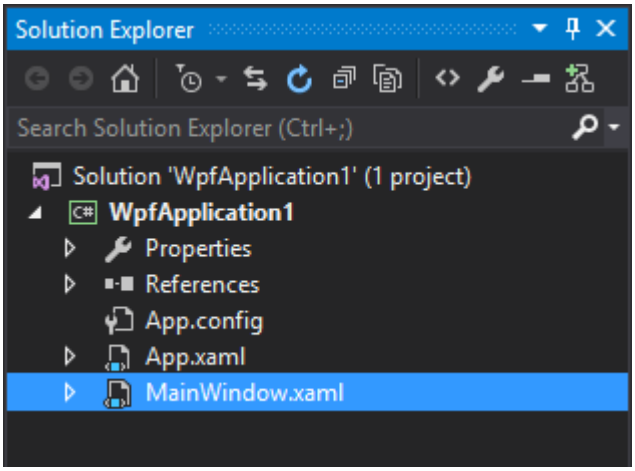
1. Click **File → New → Project**



2. Select template by clicking **Templates → Visual C# → Windows → WPF Application** and press OK:



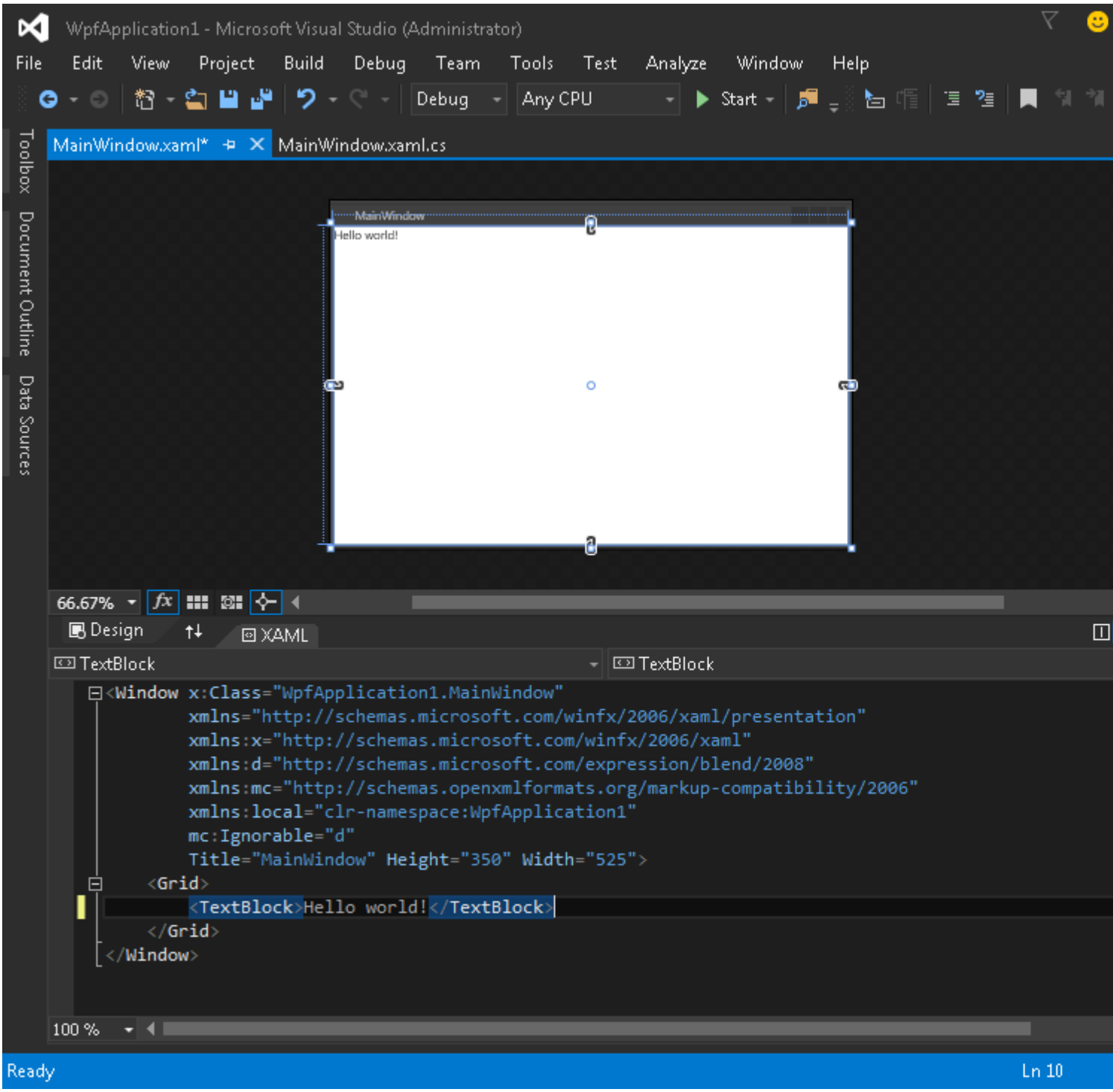
3. Open **MainWindow.xaml** file in *Solution Explorer* (if you don't see *Solution Explorer* window, open it by clicking **View → Solution Explorer**):



4. In the *XAML* section (by default below *Design* section) add this code

```
<TextBlock>Hello world!</TextBlock>
```

inside `Grid` tag:

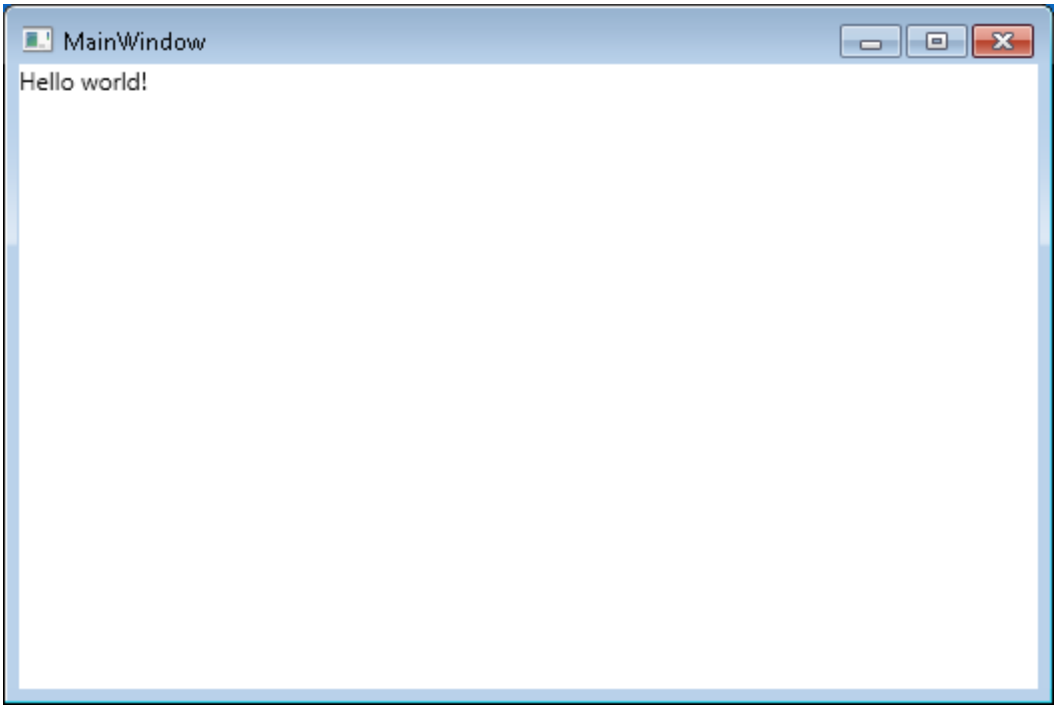


Code should look like:

```
<Window x:Class="WpfApplication1.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:WpfApplication1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <TextBlock>Hello world!</TextBlock>
    </Grid>
</Window>
```

5. Run the application by pressing `F5` or clicking menu **Debug → Start Debugging**. It should look like:



Read Getting started with wpf online: https://riptutorial.com/wpf/topic/820/getting-started-with-wpf

Chapter 2: "Half the Whitespace" Design principle

Introduction

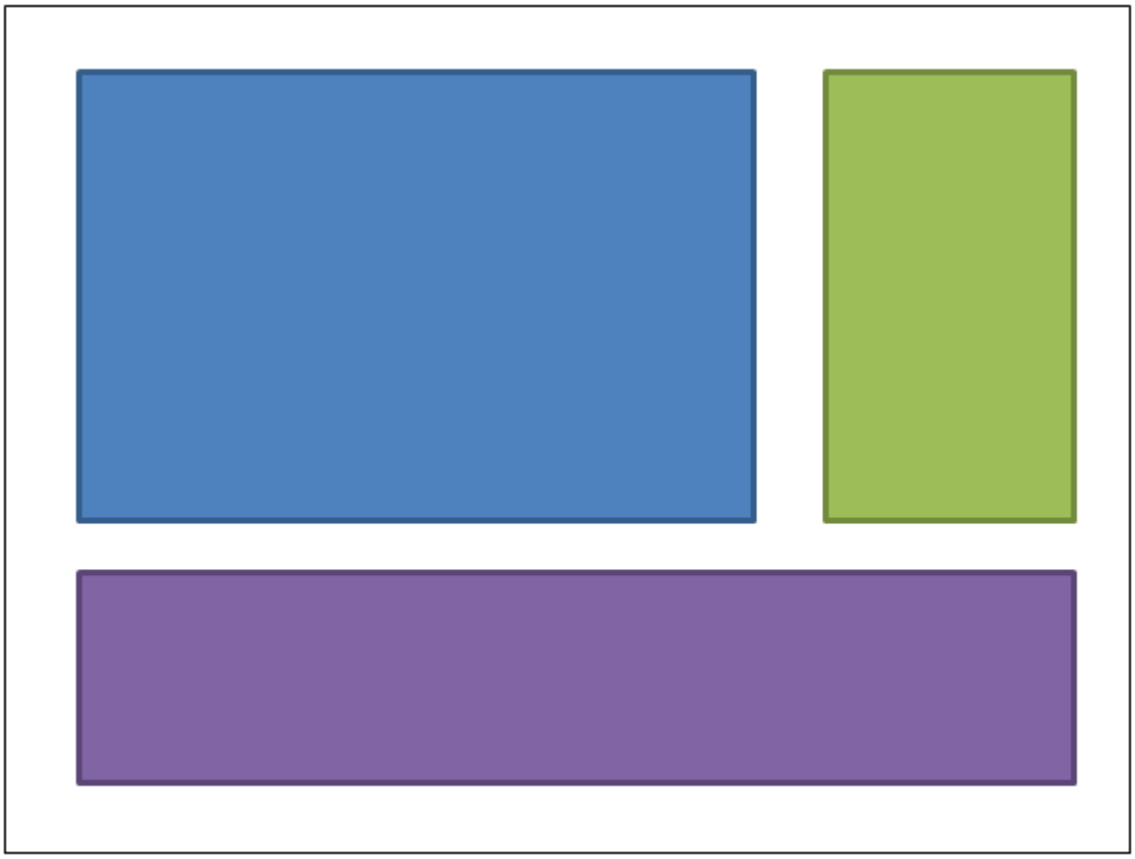
When laying out controls, it is easy to hard-code specific values in margins and paddings to make things fit the desired layout. However, by hard-coding these values, maintenance becomes much more expensive. If the layout changes, in what might be considered a trivial way, then a lot of work has to go into correcting these values.

This design principle reduces the cost of maintenance of the layout by thinking about the layout in a different way.

Examples

Demonstration of the problem and the solution

For example, imagine a screen with 3 sections, laid out like this:



The blue box might be given a margin of 4,4,0,0. The green box might be given a margin of 4,4,4,0. The purple box margin would be 4,4,4,4. Here's the XAML: (I'm using a grid to achieve the layout; but this design principle applies regardless of how you choose to achieve the layout):

```
<UserControl x:Class="WpfApplication5.UserControl1HardCoded"
             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.ColumnDefinitions>
        <ColumnDefinition Width="3*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="2*"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>

    <Border Grid.Column="0" Grid.Row="0" Margin="4,4,0,0" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
    <Border Grid.Column="1" Grid.Row="0" Margin="4,4,4,0" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
    <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="4,4,4,4"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
</Grid>
</UserControl>
```

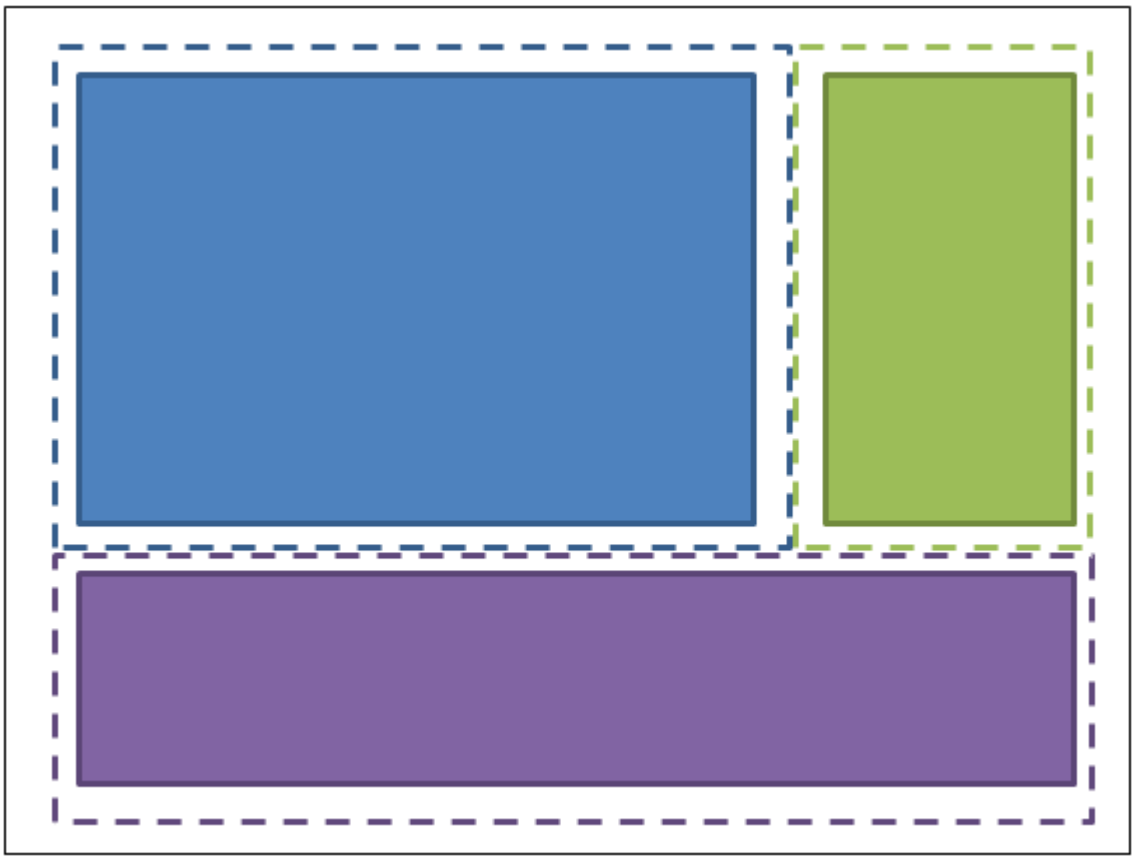
Now imagine that we want to change the layout, to put the green box on the left of the blue box. Should be simple, shouldn't it? Except that when we move that box, we now need to tinker with the margins. Either we can change the blue box's margins to 0,4,4,0; or we could change blue to 4,4,4,0 and green to 4,4,0,0. Here's the XAML:

```
<UserControl x:Class="WpfApplication5.UserControl2HardCoded"
             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.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="0" Margin="4,4,4,0" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="0" Margin="4,4,0,0" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="4,4,4,4"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

Now let's put the purple box at the top. So blue's margins become 4,0,4,4; green becomes 4,0,0,4.

```
<UserControl x:Class="WpfApplication5.UserControl3HardCoded"
             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.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="2*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="1" Margin="4,0,4,4" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="1" Margin="4,0,0,4" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" Margin="4,4,4,4"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

Wouldn't it be nice if we could move things around so that we didn't need to adjust these values at all. This can be achieved by just thinking about the whitespace in a different way. Rather than allocating all the whitespace to one control or the other, imagine half the whitespace being allocated to each box: (my drawing is not quite to scale - the dotted lines should be half-way between the edge of the box and its neighbour).



So the blue box has margins of 2,2,2,2; the green box has margins of 2,2,2,2; the purple box has margins of 2,2,2,2. And the container in which they are housed is given a padding (not margin) of 2,2,2,2. Here's the XAML:

```
<UserControl x:Class="WpfApplication5.UserControl1HalfTheWhitespace"
             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"
             Padding="2,2,2,2">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="3*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="0" Grid.Row="0" Margin="2,2,2,2" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="1" Grid.Row="0" Margin="2,2,2,2" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="2,2,2,2"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

Now let’s try moving the boxes around, the same way as before...Let’s put the green box on the left of the blue box. OK, done. And there was no need to change any padding or margins. Here's the XAML:

```
<UserControl x:Class="WpfApplication5.UserControl2HalfTheWhitespace"
             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"
             Padding="2,2,2,2">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="0" Margin="2,2,2,2" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="0" Margin="2,2,2,2" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Margin="2,2,2,2"
```

```
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

Now let’s put the purple box at the top. OK, done. And there was no need to change any padding or margins. Here's the XAML:

```
<UserControl x:Class="WpfApplication5.UserControl3HalfTheWhitespace"
             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"
             Padding="2,2,2,2">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="2*"/>
        </Grid.RowDefinitions>

        <Border Grid.Column="1" Grid.Row="1" Margin="2,2,2,2" Background="DodgerBlue"
BorderBrush="DarkBlue" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.Row="1" Margin="2,2,2,2" Background="Green"
BorderBrush="DarkGreen" BorderThickness="5"/>
        <Border Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" Margin="2,2,2,2"
Background="MediumPurple" BorderBrush="Purple" BorderThickness="5"/>
    </Grid>
</UserControl>
```

How to use this in real code

To generalise what we've demonstrated above: individual things contain a fixed *margin* of "half-the-whitespace", and the container they are held in should have a *padding* of "half-the-whitespace". You can apply these styles in your application resource dictionary, and then you won't even need to mention them on the individual items. Here's how you could define "HalfTheWhiteSpace":

```
<system:Double x:Key="DefaultMarginSize">2</system:Double>
<Thickness x:Key="HalfTheWhiteSpace" Left="{StaticResource DefaultMarginSize}"
Top="{StaticResource DefaultMarginSize}" Right="{StaticResource DefaultMarginSize}"
Bottom="{StaticResource DefaultMarginSize}"/>
```

Then I can define a base style to base my other controls styles on: (this could also contain your default FontFamily, FontSize, etc, etc)

```
<Style x:Key="BaseStyle" TargetType="{x:Type Control}">
    <Setter Property="Margin" Value="{StaticResource HalfTheWhiteSpace}"/>
</Style>
```

Then I can define my default styling for TextBox to use this margin:

```
<Style TargetType="TextBox" BasedOn="{StaticResource BaseStyle}"/>
```

I can do this kind of thing for DatePickers, Labels, etc, etc. (anything which might be held within a container). Beware of styling TextBlock like this... that control is used internally by a lot of controls. I'd suggest you create your own control which simply derives from TextBlock. You can style *your* TextBlock to use the default margin; and you should use *your* TextBlock whenever you explicitly use a TextBlock in your XAML.

You can use a similar approach to apply the padding to common containers (e.g. ScrollViewer, Border, etc).

Once you've done this, *most* of your controls will not need margins and padding - and you will only need to specify values in places where you intentionally want to deviate from this design principle.

Read "Half the Whitespace" Design principle online: https://riptutorial.com/wpf/topic/9407/-half-the-whitespace--design-principle

Chapter 3: An Introduction to WPF Styles

Introduction

A style allows the complete modification of the visual appearance of a WPF control. Here are some examples of some basic styling, and an introduction to resource dictionaries and animation.

Examples

Styling a Button

The easiest way to create a style is to copy an existing one and edit it.

Create a simple window with two buttons:

```
<Window x:Class="WPF_Style_Example.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"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="MainWindow"
    Height="150" Width="200">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Margin="5" Content="Button 1"/>
    <Button Margin="5" Grid.Row="1" Content="Button 2"/>
</Grid>
```

In Visual Studio, copying can be done by right-clicking on the first button in the editor and choosing "Edit a Copy..." under the "Edit Template" menu.

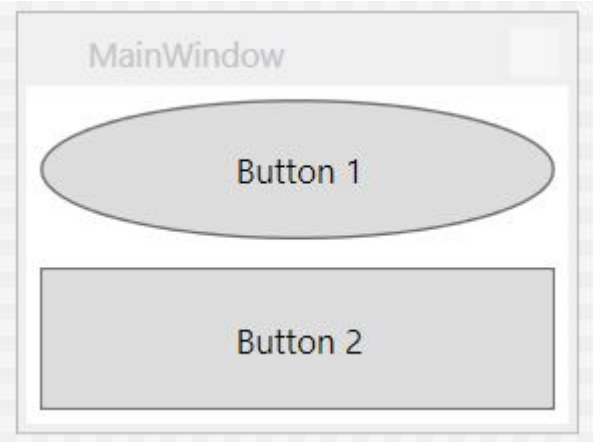
Define in "Application".

The following example shows a modified template to create an ellipse button:

```
<Style x:Key="ButtonStyle1" TargetType="{x:Type Button}">
        <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
        <Setter Property="Background" Value="{StaticResource Button.Static.Background}"/>
        <Setter Property="BorderBrush" Value="{StaticResource Button.Static.Border}"/>
        <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
        <Setter Property="BorderThickness" Value="1"/>
        <Setter Property="HorizontalContentAlignment" Value="Center"/>
        <Setter Property="VerticalContentAlignment" Value="Center"/>
        <Setter Property="Padding" Value="1"/>
        <Setter Property="Template">
            <Setter.Value>
```

```
                <ControlTemplate TargetType="{x:Type Button}">
                    <Grid>
                        <Ellipse x:Name="ellipse" StrokeThickness="{TemplateBinding
BorderThickness}" Stroke="{TemplateBinding BorderBrush}" Fill="{TemplateBinding Background}"
SnapsToDevicePixels="true"/>
                        <ContentPresenter x:Name="contentPresenter" Focusable="False"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="{TemplateBinding
Padding}" RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}" VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
                    </Grid>
                    <ControlTemplate.Triggers>
                        <Trigger Property="IsDefaulted" Value="true">
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                        </Trigger>
                        <Trigger Property="IsMouseOver" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Border}"/>
                        </Trigger>
                        <Trigger Property="IsPressed" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Border}"/>
                        </Trigger>
                        <Trigger Property="IsEnabled" Value="false">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Border}"/>
                            <Setter Property="TextElement.Foreground"
TargetName="contentPresenter" Value="{StaticResource Button.Disabled.Foreground}"/>
                        </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

The result:



Style Applied to All Buttons

Taking the previous example, removing the x:Key element of the style applies the style to all buttons in the Application scope.

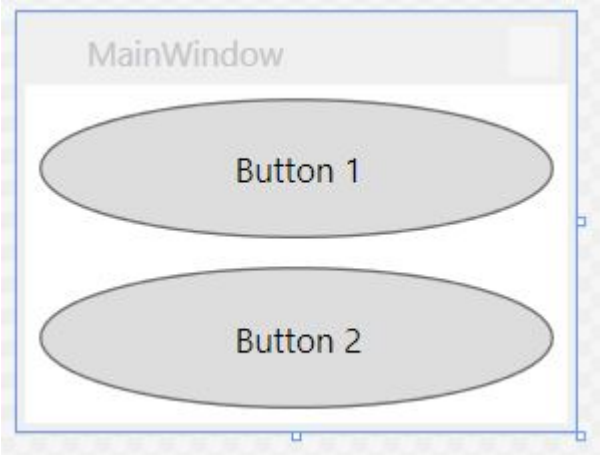
```
<Style TargetType="{x:Type Button}">
        <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
        <Setter Property="Background" Value="{StaticResource Button.Static.Background}"/>
        <Setter Property="BorderBrush" Value="{StaticResource Button.Static.Border}"/>
        <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
        <Setter Property="BorderThickness" Value="1"/>
        <Setter Property="HorizontalContentAlignment" Value="Center"/>
        <Setter Property="VerticalContentAlignment" Value="Center"/>
        <Setter Property="Padding" Value="1"/>
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type Button}">
                    <Grid>
                        <Ellipse x:Name="ellipse" StrokeThickness="{TemplateBinding
BorderThickness}" Stroke="{TemplateBinding BorderBrush}" Fill="{TemplateBinding Background}"
SnapsToDevicePixels="true"/>
                        <ContentPresenter x:Name="contentPresenter" Focusable="False"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="{TemplateBinding
Padding}" RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}" VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
                    </Grid>
                    <ControlTemplate.Triggers>
                        <Trigger Property="IsDefaulted" Value="true">
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                        </Trigger>
                        <Trigger Property="IsMouseOver" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.MouseOver.Border}"/>
                        </Trigger>
                        <Trigger Property="IsPressed" Value="true">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Pressed.Border}"/>
                        </Trigger>
                        <Trigger Property="IsEnabled" Value="false">
                            <Setter Property="Fill" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Background}"/>
                            <Setter Property="Stroke" TargetName="ellipse"
Value="{StaticResource Button.Disabled.Border}"/>
                            <Setter Property="TextElement.Foreground"
TargetName="contentPresenter" Value="{StaticResource Button.Disabled.Foreground}"/>
                        </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

Note that the Style no longer needs to be specified for individual buttons:

```
<Window x:Class="WPF_Style_Example.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"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="MainWindow"
    Height="150" Width="200">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Margin="5" Content="Button 1"/>
    <Button Margin="5" Grid.Row="1" Content="Button 2"/>
</Grid>
```

Both buttons are now styled.



Styling a ComboBox

Starting with the following `ComboBox`es:

```
<Window x:Class="WPF_Style_Example.ComboBoxWindow"
    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"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="ComboBoxWindow"
    Height="100" Width="150">
<StackPanel>
    <ComboBox Margin="5" SelectedIndex="0">
        <ComboBoxItem Content="Item A"/>
        <ComboBoxItem Content="Item B"/>
        <ComboBoxItem Content="Item C"/>
    </ComboBox>
    <ComboBox IsEditable="True" Margin="5" SelectedIndex="0">
        <ComboBoxItem Content="Item 1"/>
        <ComboBoxItem Content="Item 2"/>
        <ComboBoxItem Content="Item 3"/>
    </ComboBox>
</StackPanel>
```

Right click on the first `ComboBox` in the designer, choose "Edit Template --> Edit a Copy". Define the style in the application scope.

There are 3 styles created:

```
ComboBoxToggleButton
ComboBoxEditableTextBox
ComboBoxStyle1
```

And 2 templates:

```
ComboBoxTemplate
ComboBoxEditableTemplate
```

An example of editing the `ComboBoxToggleButton` style:

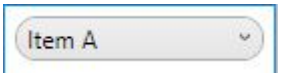
```
<SolidColorBrush x:Key="ComboBox.Static.Border" Color="#FFACACAC"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Border" Color="#FFABADB3"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Button.Background" Color="Transparent"/>
    <SolidColorBrush x:Key="ComboBox.Static.Editable.Button.Border" Color="Transparent"/>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Glyph" Color="#FF000000"/>
    <LinearGradientBrush x:Key="ComboBox.MouseOver.Background" EndPoint="0,1"
StartPoint="0,0">
        <GradientStop Color="Orange" Offset="0.0"/>
        <GradientStop Color="OrangeRed" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Border" Color="Red"/>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Editable.Border" Color="#FF7EB4EA"/>
    <LinearGradientBrush x:Key="ComboBox.MouseOver.Editable.Button.Background" EndPoint="0,1"
StartPoint="0,0">
        <GradientStop Color="#FFEBF4FC" Offset="0.0"/>
        <GradientStop Color="#FFDCECFC" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.MouseOver.Editable.Button.Border" Color="#FF7EB4EA"/>
    <SolidColorBrush x:Key="ComboBox.Pressed.Glyph" Color="#FF000000"/>
    <LinearGradientBrush x:Key="ComboBox.Pressed.Background" EndPoint="0,1" StartPoint="0,0">
        <GradientStop Color="OrangeRed" Offset="0.0"/>
        <GradientStop Color="Red" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.Pressed.Border" Color="DarkRed"/>
    <SolidColorBrush x:Key="ComboBox.Pressed.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.Pressed.Editable.Border" Color="#FF569DE5"/>
    <LinearGradientBrush x:Key="ComboBox.Pressed.Editable.Button.Background" EndPoint="0,1"
StartPoint="0,0">
        <GradientStop Color="#FFDAEBFC" Offset="0.0"/>
        <GradientStop Color="#FFC4E0FC" Offset="1.0"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ComboBox.Pressed.Editable.Button.Border" Color="#FF569DE5"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Glyph" Color="#FFBFBFBF"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Background" Color="#FFF0F0F0"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Border" Color="#FFD9D9D9"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Background" Color="#FFFFFFFF"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Border" Color="#FFBFBFBF"/>
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Button.Background"
Color="Transparent"/>
```

```
    <SolidColorBrush x:Key="ComboBox.Disabled.Editable.Button.Border" Color="Transparent"/>
    <SolidColorBrush x:Key="ComboBox.Static.Glyph" Color="#FF606060"/>
    <Style x:Key="ComboBoxToggleButton" TargetType="{x:Type ToggleButton}">
        <Setter Property="OverridesDefaultStyle" Value="true"/>
        <Setter Property="IsTabStop" Value="false"/>
        <Setter Property="Focusable" Value="false"/>
        <Setter Property="ClickMode" Value="Press"/>
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type ToggleButton}">
                    <Border x:Name="templateRoot" CornerRadius="10"
BorderBrush="{StaticResource ComboBox.Static.Border}" BorderThickness="{TemplateBinding
BorderThickness}" Background="{StaticResource ComboBox.Static.Background}"
SnapsToDevicePixels="true">
                        <Border x:Name="splitBorder" BorderBrush="Transparent"
BorderThickness="1" HorizontalAlignment="Right" Margin="0" SnapsToDevicePixels="true"
Width="{DynamicResource {x:Static SystemParameters.VerticalScrollBarWidthKey}}">
                            <Path x:Name="arrow" Data="F1 M 0,0 L 2.667,2.66665 L 5.3334,0 L
5.3334,-1.78168 L 2.6667,0.88501 L0,-1.78168 L0,0 Z" Fill="{StaticResource
ComboBox.Static.Glyph}" HorizontalAlignment="Center" Margin="0" VerticalAlignment="Center"/>
                        </Border>
                    </Border>
                    <ControlTemplate.Triggers>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                                <Condition Binding="{Binding IsMouseOver,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsPressed,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsEnabled,
RelativeSource={RelativeSource Self}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Static.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Static.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.Static.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.Static.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                        <Trigger Property="IsMouseOver" Value="true">
                            <Setter Property="BorderThickness" TargetName="templateRoot"
Value="2"/>
                        </Trigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsMouseOver,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="false"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Border}"/>
                        </MultiDataTrigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
```

```
                                <Condition Binding="{Binding IsMouseOver,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.MouseOver.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.MouseOver.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.MouseOver.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                        <Trigger Property="IsPressed" Value="true">
                            <Setter Property="Fill" TargetName="arrow" Value="{StaticResource
ComboBox.Pressed.Glyph}"/>
                        </Trigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsPressed,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="false"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Border}"/>
                        </MultiDataTrigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsPressed,
RelativeSource={RelativeSource Self}}" Value="true"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Pressed.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.Pressed.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.Pressed.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                        <Trigger Property="IsEnabled" Value="false">
                            <Setter Property="Fill" TargetName="arrow" Value="{StaticResource
ComboBox.Disabled.Glyph}"/>
                        </Trigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsEnabled,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="false"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Border}"/>
```

```
                        </MultiDataTrigger>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding IsEnabled,
RelativeSource={RelativeSource Self}}" Value="false"/>
                                <Condition Binding="{Binding IsEditable,
RelativeSource={RelativeSource AncestorType={x:Type ComboBox}}}" Value="true"/>
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Background" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Editable.Background}"/>
                            <Setter Property="BorderBrush" TargetName="templateRoot"
Value="{StaticResource ComboBox.Disabled.Editable.Border}"/>
                            <Setter Property="Background" TargetName="splitBorder"
Value="{StaticResource ComboBox.Disabled.Editable.Button.Background}"/>
                            <Setter Property="BorderBrush" TargetName="splitBorder"
Value="{StaticResource ComboBox.Disabled.Editable.Button.Border}"/>
                        </MultiDataTrigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

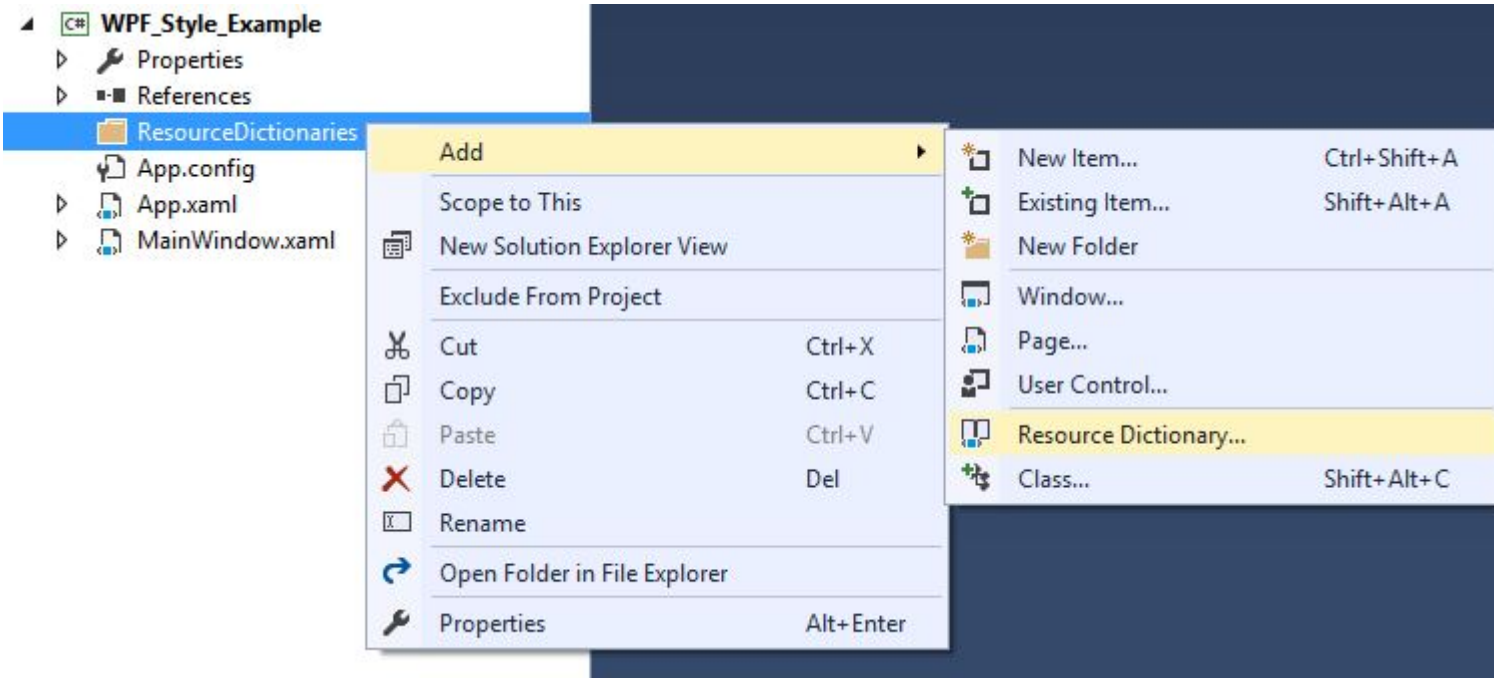
This creates a rounded `ComboBox` that highlights orange on mouse over and turns red when pressed.



Note that this will not change the Editable combobox below it; modifying that requires changing the `ComboBoxEditableTextBox` style or the `ComboBoxEditableTemplate`.

Creating a Resource Dictionary

Having lots of styles in App.xaml will quickly become complex, so they can be placed in separate resource dictionaries.



In order to use the dictionary, it must be merged with App.xaml. So, in App.xaml, after the resource dictionary has been created:

```
<Application
         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
         x:Class="WPF_Style_Example.App"
         StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="ResourceDictionaries/Dictionary1.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

New styles can now be created in the Dictionary1.xaml and they can be referenced as if they were in App.xaml. After building the project, the option will also appear in Visual Studio when copying a style to locate it in the new resource dictionary.

Button Style DoubleAnimation

The following `Window` has been created:

```
<Window x:Class="WPF_Style_Example.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"
    mc:Ignorable="d" ResizeMode="NoResize"
    Title="MainWindow"
    Height="150" Width="250">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Button Margin="5" Content="Button 1" Width="200"/>
    <Button Margin="5" Grid.Row="1" Content="Button 2" Width="200"/>
</Grid>
```

A style (created in App.xaml) has been applied to the buttons, which animates the width from 200 to 100 when the mouse enters the control and from 100 to 200 when it leaves:

```
<Style TargetType="{x:Type Button}">
    <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
    <Setter Property="Background" Value="{StaticResource Button.Static.Background}"/>
    <Setter Property="BorderBrush" Value="{StaticResource Button.Static.Border}"/>
    <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
    <Setter Property="BorderThickness" Value="1"/>
    <Setter Property="HorizontalContentAlignment" Value="Center"/>
    <Setter Property="VerticalContentAlignment" Value="Center"/>
    <Setter Property="Padding" Value="1"/>
```

```
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Button}">
                <Grid Background="White">
                    <Border x:Name="border" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" Background="{TemplateBinding Background}"
SnapsToDevicePixels="true">
                        <ContentPresenter x:Name="contentPresenter" Focusable="False"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}" Margin="{TemplateBinding
Padding}" RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}" VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
                    </Border>
                </Grid>
                <ControlTemplate.Triggers>
                    <EventTrigger RoutedEvent="MouseEnter">
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation To="100" From="200"
Storyboard.TargetProperty="Width" Storyboard.TargetName="border" Duration="0:0:0.25"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                    <EventTrigger RoutedEvent="MouseLeave">
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation To="200" From="100"
Storyboard.TargetProperty="Width" Storyboard.TargetName="border" Duration="0:0:0.25"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                    <Trigger Property="IsDefaulted" Value="true">
                        <Setter Property="BorderBrush" TargetName="border"
Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                    </Trigger>
                    <Trigger Property="IsMouseOver" Value="true">
                        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.MouseOver.Background}"/>
                        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.MouseOver.Border}"/>
                    </Trigger>
                    <Trigger Property="IsPressed" Value="true">
                        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.Pressed.Background}"/>
                        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.Pressed.Border}"/>
                    </Trigger>
                    <Trigger Property="IsEnabled" Value="false">
                        <Setter Property="Background" TargetName="border"
Value="{StaticResource Button.Disabled.Background}"/>
                        <Setter Property="BorderBrush" TargetName="border"
Value="{StaticResource Button.Disabled.Border}"/>
                        <Setter Property="TextElement.Foreground"
TargetName="contentPresenter" Value="{StaticResource Button.Disabled.Foreground}"/>
                    </Trigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Read An Introduction to WPF Styles online: https://riptutorial.com/wpf/topic/9670/an-introduction-

to-wpf-styles

Chapter 4: Creating custom UserControls with data binding

Remarks

Note that a UserControl is very different from a Control. One of the primary differences is that a UserControl makes use of a XAML layout file to determine where to place several individual Controls. A Control, on the other hand, is just pure code - there's no layout file at all. In some ways, creating a custom Control can be more effective than creating a custom UserControl.

Examples

ComboBox with custom default text

This custom UserControl will appear as a regular combobox, but unlike the built-in ComboBox object, it can show the user a default string of text if they have not made a selection yet.

In order to accomplish this, our UserControl will be made up of two Controls. Obviously we need an actual ComboBox, but we will also use a regular Label to show the default text.

CustomComboBox.xaml

```
<UserControl x:Class="UserControlDemo.CustomComboBox"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:cnvrt="clr-namespace:UserControlDemo"
             x:Name="customComboBox">
    <UserControl.Resources>
        <cnvrt:InverseNullVisibilityConverter x:Key="invNullVisibleConverter" />
    </UserControl.Resources>
    <Grid>
        <ComboBox x:Name="comboBox"
                  ItemsSource="{Binding ElementName=customComboBox, Path=MyItemsSource}"
                  SelectedItem="{Binding ElementName=customComboBox, Path=MySelectedItem}"
                  HorizontalContentAlignment="Left" VerticalContentAlignment="Center"/>

        <Label HorizontalAlignment="Left" VerticalAlignment="Center"
               Margin="0,2,20,2" IsHitTestVisible="False"
               Content="{Binding ElementName=customComboBox, Path=DefaultText}"
               Visibility="{Binding ElementName=comboBox, Path=SelectedItem,
Converter={StaticResource invNullVisibleConverter}}"/>
    </Grid>
</UserControl>
```

As you can see, this single UserControl is actually group of two individual Controls. This allows us some flexibility that is not available in a single ComboBox alone.

Here are several important things to note:

- The UserControl itself has an `x:Name` set. This is because we want to bind to properties that are located in the code-behind, which means it needs some way to reference itself.
- Each of the binding on the ComboBox have the UserControl's name as the `ElementName`. This is so that the UserControl knows to look at itself to locate bindings.
- The Label is not hit-test visible. This is to give the user the illusion that the Label is part of the ComboBox. By setting `IsHitTestVisible=false`, we disallow the user from hovering over or clicking on the Label - all input is passed through it to the ComboBox below.
- The Label uses an `InverseNullVisibility` converter to determine whether it should show itself or not. You can find the code for this at the bottom of this example.

CustomComboBox.xaml.cs

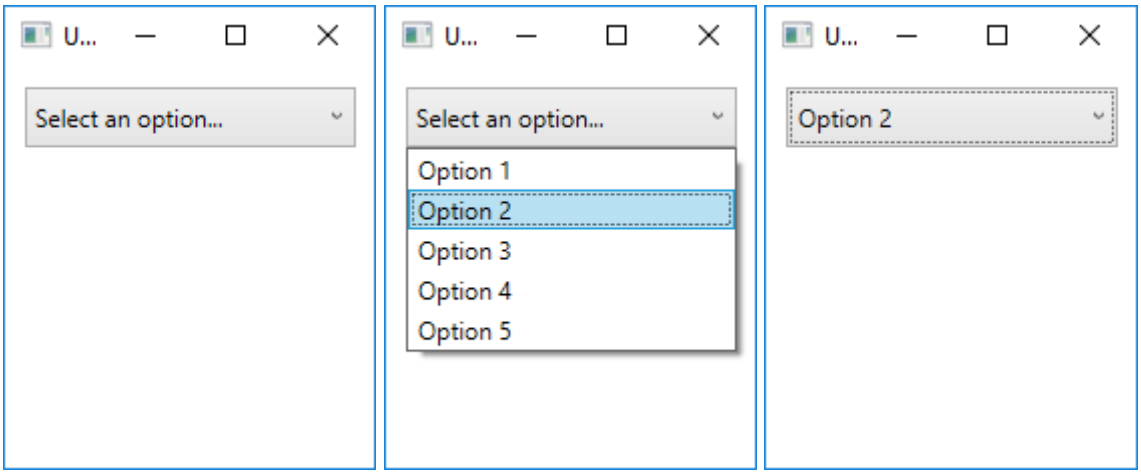
```
public partial class CustomComboBox : UserControl
{
    public CustomComboBox()
    {
        InitializeComponent();
    }

    public static DependencyProperty DefaultTextProperty =
        DependencyProperty.Register("DefaultText", typeof(string), typeof(CustomComboBox));

    public static DependencyProperty MyItemsSourceProperty =
        DependencyProperty.Register("MyItemsSource", typeof(IEnumerable),
typeof(CustomComboBox));

    public static DependencyProperty SelectedItemProperty =
        DependencyProperty.Register("SelectedItem", typeof(object), typeof(CustomComboBox));

    public string DefaultText
    {
        get { return (string)GetValue(DefaultTextProperty); }
        set { SetValue(DefaultTextProperty, value); }
    }

    public IEnumerable MyItemsSource
    {
        get { return (IEnumerable)GetValue(MyItemsSourceProperty); }
        set { SetValue(MyItemsSourceProperty, value); }
    }

    public object MySelectedItem
    {
        get { return GetValue(MySelectedItemProperty); }
        set { SetValue(MySelectedItemProperty, value); }
    }
}
```

In the code-behind, we're simply exposing which properties we want to be available to the programmer using this UserControl. Unfortunately, because we don't have direct access to the ComboBox from outside this class, we need to expose duplicate properties (`MyItemsSource` for the ComboBox's `ItemsSource`, for example). However, this is a minor tradeoff considering that we can now use this similarly to a native control.

Here's how the `CustomComboBox` UserControl might be used:

```
<Window x:Class="UserControlDemo.UserControlDemo"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:cntrls="clr-namespace:UserControlDemo"
        Title="UserControlDemo" Height="240" Width=200>
    <Grid>
        <cntrls:CustomComboBox HorizontalAlignment="Left" Margin="10,10,0,0"
VerticalAlignment="Top" Width="165"
                               MyItemsSource="{Binding Options}"
                               MySelectedItem="{Binding SelectedOption, Mode=TwoWay}"
                               DefaultText="Select an option..."/>
    <Grid>
</Window>
```

And the end result:



Here's the InverseNullVisibilityConverter needed for the Label on the UserControl, which is just a slight variation on lll's version:

```
public class InverseNullVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return value == null ? Visibility.Visible : Visibility.Hidden;
    }

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

Read Creating custom UserControls with data binding online:
https://riptutorial.com/wpf/topic/6508/creating-custom-usercontrols-with-data-binding

Chapter 5: Creating Splash Screen in WPF

Introduction

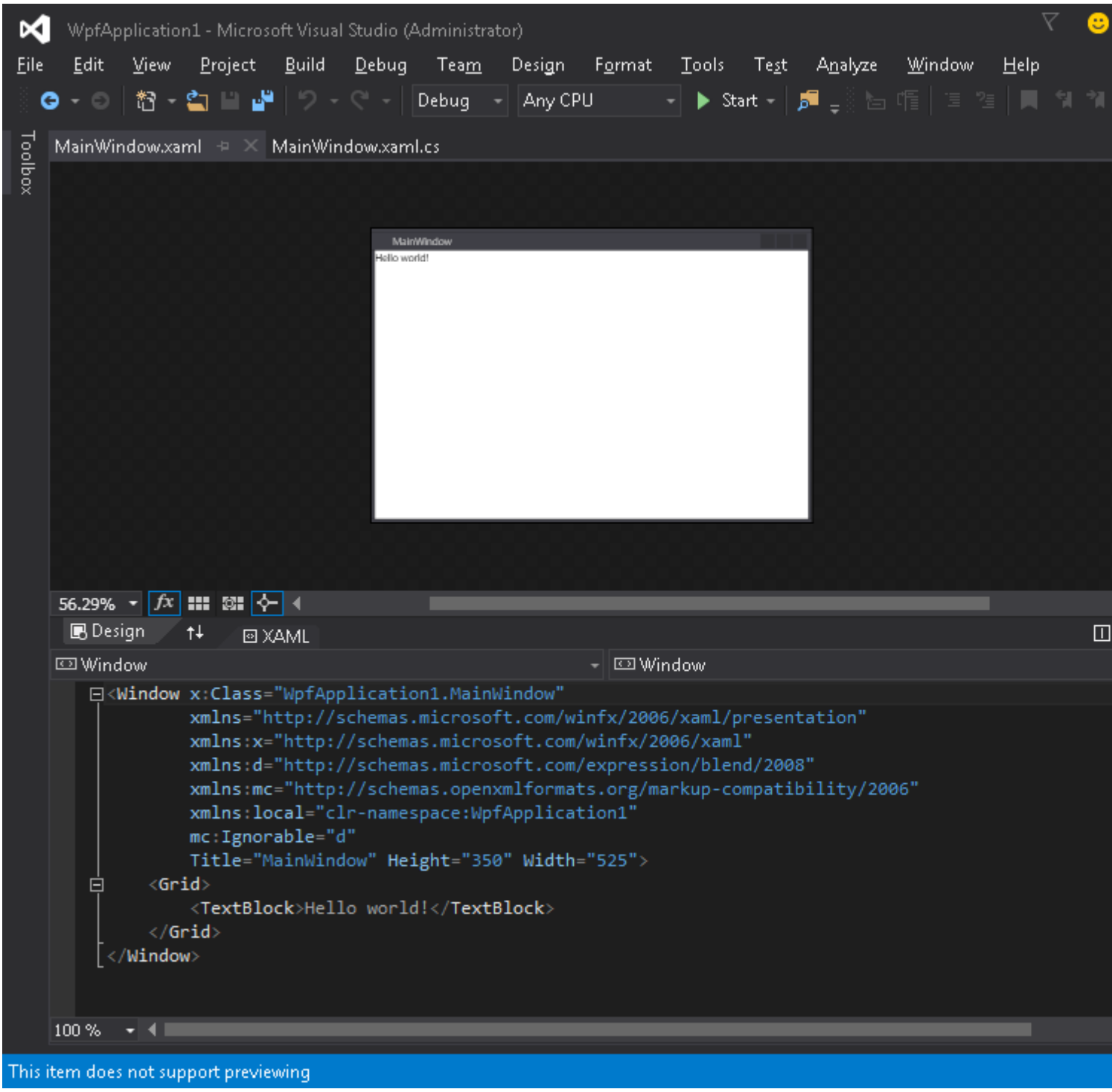
When WPF application launched, it could take a while for a current language runtime (CLR) to initialize .NET Framework. As a result, first application window can appear some time after application was launched, depending of application complexity. Splash screen in WPF allows application to show either static image or custom dynamic content during initialization before first window appears.

Examples

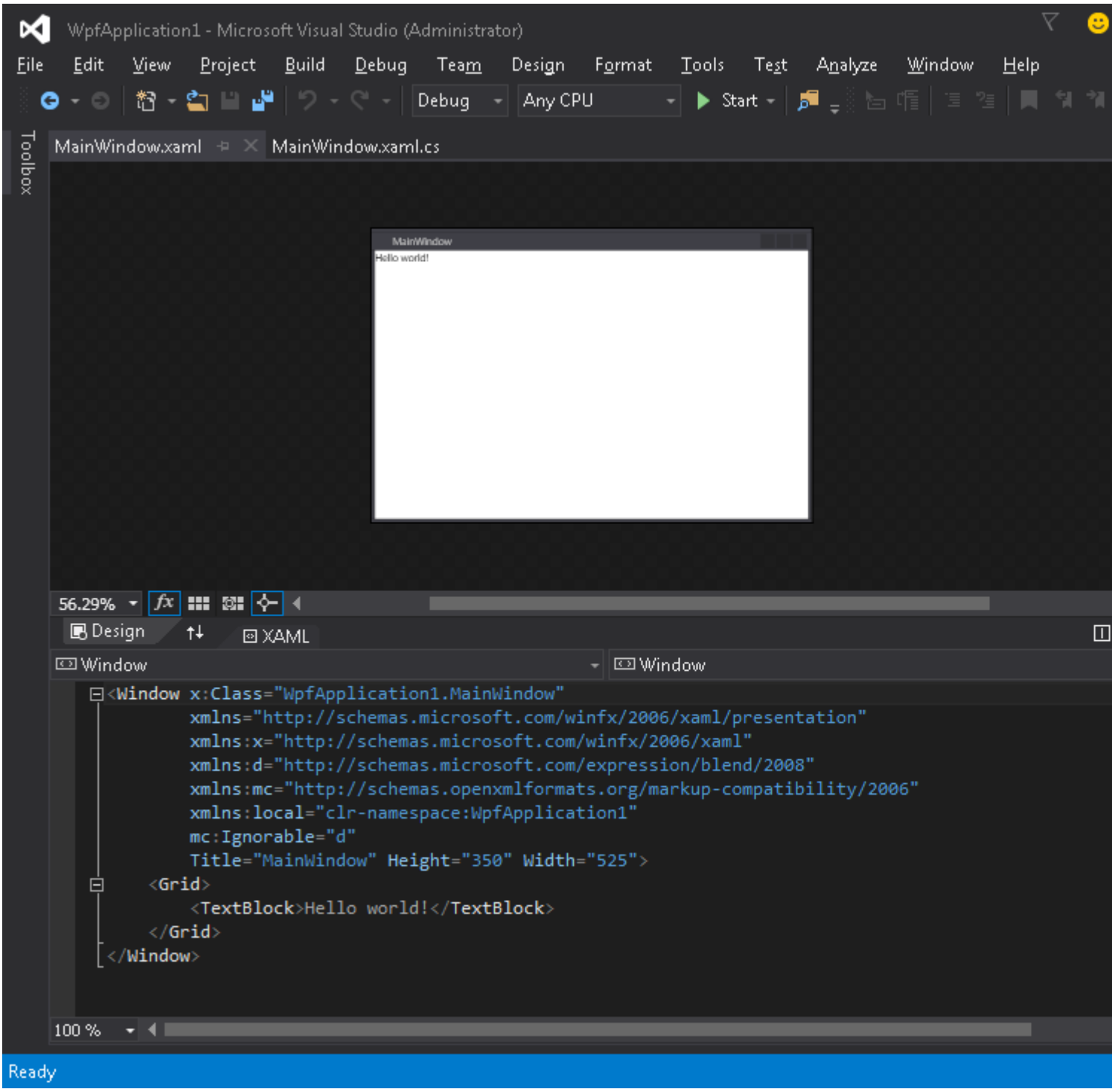
Adding simple Splash Screen

Follow this steps for adding splash screen into WPF application in Visual Studio:

1. Create or get any image and add it to your project (e.g. inside *Images* folder):



2. Open properties window for this image (**View → Properties Window**) and change **Build Action** setting to **SplashScreen** value:



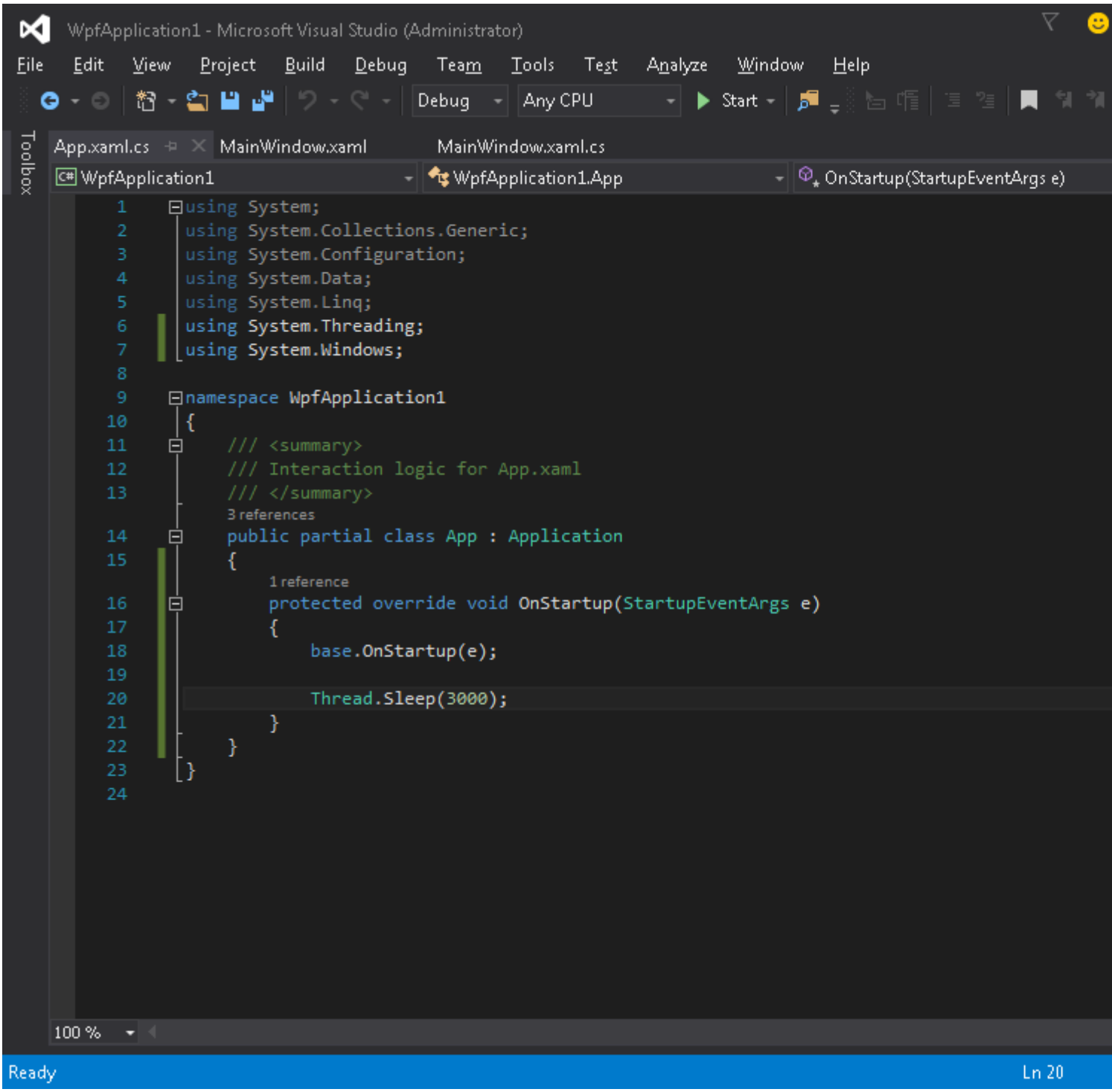
3. Run the application. You'll see your splash screen image on the center of the screen before application window appears (after window appears, splash screen image will faded out within about 300 milliseconds).

Testing Splash Screen

If your application is lightweight and simple, it will launch very fast, and with similar speed will appear and disappear splash screen.

As soon as splash screen disappearing after `Application.Startup` method completed, you can simulate application launch delay by following this steps:

1. Open **App.xaml.cs** file
2. Add *using* namespace `using System.Threading;`
3. Override `OnStartup` method and add `Thread.Sleep(3000);` inside it:



Code should look like:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading;
using System.Windows;
```

```
namespace WpfApplication1
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            Thread.Sleep(3000);
        }
    }
}
```

4. Run the application. Now it will be launch for about 3 seconds longer, so you'll have more time to test your splash screen.

Creating custom splash screen window

WPF does not support displaying anything other than an image as a splash screen out-of-the-box, so we'll need to create a `Window` which will serve as a splash screen. We're assuming that we've already created a project containing `MainWindow` class, which is to be the application main window.

First off we add a `SplashScreenWindow` window to our project:

```
<Window x:Class="SplashScreenExample.SplashScreenWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        WindowStartupLocation="CenterScreen"
        WindowStyle="None"
        AllowsTransparency="True"
        Height="30"
        Width="200">
    <Grid>
        <ProgressBar IsIndeterminate="True" />
        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Center">Loading...</TextBlock>
    </Grid>
</Window>
```

Then we override the `Application.OnStartup` method to show the splash screen, do some work and finally show the main window (***App.xaml.cs***):

```
public partial class App
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        //initialize the splash screen and set it as the application main window
        var splashScreen = new SplashScreenWindow();
        this.MainWindow = splashScreen;
        splashScreen.Show();
```

```
        //in order to ensure the UI stays responsive, we need to
        //do the work on a different thread
        Task.Factory.StartNew(() =>
        {
            //simulate some work being done
            System.Threading.Thread.Sleep(3000);

            //since we're not on the UI thread
            //once we're done we need to use the Dispatcher
            //to create and show the main window
            this.Dispatcher.Invoke(() =>
            {
                //initialize the main window, set it as the application main window
                //and close the splash screen
                var mainWindow = new MainWindow();
                this.MainWindow = mainWindow;
                mainWindow.Show();
                splashScreen.Close();
            });
        });
    }
}
```

Lastly we need to take care of the default mechanism which shows the `MainWindow` on application startup. All we need to do is to remove the `StartupUri="MainWindow.xaml"` attribute from the root `Application` tag in ***App.xaml*** file.

Creating splash screen window with progress reporting

WPF does not support displaying anything other than an image as a splash screen out-of-the-box, so we'll need to create a `Window` which will serve as a splash screen. We're assuming that we've already created a project containing `MainWindow` class, which is to be the application main window.

First off we add a `SplashScreenWindow` window to our project:

```
<Window x:Class="SplashScreenExample.SplashScreenWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        WindowStartupLocation="CenterScreen"
        WindowStyle="None"
        AllowsTransparency="True"
        Height="30"
        Width="200">
    <Grid>
        <ProgressBar x:Name="progressBar" />
        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Center">Loading...</TextBlock>
    </Grid>
</Window>
```

Then we expose a property on the `SplashScreenWindow` class so that we can easily update the current progress value (***SplashScreenWindow.xaml.cs***):

```
public partial class SplashScreenWindow : Window
```

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

    public double Progress
    {
        get { return progressBar.Value; }
        set { progressBar.Value = value; }
    }
}
```

Next we override the `Application.OnStartup` method to show the splash screen, do some work and finally show the main window (***App.xaml.cs***):

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        //initialize the splash screen and set it as the application main window
        var splashScreen = new SplashScreenWindow();
        this.MainWindow = splashScreen;
        splashScreen.Show();

        //in order to ensure the UI stays responsive, we need to
        //do the work on a different thread
        Task.Factory.StartNew(() =>
        {
            //we need to do the work in batches so that we can report progress
            for (int i = 1; i <= 100; i++)
            {
                //simulate a part of work being done
                System.Threading.Thread.Sleep(30);

                //because we're not on the UI thread, we need to use the Dispatcher
                //associated with the splash screen to update the progress bar
                splashScreen.Dispatcher.Invoke(() => splashScreen.Progress = i);
            }

            //once we're done we need to use the Dispatcher
            //to create and show the main window
            this.Dispatcher.Invoke(() =>
            {
                //initialize the main window, set it as the application main window
                //and close the splash screen
                var mainWindow = new MainWindow();
                this.MainWindow = mainWindow;
                mainWindow.Show();
                splashScreen.Close();
            });
        });
    }
}
```

Lastly we need to take care of the default mechanism which shows the `MainWindow` on application startup. All we need to do is to remove the `StartupUri="MainWindow.xaml"` attribute from the root

`Application` tag in ***App.xaml*** file.

Read Creating Splash Screen in WPF online: https://riptutorial.com/wpf/topic/3948/creating-splash-screen-in-wpf

Chapter 6: Dependency Properties

Introduction

Dependency Properties are a type of property that extend out a CLR property. Whereas a CLR property is read directly from a member of your class, a Dependency Property will be dynamically resolved when calling the GetValue() method that your object gains via inheritance from the base DependencyObject class.

This section will break down Dependency Properties and explain their usage both conceptually and through code examples.

Syntax

- DependencyProperty.Register(string name, Type propertyType, Type ownerType)
- DependencyProperty.Register(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.Register(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterAttached(string name, Type propertyType, Type ownerType)
- DependencyProperty.RegisterAttached(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterAttached(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterReadOnly(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterReadOnly(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)
- DependencyProperty.RegisterAttachedReadOnly(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata)
- DependencyProperty.RegisterAttachedReadOnly(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback)

Parameters

Parameter	Details
name	The `String` representation of the property's name
propertyType	The `Type` of the property, e.g. `typeof(int)`
ownerType	The `Type` of the class in which the property is being defined, e.g. `typeof(MyControl)` or `typeof(MyAttachedProperties)`.
typeMetadata	Instance of `System.Windows.PropertyMetadata` (or one of its subclasses)

Parameter	Details
	that defines default values, property changed callbacks, `FrameworkPropertyMetadata` allows for defining binding options like `System.Windows.Data.BindingMode.TwoWay`.
validateValueCallback	Custom callback that returns true if the property's new value is valid, otherwise false.

Examples

Standard dependency properties

When to use

Virtually all WPF controls make heavy use of dependency properties. A dependency property allows for the use of many WPF features that are not possible with standard CLR properties alone, including but not limited to support for styles, animations, data binding, value inheritance, and change notifications.

The `TextBox.Text` property is a simple example of where a standard dependency property is needed. Here, data binding wouldn't be possible if `Text` was a standard CLR property.

```
<TextBox Text="{Binding FirstName}" />
```

How to define

Dependency properties can only be defined in classes derived from `DependencyObject`, such as `FrameworkElement`, `Control`, etc.

One of the fastest ways to create a standard dependency property without having to remember the syntax is to use the "propdp" snippet by typing `propdp` and then pressing `Tab`. A code snippet will be inserted that can then be modified to suit your needs:

```
public class MyControl : Control
{
    public int MyProperty
    {
        get { return (int)GetValue(MyPropertyProperty); }
        set { SetValue(MyPropertyProperty, value); }
    }

    // Using a DependencyProperty as the backing store for MyProperty.
    // This enables animation, styling, binding, etc...
    public static readonly DependencyProperty MyPropertyProperty =
        DependencyProperty.Register("MyProperty", typeof(int), typeof(MyControl),
            new PropertyMetadata(0));
```

```
}
```

You should `Tab` through the different parts of the code snippet to make the necessary changes, including updating the property name, property type, containing class type, and the default value.

Important conventions

There are a few important conventions/rules to follow here:

1. **Create a CLR property for the dependency property.** This property is used in your object's code-behind or by other consumers. It should invoke `GetValue` and `SetValue` so consumers don't have to.

2. **Name the dependency property correctly.** The `DependencyProperty` field should be `public static readonly`. It should have a name that corresponds with the CLR property name and ends with "Property", e.g. `Text` and `TextProperty`.

3. **Do not add additional logic to CLR property's setter.** The dependency property system (and XAML specifically) does not make use of the CLR property. If you want to perform an action when the property's value changes, you must provide a callback via `PropertyMetadata`:

```
public static readonly DependencyProperty MyPropertyProperty =
    DependencyProperty.Register("MyProperty", typeof(int), typeof(MyControl),
        new PropertyMetadata(0, MyPropertyChangedHandler));

private static void MyPropertyChangedHandler(DependencyObject sender,
DependencyPropertyChangedEventArgs args)
{
    // Use args.OldValue and args.NewValue here as needed.
    // sender is the object whose property changed.
    // Some unboxing required.
}
```

Binding mode

To eliminate the need for specifying `Mode=TwoWay` in bindings (akin to the behavior of `TextBox.Text`) update the code to use `FrameworkPropertyMetadata` instead of `PropertyMetadata` and specify the appropriate flag:

```
public static readonly DependencyProperty MyPropertyProperty =
    DependencyProperty.Register("MyProperty", typeof(int), typeof(MyControl),
        new FrameworkPropertyMetadata(0,
FrameworkPropertyMetadataOptions.BindsTwoWayByDefault));
```

Attached dependency properties

When to use

An attached property is a dependency property that can be applied to any `DependencyObject` to enhance the behavior of various controls or services that are aware of the property's existence.

Some use cases for attached properties include:

1. Having a parent element iterate through its children and act upon the children in a certain way. For example, the `Grid` control uses the `Grid.Row`, `Grid.Column`, `Grid.RowSpan`, and `Grid.ColumnSpan` attached properties to arrange elements into rows and columns.
2. Adding visuals to existing controls with custom templates, e.g adding watermarks to empty text boxes app-wide without having to subclass `TextBox`.
3. Providing a generic service or feature to some or all existing controls, e.g. `ToolTipService` or `FocusManager`. These are commonly referred to as *attached behaviors*.
4. When inheritance down the visual tree is required, e.g. similar to the behavior of `DataContext`.

This further demonstrates what is happening in the `Grid` use case:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Label Grid.Column="0" Content="Your Name:" />
    <TextBox Grid.Column="1" Text="{Binding FirstName}" />
</Grid>
```

`Grid.Column` is not a property that exists on either `Label` or `TextBox`. Rather, the `Grid` control looks through its child elements and arranges them according to the values of the attached properties.

How to define

We'll continue to use `Grid` for this example. The definition of `Grid.Column` is shown below, but the `DependencyPropertyChangedEventHandler` is excluded for brevity.

```
public static readonly DependencyProperty RowProperty =
    DependencyProperty.RegisterAttached("Row", typeof(int), typeof(Grid),
        new FrameworkPropertyMetadata(0, ...));

public static void SetRow(UIElement element, int value)
{
    if (element == null)
        throw new ArgumentNullException("element");

    element.SetValue(RowProperty, value);
}

public static int GetRow(UIElement element)
{
```

```
    if (element == null)
        throw new ArgumentNullException("element");

    return ((int)element.GetValue(RowProperty));
}
```

Because the attached properties can be attached to a wide variety of items, they cannot be implemented as CLR properties. A pair of static methods is introduced instead.

Hence, in contrast to standard dependency properties, attached properties can also be defined in classes that are not derived from `DependencyObject`.

The same naming conventions that apply to regular dependency properties also apply here: the dependency property `RowProperty` has the corresponding methods `GetRow` and `SetRow`.

Caveats

As documented on MSDN:

> Although property value inheritance might appear to work for nonattached dependency properties, the inheritance behavior for a nonattached property through certain element boundaries in the run-time tree is undefined. Always use RegisterAttached to register properties where you specify Inherits in the metadata.

Read-only dependency properties

When to use

A read-only dependency property is similar to a normal dependency property, but it is structured to not allow having its value set from outside the control. This works well if you have a property that is purely informational for consumers, e.g. `IsMouseOver` or `IsKeyboardFocusWithin`.

How to define

Just like standard dependency properties, a read-only dependency property must be defined on a class that derives from `DependencyObject`.

```
public class MyControl : Control
{
    private static readonly DependencyPropertyKey MyPropertyPropertyKey =
        DependencyProperty.RegisterReadOnly("MyProperty", typeof(int), typeof(MyControl),
            new FrameworkPropertyMetadata(0));

    public static readonly DependencyProperty MyPropertyProperty =
MyPropertyPropertyKey.DependencyProperty;

    public int MyProperty
```

```
    {
        get { return (int)GetValue(MyPropertyProperty); }
        private set { SetValue(MyPropertyPropertyKey, value); }
    }
}
```

The same conventions that apply to regular dependency properties also apply here, but with two key differences:

1. The `DependencyProperty` is sourced from a `private DependencyPropertyKey`.
2. The CLR property setter is `protected` or `private` instead of `public`.

Note that the setter passes `MyPropertyPropertyKey` and not `MyPropertyProperty` to the `SetValue` method. Because the property was defined read-only, any attempt to use `SetValue` on the property must be used with overload that receives `DependencyPropertyKey`; otherwise, an `InvalidOperationException` will be thrown.

Read Dependency Properties online: https://riptutorial.com/wpf/topic/2914/dependency-properties

Chapter 7: Grid control

Examples

A simple Grid

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Column="1" Text="abc"/>
    <TextBlock Grid.Row="1" Grid.Column="1" Text="def"/>
</Grid>
```

Rows and columns are defined adding `RowDefinition` and `ColumnDefinition` elements to the corresponding collections.

There can be an abitrary amount of children in the `Grid`. To specify which row or column a child is to be placed in the attached properties `Grid.Row` and `Grid.Column` are used. Row and column numbers are zero based. If no row or column is set it defaults to `0`.

Children placed in the same row and column are drawn in order of definition. So the child defined last will be draw above the child defined before.

Grid children spanning multiple rows/columns

By using the `Grid.RowSpan` and `Grid.ColumnSpan` attached properties, children of a `Grid` can span multiple rows or columns. In the following example the second `TextBlock` will span the second and third column of the `Grid`.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Column="2" Text="abc"/>
    <TextBlock Grid.Column="1" Grid.ColumnSpan="2" Text="def"/>
</Grid>
```

Syncing rows or columns of multiple Grids

The row heigths or column widths of multiple `Grid`s can be synchronized by setting a common `SharedSizeGroup` on the rows or columns to synchronize. Then a parent control somewhere up in

the tree above the `Grid`s needs to have the attached property `Grid.IsSharedSizeScope` set to `True`.

```
<StackPanel Grid.IsSharedSizeScope="True">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" SharedSizeGroup="MyGroup"/>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        [...]
    </Grid>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" SharedSizeGroup="MyGroup"/>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        [...]
    </Grid>
</StackPanel>
```

In this example the first column of both `Grid`s will always have the same width, also when one of them is resized by its content.

Read Grid control online: https://riptutorial.com/wpf/topic/6483/grid-control

Chapter 8: Introduction to WPF Data Binding

Syntax

- {Binding PropertyName} *is equivalent to* {Binding Path=PropertyName}
- {Binding Path=SomeProperty.SomeOtherProperty.YetAnotherProperty}
- {Binding Path=SomeListProperty[1]}

Parameters

Parameter	Details
Path	Specifies the path to bind to. If unspecified, binds to the DataContext itself.
UpdateSourceTrigger	Specifies when the binding source has its value updated. Defaults to `LostFocus`. Most used value is `PropertyChanged`.
Mode	Typically `OneWay` or `TwoWay`. If unspecified by the binding, it defaults to `OneWay` unless the binding target requests it to be `TwoWay`. An error occurs when `TwoWay` is used to bind to a readonly property, e.g. `OneWay` must be explicitly set when binding a readonly string property to `TextBox.Text`.
Source	Allows for using a `StaticResource` as a binding source instead of the current DataContext.
RelativeSource	Allows for using another XAML element as a binding source instead of the current DataContext.
ElementName	Allows for using a named XAML element as a binding source instead of the current DataContext.
FallbackValue	If the binding fails, this value is provided to the binding target.
TargetNullValue	If the binding source value is `null`, this value is provided to the binding target.
Converter	Specifies the converter `StaticResource` that is used to convert the binding's value, e.g. convert a boolean to a `Visibility` enum item.
ConverterParameter	Specifies an optional parameter to be provided to the converter. This value must be static and cannot be bound.
StringFormat	Specifies a format string to be used when displaying the bound value.

Parameter	Details
Delay	(WPF 4.5+) Specifies a Delay in `milliseconds` for the binding to update the `BindingSource` in the `ViewModel`. This must be used with `Mode=TwoWay` and `UpdateSourceTrigger=PropertyChanged` to take effect.

Remarks

UpdateSourceTrigger

By default, WPF updates the binding source when the control loses focus. However, if there is only one control that can get focus -- something that's common in examples -- you will need to specify `UpdateSourceTrigger=PropertyChanged` for the updates to work.

You will want want to use `PropertyChanged` as the trigger on many two-way bindings unless updating the binding source on every keystroke is costly or live data validation is undesirable.

Using `LostFocus` has an unfortunate side effect: pressing enter to submit a form using a button marked `IsDefault` does not update the property backing your binding, effectively undoing your changes. Fortunately, some workarounds exist.

Please also note that, unlike UWP, WPF (4.5+) also has the `Delay` property in bindings, wich might just be enough for some Bindings with local-only or simple minor intelligence settings, like some `TextBox` validations.

Examples

Convert a boolean to visibility value

This example hides the red box (border) if the checkbox is not checked by making use of an `IValueConverter`.

Note: The `BooleanToVisibilityConverter` used in the example below is a built-in value converter, located in the System.Windows.Controls namespace.

XAML:

```
<Window x:Class="StackOverflowDataBindingExample.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">
    <Window.Resources>
        <BooleanToVisibilityConverter x:Key="VisibleIfTrueConverter" />
    </Window.Resources>
    <StackPanel>
        <CheckBox x:Name="MyCheckBox"
                  IsChecked="True" />
        <Border Background="Red" Width="20" Height="20"
                Visibility="{Binding Path=IsChecked,ElementName=MyCheckBox,
```

```
Converter={StaticResource VisibleIfTrueConverter}}" />
    </StackPanel>
</Window>
```

Defining the DataContext

In order to work with bindings in WPF, you need to define a **DataContext**. The DataContext tells bindings where to get their data from by default.

```
<Window x:Class="StackOverflowDataBindingExample.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:StackOverflowDataBindingExample"
        xmlns:vm="clr-namespace:StackOverflowDataBindingExample.ViewModels"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Window.DataContext>
        <vm:HelloWorldViewModel />
    </Window.DataContext>
    ...
</Window>
```

You can also set the DataContext through code-behind, but it is worth noting that XAML IntelliSense is somewhat picky: a strongly-typed DataContext must be set in XAML for IntelliSense to suggest properties available for binding.

```
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        DataContext = new HelloWorldViewModel();
    }
}
```

While there are frameworks to help you define your DataContext in a more flexible way (e.g. MVVM Light has a viewmodel locator that uses inversion of control), we use the quick and dirty method for the purposes of this tutorial.

You can define a DataContext for pretty much any visual element in WPF. The DataContext is generally inherited from ancestors in the visual tree unless it has been explicitly overridden, e.g. inside a ContentPresenter.

Implementing INotifyPropertyChanged

`INotifyPropertyChanged` is an interface used by binding sources (i.e. the DataContext) to let the user interface or other components know that a property has been changed. WPF automatically updates the UI for you when it sees the `PropertyChanged` event raised. It is desirable to have this

interface implemented on a base class that all of your viewmodels can inherit from.

In C# 6, this is all you need:

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void NotifyPropertyChanged([CallerMemberName] string name = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
    }
}
```

This allows you to invoke `NotifyPropertyChanged` in two different ways:

1. `NotifyPropertyChanged()`, which will raise the event for the setter that invokes it, thanks to the attribute CallerMemberName.
2. `NotifyPropertyChanged(nameof(SomeOtherProperty))`, which will raise the event for SomeOtherProperty.

For .NET 4.5 and above using C# 5.0, this can be used instead:

```
public abstract class ViewModelBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void NotifyPropertyChanged([CallerMemberName] string name = null)
    {
        var handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(name));
        }
    }
}
```

In versions of .NET prior to 4.5, you have to settle for property names as string constants or a solution using expressions.

Note: It is possible to bind to a property of a "plain old C# object" (POCO) that does not implement `INotifyPropertyChanged` and observe that the bindings work better than expected. This is a hidden feature in .NET and should probably be avoided. Especially as it will cause memory leaks when the binding's `Mode` is not `OneTime` (see here).

Why does the binding update without implementing INotifyPropertyChanged?

Bind to property of another named element

You can bind to a property on a named element, but the named element must be in scope.

```
<StackPanel>
```

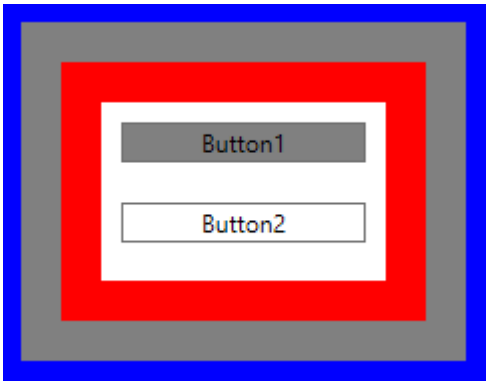
```
    <CheckBox x:Name="MyCheckBox" IsChecked="True" />
    <TextBlock Text="{Binding IsChecked, ElementName=MyCheckBox}" />
</StackPanel>
```

Bind to property of an ancestor

You can bind to a property of an ancestor in the visual tree by using a `RelativeSource` binding. The nearest control higher in the visual tree which has the same type or is derived from the type you specify will be used as the binding's source:

```
<Grid Background="Blue">
    <Grid Background="Gray" Margin="10">
        <Border Background="Red" Margin="20">
            <StackPanel Background="White" Margin="20">
                <Button Margin="10" Content="Button1" Background="{Binding Background,
RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type Grid}}}" />
                <Button Margin="10" Content="Button2" Background="{Binding Background,
RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type FrameworkElement}}}" />
            </StackPanel>
        </Border>
    </Grid>
</Grid>
```

In this example, *Button1* has a gray background because the closest `Grid` ancestor has a gray background. *Button2* has a white background because the closest ancestor derived from `FrameworkElement` is the white `StackPanel`.



Binding multiple values with a MultiBinding

The MultiBinding allows binding multiple values to the same property. In the following example multiple values are bound to the Text property of a Textbox and formatted using the StringFormat property.

```
<TextBlock>
    <TextBlock.Text>
        <MultiBinding StringFormat="{}{0} {1}">
            <Binding Path="User.Forename"/>
            <Binding Path="User.Surname"/>
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>
```

Apart from `StringFormat`, an `IMultiValueConverter`could also be used to convert the values from the Bindings to one value for the MultiBinding's target.

However, MultiBindings cannot be nested.

Read Introduction to WPF Data Binding online: https://riptutorial.com/wpf/topic/2236/introduction-to-wpf-data-binding

Chapter 9: Markup Extensions

Parameters

Method	Description
ProvideValue	MarkupExtension class has only one method that should be overridden, XAML parser then uses the value provided by this method to evaluate the result of markup extension.

Remarks

A markup extension can be implemented to provide values for properties in an attribute usage, properties in a property element usage, or both.

When used to provide an attribute value, the syntax that distinguishes a markup extension sequence to a XAML processor is the presence of the opening and closing curly braces ({ and }). The type of markup extension is then identified by the string token immediately following the opening curly brace.

When used in property element syntax, a markup extension is visually the same as any other element used to provide a property element value: a XAML element declaration that references the markup extension class as an element, enclosed within angle brackets (<>).

For more info visit https://msdn.microsoft.com/en-us/library/ms747254(v=vs.110).aspx

Examples

Markup Extension used with IValueConverter

One of the best uses for markup extensions is for easier usage of IValueConverter. In the sample below BoolToVisibilityConverter is a value converter but since it's instance independent it can be used without the normal hasles of a value converter with the help of markup extension. In XAML just use

```
Visibility="{Binding [BoolProperty], Converter={[xmlns]:BoolToVisibilityConverter}}"
```

and you can set item visibility to bool value.

```
public class BoolToVisibilityConverter : MarkupExtension, IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            if (value is bool)
```

```
                return (bool)value ? Visibility.Visible : Visibility.Collapsed;
            else
                return Visibility.Collapsed;
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
        {
            if (value is Visibility)
            {
                if ((Visibility)value == Visibility.Visible)
                    return true;
                else
                    return false;
            }
            else
                return false;
        }

        public override object ProvideValue(IServiceProvider serviceProvider)
        {
            return this;
        }
    }
```

XAML-Defined markup extensions

There are four predefined markup extensions in XAML:

`x:Type` supplies the Type object for the named type. This facility is used most frequently in styles and templates.

```
<object property="{x:Type prefix:typeNameValue}" .../>
```

`x:Static` produces static values. The values come from value-type code entities that are not directly the type of a target property's value, but can be evaluated to that type.

```
<object property="{x:Static prefix:typeName.staticMemberName}" .../>
```

`x:Null` specifies null as a value for a property and can be used either for attributes or property element values.

```
<object property="{x:Null}" .../>
```

`x:Array` provides support for the creation of general arrays in XAML syntax, for cases where the collection support provided by WPF base elements and control models is deliberately not used.

```
<x:Array Type="typeName">
  arrayContents
</x:Array>
```

Read Markup Extensions online: https://riptutorial.com/wpf/topic/6619/markup-extensions

Chapter 10: MVVM in WPF

Remarks

Models and View-Models

The definition of a model is often hotly debated, and the line between a model and a view-model can be blurred. Some prefer not to "pollute" their models with the `INotifyPropertyChanged` interface, and instead duplicate the model properties in the view-model, which *does* implement this interface. Like many things in software development, there is no right or wrong answer. Be pragmatic and do whatever feels right.

View Separation

The intention of MVVM is to separate those three distinct areas - Model, view-model, and View. While it's acceptable for the view to access the view-model (VM) and (indirectly) the model, the most important rule with MVVM is that the VM should have no access to the view or its controls. The VM should expose everything that the view needs, via public properties. The VM should not directly expose or manipulate UI controls such as `TextBox`, `Button`, etc.

In some cases, this strict separation can be difficult to work with, especially if you need to get some complex UI functionality up and running. Here, it's perfectly acceptable to resort to using events and event handlers in the view's "code-behind" file. If it's purely UI functionality then by all means utilise events in the view. It's also acceptable for these event handlers to call public methods on the VM instance - just don't go passing it references to UI controls or anything like that.

RelayCommand

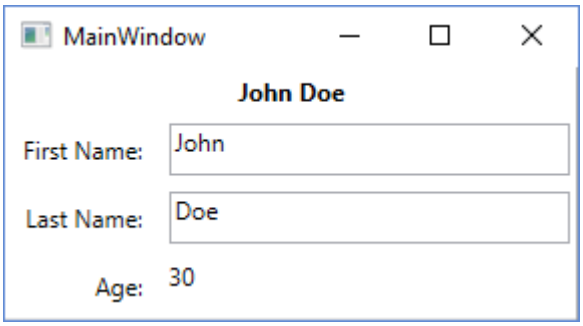
Unfortunately the `RelayCommand` class used in this example isn't part of the WPF framework (it should have been!), but you'll find it in almost every WPF developer's tool box. A quick search online will reveal plenty of code snippets that you can lift, to create your own.

A useful alternative to `RelayCommand` is `ActionCommand` which is provided as part of `Microsoft.Expression.Interactivity.Core` which provides comparable functionality.

Examples

Basic MVVM example using WPF and C#

This a Basic example for using the MVVM model in a windows desktop application, using WPF and C#. The example code implements a simple "user info" dialog.



The View

The XAML

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

    <TextBlock Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="2" Margin="4" Text="{Binding
FullName}" HorizontalAlignment="Center" FontWeight="Bold"/>

    <Label Grid.Column="0" Grid.Row="1" Margin="4" Content="First Name:"
HorizontalAlignment="Right"/>
    <!-- UpdateSourceTrigger=PropertyChanged makes sure that changes in the TextBoxes are
immediately applied to the model. -->
    <TextBox Grid.Column="1" Grid.Row="1" Margin="4" Text="{Binding FirstName,
UpdateSourceTrigger=PropertyChanged}" HorizontalAlignment="Left" Width="200"/>

    <Label Grid.Column="0" Grid.Row="2" Margin="4" Content="Last Name:"
HorizontalAlignment="Right"/>
    <TextBox Grid.Column="1" Grid.Row="2" Margin="4" Text="{Binding LastName,
UpdateSourceTrigger=PropertyChanged}" HorizontalAlignment="Left"  Width="200"/>

    <Label Grid.Column="0" Grid.Row="3" Margin="4" Content="Age:"
HorizontalAlignment="Right"/>
    <TextBlock Grid.Column="1" Grid.Row="3" Margin="4" Text="{Binding Age}"
HorizontalAlignment="Left"/>

</Grid>
```

and the code behind

```
public partial class MainWindow : Window
{
    private readonly MyViewModel _viewModel;

    public MainWindow() {
        InitializeComponent();
        _viewModel = new MyViewModel();
        // The DataContext serves as the starting point of Binding Paths
```

```
        DataContext = _viewModel;
    }
}
```

The View Model

```
// INotifyPropertyChanged notifies the View of property changes, so that Bindings are updated.
sealed class MyViewModel : INotifyPropertyChanged
{
    private User user;

    public string FirstName {
        get {return user.FirstName;}
        set {
            if(user.FirstName != value) {
                user.FirstName = value;
                OnPropertyChange("FirstName");
                // If the first name has changed, the FullName property needs to be udpated as
well.
                OnPropertyChange("FullName");
            }
        }
    }

    public string LastName {
        get { return user.LastName; }
        set {
            if (user.LastName != value) {
                user.LastName = value;
                OnPropertyChange("LastName");
                // If the first name has changed, the FullName property needs to be udpated as
well.
                OnPropertyChange("FullName");
            }
        }
    }

    // This property is an example of how model properties can be presented differently to the
View.
    // In this case, we transform the birth date to the user's age, which is read only.
    public int Age {
        get {
            DateTime today = DateTime.Today;
            int age = today.Year - user.BirthDate.Year;
            if (user.BirthDate > today.AddYears(-age)) age--;
            return age;
        }
    }

    // This property is just for display purposes and is a composition of existing data.
    public string FullName {
        get { return FirstName + " " + LastName; }
    }

    public MyViewModel() {
        user = new User {
            FirstName = "John",
            LastName = "Doe",
            BirthDate = DateTime.Now.AddYears(-30)
        };
```

```
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChange(string propertyName) {
        if(PropertyChanged != null) {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

The Model

```
sealed class User
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public DateTime BirthDate { get; set; }
}
```

The View-Model

The view-model is the "VM" in MV**VM**. This is a class that acts as a go-between, exposes the model(s) to the user interface (view), and handling requests from the view, such as commands raised by button clicks. Here is a basic view-model:

```
public class CustomerEditViewModel
{
    /// <summary>
    /// The customer to edit.
    /// </summary>
    public Customer CustomerToEdit { get; set; }

    /// <summary>
    /// The "apply changes" command
    /// </summary>
    public ICommand ApplyChangesCommand { get; private set; }

    /// <summary>
    /// Constructor
    /// </summary>
    public CustomerEditViewModel()
    {
        CustomerToEdit = new Customer
                         {
                             Forename = "John",
                             Surname = "Smith"
                         };

        ApplyChangesCommand = new RelayCommand(
            o => ExecuteApplyChangesCommand(),
            o => CustomerToEdit.IsValid);
    }

    /// <summary>
```

```
    /// Executes the "apply changes" command.
    /// </summary>
    private void ExecuteApplyChangesCommand()
    {
        // E.g. save your customer to database
    }
}
```

The constructor creates a `Customer` model object and assigns it to the `CustomerToEdit` property, so that it's visible to the view.

The constructor also creates a `RelayCommand` object and assigns it to the `ApplyChangesCommand` property, again making it visible to the view. WPF commands are used to handle requests from the view, such as button or menu item clicks.

The `RelayCommand` takes two parameters - the first is the delegate that gets called when the command is executed (e.g. in response to a button click). The second parameter is a delegate that returns a boolean value indicating whether the command can execute; in this example it's wired up to the customer object's `IsValid` property. When this returns false, it disables the button or menu item that is bound to this command (other controls may behave differently). This is a simple but effective feature, avoiding the need to write code to enable or disable controls based on different conditions.

If you do get this example up and running, try emptying out one of the `TextBox`es (to place the `Customer` model into an invalid state). When you tab away from the `TextBox` you should find that the "Apply" button becomes disabled.

Remark on Customer Creation

The view-model doesn't implement `INotifyPropertyChanged` (INPC). This means that if a different `Customer` object was to be assigned to the `CustomerToEdit` property then the view's controls wouldn't change to reflect the new object - the `TextBox`es would still contain the forename and surname of the previous customer.

The example code works because the `Customer` is created in the view-model's constructor, before it gets assigned to the view's `DataContext` (at which point the bindings are wired up). In a real-world application you might be retrieving customers from a database in methods other than the constructor. To support this, the VM should implement INPC, and the `CustomerToEdit` property should be changed to use the "extended" getter and setter pattern that you see in the example Model code, raising the `PropertyChanged` event in the setter.

The view-model's `ApplyChangesCommand` doesn't need to implement INPC as the command is very unlikely to change. You *would* need to implement this pattern if you were creating the command somewhere other than the constructor, for example some kind of `Initialize()` method.

The general rule is: implement INPC if the property is bound to any view controls *and* the property's value is able to change anywhere other than in the constructor. You don't need to implement INPC if the property value is only ever assigned in the constructor (and you'll save yourself some typing in the process).

The Model

The model is the first "M" in **M**VVM. The model is usually a class containing the data that you want to expose via some kind of user interface.

Here is a very simple model class exposing a couple of properties:-

```
public class Customer : INotifyPropertyChanged
{
    private string _forename;
    private string _surname;
    private bool _isValid;

    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// Customer forename.
    /// </summary>
    public string Forename
    {
        get
        {
            return _forename;
        }
        set
        {
            if (_forename != value)
            {
                _forename = value;
                OnPropertyChanged();
                SetIsValid();
            }
        }
    }

    /// <summary>
    /// Customer surname.
    /// </summary>
    public string Surname
    {
        get
        {
            return _surname;
        }
        set
        {
            if (_surname != value)
            {
                _surname = value;
                OnPropertyChanged();
                SetIsValid();
            }
        }
    }

    /// <summary>
    /// Indicates whether the model is in a valid state or not.
    /// </summary>
    public bool IsValid
```

```
    {
        get
        {
            return _isValid;
        }
        set
        {
            if (_isValid != value)
            {
                _isValid = value;
                OnPropertyChanged();
            }
        }
    }

    /// <summary>
    /// Sets the value of the IsValid property.
    /// </summary>
    private void SetIsValid()
    {
        IsValid = !string.IsNullOrEmpty(Forename) && !string.IsNullOrEmpty(Surname);
    }

    /// <summary>
    /// Raises the PropertyChanged event.
    /// </summary>
    /// <param name="propertyName">Name of the property.</param>
    private void OnPropertyChanged([CallerMemberName] string propertyName = "")
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

This class implements the `INotifyPropertyChanged` interface which exposes a `PropertyChanged` event. This event should be raised whenever one of the property values changes - you can see this in action in the above code. The `PropertyChanged` event is a key piece in the WPF data-binding mechanisms, as without it, the user interface would not be able to reflect the changes made to a property's value.

The model also contains a very simple validation routine that gets called from the property setters. It sets a public property indicating whether or not the model is in a valid state. I've included this functionality to demonstrate a "special" feature of WPF *commands*, which you'll see shortly. *The WPF framework provides a number of more sophisticated approaches to validation, but these are outside the scope of this article*.

The View

The View is the "V" in M**V**VM. This is your user interface. You can use the Visual Studio drag-and-drop designer, but most developers eventually end up coding the raw XAML - an experience similar to writing HTML.

Here is the XAML of a simple view to allow editing of a `Customer` model. Rather than create a new view, this can just be pasted into a WPF project's `MainWindow.xaml` file, in-between the `<Window ...>` and `</Window>` tags:-

```
<StackPanel Orientation="Vertical"
            VerticalAlignment="Top"
            Margin="20">
    <Label Content="Forename"/>
    <TextBox Text="{Binding CustomerToEdit.Forename}"/>

    <Label Content="Surname"/>
    <TextBox Text="{Binding CustomerToEdit.Surname}"/>

    <Button Content="Apply Changes"
            Command="{Binding ApplyChangesCommand}" />
</StackPanel>
```

This code creates a simple data entry form consisting of two `TextBox`es - one for the customer forename, and one for the surname. There is a `Label` above each `TextBox`, and an "Apply" `Button` at the bottom of the form.

Locate the first `TextBox` and look at it's `Text` property:

```
Text="{Binding CustomerToEdit.Forename}"
```

Rather than setting the `TextBox`'s text to a fixed value, this special curly brace syntax is instead binding the text to the "path" `CustomerToEdit.Forename`. What's this path relative to? It's the view's "data context" - in this case, our view-model. The binding path, as you may be able to figure out, is the view-model's `CustomerToEdit` property, which is of type `Customer` that in turn exposes a property called `Forename` - hence the "dotted" path notation.

Similarly, if you look at the `Button`'s XAML, it has a `Command` that is bound to the `ApplyChangesCommand` property of the view-model. That's all that's needed to wire up a button to the VM's command.

The DataContext

So how do you set the view-model to be the view's data context? One way is to set it in the view's "code-behind". Press F7 to see this code file, and add a line to the existing constructor to create an instance of the view-model and assign it to the window's `DataContext` property. It should end up looking like this:

```
    public MainWindow()
    {
        InitializeComponent();

        // Our new line:-
        DataContext = new CustomerEditViewModel();
    }
```

In real world systems, other approaches are often used to create the view model, such as dependency injection or MVVM frameworks.

Commanding in MVVM

Commands are used for handling `Events` in WPF while respecting the MVVM-Pattern.

A normal `EventHandler` would look like this (located in `Code-Behind`):

```
public MainWindow()
{
    _dataGrid.CollectionChanged += DataGrid_CollectionChanged;
}

private void DataGrid_CollectionChanged(object sender,
System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
        //Do what ever
}
```

No to do the same in MVVM we use `Commands`:

```
 <Button Command="{Binding Path=CmdStartExecution}" Content="Start" />
```

> I recommend to use some kind of prefix (`Cmd`) for your command properties, because you will mainly need them in xaml - that way they are easier to recognize.

Since it's MVVM you want to handle that Command (For `Button` "eq" `Button_Click`) in your `ViewModel` .

For that we basically need two things:

1. System.Windows.Input.ICommand
2. RelayCommand (for example taken from here.

A simple **example** could look like this:

```
private RelayCommand _commandStart;
public ICommand CmdStartExecution
{
   get
   {
      if(_commandStart == null)
      {
          _commandStart = new RelayCommand(param => Start(), param => CanStart());
      }
      return _commandStart;
   }
}

public void Start()
{
   //Do what ever
}

public bool CanStart()
{
    return (DateTime.Now.DayOfWeek == DayOfWeek.Monday); //Can only click that button on
mondays.
}
```

So what is this doing in detail:

The `ICommand` is what the `Control` in xaml is binding to. The `RelayCommand` will route your command to an `Action` (i.e call a `Method`). The Null-Check just ensures that each `Command` will only get initialized once (due to performance issues). If you've read the link for the `RelayCommand` above you may have noticed that `RelayCommand` has two overloads for it's constructor. `(Action<object> execute)` and `(Action<object> execute, Predicate<object> canExecute).`

That means you can (aditionally) add a second `Method` returning a `bool` to tell that `Control` wheather the "Event" can fire or not.

A good thing for that is that `Button`s for example will be `Enabled="false"` if the `Method` will return `false`

CommandParameters

```
<DataGrid x:Name="TicketsDataGrid">
    <DataGrid.InputBindings>
        <MouseBinding Gesture="LeftDoubleClick"
                      Command="{Binding CmdTicketClick}"
                      CommandParameter="{Binding ElementName=TicketsDataGrid,
                                                 Path=SelectedItem}" />
    </DataGrid.InputBindings>
<DataGrid />
```

In this example I want to pass the `DataGrid.SelectedItem` to the Click_Command in my ViewModel.

Your Method should look like this while the ICommand implementation itself stays as above.

```
private RelayCommand _commandTicketClick;

public ICommand CmdTicketClick
{
   get
   {
       if(_commandTicketClick == null)
       {
           _commandTicketClick = new RelayCommand(param => HandleUserClick(param));
       }
       return _commandTicketClick;
   }
}

private void HandleUserClick(object item)
{
    MyModelClass selectedItem = item as MyModelClass;
    if (selectedItem != null)
    {
        //Do sth. with that item
    }
}
```

Read MVVM in WPF online: https://riptutorial.com/wpf/topic/2134/mvvm-in-wpf

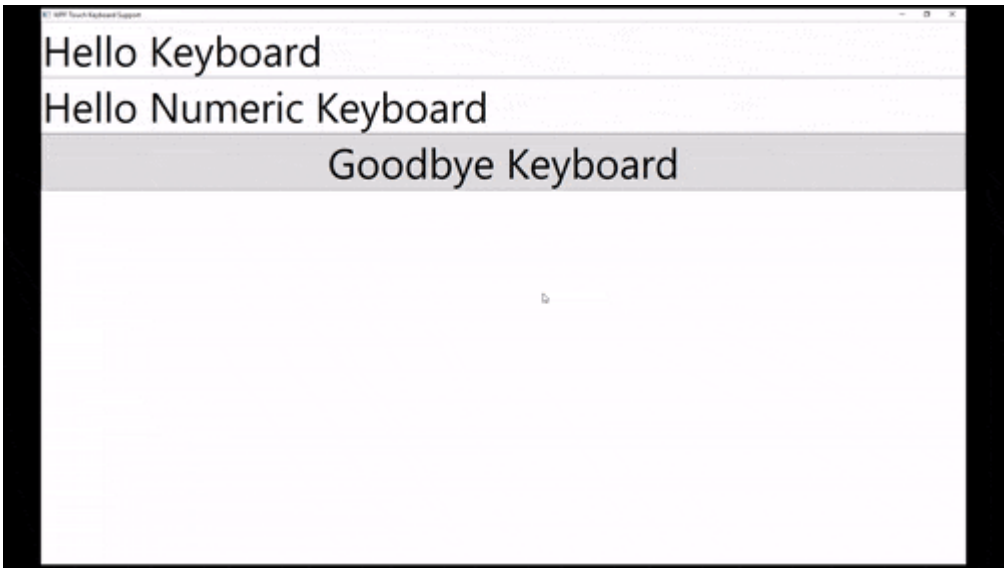
Chapter 11: Optimizing for touch interaction

Examples

Showing touch keyboard on Windows 8 and Windows 10

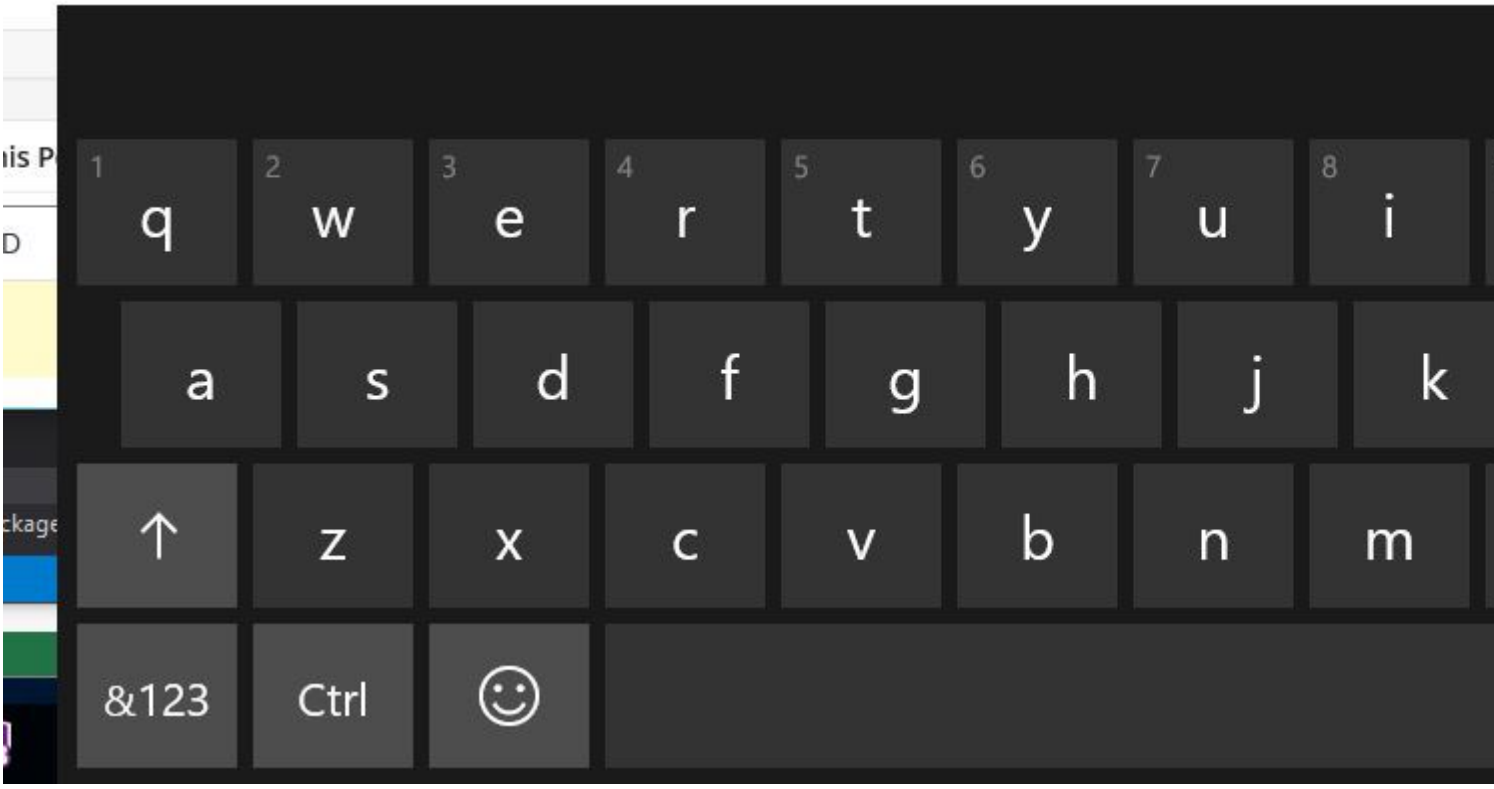
WPF apps targeting .NET Framework 4.6.2 and later

With WPF apps targeting .NET Framework 4.6.2 (and later), the soft keyboard is automatically invoked and dismissed without any additional steps required.



WPF apps targeting .NET Framework 4.6.1 and earlier

WPF is not primarily touch enabled, which means that when the user interacts with a WPF application on desktop, the app **will not automatically display the touch keyboard** when `TextBox` controls receive focus. This is an inconvenient behavior for users of tablets, forcing them to manually open the touch keyboard via the system task bar.



Workaround

The touch keyboard is actually a classic `exe` application which can be found on each Windows 8 and Windows 10 PC on the following path: `C:\Program Files\Common Files\Microsoft Shared\Ink\TabTip.exe`.

Based on this knowledge, you can create a custom control derived from `TextBox`, which listens on the `GotTouchCapture` event (this event is called when the control gets focus using touch) and **starts the touch keyboard's process**.

```
public class TouchEnabledTextBox : TextBox
{
    public TouchEnabledTextBox()
    {
        this.GotTouchCapture += TouchEnabledTextBox_GotTouchCapture;
    }

    private void TouchEnabledTextBox_GotTouchCapture(
       object sender,
       System.Windows.Input.TouchEventArgs e )
    {
        string touchKeyboardPath =
           @"C:\Program Files\Common Files\Microsoft Shared\Ink\TabTip.exe";
        Process.Start( touchKeyboardPath );
    }
}
```

You can improve this even further by caching the created process and then killing it after the control loses focus:

```
//added field
private Process _touchKeyboardProcess = null;

//replace Process.Start line from the previous listing with
_touchKeyboardProcess = Process.Start( touchKeyboardPath );
```

Now you can wire up the `LostFocus` event:

```
//add this at the end of TouchEnabledTextBox's constructor
this.LostFocus += TouchEnabledTextBox_LostFocus;

//add this method as a member method of the class
private void TouchEnabledTextBox_LostFocus( object sender, RoutedEventArgs eventArgs ){
   if ( _touchKeyboardProcess != null )
   {
      _touchKeyboardProcess.Kill();
      //nullify the instance pointing to the now-invalid process
      _touchKeyboardProcess = null;
   }
}
```

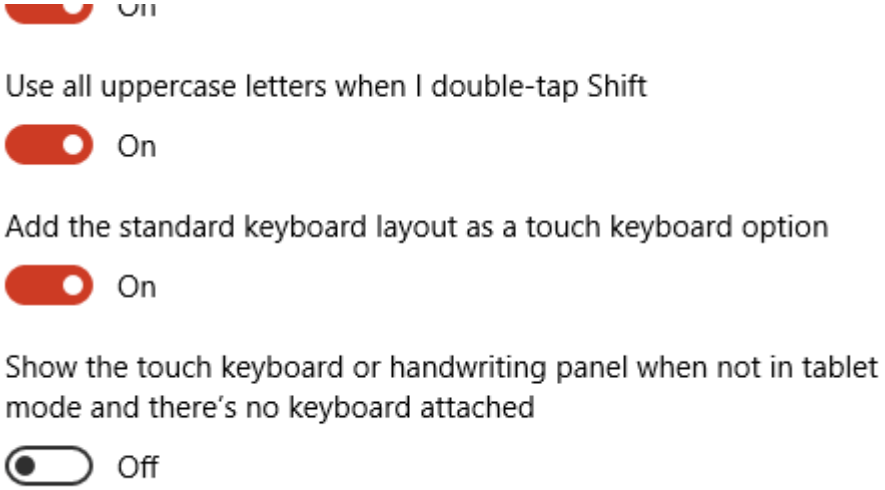
Note about the Tablet Mode in Windows 10

Windows 10 introduced a **tablet mode**, which simplifies interaction with the system when using the PC in touch-first manner. This mode, apart from other improvements, ensures, that the **touch keyboard is displayed automatically** even for classic Desktop apps including WPF apps.

Windows 10 Settings approach

In addition to the tablet mode, Windows 10 can automatically display the touch keyboard for classic apps even outside of the tablet mode. This behavior, which is disabled by default, can be enabled in the Settings app.

In the **Settings** app, go to the **Devices** category and select **Typing**. If you scroll all the way down, you can find the "Show the touch keyboard or handwriting panel when not in tablet mode and there's no keyboard attached" setting, which you can enable.



It is worth mentioning, that this setting is only visible on devices with touch capabilities.

Read Optimizing for touch interaction online: https://riptutorial.com/wpf/topic/6799/optimizing-for-touch-interaction

Chapter 12: Slider Binding: Update only on Drag Ended

Parameters

Parameter	Detail
Value (float)	The property bound to this Dependency Property will be updated whenever the user will cease dragging the slider

Remarks

- Make sure to reference the *System.Windows.Interactivity* assembly, so that the XAML Parser will recognize the *xmlns:i* declaration.
- Note that the *xmlns:b* statement matches the namespace where the behavior implementation resides
- Example assumes working knowledge of binding expressions and XAML.

Examples

Behavior Implementation

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Interactivity;

namespace MyBehaviorAssembly
{

public class SliderDragEndValueBehavior : Behavior<Slider>
{

    public static readonly DependencyProperty ValueProperty = DependencyProperty.Register(
        "Value", typeof (float), typeof (SliderDragEndValueBehavior), new
PropertyMetadata(default(float)));

    public float Value
    {
        get { return (float) GetValue(ValueProperty); }
        set { SetValue(ValueProperty, value); }
    }

    protected override void OnAttached()
    {
        RoutedEventHandler handler = AssociatedObject_DragCompleted;
        AssociatedObject.AddHandler(Thumb.DragCompletedEvent, handler);
    }
```

```
    private void AssociatedObject_DragCompleted(object sender, RoutedEventArgs e)
    {
        Value = (float) AssociatedObject.Value;
    }

    protected override void OnDetaching()
    {
        RoutedEventHandler handler = AssociatedObject_DragCompleted;
        AssociatedObject.RemoveHandler(Thumb.DragCompletedEvent, handler);
    }
}
}
```

XAML Usage

```
<UserControl x:Class="Example.View"
         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"
         xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"

         xmlns:b="MyBehaviorAssembly;assembly=MyBehaviorAssembly"

         mc:Ignorable="d"
         d:DesignHeight="200" d:DesignWidth="200"

         >
           <Slider>
            <i:Interaction.Behaviors>
                <b:SliderDragEndValueBehavior

                        Value="{Binding Value, Mode=OneWayToSource,
UpdateSourceTrigger=PropertyChanged}"

                        />
            </i:Interaction.Behaviors>
          </Slider>

</UserControl>
```

Read Slider Binding: Update only on Drag Ended online:
https://riptutorial.com/wpf/topic/6339/slider-binding--update-only-on-drag-ended

Chapter 13: Speech Synthesis

Introduction

In the `System.Speech` assembly, Microsoft has added **Speech Synthesis**, the ability to transform text into spoken words.

Syntax

1. SpeechSynthesizer speechSynthesizerObject = new SpeechSynthesizer();

   speechSynthesizerObject.Speak("Text to Speak");

Examples

Speech Synthesis Example - Hello World

```
using System;
using System.Speech.Synthesis;
using System.Windows;

namespace Stackoverflow.SpeechSynthesisExample
{
        public partial class SpeechSynthesisSample : Window
        {
                public SpeechSynthesisSample()
                {
                        InitializeComponent();
                        SpeechSynthesizer speechSynthesizer = new SpeechSynthesizer();
                        speechSynthesizer.Speak("Hello, world!");
                }
        }
}
```

Read Speech Synthesis online: https://riptutorial.com/wpf/topic/8368/speech-synthesis

Chapter 14: Styles in WPF

Remarks

Introductory remarks

In WPF, a **Style** defines the values of one or more dependency properties for a given visual element. Styles are used throughout the application to make the user interface more consistent (e.g. giving all dialog buttons a consistent size) and to make bulk changes easier (e.g. changing the width of all buttons.)

Styles are typically defined in a `ResourceDictionary` at a high level in the application (e.g. in *App.xaml* or in a theme) so it is available app-wide, but they may also be defined for a single element and its children, e.g. applying a style to all `TextBlock` elements inside a `StackPanel`.

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Margin" Value="5,5,5,0"/>
            <Setter Property="Background" Value="#FFF0F0F0"/>
            <Setter Property="Padding" Value="5"/>
        </Style>
    </StackPanel.Resources>

    <TextBlock Text="First Child"/>
    <TextBlock Text="Second Child"/>
    <TextBlock Text="Third Child"/>
</StackPanel>
```

Important notes

- The location where the style is defined affects where it is available.
- Forward references cannot be resolved by `StaticResource`. In other words, if you're defining a style that depends upon another style or resource in a resource dictionary, it must be defined after/below the resource upon which it depends.
- `StaticResource` is the recommended way to reference styles and other resources (for performance and behavioral reasons) unless you specifically require the use of `DynamicResource`, e.g. for themes that can be changed at runtime.

Resources

MSDN has thorough articles on styles and resources that have more depth than is possible to provide here.

- Resources Overview
- Styling and Templating
- Control Authoring Overview

Examples

Defining a named style

A named style requires the `x:Key` property to be set and applies only to elements that explicitly reference it by name:

```
<StackPanel>
    <StackPanel.Resources>
        <Style x:Key="MyTextBlockStyle" TargetType="TextBlock">
            <Setter Property="Background" Value="Yellow"/>
            <Setter Property="FontWeight" Value="Bold"/>
        </Style>
    </StackPanel.Resources>

    <TextBlock Text="Yellow and bold!" Style="{StaticResource MyTextBlockStyle}" />
    <TextBlock Text="Also yellow and bold!" Style="{DynamicResource MyTextBlockStyle}" />
    <TextBlock Text="Plain text." />
</StackPanel>
```

Defining an implicit style

An implicit style applies to all elements of a given type within scope. An implicit style can omit `x:Key` since it is implicitly the same as the style's `TargetType` property.

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Background" Value="Yellow"/>
            <Setter Property="FontWeight" Value="Bold"/>
        </Style>
    </StackPanel.Resources>

    <TextBlock Text="Yellow and bold!"  />
    <TextBlock Text="Also yellow and bold!" />
    <TextBlock Style="{x:Null}" Text="I'm not yellow or bold; I'm the control's default
style!" />
</StackPanel>
```

Inheriting from a style

It is common to need a base style that defines properties/values shared between multiple styles belonging to the same control, especially for something like `TextBlock`. This is accomplished by using the `BasedOn` property. Values are inherited and then can be overridden.

```
<Style x:Key="BaseTextBlockStyle" TargetType="TextBlock">
    <Setter Property="FontSize" Value="12"/>
    <Setter Property="Foreground" Value="#FFBBBBBB" />
```

```
    <Setter Property="FontFamily" Value="Arial" />
</Style>

<Style x:Key="WarningTextBlockStyle"
       TargetType="TextBlock"
       BasedOn="{StaticResource BaseTextBlockStyle">
    <Setter Property="Foreground" Value="Red"/>
    <Setter Property="FontWeight" Value="Bold" />
</Style>
```

In the above example, any `TextBlock` using the style `WarningTextBlockStyle` would be presented as 12px Arial in red and bold.

Because implicit styles have an implicit `x:Key` that matches their `TargetType`, you can inherit those as well:

```
<!-- Implicit -->
<Style TargetType="TextBlock">
    <Setter Property="FontSize" Value="12"/>
    <Setter Property="Foreground" Value="#FFBBBBBB" />
    <Setter Property="FontFamily" Value="Arial" />
</Style>

<Style x:Key="WarningTextBlockStyle"
       TargetType="TextBlock"
       BasedOn="{StaticResource {x:Type TextBlock}}">
    <Setter Property="Foreground" Value="Red"/>
    <Setter Property="FontWeight" Value="Bold" />
</Style>
```

Read Styles in WPF online: https://riptutorial.com/wpf/topic/4090/styles-in-wpf

Chapter 15: Supporting Video Streaming and Pixel Array Assignment to an Image Control

Parameters

Parameters	Details
PixelHeight (System.Int32)	The height of the image in units of image pixels
PixelWidth (System.Int32)	The width of the image in units of image pixels
PixelFormat (System.Windows.Media.PixelFormat)	The width of the image in units of image pixels
Pixels	Anything which implements IList<T>- including the C# byte array
DpiX	Specifies the horizontal Dpi - Optional
DpiY	Specifies the vertical Dpi - Optional

Remarks

- Make sure to reference the *System.Windows.Interactivity assembly*, so that the XAML Parser will recognize the *xmlns:i* declaration.
- Note that the *xmlns:b* statement matches the namespace where the behavior implementation resides
- Example assumes working knowledge of binding expressions and XAML.
- This behavior supports assigning pixels to an image in the form of a byte array - even though the Dependency Property type is specified as an *IList*. This works since the C# byte array implements the *IList*
  interface.
- Behavior achieves very high performance, and can be used for Video Streaming
- Do not assign the image's *Source* Dependency Property- bind to the *Pixels* Dependency Property instead
- The Pixels, PixelWidth, PixelHeight and PixelFormat properties must be assigned for the pixels to be rendered
- Order of Dependency Property assignment does not matter

Examples

Behavior Implementation

```
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Interactivity;
using System.Windows.Media;
using System.Windows.Media.Imaging;

namespace MyBehaviorAssembly
{
    public class PixelSupportBehavior : Behavior<Image>
    {

    public static readonly DependencyProperty PixelsProperty = DependencyProperty.Register(
        "Pixels", typeof (IList<byte>), typeof (PixelSupportBehavior), new
PropertyMetadata(default(IList<byte>),OnPixelsChanged));

    private static void OnPixelsChanged(DependencyObject d, DependencyPropertyChangedEventArgs
e)
    {
        var b = (PixelSupportBehavior) d;
        var pixels = (IList<byte>) e.NewValue;

        b.RenderPixels(pixels);
    }

    public IList<byte> Pixels
    {
        get { return (IList<byte>) GetValue(PixelsProperty); }
        set { SetValue(PixelsProperty, value); }
    }

    public static readonly DependencyProperty PixelFormatProperty =
DependencyProperty.Register(
        "PixelFormat", typeof (PixelFormat), typeof (PixelSupportBehavior), new
PropertyMetadata(PixelFormats.Default,OnPixelFormatChanged));


    private static void OnPixelFormatChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
    {
        var b = (PixelSupportBehavior) d;
        var pixelFormat = (PixelFormat) e.NewValue;

        if(pixelFormat==PixelFormats.Default)
            return;

        b._pixelFormat = pixelFormat;

        b.InitializeBufferIfAttached();
    }

    public PixelFormat PixelFormat
    {
        get { return (PixelFormat) GetValue(PixelFormatProperty); }
        set { SetValue(PixelFormatProperty, value); }
```

```
    }

    public static readonly DependencyProperty PixelWidthProperty =
DependencyProperty.Register(
        "PixelWidth", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(default(int),OnPixelWidthChanged));

    private static void OnPixelWidthChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
    {
        var b = (PixelSupportBehavior)d;
        var value = (int)e.NewValue;

        if(value<=0)
            return;

        b._pixelWidth = value;

        b.InitializeBufferIfAttached();
    }

    public int PixelWidth
    {
        get { return (int) GetValue(PixelWidthProperty); }
        set { SetValue(PixelWidthProperty, value); }
    }


    public static readonly DependencyProperty PixelHeightProperty =
DependencyProperty.Register(
        "PixelHeight", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(default(int),OnPixelHeightChanged));

    private static void OnPixelHeightChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
    {
        var b = (PixelSupportBehavior)d;
        var value = (int)e.NewValue;

        if (value <= 0)
            return;

        b._pixelHeight = value;

        b.InitializeBufferIfAttached();
    }

    public int PixelHeight
    {
        get { return (int) GetValue(PixelHeightProperty); }
        set { SetValue(PixelHeightProperty, value); }
    }

    public static readonly DependencyProperty DpiXProperty = DependencyProperty.Register(
        "DpiX", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(96,OnDpiXChanged));

    private static void OnDpiXChanged(DependencyObject d, DependencyPropertyChangedEventArgs
e)
    {
        var b = (PixelSupportBehavior)d;
```

```
        var value = (int)e.NewValue;

        if (value <= 0)
            return;

        b._dpiX = value;

        b.InitializeBufferIfAttached();
    }

    public int DpiX
    {
        get { return (int) GetValue(DpiXProperty); }
        set { SetValue(DpiXProperty, value); }
    }

    public static readonly DependencyProperty DpiYProperty = DependencyProperty.Register(
        "DpiY", typeof (int), typeof (PixelSupportBehavior), new
PropertyMetadata(96,OnDpiYChanged));

    private static void OnDpiYChanged(DependencyObject d, DependencyPropertyChangedEventArgs
e)
    {
        var b = (PixelSupportBehavior)d;
        var value = (int)e.NewValue;

        if (value <= 0)
            return;

        b._dpiY = value;

        b.InitializeBufferIfAttached();
    }

    public int DpiY
    {
        get { return (int) GetValue(DpiYProperty); }
        set { SetValue(DpiYProperty, value); }
    }

    private IntPtr _backBuffer = IntPtr.Zero;
    private int _bytesPerPixel;
    private const int BitsPerByte = 8;
    private int _pixelWidth;
    private int _pixelHeight;
    private int _dpiX;
    private int _dpiY;
    private Int32Rect _imageRectangle;
    private readonly GCHandle _defaultGcHandle = new GCHandle();
    private PixelFormat _pixelFormat;

    private int _byteArraySize;
    private WriteableBitmap _bitMap;

    private bool _attached;

    protected override void OnAttached()
    {
        _attached = true;
        InitializeBufferIfAttached();
    }
```

```
private void InitializeBufferIfAttached()
{
    if(_attached==false)
        return;

    ReevaluateBitsPerPixel();

    RecomputeByteArraySize();

    ReinitializeImageSource();
}

private void ReevaluateBitsPerPixel()
{
    var f = _pixelFormat;

    if (f == PixelFormats.Default)
    {
        _bytesPerPixel = 0;
        return;
    };

    _bytesPerPixel = f.BitsPerPixel/BitsPerByte;
}

private void ReinitializeImageSource()
{
    var f = _pixelFormat;
    var h = _pixelHeight;
    var w = _pixelWidth;

    if (w<=0|| h<=0 || f== PixelFormats.Default)
        return;

    _imageRectangle = new Int32Rect(0, 0, w, h);
    _bitMap = new WriteableBitmap(w, h, _dpiX, _dpiY, f, null);
    _backBuffer = _bitMap.BackBuffer;
    AssociatedObject.Source = _bitMap;
}

private void RenderPixels(IList<byte> pixels)
{
    if (pixels == null)
    {
        return;
    }

    var buffer = _backBuffer;
    if (buffer == IntPtr.Zero)
        return;

    var size = _byteArraySize;

    var gcHandle = _defaultGcHandle;
    var allocated = false;
    var bitMap = _bitMap;
    var rect = _imageRectangle;
    var w = _pixelWidth;
    var locked = false;
    try
```

```
        {
            gcHandle = GCHandle.Alloc(pixels, GCHandleType.Pinned);
            allocated = true;

            bitMap.Lock();
            locked = true;
            var ptr = gcHandle.AddrOfPinnedObject();
            _bitMap.WritePixels(rect, ptr, size,w);
        }
        finally
        {
            if(locked)
                bitMap.Unlock();

            if (allocated)
                gcHandle.Free();
        }
    }

    private void RecomputeByteArraySize()
    {
        var h = _pixelHeight;
        var w = _pixelWidth;
        var bpp = _bytesPerPixel;

        if (w<=0|| h<=0 || bpp<=0)
            return;

        _byteArraySize = (w * h * bpp);
    }

    public PixelSupportBehavior()
    {
        _pixelFormat = PixelFormats.Default;
    }
}
}
```

XAML Usage

```
<UserControl x:Class="Example.View"
         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"
         xmlns:b="clr-namespace:MyBehaviorAssembly;assembly=MyBehaviorAssembly"
         xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
         mc:Ignorable="d"
         d:DesignHeight="200" d:DesignWidth="200"
         >

                <Image Stretch="Uniform">

                <i:Interaction.Behaviors>

                    <b:PixelSupportBehavior
                            PixelHeight="{Binding PixelHeight}"
                            PixelWidth="{Binding PixelWidth}"
                            PixelFormat="{Binding PixelFormat}"
```

```
                            Pixels="{Binding Pixels}"
                            />

                </i:Interaction.Behaviors>

            </Image>

</UserControl>
```

Read Supporting Video Streaming and Pixel Array Assignment to an Image Control online: https://riptutorial.com/wpf/topic/6435/supporting-video-streaming-and-pixel-array-assignment-to-an-image-control

Chapter 16: System.Windows.Controls.WebBrowser

Introduction

This allows you to put a Web browser into your WPF application.

Remarks

A key point to note, which is not obvious from the documentation, and you could go for years without knowing is that it defaults to behaving like InternetExplorer7, rather than your most up-to-date InternetExplorer installation (see https://weblog.west-wind.com/posts/2011/may/21/web-browser-control-specifying-the-ie-version).

This cannot be fixed by setting a property on the control; you must either modify the pages being displayed by adding an HTML Meta Tag, or by applying a registry setting(!). (Details of both approaches are on the link above.)

For example, this bizarre design behaviour might lead you to get a message saying "Script Error"/"An error has occurred in the script on this page". Googling this error might make you think that the solution is to try to suppress the error, rather that understanding the actual problem, and applying the correct solution.

Examples

Example of a WebBrowser within a BusyIndicator

Be aware that the WebBrowser control is not sympathetic to your XAML definition, and renders itself over the top of other things. For example, if you put it inside a BusyIndicator that has been marked as being busy, it will still render itself over the top of that control. The solution is to bind the visibility of the WebBrowser to the value that the BusyIndicator is using, and use a converter to invert the Boolean and convert it to a Visibility. For example:

```
    <telerik:RadBusyIndicator IsBusy="{Binding IsBusy}">
        <WebBrowser Visibility="{Binding IsBusy, Converter={StaticResource
InvertBooleanToVisibilityConverter}}"/>
    </telerik:RadBusyIndicator>
```

Read System.Windows.Controls.WebBrowser online:
https://riptutorial.com/wpf/topic/9115/system-windows-controls-webbrowser

Chapter 17: Thread Affinity Accessing UI Elements

Examples

Accessing a UI Element From Within a Task

All UI elements created and reside in the main thread of a program. Accessing these from another thread is forbidden by the .net framework runtime. Basically it is because all UI elements are **thread sensitive resources** and accessing a resource in a multi-threaded environment requires to be thread-safe. If this cross thread object access is allowed then consistency would be affected in the first place.

Consider this scenario:

We have a calculation happening inside a task. Tasks are run in another thread than the main thread. While calculation goes on we need to update a progressbar. To do this:

```
//Prepare the action
Action taskAction = new Action( () => {
    int progress = 0;
    Action invokeAction = new Action( () => { progressBar.Value = progress; });
    while (progress <= 100) {
        progress = CalculateSomething();
        progressBar.Dispatcher.Invoke( invokeAction );
    }
} );

//After .net 4.5
Task.Run( taskAction );

//Before .net 4.5
Task.Factory.StartNew( taskAction ,
    CancellationToken.None,
    TaskCreationOptions.DenyChildAttach,
    TaskScheduler.Default);
```

Every UI element has a Dispatcher object that comes from its `DispatcherObject` ancestor (inside `System.Windows.Threading` namespace). Dispatcher executes the specified delegate synchronously at the specified priority on the thread on which the Dispatcher is associated with. Since execution is syncronised, caller task should wait for its result. This gives us the opportunity to use `int progress` also inside a dispatcher delegate.

We may want to update a UI element asynchronously then `invokeAction` definition changes:

```
//Prepare the action
Action taskAction = new Action( () => {
    int progress = 0;
    Action<int> invokeAction = new Action<int>( (i) => { progressBar.Value = i; } )
```

```
    while (progress <= 100) {
        progress = CalculateSomething();
        progressBar.Dispatcher.BeginInvoke(
            invokeAction,
            progress );
    }
} );

//After .net 4.5
Task.Run( taskAction );

//Before .net 4.5
Task.Factory.StartNew( taskAction ,
    CancellationToken.None,
    TaskCreationOptions.DenyChildAttach,
    TaskScheduler.Default);
```

This time we packed `int progress` and use it as a parameter for delegate.

Read Thread Affinity Accessing UI Elements online: https://riptutorial.com/wpf/topic/6128/thread-affinity-accessing-ui-elements

Chapter 18: Triggers

Introduction

Discussion on the various types of Triggers available in WPF, including `Trigger`, `DataTrigger`, `MultiTrigger`, `MultiDataTrigger`, and `EventTrigger`.

Triggers allow any class that derives from `FrameworkElement` or `FrameworkContentElement` to set or change their properties based on certain conditions defined in the trigger. Basically, if an element can be styled, it can be triggered as well.

Remarks

- All triggers, except for `EventTrigger` must be defined within a `<Style>` element. An `EventTrigger` may be defined in either a `<Style>` element, or a control's `Triggers` property.
- `<Trigger>` elements may contain any number of `<Setter>` elements. These elements are responsible for setting properties on the containing element when the `<Trigger>` element's condition is met.
- If a property is defined in the root element markup, the property change defined in the `<Setter>` element will not take effect, even if the trigger condition has been met. Consider the markup `<TextBlock Text="Sample">`. The `Text` property of the proceeding code will never change based on a trigger because root property definitions take precidence over properties defined in styles.
- Like bindings, once a trigger has been used, it cannot be modified.

Examples

Trigger

The simplest of the five trigger types, the `Trigger` is responsible for setting properties based on other properties **within the same control**.

```
<TextBlock>
    <TextBlock.Style>
        <Style TargetType="{x:Type TextBlock}">
            <Style.Triggers>
                <Trigger Property="Text" Value="Pass">
                    <Setter Property="Foreground" Value="Green"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </TextBlock.Style>
</TextBlock>
```

In this example, the foreground color of the `TextBlock` will turn green when it's `Text` property is equal to the string `"Pass"`.

MultiTrigger

A `MultiTrigger` is similar to a standard `Trigger` in that it only applies to properties **within the same control**. The difference is that a `MultiTrigger` has multiple conditions which must be satisfied before the trigger will operate. Conditions are defined using the `<Condition>` tag.

```
<TextBlock x:Name="_txtBlock" IsEnabled="False">
    <TextBlock.Style>
        <Style TargetType="{x:Type TextBlock}">
            <Style.Triggers>
                <MultiTrigger>
                    <MultiTrigger.Conditions>
                        <Condition Property="Text" Value="Pass"/>
                        <Condition Property="IsEnabled" Value="True"/>
                    </MultiTrigger.Conditions>
                    <Setter Property="Foreground" Value="Green"/>
                </MultiTrigger>
            </Style.Triggers>
        </Style>
    </TextBlock.Style>
</TextBlock>
```

Notice the `MultiTrigger` will not activate until both conditions are met.

DataTrigger

A `DataTrigger` can be attached to any property, be it on it's own control, another control, or even a property in a non UI class. Consider the following simple class.

```
public class Cheese
{
    public string Name { get; set; }
    public double Age { get; set; }
    public int StinkLevel { get; set; }
}
```

Which we will attach as the `DataContext` in the following `TextBlock`.

```
<TextBlock Text="{Binding Name}">
    <TextBlock.DataContext>
        <local:Cheese Age="12" StinkLevel="100" Name="Limburger"/>
    </TextBlock.DataContext>
    <TextBlock.Style>
        <Style TargetType="{x:Type TextBlock}">
            <Style.Triggers>
                <DataTrigger Binding="{Binding StinkLevel}" Value="100">
                    <Setter Property="Foreground" Value="Green"/>
                </DataTrigger>
            </Style.Triggers>
        </Style>
    </TextBlock.Style>
</TextBlock>
```

In the preceeding code, the `TextBlock.Foreground` property will be Green. If we change the

`StinkLevel` property in our XAML to anything other than 100, the `Text.Foreground` property will revert to it's default value.

Read Triggers online: https://riptutorial.com/wpf/topic/9624/triggers

Chapter 19: Value and Multivalue Converters

Parameters

Parameter	Details
value	The value produced by the binding source.
values	The values array, produced by the binding source.
targetType	The type of the binding target property.
parameter	The converter parameter to use.
culture	The culture to use in the converter.

Remarks

What IValueConverter and IMultiValueConverterthey are

IValueConverter and IMultiValueConverter - interfaces that provides a way to apply a custom logic to a binding.

What they are useful for

1. You have a some type value but you want to show zero values in one way and positive numbers in another way
2. You have a some type value and want to show element in one case and hide in another
3. You have a numeric value of money but want to show it as words
4. You have a numeric value but want to show different images for defferent numbers

These are some of the simple cases, but there are many more.

For cases like this, you can use a value converter. These small classes, which implement the IValueConverter interface or IMultiValueConverter, will act like middlemen and translate a value between the source and the destination. So, in any situation where you need to transform a value before it reaches its destination or back to its source again, you likely need a converter.

Examples

Build-In BooleanToVisibilityConverter [IValueConverter]

Converter between boolean and visibility. Get `bool` value on input and returns `Visibility` value.

NOTE: This converter have already exists in `System.Windows.Controls` namespace.

```
public sealed class BooleanToVisibilityConverter : IValueConverter
{
    /// <summary>
    /// Convert bool or Nullable bool to Visibility
    /// </summary>
    /// <param name="value">bool or Nullable bool</param>
    /// <param name="targetType">Visibility</param>
    /// <param name="parameter">null</param>
    /// <param name="culture">null</param>
    /// <returns>Visible or Collapsed</returns>
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        bool bValue = false;
        if (value is bool)
        {
            bValue = (bool)value;
        }
        else if (value is Nullable<bool>)
        {
            Nullable<bool> tmp = (Nullable<bool>)value;
            bValue = tmp.HasValue ? tmp.Value : false;
        }
        return (bValue) ? Visibility.Visible : Visibility.Collapsed;
    }

    /// <summary>
    /// Convert Visibility to boolean
    /// </summary>
    /// <param name="value"></param>
    /// <param name="targetType"></param>
    /// <param name="parameter"></param>
    /// <param name="culture"></param>
    /// <returns></returns>
    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        if (value is Visibility)
        {
            return (Visibility)value == Visibility.Visible;
        }
        else
        {
            return false;
        }
    }
}
```

Using the converter

1. Define Resource

```
<BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
```

3. Use it in binding

```
<Button Visibility="{Binding AllowEditing,
                            Converter={StaticResource BooleanToVisibilityConverter}}"/>
```

Converter with property [IValueConverter]

Show how to create simple converter with parameter via property and then pass it in declaration. Convert `bool` value to `Visibility`. Allow invert result value by setting `Inverted` property to `True`.

```
public class BooleanToVisibilityConverter : IValueConverter
{
    public bool Inverted { get; set; }

    /// <summary>
    /// Convert bool or Nullable bool to Visibility
    /// </summary>
    /// <param name="value">bool or Nullable bool</param>
    /// <param name="targetType">Visibility</param>
    /// <param name="parameter">null</param>
    /// <param name="culture">null</param>
    /// <returns>Visible or Collapsed</returns>
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        bool bValue = false;
        if (value is bool)
        {
            bValue = (bool)value;
        }
        else if (value is Nullable<bool>)
        {
            Nullable<bool> tmp = (Nullable<bool>)value;
            bValue = tmp ?? false;
        }

        if (Inverted)
            bValue = !bValue;
        return (bValue) ? Visibility.Visible : Visibility.Collapsed;
    }

    /// <summary>
    /// Convert Visibility to boolean
    /// </summary>
    /// <param name="value"></param>
    /// <param name="targetType"></param>
    /// <param name="parameter"></param>
    /// <param name="culture"></param>
    /// <returns>True or False</returns>
    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        if (value is Visibility)
        {
            return ((Visibility) value == Visibility.Visible) && !Inverted;
        }

        return false;
    }
}
```

Using the converter

1. Define namespace

```
xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"
```

2. Define Resource

```
<converters:BooleanToVisibilityConverter x:Key="BoolToVisibilityInvertedConverter"
                                         Inverted="False"/>
```

3. Use it in binding

```
<Button Visibility="{Binding AllowEditing, Converter={StaticResource
BoolToVisibilityConverter}}"/>
```

Simple add converter [IMultiValueConverter]

Show how to create simple `IMultiValueConverter` converter and use `MultiBinding` in xaml. Get summ of all values passed by `values` array.

```
public class AddConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType, object parameter, CultureInfo
culture)
    {
        decimal sum = 0M;

        foreach (string value in values)
        {
            decimal parseResult;
            if (decimal.TryParse(value, out parseResult))
            {
                sum += parseResult;
            }
        }

        return sum.ToString(culture);
    }

    public object[] ConvertBack(object value, Type[] targetTypes, object parameter,
CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

Using the converter

1. Define namespace

```
xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"
```

2. Define Resource

```
<converters:AddConverter x:Key="AddConverter"/>
```

3. Use it in binding

```
<StackPanel Orientation="Vertical">
    <TextBox x:Name="TextBox" />
    <TextBox x:Name="TextBox1" />
    <TextBlock >
        <TextBlock.Text>
            <MultiBinding Converter="{StaticResource AddConverter}">
                <Binding Path="Text" ElementName="TextBox"/>
                <Binding Path="Text" ElementName="TextBox1"/>
            </MultiBinding>
         </TextBlock.Text>
    </TextBlock>
</StackPanel>
```

Usage converters with ConverterParameter

Show how to create simple converter and use `ConverterParameter` to pass parameter to converter. Multiply value by coefficient passed in ConverterParameter.

```
public class MultiplyConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        if (value == null)
            return 0;

        if (parameter == null)
            parameter = 1;

        double number;
        double coefficient;

        if (double.TryParse(value.ToString(), out number) &&
double.TryParse(parameter.ToString(), out coefficient))
        {
            return number * coefficient;
        }

        return 0;
    }

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

Using the converter

1. Define namespace

```
xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"
```

2. Define Resource

```
<converters:MultiplyConverter x:Key="MultiplyConverter"/>
```

3. Use it in binding

```
<StackPanel Orientation="Vertical">
    <TextBox x:Name="TextBox" />
    <TextBlock Text="{Binding Path=Text,
                              ElementName=TextBox,
                              Converter={StaticResource MultiplyConverter},
                              ConverterParameter=10}"/>
</StackPanel>
```

Group multiple converters [IValueConverter]

This converter will chain multiple converters together.

```
public class ValueConverterGroup : List<IValueConverter>, IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return this.Aggregate(value, (current, converter) => converter.Convert(current,
targetType, parameter, culture));
    }

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

In this example, the boolean result from `EnumToBooleanConverter` is used as input in `BooleanToVisibilityConverter`.

```
<local:ValueConverterGroup x:Key="EnumToVisibilityConverter">
    <local:EnumToBooleanConverter/>
    <local:BooleanToVisibilityConverter/>
</local:ValueConverterGroup>
```

The button will only be visible when the `CurrentMode` property is set to `Ready`.

```
<Button Content="Ok" Visibility="{Binding Path=CurrentMode, Converter={StaticResource
EnumToVisibilityConverter}, ConverterParameter={x:Static local:Mode.Ready}"/>
```

Using MarkupExtension with Converters to skip recource declaration

Usually to use the converter, we have to define it as resource in the following way:

```
<converters:SomeConverter x:Key="SomeConverter"/>
```

It is possible to skip this step by defining a converter as `MarkupExtension` and implementing the method `ProvideValue`. The following example converts a value to its negative:

```
namespace MyProject.Converters
{
public class Converter_Negative : MarkupExtension, IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return this.ReturnNegative(value);
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo
culture)
    {
        return this.ReturnNegative(value);
    }

    private object ReturnNegative(object value)
    {
        object result = null;
        var @switch = new Dictionary<Type, Action> {
            { typeof(bool), () => result=!(bool)value },
            { typeof(byte), () => result=-1*(byte)value },
            { typeof(short), () => result=-1*(short)value },
            { typeof(int), () => result=-1*(int)value },
            { typeof(long), () => result=-1*(long)value },
            { typeof(float), () => result=-1f*(float)value },
            { typeof(double), () => result=-1d*(double)value },
            { typeof(decimal), () => result=-1m*(decimal)value }
        };

        @switch[value.GetType()]();
        if (result == null) throw new NotImplementedException();
        return result;
    }

    public Converter_Negative()
        : base()
    {
    }

    private static Converter_Negative _converter = null;

    public override object ProvideValue(IServiceProvider serviceProvider)
    {
        if (_converter == null) _converter = new Converter_Negative();
        return _converter;
    }
}
}
```

Using the converter:

1. Define namespace

   xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"

2. Example use of this converter in binding

```
<RichTextBox IsReadOnly="{Binding Path=IsChecked, ElementName=toggleIsEnabled,
Converter={converters:Converter_Negative}}"/>
```

Use IMultiValueConverter to pass multiple parameters to a Command

It is possible to pass multiple bound values as a `CommandParameter` using `MultiBinding` with a very simple `IMultiValueConverter`:

```
namespace MyProject.Converters
{
    public class Converter_MultipleCommandParameters : MarkupExtension, IMultiValueConverter
    {
        public object Convert(object[] values, Type targetType, object parameter, CultureInfo
culture)
        {
            return values.ToArray();
        }
        public object[] ConvertBack(object value, Type[] targetTypes, object parameter,
CultureInfo culture)
        {
            throw new NotSupportedException();
        }

        private static Converter_MultipleCommandParameters _converter = null;

        public override object ProvideValue(IServiceProvider serviceProvider)
        {
            if (_converter == null) _converter = new Converter_MultipleCommandParameters();
            return _converter;
        }

        public Converter_MultipleCommandParameters()
            : base()
        {
        }
    }
}
```

Using the converter:

1. Example implementation - method called when `SomeCommand` is executed (*note: `DelegateCommand` is an implementation of `ICommand` that is not provided in this example*):

```
 private ICommand _SomeCommand;
 public ICommand SomeCommand
 {
     get { return _SomeCommand ?? (_SomeCommand = new DelegateCommand(a =>
OnSomeCommand(a))); }
 }
```

```
 private void OnSomeCommand(object item)
 {
     object[] parameters = item as object[];

     MessageBox.Show(
         string.Format("Execute command: {0}\nParameter 1: {1}\nParamter 2: {2}\nParamter
3: {3}",
         "SomeCommand", parameters[0], parameters[1], parameters[2]));
 }
```

2. Define namespace

   xmlns:converters="clr-namespace:MyProject.Converters;assembly=MyProject"

3. Example use of this converter in binding

```
<Button Width="150" Height="23" Content="Execute some command" Name="btnTestSomeCommand"
     Command="{Binding Path=SomeCommand}" >
     <Button.CommandParameter>
         <MultiBinding Converter="{converters:Converter_MultipleCommandParameters}">
             <Binding RelativeSource="{RelativeSource Self}" Path="IsFocused"/>
             <Binding RelativeSource="{RelativeSource Self}" Path="Name"/>
             <Binding RelativeSource="{RelativeSource Self}" Path="ActualWidth"/>
         </MultiBinding>
     </Button.CommandParameter>
 </Button>
```

Read Value and Multivalue Converters online: https://riptutorial.com/wpf/topic/3950/value-and-multivalue-converters

Chapter 20: WPF Architecture

Examples

DispatcherObject

Derives from

`Object`

Key members

```
public Dispatcher Dispatcher { get; }
```

Summary

Most objects in WPF derive from `DispatcherObject`, which provides the basic constructs for dealing with concurrency and threading. Such objects are associated with a Dispatcher.

Only the thread that the Dispatcher was created on may access the DispatcherObject directly. To access a DispatcherObject from a thread other than the thread the DispatcherObject was created on, a call to `Invoke` or `BeginInvoke` on the Dispatcher the object is associated with is required.

DependencyObject

Derives from

`DispatcherObject`

Key members

```
public object GetValue(DependencyProperty dp);
public void   SetValue(DependencyProperty dp, object value);
```

Summary

Classes derived from `DependencyObject` participate in the dependency property system, which includes registering dependency properties and providing identification and information about such properties. Since dependency properties are the cornerstone of WPF development, all WPF controls ultimately derive from `DependencyObject`.

Read WPF Architecture online: https://riptutorial.com/wpf/topic/3571/wpf-architecture

Chapter 21: WPF Behaviors

Introduction

WPF behaviors allow a developer to alter the way WPF controls acts in response to system and user events. Behaviors inherit from the `Behavior` class of the `System.Windows.Interactity` namespace. This namespace is a part of the overarching Expression Blend SDK, but a lighter version, suitable for behavior libraries, is available as a [nuget package][1]. [1]: https://www.nuget.org/packages/System.Windows.Interactivity.WPF/

Examples

Simple Behavior to Intercept Mouse Wheel Events

Implementing the Behavior

This behavior will cause mouse wheel events from an inner `ScrollViewer` to bubble up to the parent `ScrollViewer` when the inner one is at either its upper or lower limit. Without this behavior, the events will never make it out of the inner `ScrollViewer`.

```
public class BubbleMouseWheelEvents : Behavior<UIElement>
{
    protected override void OnAttached()
    {
        base.OnAttached();
        this.AssociatedObject.PreviewMouseWheel += PreviewMouseWheel;
    }

    protected override void OnDetaching()
    {
        this.AssociatedObject.PreviewMouseWheel -= PreviewMouseWheel;
        base.OnDetaching();
    }

    private void PreviewMouseWheel(object sender, MouseWheelEventArgs e)
    {
        var scrollViewer = AssociatedObject.GetChildOf<ScrollViewer>(includeSelf: true);
        var scrollPos = scrollViewer.ContentVerticalOffset;
        if ((scrollPos == scrollViewer.ScrollableHeight && e.Delta < 0) || (scrollPos == 0 &&
e.Delta > 0))
        {
            UIElement rerouteTo = AssociatedObject;
            if (ReferenceEquals(scrollViewer, AssociatedObject))
            {
                rerouteTo = (UIElement) VisualTreeHelper.GetParent(AssociatedObject);
            }

            e.Handled = true;
            var e2 = new MouseWheelEventArgs(e.MouseDevice, e.Timestamp, e.Delta);
            e2.RoutedEvent = UIElement.MouseWheelEvent;
            rerouteTo.RaiseEvent(e2);
        }
```

```
    }
}
```

Behaviors subclass the `Behavior<T>` base-class, with `T` being the type of control that it is able to attach to, in this case `UIElement`. When the `Behavior` is instantiated from XAML, the `OnAttached` method is called. This method allows the behavior to hook in to events from the control it is attached to (via `AssociatedControl`). A similar method, `OnDetached` is called when the behavior need to be unhooked from the associated element. Care should be taken to remove any event handlers, or otherwise clean up objects to avoid memory leaks.

This behavior hooks in to the `PreviewMouseWheel` event, which gives it a change to intercept the event before the `ScrollViewer` has a chance to see it. It checks the position to see if it needs to forward the event up the visual tree to any `ScrollViewer` higher hierarchy. If so, it sets `e.Handled` to `true` to prevent the default action of the event. It then raises a new `MouseWheelEvent` routed to `AssociatedObject`. Otherwise, the event is routed as normal.

Attaching the Behavior to an Element in XAML

First, the `interactivity` xml-namespace must be brought in to scope before it can be used in XAML. Add the following line to the namespaces of your XAML.

xmlns:interactivity="http://schemas.microsoft.com/expression/2010/interactivity"

The behavior can be attached like so:

```
<ScrollViewer>
    <!--...Content...-->
    <ScrollViewer>
        <interactivity:Interaction.Behaviors>
            <behaviors:BubbleMouseWheelEvents />
        </interactivity:Interaction.Behaviors>
        <!--...Content...-->
    </ScrollViewer>
    <!--...Content...-->.
</ScrollViewer>
```

This creates a `Behaviors` collection as an Attached Property on the inner `ScrollViewer` that contains a `BubbleMouseWheelEvents` behavior.

This particular behavior could also be attached to any existing control that contains an embedded `ScrollViewer`, such as a `GridView`, and it would still function correctly.

Read WPF Behaviors online: https://riptutorial.com/wpf/topic/8365/wpf-behaviors

Chapter 22: WPF Localization

Remarks

Content of controls can be localized using Resource files, just as this is possible in classes. For XAML there is a specific syntax, that is different between a C# and a VB application.

The steps are:

- For any WPF project: make the resource file public, the default is internal.
- For C# WPF projects use the XAML provided in the example
- For VB WPF projects use the XAML provided in the example and change the Custom Tool property to `PublicVbMyResourcesResXFileCodeGenerator`.
- To select the Resources.resx file in a VB WPF project:
    - Select the project in solution explorer
    - Select "Show all files"
    - Expand My Project

Examples

XAML for VB

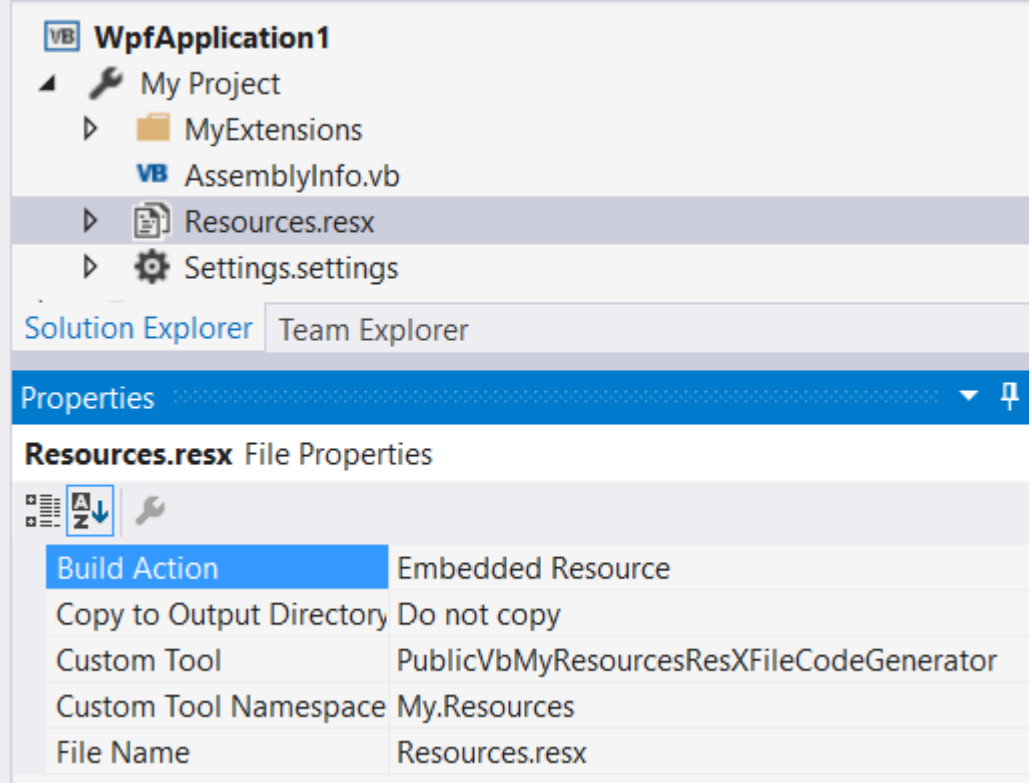
```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfApplication1"
    xmlns:my="clr-namespace:WpfApplication1.My.Resources"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
<Grid>
    <StackPanel>
        <Label Content="{Binding Source={x:Static my:Resources.MainWindow_Label_Country}}" />
    </StackPanel>
</Grid>
```

Properties for the resource file in VB

By default the Custom Tool property for a VB resource file is `VbMyResourcesResXFileCodeGenerator`. However, with this code generator the view (XAML) will not be able to access the resources. To solve this problem add `Public` before the Custom Tool property value.

To select the Resources.resx file in a VB WPF project:

- Select the project in solution explorer
- Select "Show all files"
- Expand "My Project"

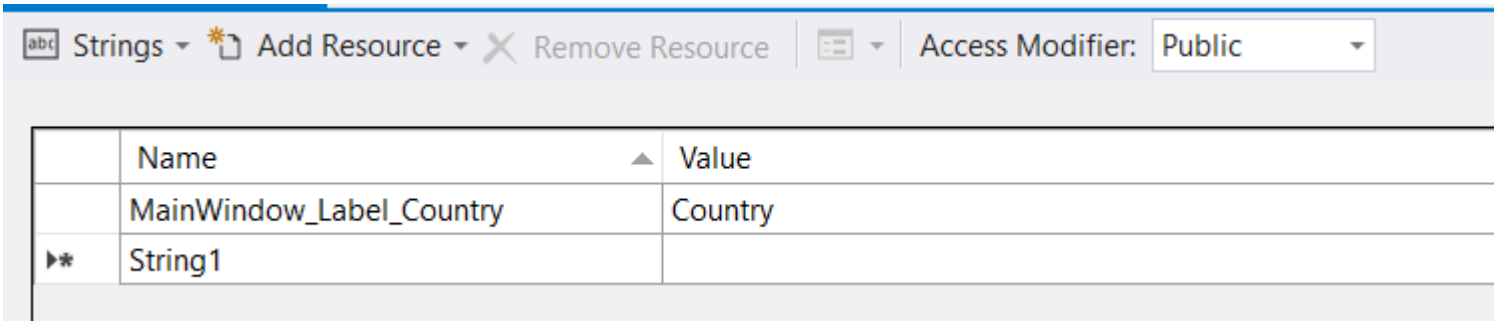


XAML for C#

```
<Window x:Class="WpfApplication2.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:WpfApplication2"
    xmlns:resx="clr-namespace:WpfApplication2.Properties"
    mc:Ignorable="d"
    Title="MainWindow" Height="350" Width="525">
<Grid>
    <StackPanel>
        <Label Content="{Binding Source={x:Static resx:Resources.MainWindow_Label_Country}}"/>
    </StackPanel>
</Grid>
```

Make the resources public

Open the resource file by double clicking it. Change the Access Modifier to "Public".



Read WPF Localization online: https://riptutorial.com/wpf/topic/3905/wpf-localization

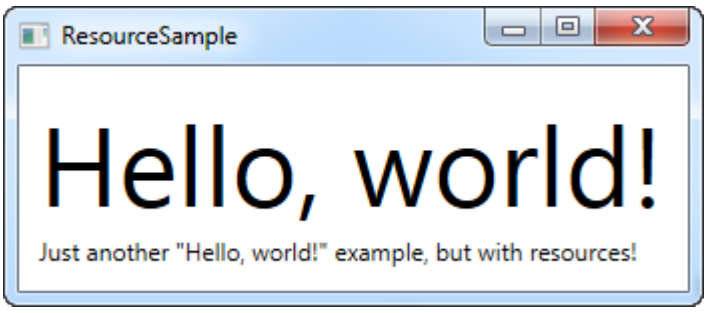
Chapter 23: WPF Resources

Examples

Hello Resources

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 pretty much 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.The concept is used a lot for styles and templates.

```
<Window x:Class="WPFApplication.ResourceSample"
        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"
        Title="ResourceSample" Height="150" Width="350">
    <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>
</Window>
```



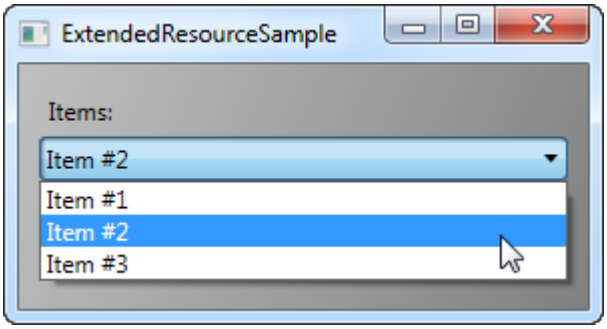
Resources are given a key, using the x:Key attribute, which allows you to reference it from other parts of the application by using this key, in combination with the StaticResource markup extension. In this example, I just store a simple string, which I then use from two different TextBlock controls.

Resource Types

Sharing a simple string was easy, but you can do much more. In this example, I'll also store a complete array of strings, along with a gradient brush to be used for the background. This should give you a pretty good idea of just how much you can do with resources:

```
<Window x:Class="WPFApplication.ExtendedResourceSample"
        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"
        Title="ExtendedResourceSample" Height="160" Width="300"
        Background="{DynamicResource WindowBackgroundBrush}">
    <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"/>
            <GradientStop Offset="1" Color="Gray"/>
        </LinearGradientBrush>
    </Window.Resources>
    <StackPanel Margin="10">
        <Label Content="{StaticResource ComboBoxTitle}" />
        <ComboBox ItemsSource="{StaticResource ComboBoxItems}" />
    </StackPanel>
</Window>
```



This time, we've added a couple of extra resources, so that our Window now contains a simple string, an array of strings and a LinearGradientBrush. The string is used for the label, the array of strings is used as items for the ComboBox control and the gradient brush is used as background for the entire window. So, as you can see, pretty much anything can be stored as a resource.

Local and application wide resources

If you only need a given resource for a specific control, you can make it more local by adding it to this specific control, instead of the window. It works exactly the same way, the only difference being that you can now only access from inside the scope of the control where you put it:

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

In this case, we add the resource to the StackPanel and then use it from its child control, the Label. Other controls inside of the StackPanel could have used it as well, just like children of these child controls would have been able to access it. Controls outside of this particular StackPanel

wouldn't have access to it, though.

If you need the ability to access the resource from several windows, this is possible as well. The App.xaml file can contain resources just like the window and any kind of WPF control, and when you store them in App.xaml, they are globally accessible in all of windows and user controls of the project. It works exactly the same way as when storing and using from a Window:

```
<Application x:Class="WpfSamples.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="WPFApplication/ExtendedResourceSample.xaml">
    <Application.Resources>
        <sys:String x:Key="ComboBoxTitle">Items:</sys:String>
    </Application.Resources>
</Application>
```

Using it is also the same - WPF will automatically go up the scope, from the local control to the window and then to App.xaml, to find a given resource:

```
<Label Content="{StaticResource ComboBoxTitle}" />
```

Resources from Code-behind

In this example, we'll be accessing three different resources from Code-behind, each stored in a different scope

App.xaml:

```
<Application x:Class="WpfSamples.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="WPFApplication/ResourcesFromCodeBehindSample.xaml">
    <Application.Resources>
        <sys:String x:Key="strApp">Hello, Application world!</sys:String>
    </Application.Resources>
</Application>
```

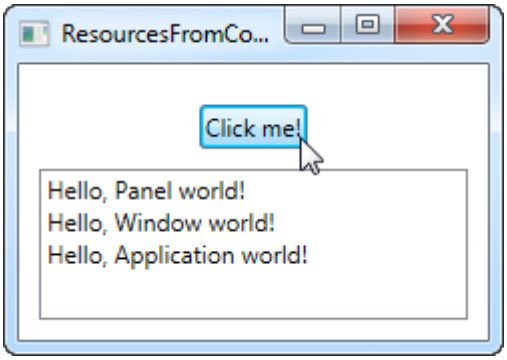
Window:

```
<Window x:Class="WpfSamples.WPFApplication.ResourcesFromCodeBehindSample"
        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"
        Title="ResourcesFromCodeBehindSample" Height="175" Width="250">
    <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>
```

```
        <WrapPanel DockPanel.Dock="Top" HorizontalAlignment="Center" Margin="10">
            <Button Name="btnClickMe" Click="btnClickMe_Click">Click me!</Button>
        </WrapPanel>

        <ListBox Name="lbResult" />
    </DockPanel>
</Window>
```

Code-behind:

```
using System;
using System.Windows;

namespace WpfSamples.WPFApplication
{
        public partial class ResourcesFromCodeBehindSample : Window
        {
                public ResourcesFromCodeBehindSample()
                {
                        InitializeComponent();
                }

                private void btnClickMe_Click(object sender, RoutedEventArgs e)
                {
                        lbResult.Items.Add(pnlMain.FindResource("strPanel").ToString());
                        lbResult.Items.Add(this.FindResource("strWindow").ToString());

lbResult.Items.Add(Application.Current.FindResource("strApp").ToString());
                }
        }
}
```



So, as you can see, we store three different "Hello, world!" messages: One in App.xaml, one inside the window, and one locally for the main panel. The interface consists of a button and a ListBox.

In Code-behind, we handle the click event of the button, in which we add each of the text strings to the ListBox, as seen on the screenshot. We use the FindResource() method, which will return the resource as an object (if found), and then we turn it into the string that we know it is by using the ToString() method.

Notice how we use the FindResource() method on different scopes - first on the panel, then on the window and then on the current Application object. It makes sense to look for the resource where

we know it is, but as already mentioned, if a resource is not found, the search progresses up the hierarchy, so in principal, we could have used the FindResource() method on the panel in all three cases, since it would have continued up to the window and later on up to the application level, if not found.

The same is not true the other way around - the search doesn't navigate down the tree, so you can't start looking for a resource on the application level, if it has been defined locally for the control or for the window.

Read WPF Resources online: https://riptutorial.com/wpf/topic/4371/wpf-resources

Credits

S. No	Chapters	Contributors
1	Getting started with wpf	Community, Derpcode, Gusdor, Matthew Cargille, Nasreddine, Sam, Stephen Wilson
2	"Half the Whitespace" Design principle	Richardissimo
3	An Introduction to WPF Styles	J R
4	Creating custom UserControls with data binding	Itiveron, Mage Xy
5	Creating Splash Screen in WPF	Grx70, Sam
6	Dependency Properties	Adi Lester, auticus, Clemens, Guttsy
7	Grid control	Alexander Mandt, vkluge
8	Introduction to WPF Data Binding	Adi Lester, Arie, Gabor Barat, Guttsy, Ian Wold, Jirajha, vkluge, wkl
9	Markup Extensions	Alexander Pacha, Emad
10	MVVM in WPF	Andrew Stephens, Athafoud, Dutts, Felix D., Felix Too, H.B., James LaPenn, Kcvin, kowsky, Matt Klein, RamenChef, STiLeTT, TrBBoI, vkluge
11	Optimizing for touch interaction	Martin Zikmund
12	Slider Binding: Update only on Drag Ended	Eyal Perry
13	Speech Synthesis	BKO
14	Styles in WPF	Guttsy, Jakub Lokša
15	Supporting Video Streaming and Pixel Array Assignment to an Image Control	Eyal Perry
16	System.Windows.Controls.WebBrowser	Richardissimo
17	Thread Affinity Accessing UI Elements	Mert Gülsoy
18	Triggers	John Strit, Maxim

19	Value and Multivalue Converters	Adi Lester, Arie, Dalstroem, galakt, ltiveron
20	WPF Architecture	Adi Lester
21	WPF Behaviors	Bradley Uffner
22	WPF Localization	Dabblernl
23	WPF Resources	Elangovan, John Strit, SUB-HDR