WPF

Application Development Using WPF

Lesson 2

Controls

Contents

1. Content Control Model 5

1.1. Variations of Content Model 6

1.1.1. The HeaderedContentControl Class 7

1.1.2. The ItemsControl Class 8

1.1.3. The HeaderedItemsControl Class 9

2. Background and Foreground Brushes 10

2.1. The SolidColorBrush 12

2.2. Gradient Brushes 13

2.2.1. LinearGradientBrush 15

2.2.2. RadialGradientBrush 18

2.3. Tile Brushes 20

2.3.1. ImageBrush 22

2.3.2. VisualBrush 24

3. Fonts 26

3.1. Font Size 26

3.2. Font Stretch 26

3.3. Text density 27

3.4. Font Style 27
3.5. Font Family 28
3.6. Inheritance of Fonts 28
4. Primitive Elements 29
4.1. TextBlock Element 29
4.2. Image Element 32
4.3. MediaElement 34
4.4. Border Element 38
5. Label Control 41
6. Grouping Controls 44
6.1. GroupBox Control 44
6.2. Expander Control 46
7. Range Controls 49
7.1. ProgressBar Control 50
7.2. Slider Control 52
7.3. ScrollBar Control 58
8. Items Control 61
8.1. ListBox Control 63
8.2. ListView Control 66
8.3. ComboBox Control 66
8.4. TreeView Control 68
8.5. TabControl 73
9. Popup Windows 76
9.1. Popup Window Placement 79
9.1.1. Target Object 82
9.1.2. Target Area 86
9.1.3. Target Origin Point
and Popup Alignment Point 90

9.2. Interaction Between Placement Properties.....92
9.3. 2.18.5 Overlapping of a Popup Window by Screen Borders.....95
9.4. Popup Positioning Configuration.....97
10. Menu.....98
10.1. Menu Control.....98
10.2. ContextMenu Control.....104
11. Tooltips.....108
12. Draggable Controls.....114
12.1. GridSplitter Control.....115
13. Events Routing.....121
13.1. Direct Routing.....122
13.2. Hit Testing in the Visual Layer.....123
13.3. Parameters of the Routed Events.....125
14. Input Handling.....128
14.1. Mouse.....128
14.2. Keyboard.....132
14.2.1. Focus.....134
14.2.2. Access Key.....139
15. Review of Basic Classes of Visual Tree Elements.....140
15.1. UIElement Class.....140
15.2. FrameworkElement Class.....141
15.3. Control Class.....143
16. Home Task.....145

Lesson materials are attached to this PDF file. In order to get access to the materials, open the lesson in Adobe Acrobat Reader.

1. Content Control Model

Previously, when considering base classes used for elements that are part of visual tree, the System.Windows.Controls.ContentControl class was mentioned, which in turn is inherited from the System.Windows.Controls.Control class. In other words, it is a class that describes a more specialized type of controls. Its specialization is that it can support only one child element placed in the System.Windows.Controls.ContentControl.**Content** property. This property has the System.Object data type, which enables it to be assigned with an object of any type.

When a content element is displayed on the screen, its content is also displayed. At that, if the object set as content has a type derived from System.Windows.UIElement, then it is displayed according to the appearance inherent to it. If the content object has another data type, then the System.Object.**ToString** method is called, which returns string representation of an object. This string is displayed as content.

Let's take the following piece of markup as an example:

```
<Button Background="Green">
    <StackPanel Margin="5" Orientation="Horizontal">
        <Button>Nested Button</Button>
        <TextBox Margin="10,0,0,0" Width="100">Nested
                TextBox</TextBox>
    </StackPanel>
</Button>
```

You can see the result of the above markup on the **Figure 1.**

Figure 1. Content property

This control model provides many possibilities when creating user interfaces. It may seem at first that elements that can only contain one other element as content are too limited in order to use them for building complex interfaces. However, this is not true. You can actually bypass this limitation by placing some panel as a child element. In this case, you can add more than one element to the panel, since they all support this functionality. By the way, the System.Windows.Window class, describing an upper level window, is also derived from the System.Windows.Controls.ContentControl class, and it can also contain only one content element. This is why when creating a new window in Visual Studio, a stub is created that contains a window and one child element — panel — which incorporates all other elements.

Properties of the System.Windows.Controls.ContentControl class:

- Content (System.Object). Gets or sets content of an element. Its default value is null.
- HasContent (System.Boolean). Gets value specifying if the element has content. True if it does, otherwise false.

1.1. Variations of Content Model

The above content model is basic. At that, WPF has several main variations of this model (Fig. 2).

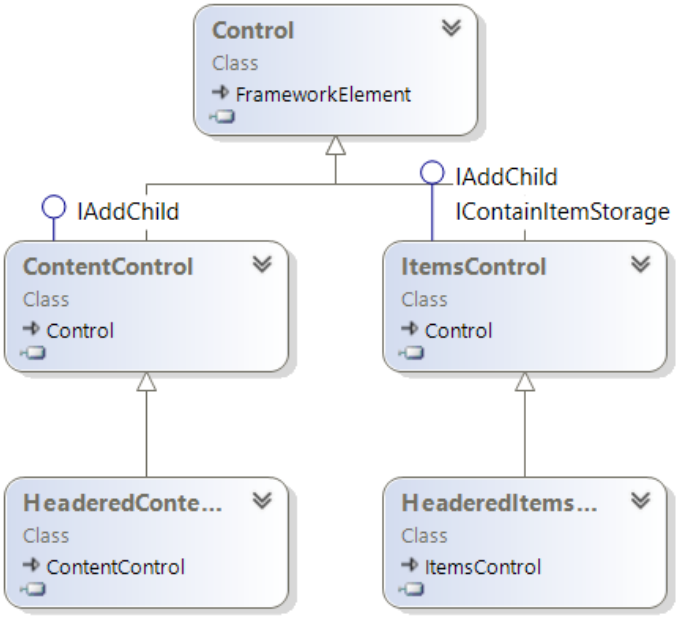

Figure 2. Class diagram showing main content models

1.1.1. The HeaderedContentControl Class

The System.Windows.Controls.HeaderedContentControl class extends the System.Windows.Controls.ContentControl class by using a header, which is used by derived classes to describe content of an element.

In order to set header to an element, use the System.Windows.Controls.HeaderedContentControl property, which, just like the property for specifying content, has the System.Object type, thus making it possible to place any object there. Header objects are displayed in the same way as the System.Windows.Controls.ContentControl.**Content** property, i.e., they either convert object to a string and display the string or display is described by the element.

Properties of the System.Windows.Controls.HeaderedContentControl class:

- **HasHeader** (System.Boolean). Gets value which specifies whether the element has a header. True if it does, otherwise false.
- **Header** (System.Object). Gets or sets element header. Its default value is null.

1.1.2. The ItemsControl Class

The System.Windows.Controls.ItemsControl class is a basic for controls that contain a list of child elements instead of one element. The content property of this class is System.Windows.Controls.ItemsControl.**Items**, which is also a collection. It allows using a simplified variant of markup, without explicitly specifying property name and collection, where child elements are placed. Data type of collection objects is the System.Object type, which allows incorporating any objects. At that, if the object type is derived from System.Windows.UIElement, then it is displayed according to the appearance inherent for it. But if there is another object data type in collection, then the System.Object.**ToString** method is called for it, which returns a string representation of the object. This string is displayed as a list item.

Properties of the System.Windows.Controls.ItemsControl class:

- **HasItems** (System.Boolean). Gets value specifying whether the element contains at least one object. True if it does, otherwise false.
- **Items** (System.Windows.Controls.ItemCollection). Gets a collection of objects used to generate element content. Empty collection is default value.

1.1.3. The HeaderedItemsControl Class

The System.Windows.Controls.HeaderedItemsControl class extends the System.Windows.Controls.ItemsControl class with a header used by derived classes in order to describe a list of child elements.

Properties of the System.Windows.Controls.HeaderedItemsControl class:

- **HasHeader** (System.Boolean). Gets value specifying whether the element has a header. True if it does, otherwise false.
- **Header** (System.Object). Gets or sets element header. Its default value is null.

2. Background and Foreground Brushes

WPF controls and some primitive elements have a set of properties describing the color of this or that part. The most common are the properties responsible for background and foreground colors. In most cases, background is the surface of a control, and foreground property is used for font color. In order to set values to such properties, one uses various brushes that are described with classes derived from the System.Windows.Media.Brush class (Fig. 3).

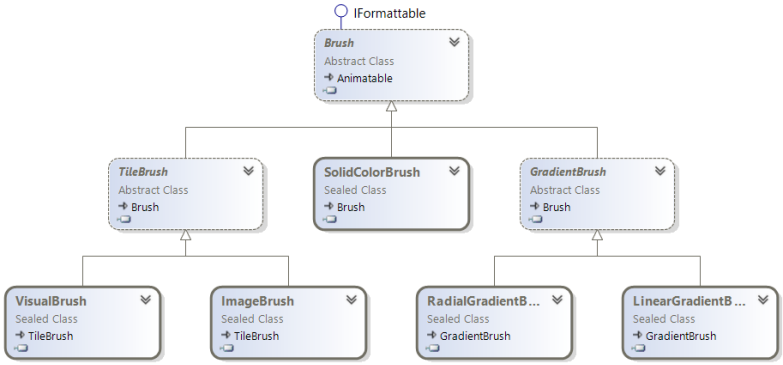

Figure 3. Class diagram describing brushes

Properties of the System.Windows.Media.Brush class:

- **Opacity** (System.Double). Gets or sets brush opacity. This property can take values from 0.0 (transparent) to 1.0 (opaque). Its default value is 1.0.

Some of the brushes require colors for different parameters to be specified as an instance of the System.Windows.Media.

Color structure, which describes color in terms of alpha, red, green, and blue channels. There are several ways to get this instance:

- Use one of static properties of the System.Windows.Media.Colors class. Each property of this class describes one of the most common colors. For example, System.Windows.Media.Colors.**Yellow**.
- Use one of static methods of the System.Windows.Media.Color structure designed to create new instance, for example, System.Windows.Media.Color.**FromArgb**.

When there is a need to set color value within the markup, you can specify a property name of the System.Windows.Media.Colors class or explicitly set values for each channel. The second option allows the following entry formats: "#AARRGGBB", "#ARGB", "#RRGGBB" и "#RGB". Each of the symbols must be substituted with any hexadecimal number in this entry.

Properties of the System.Windows.Media.Color structure:

- **A** (System.Byte). Gets or sets a value of the alpha channel. This property can take values ranged from 0 to 255.
- **B** (System.Byte). Gets or sets a value of the blue channel. This property can take values ranged from 0 to 255.
- **G** (System.Byte). Gets or sets a value of the green channel. This property can take values ranged from 0 to 255.
- **R** (System.Byte). Gets or sets a value of the red channel. This property can take values ranged from 0 to 255.
- **ScA** (System.Single). Gets or sets a value of the alpha channel. This property can take values ranged from 0.0 to 1.0.
- **ScB** (System.Single). Gets or sets a value of the blue channel. This property can take values ranged from 0.0 to 1.0.

- **ScG** (System.Single). Gets or sets a value of the green channel. This property can take values ranged from 0.0 to 1.0.
- **ScR** (System.Single). Gets or sets a value of the red channel. This property can take values ranged from 0.0 to 1.0.

Static methods of the System.Windows.Media.Color structure:

- **FromArgb**(a, r, g, b). Returns color based on the specified alpha channel and color channel values.
- **FromRgb**(r, g, b). Returns color based on the specified color channel values.
- **FromScRgb**(a, r, g, b). Returns color based on the specified alpha channel and color channel values.

2.1. The SolidColorBrush

System.Windows.Media.SolidColorBrush is used for solid color fill.

The System.Windows.Media.SolidColorBrush class properties:

- Color (System.Windows.Media.Color). Gets or sets brush color. Its default value is System.Windows.Media.Colors.**Transparent**.

The below markup snippet shows the brush (full example is in the folder Wpf.Brushes.SolidColorBrush.Xaml):

XAML

```
<Button Background="Green"
        FontSize="25"
        Foreground="#FFFFFF00"
        Height="46"
```

```
        Width="150">
    OK
</Button>
```

Figure 4 shows result of the above markup.

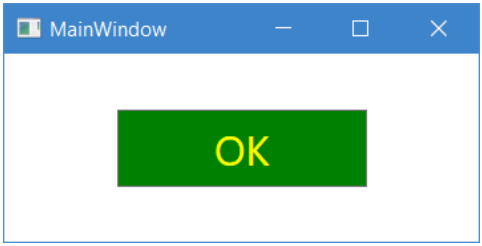

Figure 4. SolidColorBrush

The following code corresponds to the above markup (full example is in the folder Wpf.Brushes.SolidColorBrush.CSharp):

C#

```
var button = new Button
{
    Background = Brushes.Green,
    Content = "OK",
    FontSize = 25.0,
    Foreground = new SolidColorBrush(
        Color.FromArgb(a: 255, r: 255, g: 255, b: 0)
    ),
    Height = 46.0,
    Width = 150.0
};
```

2.2. Gradient Brushes

Brushes that describe gradient fill are inherited from the System.Windows.Media.GradientBrush class, which con-

tains a collection of gradient dots describing color and location of the transition point represented as the System.Windows.Media.GradientStop class. Each of the derived classes interprets this collection in its own way.

Properties of the System.Windows.Media.GradientBrush class:

- ColorInterpolationMode (System.Windows.Media.ColorInterpolationMode). Gets or sets a color interpolation mode of a gradient. Its default value is System.Windows.Media.ColorInterpolationMode.**SRgbLinearInterpolation**.
- GradientStops (System.Windows.Media.GradientStopCollection). Gets or sets a collection of gradient dots. Its default value is blank collection.
- SpreadMethod (System.Windows.Media.GradientSpreadMethod). Gets or sets a gradient spread mode, to be more specific, description of how the gradient must be displayed, which begins or ends within the scope of element display. Its default value is System.Windows.Media.GradientSpreadMethod.**Pad**.

In order to specify color interpolation mode, use the System.Windows.Media.ColorInterpolationMode enumeration which contains the following variants:

- **ScRgbLinearInterpolation**. The following properties will be used for interpolation: System.Windows.Media.Color.**ScA**, System.Windows.Media.Color.**ScR**, System.Windows.Media.Color.**ScG**, System.Windows.Media.Color.**ScB**.
- **SRgbLinearInterpolation**. The following properties will be used for interpolation: System.Windows.Media.Color.**A**,

System.Windows.Media.Color.**R**, System.Windows.Media.Color.**G**, System.Windows.Media.Color.**B**.

In order to specify gradient spread mode, use the System.Windows.Media.ColorInterpolationMode enumeration which contains the following variants:

- **Pad**. The color values at the ends of the gradient vector fill the remaining space in their direction.
- **Reflect**. Gradient is repeated in the reverse direction.
- **Repeat**. Gradient is repeated in the original direction.

Properties of the System.Windows.Media.GradientStop class:

- Color (System.Windows.Media.Color). Gets or sets a gradient dot color. Its default value is System.Windows.Media.Colors.**Transparent**.
- Offset (System.Double). Gets or sets offset on the gradient vector. This property can take values in the range from 0.0 (start point of the vector) to 1.0 (end point of the vector). Its default value is 0.0.

When describing gradient dots, you can use both positive and negative values, as well as 0. At that the (0,0) coordinate is bound to the upper left corner of the filled area, and the (1,1) coordinate to the lower right corner, regardless of size and proportions of the filled area.

2.2.1. LinearGradientBrush

System.Windows.Media.LinearGradientBrush is used for linear gradient fill which blends two or more colors across the straight line (gradient vector).

Properties of the System.Windows.Media.LinearGradientBrush class:

- **EndPoint** (System.Windows.Point). Gets or sets coordinates (in a two-dimensional coordinate system) that describe end point of the gradient vector. Its default value is (1,1).
- **StartPoint** (System.Windows.Point). Gets or sets coordinates (in a two-dimensional coordinate system) that describe start point of the gradient vector. Its default value is (0,0).

The below markup snippet shows this brush (full example is in the folder Wpf.Brushes.LinearGradientBrush.Xaml):

XAML

```
<Button Content="OK" FontSize="25" Height="46"
Width="150">
    <Button.Background>
        <LinearGradientBrush EndPoint="1,1"
StartPoint="0,0">
            <GradientStop Color="Red" Offset="0"/>
            <GradientStop Color="Green"
Offset="0.5"/>
            <GradientStop Color="Blue" Offset="1"/>
        </LinearGradientBrush>
    </Button.Background>
    <Button.Foreground>
        <LinearGradientBrush EndPoint="0,1"
StartPoint="0,0">
            <GradientStop Color="White" Offset="0"/>
            <GradientStop Color="#FF777777"
Offset="0.5"/>
            <GradientStop Color="White" Offset="1"/>
        </LinearGradientBrush>
    </Button.Foreground>
</Button>
```

Result of the above markup is in the Fig. 5.

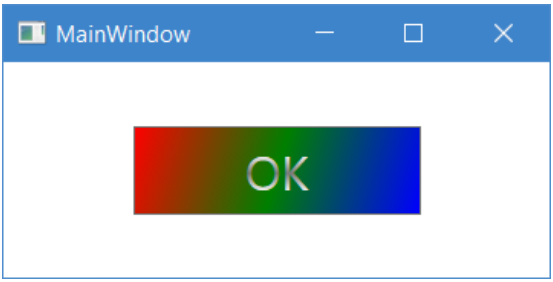

Figure 5. LinearGradientBrush

The following code corresponds to the above markup (full example is in the folder Wpf.Brushes.LinearGradientBrush.CSharp):

C#

```
var backgroundBrush = new LinearGradientBrush
{
    EndPoint = new Point(x: 1.0, y: 1.0),
    StartPoint = new Point(x: 0.0, y: 0.0)
};

backgroundBrush.GradientStops.Add(
        new GradientStop(Colors.Red, offset: 0.0));
backgroundBrush.GradientStops.Add(
        new GradientStop(Colors.Green, offset: 0.5));
backgroundBrush.GradientStops.Add(
        new GradientStop(Colors.Blue, offset: 1.0));

var foregroundBrush = new LinearGradientBrush
{
    EndPoint = new Point(x: 0.0, y: 1.0),
    StartPoint = new Point(x: 0.0, y: 0.0)
};

foregroundBrush.GradientStops.Add(
        new GradientStop(Colors.White, offset: 0.0));
```

```
foregroundBrush.GradientStops.Add(
        new GradientStop(Color.FromArgb(a: 0xFF,
        r: 0x77, g: 0x77, b: 0x77), offset: 0.5
    )
);

foregroundBrush.GradientStops.Add(
        new GradientStop(Colors.White, offset: 1.0));

var button = new Button
{
    Background = backgroundBrush,
    Content = "OK",
    FontSize = 25.0,
    Foreground = foregroundBrush,
    Height = 46.0,
    Width = 150.0
};
```

2.2.2. RadialGradientBrush

System.Windows.Media.RadialGradientBrush is used for radial gradient fill, which blends two or more colors across a circle.

Properties of the System.Windows.Media.RadialGradientBrush class:

- **Center** (System.Windows.Point). Gets or sets coordinates (in a two-dimensional coordinate system) of the gradient circle center.
- **GradientOrigin** (System.Windows.Point). Gets or sets coordinates (in a two-dimensional coordinate system) of the focal point that specifies the beginning of the gradient. Default value is (0.5, 0.5).

- **RadiusX** (System.Double). Gets or sets a horizontal radius of the gradient circle. Its default value is 0.5.
- **RadiusY** (System.Double). Gets or sets a vertical radius of the gradient circle. Its default value is 0.5.

The below markup snippet shows this brush (full example is in the folder Wpf.Brushes.RadialGradientBrush.Xaml):

XAML

```
<Button Content="OK" FontSize="25" Foreground="Gray"
        Height="46" Width="150">
    <Button.Background>
        <RadialGradientBrush GradientOrigin="0.5,0.5">
            <GradientStop Color="Red" Offset="0"/>
            <GradientStop Color="Green" Offset="0.5"/>
            <GradientStop Color="Blue" Offset="1"/>
        </RadialGradientBrush>
    </Button.Background>
</Button>
```

Figure 6 shows the result of the above markup.

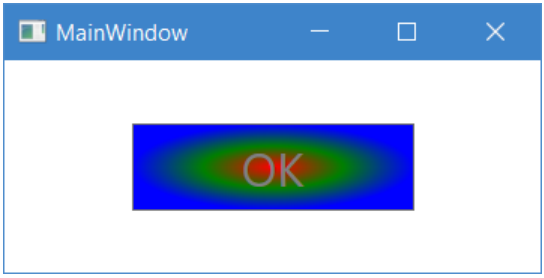

Figure 6. RadialGradientBrush

The following code corresponds to the above markup (full example is in the folder Wpf.Brushes.RadialGradientBrush.CSharp):

C#

```
var backgroundBrush = new RadialGradientBrush {
    GradientOrigin = new Point(0.5, 0.5) };

backgroundBrush.GradientStops.Add(
    new GradientStop(Colors.Red, offset: 0.0));

backgroundBrush.GradientStops.Add(
    new GradientStop(Colors.Green, offset: 0.5));

backgroundBrush.GradientStops.Add(
    new GradientStop(Colors.Blue, offset: 1.0));

var button = new Button
{
    Background = backgroundBrush,
    Content = "OK",
    FontSize = 25.0,
    Foreground = Brushes.Gray,
    Height = 46.0,
    Width = 150.0
};
```

2.3. Tile Brushes

Brushes that allow using images as a fill, they are derived from the System.Windows.Media.TileBrush class, which describes a tile principle of fill.

Properties of the System.Windows.Media.TileBrush class:

- **AlignmentX** (System.Windows.Media.AlignmentX). Gets or sets horizontal alignment for basic tile content. Its default value is System.Windows.Media.AlignmentX.**Center**.
- **AlignmentY** (System.Windows.Media.AlignmentY). Gets or sets vertical alignment for basic tile content. Its default value is System.Windows.Media.AlignmentY.**Center**.

- **Stretch** (System.Windows.Media.Stretch). Gets or sets content stretch mode. Its default value is System.Windows.Media.Stretch.**Fill**.
- **TileMode** (System.Windows.Media.TileMode). Gets or sets tile fill mode, which is used if basic tile is less than the filled area. Its default value is System.Windows.Media.TileMode.**None**.

In order to specify horizontal alignment for basic tile content, use the System.Windows.Media.AlignmentX enumeration, which contains the following options:

- **Center**. The content is centered.
- **Left**. The content is aligned left.
- **Right**. The content is aligned right.

In order to specify vertical alignment for basic tile content, use the System.Windows.Media.AlignmentY enumeration, which contains the following options:

- **Bottom**. The content is aligned toward the lower edge.
- **Center**. The content is centered.
- **Top**. The content is aligned toward the upper edge.

In order to specify stretch mode of the content, use the System.Windows.Media.Stretch enumeration, which contains the following options:

- **Fill**. The content is stretched to fill the whole space available. The aspect ratio is not preserved.
- **None**. No stretch. The content preserves its original size.
- **Uniform**. The content is stretched to fill the whole space available. The aspect ratio is preserved. At that, the content is shown in full.

- **UniformToFill**. The content is stretched to fill the whole space available. The aspect ratio is preserved. At that, if the content doesn't fit in full, it is cut.

In order to specify tile mode, use the System.Windows.Media.TileMode enumeration, which contains the following options:

- **FlipX**. It's the same as System.Windows.Media.TileMode.**Tile**, but alternating columns are flipped. Basic tile is displayed in its original form.
- **FlipXY**. It's a combination of System.Windows.Media.TileMode.**FlipX** and System.Windows.Media.TileMode.**FlipY**.
- **FlipY**. It's the same as System.Windows.Media.TileMode.**Tile**, but alternating lines are flipped. Basic tile is displayed in its original form.
- **None**. Basic tile is displayed without repetitions, and the remaining space remains transparent.
- **Tile**. Basic tile is displayed, and the remaining space is filled by repeating the base tile. The right edge of the tile joins the left edge of another tile, and top edge joins the bottom edge.

2.3.1. ImageBrush

System.Windows.Media.ImageBrush is used for tile fill with an image.

The System.Windows.Media.ImageBrush class properties:

- **ImageSource** (System.Windows.Media.ImageSource). Gets or sets image source.

The below markup snippet shows this brush (full example is in the folder Wpf.Brushes.ImageBrush.Xaml):

XAML

```
<Button Content="OK" FontSize="25" Foreground="Gray"
        Height="46" Width="150">
    <Button.Background>
        <ImageBrush ImageSource="/Succulent.jpg"
                    Stretch="UniformToFill"/>
    </Button.Background>
</Button>
```

Figure 7 shows result of the above markup.

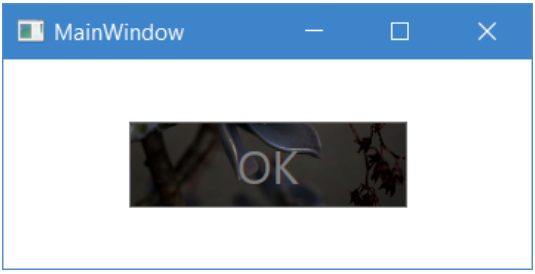

Figure 7. ImageBrush

The following code corresponds to the above markup (full example is in the folder Wpf.Brushes.ImageBrush.CSharp):

C#

```
var backgroundBrush = new ImageBrush
{
    ImageSource = new BitmapImage(new Uri("Succulent.
        jpg", UriKind.Relative)),
        Stretch = Stretch.UniformToFill
};

var button = new Button
{
    Background = backgroundBrush,
```

```
    Content = "OK",
    FontSize = 25.0,
    Foreground = Brushes.Gray,
    Height = 46.0,
    Width = 150.0
};
```

2.3.2. VisualBrush

System.Windows.Media.VisualBrush is used to fill a visual object with a visual.

Properties of the System.Windows.Media.VisualBrush class:

- **Visual** (System.Windows.Media.Visual). Gets or sets a visual element, the appearance of which is the fill content. Its default value is null.

The below markup snippet shows this brush (full example is in the folder Wpf.Brushes.VisualBrush.Xaml):

XAML

```
<StackPanel HorizontalAlignment="Center"
            VerticalAlignment="Center">
    <TextBox x:Name="visual" FontSize="25"
             Text="123" Width="150"/>
    <TextBlock>
        <TextBlock.Background>
            <VisualBrush Visual="{Binding
                        ElementName=visual}"/>
        </TextBlock.Background>
    </TextBlock>
</StackPanel>
```

Figure 8 shows result of the above markup.

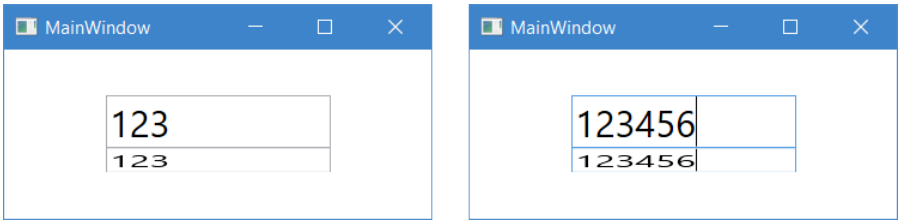

Figure 8. VisualBrush

This example used markup extension to specify which element is the content: Visual="{Binding ElementName=visual}". This markup extension is required to describe data binding. Its operating principle will be discussed in detail in the next sections.

The following code corresponds to the above markup (full example is in the folder Wpf.Brushes.VisualBrush.CSharp):

C#

```
var textBox = new TextBox { FontSize = 25.0,
                            Text = "123",
                            Width = 150.0 };
var textBlock = new TextBlock { Background =
                new VisualBrush(visual: textBox) };

var stackPanel = new StackPanel
{
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
stackPanel.Children.Add(textBox);
stackPanel.Children.Add(textBlock);
```

3. Fonts

Controls have a set of properties responsible for the font used at text display.

3.1. Font Size

The **FontSize** property can take only positive values. The specified value is measured in device independent units, which deviates from the commonly used units describing font size — points. In order to convert units used in WPF to points, you have to multiply them by 0.75, i.e., 26 points are equal to 48 WPF units.

3.2. Font Stretch

In order to specify the font stretch mode, use static properties of the System.Windows.FontStretches class, each of them specifies the percentage to which the font must be stretched from its normal aspect ratio.

FontStretch	% от нормального
UltraCondensed	50,0%
ExtraCondensed	62,5%
Condensed	75,0%
SemiCondensed	87,5%
Normal	100,0%
Medium	100,0%
SemiExpanded	112,5%
Expanded	125,0%
ExtraExpanded	150,0%
UltraExpanded	200,0%

3.3. Text density

In order to specify font weight, use static properties of the System.Windows.FontWeights class, each of them specifies value in the range from 1 to 999. The less the value, the less is the font weight and vice versa, the greater the value, the greater the font weight.

FontWeight	Value
Thin	100
ExtraLight/UltraLight	200
Light	300
Normal/Regular	400
Medium	500
DemiBold/SemiBold	600
Bold	700
ExtraBold/UltraBold	800
Black/Heavy	900
ExtraBlack/ExtraHeavy	950

3.4. Font Style

In order to specify the font style, use static properties of the System.Windows.FontStyles class.

FontStyle	Description
Normal	The font is not italic.
Italic	Italic font is used for.
Oblique	Italic font is generated automatically by transforming a normal one. It is used in the cases, when font doesn't have initial italic variant.

3.5. Font Family

Font family is defined with a set of related typefaces; typography rules and symbol appearance are specified separately for them. They are treated as components of one family. For example, **Arial Regular**, **Arial Bold**, **Arial Italic**, and **Arial Bold Italic** are components of the **Arial** font family.

If you specify that the element must use the **Arial** font family and specify font weight as System.Windows.FontWeights.**Bold**, them the **Arial Bold** typeface will be set.

The below markup creates two buttons, which fonts are considered the same in terms of result:

XAML

```
<Button FontFamily="Arial"
        FontWeight="Bold">Button Text</Button>
<Button FontFamily="Arial Bold">Button Text</Button>
```

3.6. Inheritance of Fonts

If you set a property responsible for some font characteristic, the value of this property will be inherited by all child elements until one of them redefines it explicitly. In other words, if you set family font to an upper level window (<Window> element), the same font family will be used for all elements of this window.

These properties can be inherited even by the elements that don't support the inheritance, for example, the <StackPanel> element.

4. Primitive Elements

WPF provides a wide variety of controls to build user interface. However, as it was said before, not all WPF elements are controls. These are the elements that have a certain preset behavior and possibility to interact with the user and appearance template. There is a row of so-called primitive elements (Fig. 9), which provide some very basic functionality (output of text, image, video, etc.).

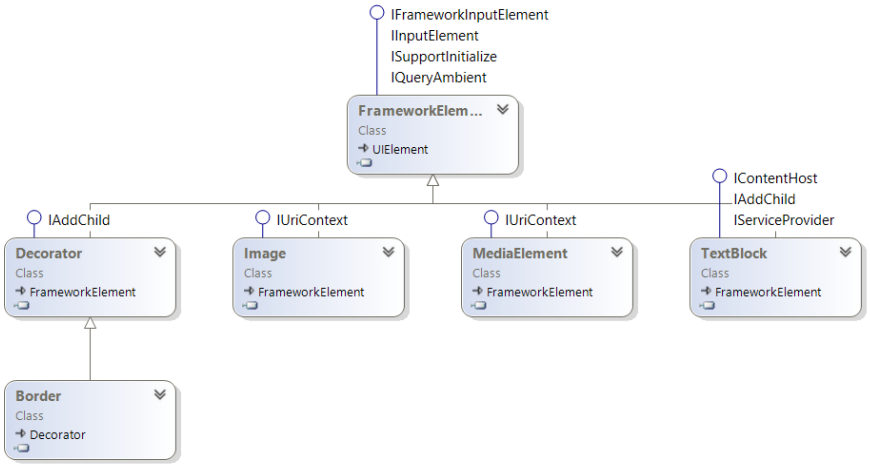

Figure 9. Class diagram that describes primitive elements

4.1. TextBlock Element

System.Windows.Controls.TextBlock element represents a lightweight control for displaying text.

Properties of the System.Windows.Controls.TextBlock class:

- **Background** (System.Windows.Media.Brush). Gets or sets a brush used for filling element background. Its default value is null.
- **BaselineOffset** (System.Double). Gets or sets offset from the text baseline. This property can take the System.Double.**NaN** value, which means that offset must be calculated as optimal based on the current font settings. Its default value is System.Double.**NaN**.
- **FontFamily** (System.Windows.Media.FontFamily). Gets or sets a font family. Its default value is System.Windows.SystemFonts.**MessageFontFamily**.
- **FontSize** (System.Double). Gets or sets a font size. This property can take only positive values. Its default value is System.Windows.SystemFonts.**MessageFontSize**.
- **FontStretch** (System.Windows.FontStretch). Gets or sets font stretch mode. Its default value is System.Windows.FontStretches.**Normal**.
- **FontStyle** (System.Windows.FontStyle). Gets or sets a font style. Its default value is System.Windows.SystemFonts.**MessageFontStyle**.
- **FontWeight** (System.Windows.FontWeight). Gets or sets a font weight. Its default value is System.Windows.SystemFonts.**MessageFontWeight**.
- **Foreground** (System.Windows.Media.Brush). Gets or sets a brush used to fill foreground of an element (text). Its default value is System.Windows.Media.Brushes.**Black**.
- **IsHyphenationEnabled** (System.Boolean). Gets or sets a value that specifies whether the hyphenation is enables. True if it does; otherwise false. Its default value is false.

- **LineHeight** (System.Double). Gets or sets a line height. This property can take the System.Double.**NaN** value, which means that the line height must be calculated as optimal based on the current font settings. Its default value is System.Double.**NaN**.
- **Text** (System.String). Gets or sets a text. Its default value is System.Double.**NaN**.
- **TextAlignment** (System.Windows.TextAlignment). Gets or sets horizontal alignment mode for text. Its default value is System.Windows.TextAlignment.**Left**.
- **TextTrimming** (System.Windows.TextTrimming). Gets or sets trimming of content at overflow. Its default value is System.Windows.TextTrimming.**None**.
- **TextWrapping** (System.Windows.TextWrapping). Gets or sets text wrapping mode. Its default value is System.Windows.TextWrapping.**NoWrap**.

In order to specify text trimming mode at overflow, use the System.Windows.Controls.TextTrimming enumeration, which contains the following options:

- **CharacterEllipsis**. Text is trimmed at a character boundary. An ellipsis (…) is drawn in place of remaining text.
- **None**. Text is not trimmed.
- **WordEllipsis**. Text is trimmed at a word boundary. Ellipsis (…) is drawn in place of remaining text.

The below markup snippet shows this element (full example is in the folder Wpf.Primitives.TextBlock.Xaml):

XAML

```
<TextBlock FontStyle="Italic" FontWeight="Bold"
Text="Simple Text"/>
```

Figure 10 shows result of the above markup.

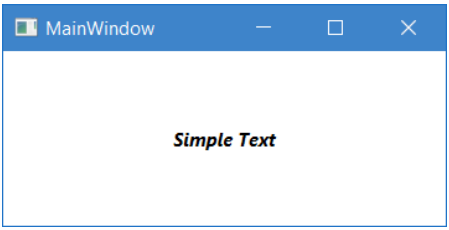

Figure 10. The TextBlock primitive element

The following code corresponds to the above markup (full example is in the folder Wpf.Primitives.TextBlock.CSharp):

C#

```
var textBlock = new TextBlock
{
    FontStyle = FontStyles.Italic,
    FontWeight = FontWeights.Bold,
    Text = "Simple Text"
};
```

4.2. Image Element

System.Windows.Controls.Image is designed for image output.

Properties of the System.Windows.Controls.Image class:

- **Source** (System.Windows.Media.ImageSource). Gets or sets an image source. Its default value is null.
- **Stretch** (System.Windows.Media.Stretch). Gets or sets stretch mode of an image. Its default value is System.Windows.Media.Stretch.**Uniform**.
- **StretchDirection** (System.Windows.Controls.StretchDirection). Gets or sets image scaling mode. Its default

value is System.Windows.Controls.StretchDirection.**Both**.

Events of the System.Windows.Controls.Image class:

- **ImageFailed** (System.EventHandler<System.Windows.ExceptionRoutedEventArgs>). Is triggered when exception is thrown at reading image source.

In order to specify image scaling, use the System.Windows.Media.StretchDirection enumeration, which contains the following options:

- **Both**. Image is stretched in any direction with the value of the System.Windows.Controls.Image.**Stretch** property taken into account.
- **DownOnly**. Image can be only stretched down in case it is bigger than the parent element space available.
- **UpOnly**. Image can be only stretched up in case it is smaller than the parent element space available.

The below markup snippet shows this element (full example is in the folder Wpf.Primitives.Image.Xaml):

XAML

```
<Image Margin="10" Source="Succulent.jpg"/>
```

Figure 11 shows result of the above markup.

The following code corresponds to the above markup (full example is in the folder Wpf.Primitives.Image.CSharp):

C#

```
var image = new Image
{
```

```
    Margin = new Thickness(10.0),
    Source = new BitmapImage(new Uri("/Succulent.jpg",
             UriKind.Relative))
};
```

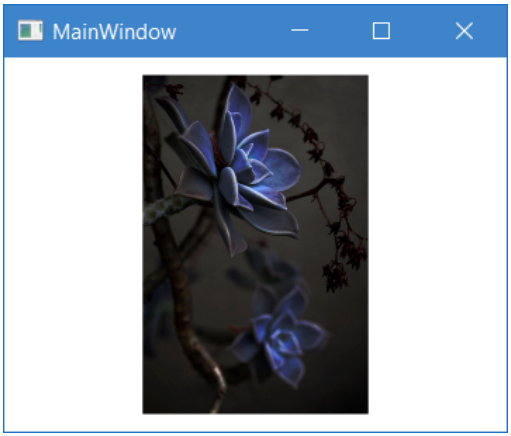

Figure 11. The Image primitive element

4.3. MediaElement

System.Windows.Controls.MediaElement plays audio and video of the content.

Property of the System.Windows.Controls.MediaElement class:

- **Balance** (System.Double). Gets or sets a ratio of volume across speakers. This property can take values from -1.0 (volume is on the left speaker) to 1.0 (volume is on the right speaker). Its default value is 0.0.
- **BufferingProgress** (System.Double). Gets content buffering progress. This property returns value in the range from 0.0 (progress 0%) to 1.0 (progress 100%).

- **CanPause** (System.Boolean). Gets value indicating whether the content being played back can be paused. True if it can, otherwise false.
- **DownloadProgress** (System.Double). Gets a progress of the content download, which is placed on a remote server. This property returns value in the range from 0.0 (progress 0%) to 1.0 (progress 100%).
- **HasAudio** (System.Boolean). Gets a value indicating whether the content has audio. True if the content has audio, otherwise false.
- **HasVideo** (System.Boolean). Gets a value indicating whether the content has video. True if it does, otherwise false.
- **IsBuffering** (System.Boolean). Gets a value indicating whether the content is buffered. True if it does, otherwise false.
- **IsMuted** (System.Boolean). Gets or sets a value indicating whether the audio is muted for the content. True if it does, otherwise false. Its default value is false.
- **LoadedBehavior** (System.Windows.Controls.MediaState). Gets or sets a value for content after load. Its default value is System.Windows.Controls.MediaState.**Play**.
- **NaturalDuration** (System.Windows.Duration). Gets natural duration of the content.
- **NaturalVideoHeight** (System.Int32). Gets natural height of a video.
- **NaturalVideoWidth** (System.Int32). Gets natural width of a video.
- **ScrubbingEnabled** (System.Boolean). Gets or sets a value indicating whether the element will update frames for seek

operation while paused. True if frames must be updated, otherwise false. Its default value is false.

- **Source** (System.Uri). Gets or sets content source. Its default value is null.
- **SpeedRatio** (System.Double). Gets or sets content playback speed ratio. This property can take values in the range from 0.0 to System.Double.**PositiveInfinity**. Negative values are converted to 0.0. Values in the range from 0.0 to 1.0 slow down the playback, and values greater than 1.0 speed it up. Its default value is 1.0.
- **Stretch** (System.Windows.Media.Stretch). Gets or sets stretch mode of the content. Its default value is System.Windows.Media.Stretch.**Uniform**.
- **StretchDirection** (System.Windows.Controls.StretchDirection). Gets or sets scaling mode of the content. Its default value is System.Windows.Controls.StretchDirection.**Both**.
- **Position** (System.TimeSpan). Gets or sets the current playback position for content. Its default value is System.TimeSpan.**Zero**.
- **UnloadedBehavior** (System.Windows.Controls.MediaState). Gets or sets content behavior after it is unloaded.
- **Volume** (System.Double). Gets or sets volume. This property can take values in the range from 0.0 (volume off) to 1.0 (maximum volume). Its default value is 0.5.

Events of the System.Windows.Controls.MediaElement class:

- **BufferingEnded** (System.Windows.RoutedEventHandler). Occurs whenever the content buffering ends.

- **BufferingStarted** (System.Windows.RoutedEventHandler). Occurs whenever the content buffering starts.
- **MediaEnded** (System.Windows.RoutedEventHandler). Occurs whenever the content playback ends.
- **MediaFailed** (System.EventHandler<System.Windows.ExceptionRoutedEventArgs>). Occurs whenever an exception is thrown at content reading.
- **MediaOpened** (System.Windows.RoutedEventHandler). Occurs whenever the content load ends.

Methods of the System.Windows.Controls.MediaElement class:

- **Close**(). Closes the content.
- **Pause**(). Pauses the content.
- **Play**(). Plays the content.
- **Stop**(). Stops the content.

In order to indicate behavior of the content at load or unload, use the System.Windows.Controls.MediaState enumeration, which has the following options:

- **Close**. The content must be closed and all its resources released.
- **Manual**. The content is controlled manually. The content will be loaded, but it won't be played automatically.
- **Pause**. The content must be paused.
- **Play**. The content will be played.
- **Stop**. The content will be stopped. At that, its resources won't be released.

The below markup snippet shows this element (full example is in the folder Wpf.Primitives.MediaElement.Xaml):

XAML

```
<MediaElement Source="KellyMcGarry.mp4"/>
```

Figure 12 shows result of the above markup.

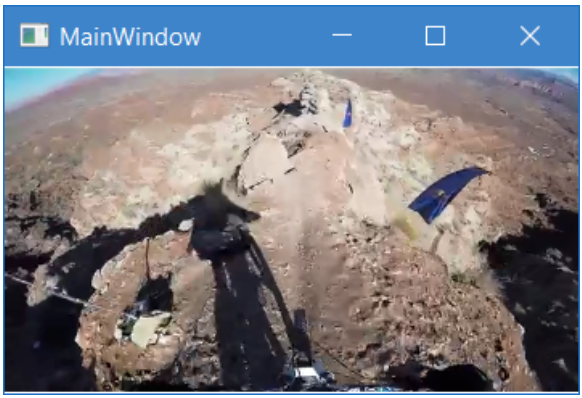

Figure 12. Primitive MediaElement

The following code corresponds to the above markup (full example is in the folder Wpf.Primitives.MediaElement.CSharp):

C#

```
var mediaElement = new MediaElement
{
    Source = new Uri("KellyMcGarry.mp4",
                     UriKind.Relative)
};
```

4.4. Border Element

The System.Windows.Controls.Border element outputs background, border, or both.

Properties of the System.Windows.Controls.Border class:

- **Background** (System.Windows.Media.Brush). Gets or sets a brush used to fill areas between element borders.
- **BorderBrush** (System.Windows.Media.Brush). Gets or sets a brush used to fill element border.
- **BorderThickness** (System.Windiws.Thickness). Gets or sets thickness of a border.
- **CornerRadius** (System.Windows.CornerRadius). Gets or sets a corner radius of each of border corners.
- **Padding** (System.Windows.Thickness). Gets or sets inner paddings for the element (between the border and content).

The below markup snippet shows this element (full example is in the folder Wpf.Primitives.Border.Xaml):

XAML

```
<Border Background="Yellow"
        BorderBrush="Green"
        BorderThickness="5"
        CornerRadius="10"
        Margin="10"/>
```

Figure 13 shows result of the above markup.

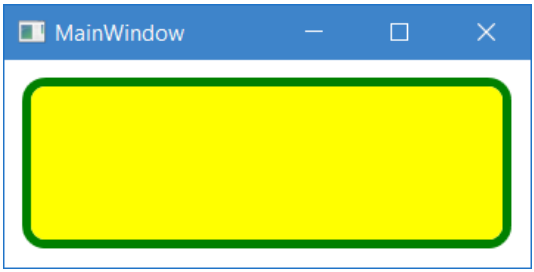

Figure 13. The Border primitive element

The following code corresponds to the above markup (full example is in the folder Wpf.Primitives.MediaElement.CSharp):

C#

```
var border = new Border
{
    Background = Brushes.Yellow,
    BorderBrush = Brushes.Green,
    BorderThickness = new Thickness(5.0),
    CornerRadius = new CornerRadius(10.0),
    Margin = new Thickness(10.0)
};
```

5. Label Control

The System.Windows.Controls.Label control is used to display text connected with another control. This control is inherited from the System.Windows.Controls.ContentControl class (Fig. 14), and thus it supports the above described content control model, i.e., it can contain not only text, but virtually anything. Despite this, it is often used to display text because of its distinctive feature which the previously discussed System.Windows.Controls.TextBlock primitive element doesn't have. By this I mean support of access keys (also known as mnemonics), pressing shortcuts to pass focus to the control connected to it.

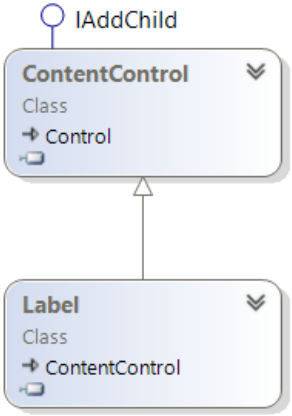

Figure 14. Diagram of classes that describe labels

Properties of the System.Windows.Controls.Label class:

- **Target** (System.Windows.UIElement). Gets or sets an element that has to get focus when clicking a control key connected to label. Its default value is null.

The below markup snippet shows this element (full example is in the folder Wpf.Controls.Label.Xaml):

XAML

```
<Grid HorizontalAlignment="Center"
      VerticalAlignment="Center">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Label Grid.Column="0"
           Target="{Binding ElementName=username}">
           _Username:
    </Label>
    <TextBox x:Name="username"
             Grid.Column="1" Width="200"/>
</Grid>
```

Figure 15 shows result of the above markup.

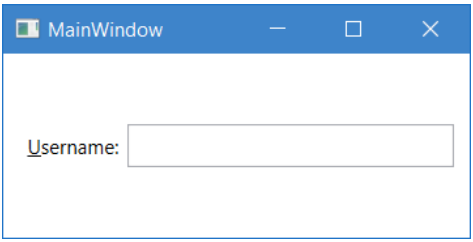

Figure 15. Label Control

In order to indicate a shortcut, press the underscore symbol (_) before the symbol of the required key. If you need to output the underscore symbol itself, just double it: (__).

In order to indicate element that must get focus when clicking a shortcut, use the System.Windows.Controls.Label. **Target** property.

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Label.CSharp):

C#

```
var textBox = new TextBox { Width = 200.0 };
var label = new Label { Content = "_Username:",
                        Target = textBox };
var grid = new Grid
{
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};
grid.ColumnDefinitions.Add(new ColumnDefinition
     { Width = GridLength.Auto });
grid.ColumnDefinitions.Add(new ColumnDefinition());

grid.Children.Add(label);
grid.Children.Add(textBox);

Grid.SetColumn(label, 0);
Grid.SetColumn(textBox, 1);
```

6. Grouping Controls

WPF grouping controls are derived from the System.Windows.Controls.HeaderedContentControl class (Fig. 16). They group several logically connected controls together and name the appeared group.

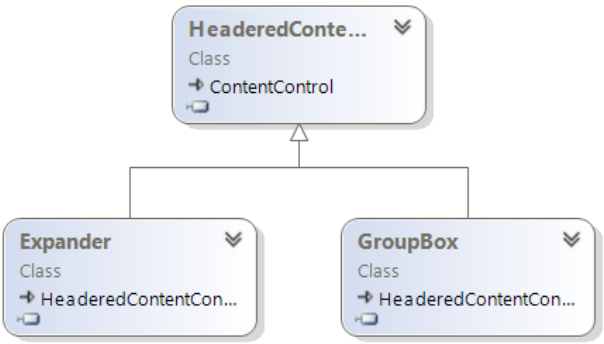

Figure 16. Class diagram that describes grouping elements

6.1. GroupBox Control

The System.Windows.Controls.GroupBox control allows "wrapping" content in a border with header.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.GroupBox.Xaml):

XAML

```
<GroupBox Header="Languages"
          HorizontalAlignment="Center"
          VerticalAlignment="Center"
          Width="150">
```

```
    <StackPanel Margin="5">
        <RadioButton>English</RadioButton>
        <RadioButton Margin="0,5,0,0">French
              </RadioButton>
        <RadioButton Margin="0,5,0,0">Spanish
              </RadioButton>
    </StackPanel>
</GroupBox>
```

Figure 17 shows result of the above markup.

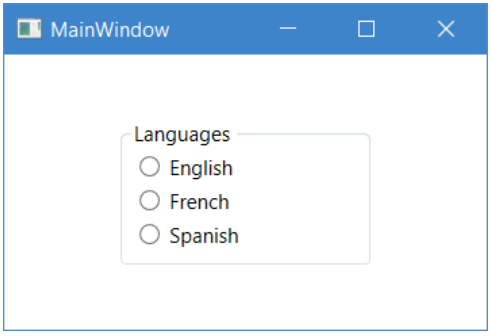

Figure 17. The GroupBox control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.GroupBox.CSharp):

C#

```
var radioButton1 = new RadioButton { Content =
                   "English" };
var radioButton2 = new RadioButton
{
    Content = "French",
    Margin = new Thickness(0.0, 5.0, 0.0, 0.0)
};
var radioButton3 = new RadioButton
{
```

```
    Content = "Spanish",
    Margin = new Thickness(0.0, 5.0, 0.0, 0.0)
};

var stackPanel = new StackPanel { Margin =
                 new Thickness(5.0) };
stackPanel.Children.Add(radioButton1);
stackPanel.Children.Add(radioButton2);
stackPanel.Children.Add(radioButton3);

var groupBox = new GroupBox
{
    Content = stackPanel,
    Header = "Languages",
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center,
    Width = 150.0
};
```

6.2. Expander Control

The System.Windows.Controls.Expander control allows creating a header to its content placed in an area, which can be collapsed or expanded if needed.

Properties of the System.Windows.Controls.Expander class:

- **ExpandDirection** (System.Windows.Controls.ExpandDirection). Gets or sets direction of the content expanded. Its default value is System.Windows.Controls.ExpandDirection.**Down**.
- **IsExpanded** (System.Boolean). Gets or sets values which indicate whether the element content is displayed. True if they do, otherwise false. Its default value is false.

Properties of the System.Windows.Controls.Expander class:

- **Collapsed** (System.Windows.RoutedEventHandler). Occurs when the content of grouping element is collapsed.
- **Expanded** (System.Windows.RoutedEventHandler). Occurs when the content of grouping element is expanded.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.Expander.Xaml):

XAML

```
<Expander Header="Languages" Margin="30" Width="200">
    <StackPanel Margin="5">
        <RadioButton>English</RadioButton>
        <RadioButton Margin="0,5,0,0">French
              </RadioButton>
        <RadioButton Margin="0,5,0,0">Spanish
              </RadioButton>
    </StackPanel>
</Expander>
```

Figure 18 shows result of the above markup.

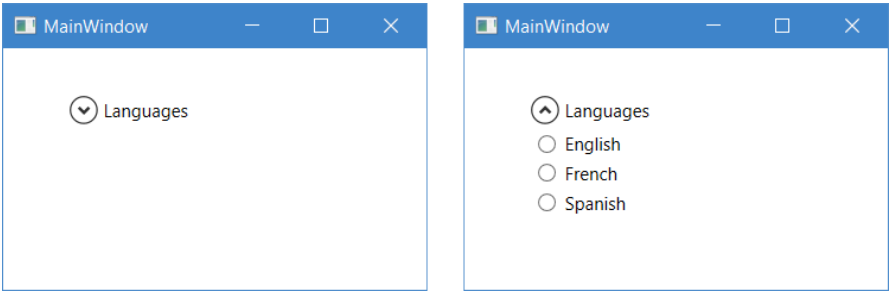

Figure 18. The Expander control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Expander.CSharp):

C#

```
var radioButton1 = new RadioButton { Content =
                    "English" };
var radioButton2 = new RadioButton
{
    Content = "French",
    Margin = new Thickness(0.0, 5.0, 0.0, 0.0)
};

var radioButton3 = new RadioButton
{
    Content = "Spanish",
    Margin = new Thickness(0.0, 5.0, 0.0, 0.0)
};

var stackPanel = new StackPanel { Margin =
                 new Thickness(5.0) };
stackPanel.Children.Add(radioButton1);
stackPanel.Children.Add(radioButton2);
stackPanel.Children.Add(radioButton3);

var expander = new Expander
{
    Content = stackPanel,
    Header = "Languages",
    Margin = new Thickness(30.0),
    Width = 200.0
};
```

7. Range Controls

Selection of a specific value from a valid range of values is a pretty common problem. Controls, which need this functionality, are derived from the System.Windows.Controls.Primitives.RangeBase base class (Fig. 19).

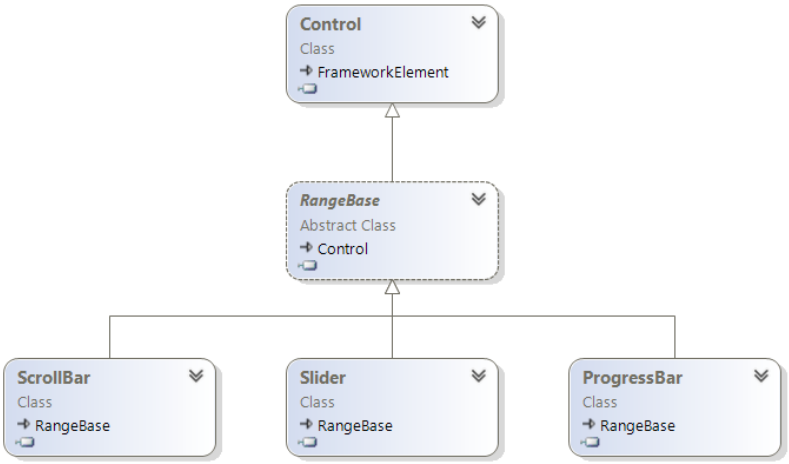

Figure 19. Class diagram that describes range controls

Properties of the System.Windows.Controls.Primitives.RangeBase class:

- **LargeChange** (System.Double). Gets or sets a value which is subtracted from or added to the current value of the System.Windows.Controls.Primitives.RangeBase. **Value** property. Its default value is 1.0.
- **Maximum** (System.Double). Gets or sets a maximum value for the System.Windows.Controls.Primitives.RangeBase. **Value** property. Its default value is 1.0.

- **Minimum** (System.Double). Gets or sets a minimum value for the System.Windows.Controls.Primitives.RangeBase.**Value** property. Its default value is 0.0.
- **SmallChange** (System.Double). Gets or sets a value, which must be subtracted from or added to the current value of the System.Windows.Controls.Primitives.RangeBase.**Value** property. Its default value is 0.1.
- **Value** (System.Double). Gets or sets the current value set in the element. Its default value is 0.0.

Events of the System.Windows.Controls.Primitives.RangeBase class:

- **ValueChanged** (System.Windows.RoutedPropertyChangedEventHandler<System.Double>). Occurs when value of the System.Windows.Controls.Primitives.RangeBase.**Value** property is changed.

Properties System.Windows.Controls.Primitives.RangeBase.**SmallChange** and System.Windows.Controls.Primitives.RangeBase.**LargeChange** can be interpreted by derived classes in a different way. For example, the System.Windows.Controls.ProgressBar class doesn't use them at all, and the System.Windows.Controls.Primitives.ScrollBar class uses one value when clicking a scroll bar arrow, and another value when clicking the scroll bar itself.

7.1. ProgressBar Control

The System.Windows.Controls.ProgressBar control is used in the situations which require progress of an operation to be visualized.

Properties of the System.Windows.Controls.ProgressBar class:

- **IsIndeterminate** (System.Boolean). Gets or sets a value, which indicates whether the progress is displayed taking into account the current value set in an element or progress is displayed in general form. True if progress must be displayed in general form, otherwise false. Its default value is false.
- **Orientation** (System.Windows.Controls.Orientation). Gets or sets an element orientation (horizontal or vertical). Its default value is System.Windows.Controls.Orientation.**Horizontal**.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.ProgressBar.Xaml):

XAML

```
<ProgressBar Height="30" Maximum="100" Minimum="0"
             Value="35" Width="200"/>
```

Figure 20 shows result of the above markup.

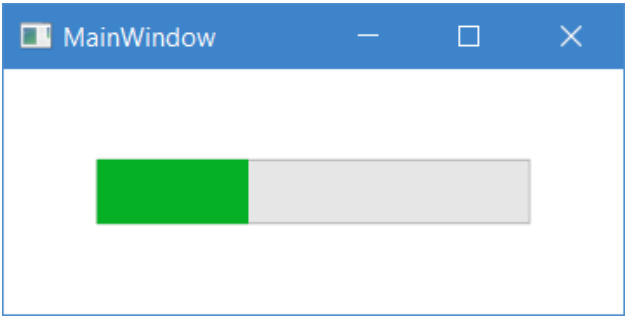

Figure 20. The ProgressBar control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.ProgressBar.CSharp):

C#

```
var progressBar = new ProgressBar
{
    Height = 30.0,
    Maximum = 100.0,
    Minimum = 0.0,
    Value = 35.0,
    Width = 200.0
};
```

7.2. Slider Control

The System.Windows.Controls.Slider control allows the user to select value of a specific parameter using slider from the valid range of values.

Properties of the System.Windows.Controls.Slider class:

- **AutoToolTipPlacement** (System.Windows.Controls.Primitives.AutoToolTipPlacement). Gets or sets display mode of a tip when clicking slider, which displays the current selected value. When specifying display mode, placement is also specified. Its default value is System.Windows.Controls.Primitives.AutoToolTipPlacement.**None**.
- **AutoToolTipPrecision** (System.Int32). Gets or sets a number of digits, which must be displayed after decimal point, when displaying a tip with the current selected value. This property can take only positive values. Its default value is 0.
- **Delay** (System.Int32). Gets or sets the amount of time in milliseconds that must pass before the slider starts to move when clicking areas to the left or to the right from the slider. Its default value is System.Windows.SystemParameters.**KeyboardDelay**.

- **Interval** (System.Int32). Gets or sets amount of time in milliseconds that must pass before the slider is moved for the second time when clicking areas to the left or to the right from the slider. Its default value is System.Windows.SystemParameters.**KeyboardSpeed**.
- **IsDirectionReversed** (System.Boolean). Gets or sets direction, in which values are increased. True if values are increased to the left, for horizontal slider, and down for vertical slider, otherwise false. Its default value is false.
- **IsMoveToPointEnabled** (System.Boolean). Gets or sets a value, which indicates whether the slider must be moved immediately to the location of mouse click. True if the slider must be moved immediately to the location of mouse click, otherwise false. Its default value is false.
- **IsSelectionRangeEnabled** (System.Boolean). Gets or sets a value, which indicates whether the selection range must be displayed on an element. True if the selection range must be displayed, otherwise false. Its default value is false.
- **IsSnapToTickEnabled** (System.Boolean). Gets or sets a value, which indicates whether the slider must be moved between ticks or freely. True if the slider must be moved between ticks, otherwise false. Its default value is false.
- **Orientation** (System.Windows.Controls.Orientation). Gets or sets element orientation (horizontal or vertical). Its default value is System.Windows.Controls.Orientation.**Horizontal**.
- **SelectionEnd** (System.Double). Gets or sets a maximum value of the selection range on an element. Its default value is 0.0.

- **SelectionStart** (System.Double). Gets or sets a minimum value of the selection range on an element. Its default value is 0.0.
- **TickFrequency** (System.Double). Gets or sets frequency, at which ticks are placed on an element. Its default value is 1.0.
- **TickPlacement** (System.Windows.Controls.Primitives.TickPlacement). Gets or sets mode of tick marks display on an element. When specifying mode, placement is also specified. Its default value is System.Windows.Controls.Primitives.TickPlacement**.None**.
- **Ticks** (System.Windows.Media.DoubleCollection). Gets or sets a collection of values, for which tick marks must be displayed on an element. Its default value is empty collection.

In order to specify display mode of a tip when clicking on a slider, use the System.Windows.Controls.Primitives.AutoToolTipPlacement enumeration, which contains the following options:

- **BottomRight.** Tip is displayed under a slider for horizontal slider. Tip is displayed to the right of the slider for horizontal slider.
- **None.** Tip is not displayed.
- **TopLeft.** For a horizontal slider, tip is displayed under it. For vertical slider, tip is displayed to the left of it.

In order to display tick placement on a slider, use the System.Windows.Controls.Primitives.TickPlacement enumeration, which has the following options:

- **Both**. For horizontal slider, tick marks are displayed under it. For vertical slider, tick marks are displayed to the left or to the right of it.
- **BottomRight**. For horizontal slider, tick marks are displayed under it. For vertical slider, tick marks are displayed to the right of it.
- **None**. Tick marks are not displayed.
- **TopLeft**. For horizontal slider, tick marks are displayed under it. For vertical slider, tick marks are displayed to the left of it.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.Slider.Xaml):

XAML

```
<Grid Margin="10">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Slider Grid.Row="0" Maximum="100" Value="10"/>
    <Slider Grid.Row="1"
            Maximum="100"
            TickFrequency="10"
            TickPlacement="BottomRight"
            Value="10"/>
    <Slider Grid.Row="2"
            Maximum="100"
            Ticks="0,5,10,15,25,50,100"
            TickPlacement="BottomRight"
            Value="10"/>
    <Slider Grid.Row="3"
```

```
            IsSelectionRangeEnabled="True"
            Maximum="100"
            TickFrequency="10"
            TickPlacement="BottomRight"
            SelectionEnd="75"
            SelectionStart="25"
            Value="10"/>
</Grid>
```

Figure 21 shows result of the above markup.

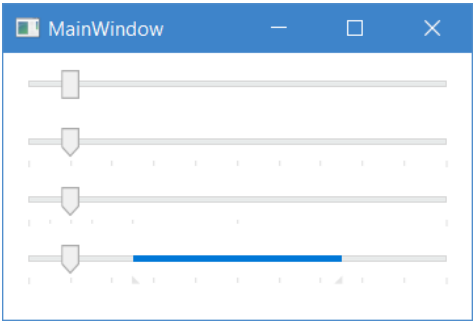

Figure 21. The Slider control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Slider.CSharp):

C#

```
var slider1 = new Slider { Maximum = 100.0, Value =
10.0 };
var slider2 = new Slider
{
    Maximum = 100.0,
    TickFrequency = 10.0,
    TickPlacement = TickPlacement.BottomRight,
    Value = 10.0
};
```

```
var slider3 = new Slider
{
    Maximum = 100.0,
    TickPlacement = TickPlacement.BottomRight,
    Ticks = new DoubleCollection(new[] { 0.0, 5.0,
            10.0, 15.0, 25.0, 50.0, 100.0 }),
            Value = 10.0
};

var slider4 = new Slider
{
    IsSelectionRangeEnabled = true,
    Maximum = 100.0,
    SelectionEnd = 75.0,
    SelectionStart = 25.0,
    TickFrequency = 10.0,
    TickPlacement = TickPlacement.BottomRight,
    Value = 10.0
};

var grid = new Grid { Margin = new Thickness(10.0) };
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());

Grid.SetRow(slider1, 0);
Grid.SetRow(slider2, 1);
Grid.SetRow(slider3, 2);
Grid.SetRow(slider4, 3);

grid.Children.Add(slider1);
grid.Children.Add(slider2);
grid.Children.Add(slider3);
grid.Children.Add(slider4);
```

7.3. ScrollBar Control

The System.Windows.Controls.Primitives.ScrollBar control represents a scroll bar, which is usually used to control the display area of the scrolled content.

Properties of the System.Windows.Controls.Primitives.ScrollBar class:

- **Orientation** (System.Windows.Controls.Orientation). Gets or sets element orientation (horizontal or vertical). Its default value is System.Windows.Controls.Orientation.**Vertical**.
- **ViewportSize** (System.Double). Gets or sets size of the viewed element content, for which the current scroll bar is used. This value is used to calculate the slider size. The more content is visible, the bigger the size of the slider is. Its default value is 0.0.

Events of the System.Windows.Controls.Primitives.ScrollBar class:

- **Scroll** (System.Windows.Controls.Primitives.ScrollEventHandler). Occurs when the slider is moved after scroll.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.ScrollBar.Xaml):

XAML

```
<Grid Margin="10">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition Width="Auto"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
```

```
    </Grid.RowDefinitions>
    <ScrollBar Grid.Column="1"
               Grid.Row="0"
               Maximum="50"
               Value="15"/>
    <ScrollBar Grid.Column="0"
               Grid.Row="1"
               Maximum="30"
               Orientation="Horizontal"
               Value="10"/>
</Grid>
```

Figure 22 shows result of the above markup.

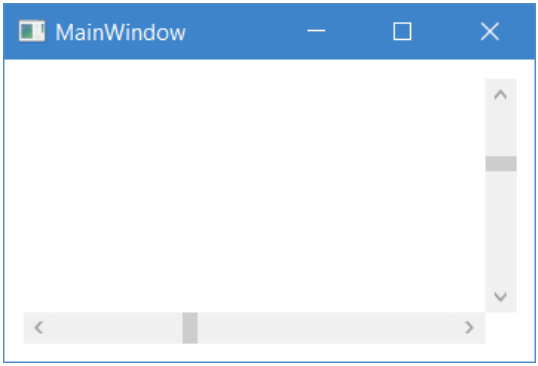

Figure 22. The ScrollBar element

The following code corresponds to the above markup (full example is in the folder Wpf.Controls. ScrollBar.CSharp):

C#

```
var scrollBar1 = new ScrollBar { Maximum = 50.0,
Value = 15.0 };
var scrollBar2 = new ScrollBar
{
    Maximum = 30.0,
```

```
    Orientation = Orientation.Horizontal,
    Value = 10.0
};

var grid = new Grid { Margin = new Thickness(10.0) };
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition { Width =
                           GridLength.Auto });
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition { Height =
                           GridLength.Auto });

Grid.SetColumn(scrollBar1, 1);
Grid.SetRow(scrollBar1, 0);
Grid.SetColumn(scrollBar2, 0);
Grid.SetRow(scrollBar2, 1);

grid.Children.Add(scrollBar1);
grid.Children.Add(scrollBar2);
```

8. Items Control

WPF has a group of controls intended to work with a set of objects (list). Base class for them is the System.Windows.Controls.ItemsControl class (Fig. 23), which provides basic functionality to store and process collections of objects. The controls that store objects placed in them as a single-level collection are derived from the intermediate class System.Windows.Controls.Primitives.Selector, which provides functionality for object selection processing.

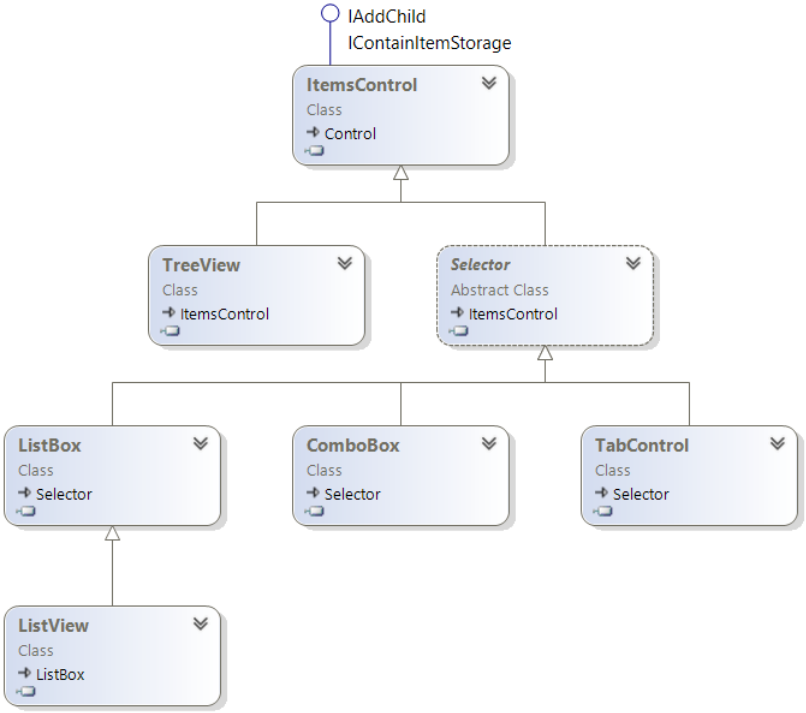

Figure 23. Class diagram that describes items controls

Properties of the System.Windows.Controls.Primitives.Selector class:

- **SelectedIndex** (System.Int32). Gets or sets index of the first object in the current selection. If there is no selection, the -1 value is used. Its default value is -1.
- **SelectedItem** (System.Object). Gets or sets the first object in the current selection. If there is no selection, the null value is used. Its default value is null.

Events of the System.Windows.Controls.Primitives.Selector class:

- **SelectionChanged** (System.Windows.Controls.SelectionChangedEventHandler). Occurs when the selection statin the element is not changed.

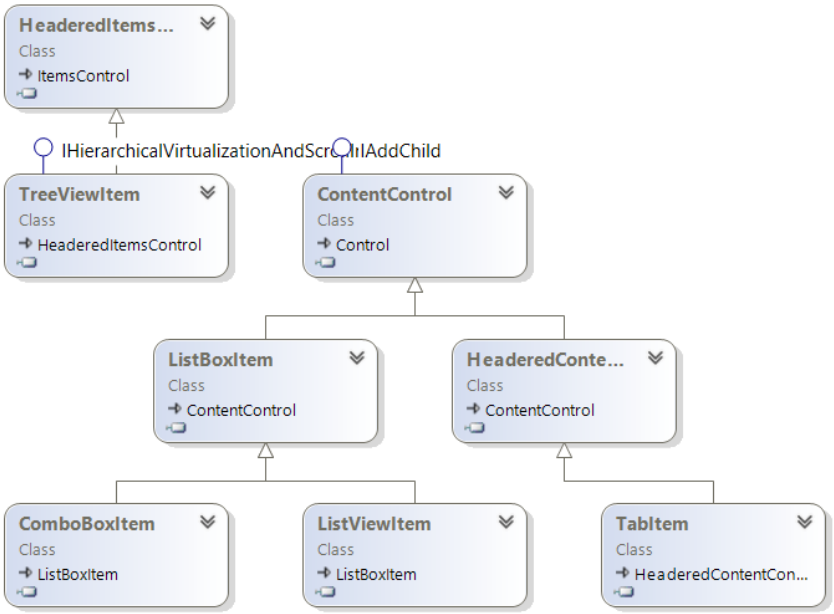

Figure 24. Class diagram that describes element containers

Content control model provided by the System.Windows.Controls.ItemsControl class works in such a way that each element placed in a collection of child elements is wrapped in a special element container. Each items control has its own type of this container. If fine tuning of an element container is required, there is a possibility to describe its creation explicitly. Figure 24 shows element container classes intended specifically to be used as wrappings above the content of each item element.

8.1. ListBox Control

The System.Windows.Controls.ListBox control is intended to display a list of objects. If there are too many objects placed in an element, scroll bars appear to navigate through the content. Element also supports several selection modes.

Properties of the System.Windows.Controls.ListBox class:

- **SelectedItems** (System.Collections.IList). Gets an object collection of the current selection. Its default value is empty collection.
- **SelectionMode** (System.Windows.Controls.SelectionMode). Gets or sets object selection mode in an element. Its default value is System.Windows.Controls.SelectionMode.**Single**.

Methods of the System.Windows.Controls.ListBox class:

- **ScrollIntoView**(item). Scrolls element content in such a way to make the indicated object appear in view area.
- **SelectAll**(). Selects all objects in an element.
- **UnselectAll**(). Unselects all objects in an element.

In order to indicate object selection mode, use the System.Windows.Controls.SelectionMode enumeration, which has the following options:

- **Extended**. More than one object can be selected. By pressing Shift, you can select more than one element at a time.
- **Multiple**. More than one object can be selected. Selection with Shift being pressed is not supported.
- **Single**. Only one object can be selected at a time.

The System.Windows.Controls.ListBoxItem class is applied as an element container for the System.Windows.Controls.ListBox control.

Properties of the System.Windows.Controls.ListBoxItem class:

- **IsSelected** (System.Boolean). Gets or sets a value, which indicates whether the element is selected. True if it is, otherwise false. Its default value is false.

Events of the System.Windows.Controls.ListBoxItem class:

- **Selected** (System.Windows.RoutedEventHandler). Occurs when element is selected.
- **Unselected** (System.Windows.RoutedEventHandler). Occurs when element loses selection.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.ListBox.Xaml):

XAML

```
<ListBox Height="100"
         SelectionMode="Multiple"
         Width="200">
    <ListBoxItem>Red</ListBoxItem>
```

```
    <ListBoxItem IsSelected="True">Green</ListBoxItem>
    <ListBoxItem IsSelected="True">Blue</ListBoxItem>
</ListBox>
```

Figure 25 shows result of the above markup.

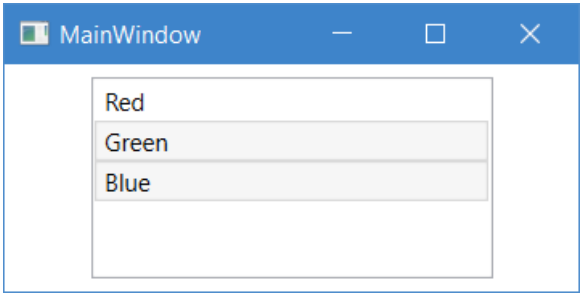

Figure 25. The ListBox control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.ListBox.CSharp):

C#

```
var item1 = new ListBoxItem { Content = "Red" };
var item2 = new ListBoxItem { Content = "Green",
IsSelected = true };
var item3 = new ListBoxItem { Content = "Blue",
IsSelected = true };

var listBox = new ListBox
{
    Height = 100.0,
    SelectionMode = SelectionMode.Multiple,
    Width = 200.0
};
listBox.Items.Add(item1);
listBox.Items.Add(item2);
listBox.Items.Add(item3);
```

8.2. ListView Control

The System.Windows.Controls.ListView control displays a list of objects. This element, unlike the System.Windows.Controls.ListBox control, can display objects as a table with named columns. To implement a table view, use data binding, which will be discussed in the next sections.

8.3. ComboBox Control

The System.Windows.Controls.ComboBox control is different from other items control because it displays not more than one selected object. The rest of objects are hidden in the drop-down menu, which is displayed when clicking the element.

Properties of the System.Windows.Controls.ComboBox class:

- **IsDropDownOpen** (System.Boolean). Gets or sets a value, which indicates whether the drop-down menu is open. True if it is, otherwise false. Its default value is false.
- **IsEditable** (System.Boolean). Gets or sets a value, which indicates whether the selected value is displayed as a button or as a text box. True if the element displays a text box, otherwise false. Its default value is false.
- **IsReadOnly** (System.Boolean). Gets or sets a value, which indicates whether text in a text box is editable (if its display mode is enabled, of course). True if text is read only, otherwise false. Its default value is false.
- **MaxDropDownHeight** (System.Double). Gets or sets maximum height of a drop-down menu.
- **ShouldPreserveUserEnteredPrefix** (System.Boolean). Gets or sets a value, which indicates whether the text entered by

the user must be displayed in its original form or it must be replaced with the value of the existing object. True if the value entered by the user must be left, otherwise false. Its default value is false.

- **StaysOpenOnEdit** (System.Boolean). Gets or sets a value, which indicates whether the drop-down list must remain open after the user clicks on the text box. True if the drop-down list must remain open, otherwise false. Its default value is false.
- **Text** (System.String). Gets or sets text of the current object. Its default value is System.String.**Empty**.

Events of the System.Windows.Controls.ComboBox class:

- **DropDownClosed** (System.EventHandler). Occurs when the drop-down list is closed.
- **DropDownOpened** (System.EventHandler). Occurs when the drop-down list is opened.

The System.Windows.Controls.ComboBoxItem class is applied as an element container for the System.Windows.Controls.ComboBox control.

Properties of the System.Windows.Controls.ComboBoxItem class:

- **IsHighlighted** (System.Boolean). Gets or sets a value, which indicates if the element is highlighted. True if is, otherwise false. Its default value is false.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.ComboBox.Xaml):

XAML

```
<ComboBox Height="23" Width="200">
    <ComboBoxItem>Red</ComboBoxItem>
```

```
    <ComboBoxItem IsSelected="True">Green
          </ComboBoxItem>
    <ComboBoxItem>Blue</ComboBoxItem>
</ComboBox>
```

Figure 26 shows result of the above markup.

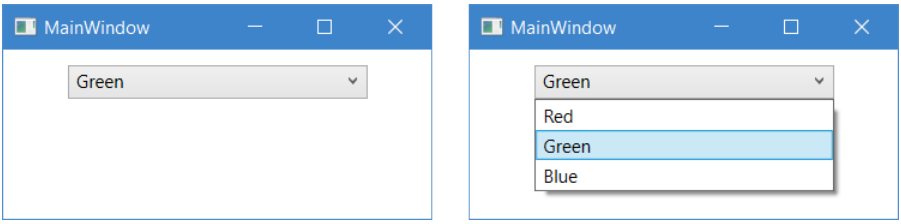

Figure 26. The ComboBox control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.ComboBox.CSharp):

C#

```
var item1 = new ComboBoxItem { Content = "Red" };
var item2 = new ComboBoxItem { Content = "Green",
            IsSelected = true };
var item3 = new ComboBoxItem { Content = "Blue" };
var comboBox = new ComboBox { Height = 23.0,
                              Width = 200.0};
comboBox.Items.Add(item1);
comboBox.Items.Add(item2);
comboBox.Items.Add(item3);
```

8.4. TreeView Control

The System.Windows.Controls.TreeView control displays a hierarchical (tree-like) data structure. Each object placed in an element is a node, which has its own value and collection of child nodes.

Properties of the System.Windows.Controls.TreeView class:

- **SelectedItem** (System.Object). Gets or sets selected item in the element. If there is no selection, the null value is used. Its default value is false.

Events of the System.Windows.Controls.TreeView class:

- **SelectedItemChanged** (System.Windows.RoutedPropertyChangedEventHandler<System.Object>). Occurs when selection state in an element is changed.

Element description placed in the System.Windows.Controls.TreeView items control differs from the ones discussed above. The difference is that the System.Windows.Controls.ListBox and System.Windows.Controls.ComboBox elements contain a single-layer element collection intended for display. The System.Windows.Controls.TreeView element describes a hierarchical element structure. This means that each element, apart from its content, can also contain a set of other (child) elements designed in the same way. The System.Windows.Controls.TreeViewItem class is used to describe such structure.

Properties of the System.Windows.Controls.TreeViewItem class:

- **IsExpanded** (System.Boolean). Gets or sets a value, which indicates whether child elements are expanded. True if they are, otherwise false. Its default value is false.
- **IsSelected** (System.Boolean). Gets or sets a value, which indicates whether an element is selected. True if it is, otherwise false. Its default value is false.
- **IsSelectionActive** (System.Boolean). Gets value, which indicates whether the element has an entry focus. True if it has, otherwise false. Its default value is false.

Events of the System.Windows.Controls.TreeViewItem class:

- **Collapsed** (System.Windows.RoutedEventArgs). Occurs when an element collapses its child elements.
- **Expanded** (System.Windows.RoutedEventArgs). Occurs when an element expands its child elements.
- **Selected** (System.Windows.RoutedEventArgs). Occurs when an element is selected.
- **Unselected** (System.Windows.RoutedEventArgs). Occurs when an element loses selection.

Methods of the System.Windows.Controls.TreeViewItem class:

- **ExpandSubtree**(). Expands the current element and all its child elements.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.TreeView.Xaml):

XAML

```
<TreeView>
    <TreeViewItem Header="Europe">
        <TreeViewItem Header="England">
            <TreeViewItem Header="London"/>
            <TreeViewItem Header="Liverpool"/>
        </TreeViewItem>

        <TreeViewItem Header="France">
            <TreeViewItem Header="Marseille"/>
        </TreeViewItem>
    </TreeViewItem>
    <TreeViewItem Header="North America">
        <TreeViewItem Header="Canada">
```

```
            <TreeViewItem Header="Toronto"/>
            <TreeViewItem Header="Montreal"
                          IsSelected="True"/>
            <TreeViewItem Header="Kingston"/>
        </TreeViewItem>
    </TreeViewItem>

    <TreeViewItem Header="South America">
        <TreeViewItem Header="Argentina">
            <TreeViewItem Header="Buenos Aires"/>
            <TreeViewItem Header="Rosario"/>
        </TreeViewItem>
    </TreeViewItem>
</TreeView>
```

Figure 27 shows result of the above markup.

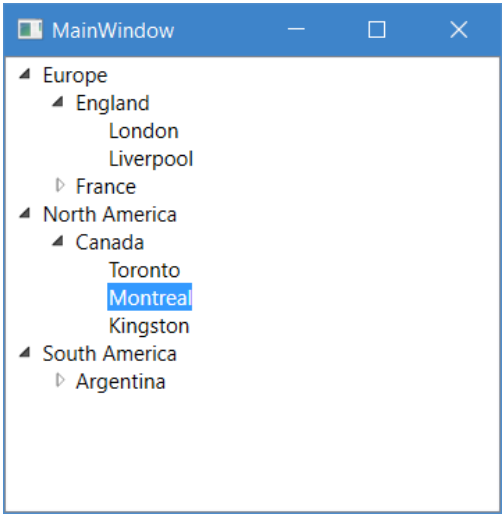

Figure 27. The TreeView control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.TreeView.CSharp):

C#

```
var item31 = new TreeViewItem { Header = "London" };
var item32 = new TreeViewItem { Header = "Liverpool" };
var item33 = new TreeViewItem { Header = "Marseille" };
var item34 = new TreeViewItem { Header = "Toronto" };
var item35 = new TreeViewItem { Header = "Montreal",
             IsSelected = true };
var item36 = new TreeViewItem { Header = "Kingston" };
var item37 = new TreeViewItem { Header = "Buenos Aires"};
var item38 = new TreeViewItem { Header = "Rosario" };

var item21 = new TreeViewItem { Header = "England" };
item21.Items.Add(item31);
item21.Items.Add(item32);

var item22 = new TreeViewItem { Header = "France" };
item22.Items.Add(item33);

var item23 = new TreeViewItem { Header = "Canada" };
item23.Items.Add(item34);
item23.Items.Add(item35);
item23.Items.Add(item36);

var item24 = new TreeViewItem { Header = "Argentina" };
item24.Items.Add(item37);
item24.Items.Add(item38);

var item11 = new TreeViewItem { Header = "Europe" };
item11.Items.Add(item21);
item11.Items.Add(item22);

var item12 = new TreeViewItem { Header = "North America"};
item12.Items.Add(item23);

var item13 = new TreeViewItem { Header = "South America"};
item13.Items.Add(item24);

var treeView = new TreeView();
treeView.Items.Add(item11);
treeView.Items.Add(item12);
treeView.Items.Add(item13);
```

8.5. TabControl

System.Windows.Controls.TabControl displays content as tabs. At first it may seem that this element has nothing in common with the previous items controls, but it is not true. This control can display only one of the selected tabs, which, in fact, are a list of elements available to be displayed. This is why its base class is System.Windows.Controls.Primitives.Selector, which supports selection of one element from a set of available ones.

Properties of the System.Windows.Controls.TabControl class:

- **TabStripPlacement** (System.Windows.Controls.Dock). Gets or sets the tab strip alignment mode in relation to the content of an active tab. Its default value is System.Windows.Controls.Dock.**Top**.

The System.Windows.Controls.TabItem class is applied as an element container for System.Windows.Controls.TabControl.

Properties of the System.Windows.Controls.TabItem class:

- **IsSelected** (System.Boolean). Gets or sets a value, which indicates if the current element is selected. True if it is, otherwise false. Its default value is false.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.TabControl.Xaml):

XAML

```
<TabControl>
    <TabItem Header="Colors">
        <ListBox Margin="10">
            <ListBoxItem>Red</ListBoxItem>
            <ListBoxItem>Green</ListBoxItem>
```

```
            <ListBoxItem>Blue</ListBoxItem>
        </ListBox>
    </TabItem>
    <TabItem Header="Cities" IsSelected="True">
        <ListBox Margin="10">
            <ListBoxItem>London</ListBoxItem>
            <ListBoxItem>Paris</ListBoxItem>
            <ListBoxItem>Berlin</ListBoxItem>
        </ListBox>
    </TabItem>
</TabControl>
```

Figure 28 shows result of the above markup.

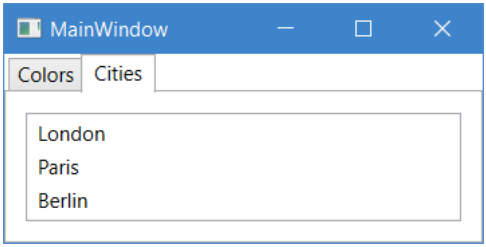

Figure 28. TabControl

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.TabControl.CSharp):

C#

```
var item1 = new ListBoxItem { Content = "Red" };
var item2 = new ListBoxItem { Content = "Green" };
var item3 = new ListBoxItem { Content = "Blue" };

var listBox1 = new ListBox { Margin =
               new Thickness(10.0) };
listBox1.Items.Add(item1);
listBox1.Items.Add(item2);
listBox1.Items.Add(item3);
```

```
var item4 = new ListBoxItem { Content = "London" };
var item5 = new ListBoxItem { Content = "Paris" };
var item6 = new ListBoxItem { Content = "Berlin" };

var listBox2 = new ListBox { Margin =
               new Thickness(10.0) };
listBox2.Items.Add(item4);
listBox2.Items.Add(item5);
listBox2.Items.Add(item6);

var tabItem1 = new TabItem { Content = listBox1,
               Header = "Colors" };
var tabItem2 = new TabItem { Content = listBox2,
               Header = "Cities", IsSelected = true };

var tabControl = new TabControl();
tabControl.Items.Add(tabItem1);
tabControl.Items.Add(tabItem2);
```

9. Popup Windows

The System.Windows.Controls.Primitives.Popup control is a popup window with content. It is used to display a part of user interface in a separate window, which is displayed over the current window of an application in relation to the assigned element or a point on the screen.

Properties of the System.Windows.Controls.Primitives.Popup class:

- **CustomPopupPlacementCallback** (System.Windows.Controls.Primitives.CustomPopupPlacementCallback)**. Gets or sets callback method, which is used to identify position of a popup window, if the** System.Windows.Controls.ContextMenu**.Placement property value equals to** System.Windows.Controls.Primitives.PlacementMode**.Custom.**
- **HasDropShadow** (System.Boolean). Gets or sets a value, which indicates, whether the popup window must drop shadow. True if a popup window must drop shadow, otherwise false. Its default value is false.
- **HorizontalOffset** (System.Double). Gets or sets horizontal offset in respect to target area. Its default value is 0.0.
- **IsOpen** (System.Boolean). Gets or sets a value, which indicates whether a popup window is displayed. True if it is, otherwise false. Its default value is false.
- **Placement** (System.Windows.Controls.Primitives.PlacementMode). Gets or sets placement mode of a popup window. Its default value is System.Windows.Controls.Primitives.PlacementMode**.MousePoint**.

- **PlacementRectangle** (System.Windows.Rect). Gets or sets area in respect to which a popup window is positioned when displayed. Its default value is System.Windows.Rect.**Empty**.
- **PlacementTarget** (System.Windows.UIElement). Gets or sets an element, in respect to which a popup window is positioned when displayed. Its default value is null.
- **StaysOpen** (System.Boolean). Gets or sets a value, which indicates whether a popup window is closed automatically. True if the popup window must remain opened until this property is assigned false; otherwise false. Its default value is false.
- **VerticalOffset** (System.Double). Gets or sets vertical offset in relation to target area. Its default value is 0.0.

Events of the System.Windows.Controls.Primitives.Popup class:

- **Closed** (System.Windows.RoutedEventArgs). Occurs when a popup window is closed.
- **Opened** (System.Windows.RoutedEventArgs). Occurs when a popup window is opened.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.Popup.Overview.Xaml):

XAML

```
<Button Click="Button_Click">
    <StackPanel>
        <TextBlock Text="Display Popup"/>
        <Popup x:Name="popup">
            <TextBlock Background="Gray"
                       Foreground="White"
                       Padding="3"
```

```
                        Text="Popup Text"/>
        </Popup>
    </StackPanel>
</Button>
```

Figure 29 shows result of the above markup.

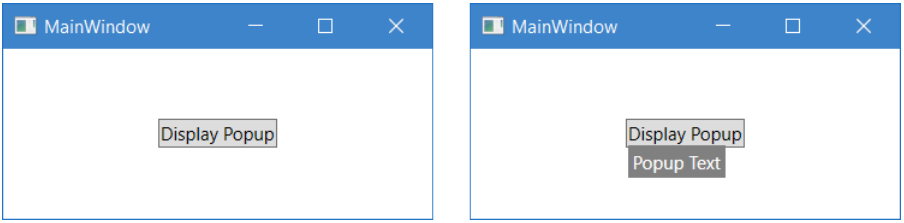

Figure 29. The Popup control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Popup.Overview.CSharp):

C#

```
var textBlock1 = new TextBlock
{
    Background = Brushes.Gray,
    Foreground = Brushes.White,
    Padding = new Thickness(3.0),
    Text = "Popup Text"
};

var popup = new Popup { Child = textBlock1 };
var textBlock2 = new TextBlock { Text = "Display Popup" };
var stackPanel = new StackPanel();
stackPanel.Children.Add(textBlock2);
stackPanel.Children.Add(popup);
var button = new Button { Content = stackPanel };
button.Click += (sender, e) => popup.IsOpen = true;
```

The System.Windows.Controls.Primitives.Popup control contains basic functionality to describe popup windows and it is worth using if you need to create a new control or to describe template for the existing one. The most common types of WPF popup windows have a number of ready classes:

- System.Windows.Controls.ToolTip. Tip.
- System.Windows.Controls.ContextMenu. Context menu.
- System.Windows.Controls.ComboBox. Combo box.

9.1. Popup Window Placement

Popup window may be placed in respect to a control, mouse, or screen by using the following set of properties: System.Windows.Controls.Primitives.Popup**.Placement**, System.Windows.Controls.Primitives.Popup**.PlacementTarget**, System.Windows.Controls.Primitives.Popup**.PlacementRectangle**, System.Windows.Controls.Primitives.Popup**.HorizontalOffset** and System.Windows.Controls.Primitives.Popup**.VerticalOffset**.

The below markup snippet shows how popup windows can be placed by using the System.Windows.Controls.Primitives.Popup**.Placement** property only (full example is in the folder Wpf.Controls.Popup.Placement.Xaml):

XAML

```
<Grid HorizontalAlignment="Center"
VerticalAlignment="Center">
    <Image Height="250" Source="Succulent.jpg"/>
    <Popup IsOpen="True" Placement="Bottom">
        <TextBlock Background="LightBlue"
                   Padding="3"
                   Text="Placement = Bottom"/>
    </Popup>
```

```
    <Popup IsOpen="True" Placement="Left">
        <TextBlock Background="LightBlue"
                   Padding="3" Text="Placement = Left"/>
    </Popup>
    <Popup IsOpen="True" Placement="Right">
        <TextBlock Background="LightBlue" Padding="3"
                   Text="Placement = Right"/>
    </Popup>
    <Popup IsOpen="True" Placement="Top">
        <TextBlock Background="LightBlue" Padding="3"
                   Text="Placement = Top"/>
    </Popup>
</Grid>
```

Figure 30 shows result of the above markup.

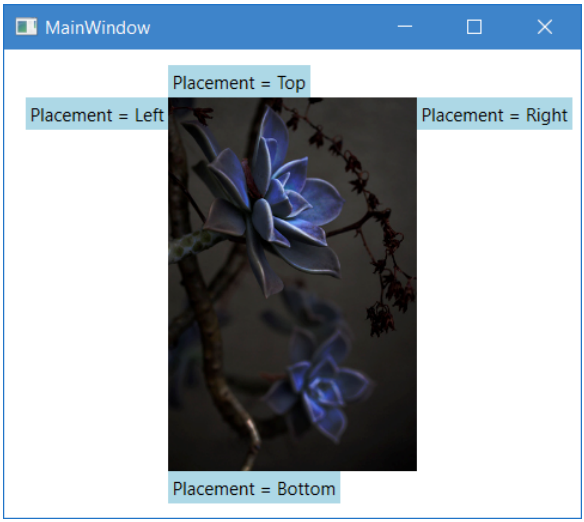

Figure 30. The Popup control, placed in respect to the parent element

In this case, the popup window is placed in respect to its parent element.

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Popup.Placement.CSharp):

C#

```
var image = new Image
{
    Height = 250.0,
    Source = new BitmapImage(new Uri("/Succulent.jpg",
             UriKind.Relative))
};

var textBlock1 = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Placement = Bottom"
};

var popup1 = new Popup { Child = textBlock1,
             Placement = PlacementMode.Bottom };
var textBlock2 = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Placement = Left"
};

var popup2 = new Popup { Child = textBlock2,
             Placement = PlacementMode.Left };
var textBlock3 = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Placement = Right"
};
```

```
var popup3 = new Popup { Child = textBlock3,
             Placement = PlacementMode.Right };
var textBlock4 = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Placement = Top"
};

var popup4 = new Popup { Child = textBlock4,
             Placement = PlacementMode.Top };
var grid = new Grid
{
    HorizontalAlignment = HorizontalAlignment.Center,
    VerticalAlignment = VerticalAlignment.Center
};

grid.Children.Add(image);
grid.Children.Add(popup1);
grid.Children.Add(popup2);
grid.Children.Add(popup3);
grid.Children.Add(popup4);
```

In order to make explanation of popup window placement properties more clear, we have to introduce several terms: target object, target area, target origin point, and popup alignment point. This terminology provides a convenient way of addressing to various aspects of a popup window and a control associated with it.

9.1.1. Target Object

Target object is a control associates with a popup window. If a value is set to System.Windows.Controls.Primitives.Popup.**PlacementTarget**, then it describes a target object.

If this property is not set and popup window has a parent element, then parent element is a target object. If a value is not set to the System.Windows.Controls.Primitives.Popup.**PlacementTarget** property and popup window doesn't have a parent element, it is placed in respect to the screen.

The below markup snippet shows creation of a popup window with explicit indication to a target object and without it (full example is in the folder Wpf.Controls.Popup.TargetObject.Xaml):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Canvas Background="Green" Grid.Column="0"
            Margin="30,30,15,30">
        <Border Background="Yellow"
                Canvas.Left="20"
                Canvas.Top="20"
                Height="50"
                Width="50"/>
        <Popup IsOpen="True">
            <TextBlock Background="LightBlue"
                       Padding="3"
                       Text="Popup Text"/>
        </Popup>
    </Canvas>
    <Canvas Background="Red" Grid.Column="1"
            Margin="15,30,30,30">
        <Border x:Name="border"
                Background="Yellow"
                Canvas.Left="20"
                Canvas.Top="20"
```

```
                Height="50"
                Width="50"/>
        <Popup IsOpen="True"
               PlacementTarget="{Binding
               ElementName=border}">
            <TextBlock Background="LightBlue"
                       Padding="3"
                       Text="Popup Text"/>
        </Popup>
    </Canvas>
</Grid>
```

Figure 31 shows result of the above markup.

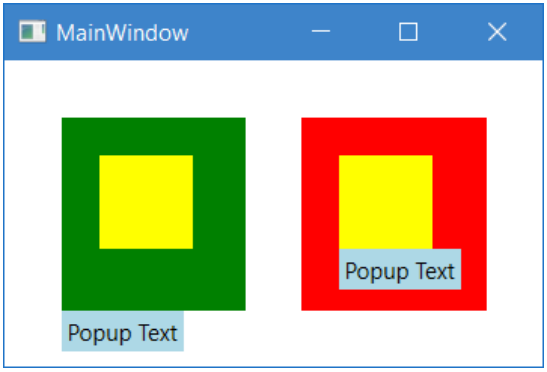

Figure 31. The Popup control with target object being indicated and being not indicated

The System.Windows.Controls.Primitives.Popup.**Placement** property was set for both popup windows, that's why its default value was taken, which indicates that the popup window must be placed under the target object. The System.Windows.Controls.Primitives.Popup.**PlacementTarget** property was not set for the first popup window, this is why the <Canvas> parent element was selected as a target object. The <Border>

element was set for the second popup window as a target object.

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Popup.TargetObject.CSharp):

C#

```
var border1 = new Border { Background =
              Brushes.Yellow, Height = 50.0,
              Width = 50.0 };

var textBlock1 = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Popup Text"
};

var popup1 = new Popup { Child = textBlock1 };
var canvas1 = new Canvas
{
    Background = Brushes.Green,
    Margin = new Thickness(30.0, 30.0, 15.0, 30.0)
};

canvas1.Children.Add(border1);
canvas1.Children.Add(popup1);
Canvas.SetLeft(border1, 20.0);
Canvas.SetTop(border1, 20.0);
popup1.IsOpen = true;

var border2 = new Border { Background =
    Brushes.Yellow, Height = 50.0, Width = 50.0 };

var textBlock2 = new TextBlock
{
```

```
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Popup Text"
};

var popup2 = new Popup { Child = textBlock2,
PlacementTarget = border2 };

var canvas2 = new Canvas
{
    Background = Brushes.Red,
    Margin = new Thickness(15.0, 30.0, 30.0, 30.0)
};

canvas2.Children.Add(border2);
canvas2.Children.Add(popup2);
Canvas.SetLeft(border2, 20.0);
Canvas.SetTop(border2, 20.0);

popup2.IsOpen = true;

var grid = new Grid();

grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.Children.Add(canvas1);
grid.Children.Add(canvas2);
Grid.SetColumn(canvas1, 0);
Grid.SetColumn(canvas2, 1);
```

9.1.2. Target Area

Target area is an area on the screen relative to which popup windows are positioned. In the above example, target area was an area that constrained the target object. However, there may be situations when the target object is set but at

that target area differs from the area occupied by it. In order to indicate target area, use the System.Windows.Controls.Primitives.Popup.**PlacementRectangle** property.

The below markup snippet shows creation of a popup window with explicit indication of a target area and without it (full example is in the folder Wpf.Controls.Popup.TargetArea.Xaml):

XAML

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Canvas Background="Green"
            Grid.Column="0"
            Margin="30,30,15,30">
        <Border Background="Yellow"
                Canvas.Left="20"
                Canvas.Top="20"
                Height="50"
                Width="50"/>
        <Popup IsOpen="True">
            <TextBlock Background="LightBlue"
                       Padding="3"
                       Text="Popup Text"/>
        </Popup>
    </Canvas>
    <Canvas Background="Red"
            Grid.Column="1"
            Margin="15,30,30,30">
        <Border Background="Yellow"
                Canvas.Left="20"
                Canvas.Top="20"
                Height="50"
                Width="50"/>
```

```
        <Popup IsOpen="True"
               PlacementRectangle="20,20,60,60">
            <TextBlock Background="LightBlue"
                       Padding="3"
                       Text="Popup Text"/>
        </Popup>
    </Canvas>
</Grid>
```

Figure 32 shows result of the above markup.

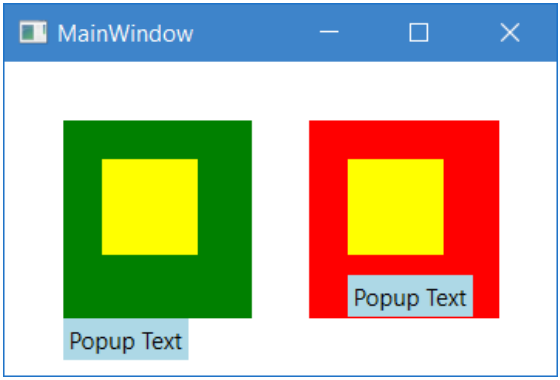

Figure 32. The Popup control with target area being specified and being not specified

Two <Canvas> elements are created in this element, each of them has the(<Border> element and the(<Popup> window. In both cases target object for the popup window is a panel, in which it is placed. In the first case, the System.Windows.Controls.Primitives.Popup.**PlacementRectangle** property is not set, so target area is the whole space occupied by the target object, i.e., panel. In the second case, target area is set explicitly. It is worth noting that coordinates that describe target area are set with respect to the target object, to be more specific,

to its upper left corner. It is also worth noting that setting of the target area is not visualized in any way.

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Popup.TargetArea.CSharp):

C#

```
var border1 = new Border { Background = Brushes.
Yellow, Height = 50.0, Width = 50.0 };

var textBlock1 = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Popup Text"
};

var popup1 = new Popup { Child = textBlock1 };
var canvas1 = new Canvas
{
    Background = Brushes.Green,
    Margin = new Thickness(30.0, 30.0, 15.0, 30.0)
};

canvas1.Children.Add(border1);
canvas1.Children.Add(popup1);
Canvas.SetLeft(border1, 20.0);
Canvas.SetTop(border1, 20.0);

popup1.IsOpen = true;
var border2 = new Border { Background = Brushes.
Yellow, Height = 50.0, Width = 50.0 };

var textBlock2 = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
```

```
    Text = "Popup Text"
};

var popup2 = new Popup
{
    Child = textBlock2,
    PlacementRectangle = new Rect(20.0, 20.0, 60.0, 60.0)
};

var canvas2 = new Canvas
{
    Background = Brushes.Red,
    Margin = new Thickness(15.0, 30.0, 30.0, 30.0)
};

canvas2.Children.Add(border2);
canvas2.Children.Add(popup2);
Canvas.SetLeft(border2, 20.0);
Canvas.SetTop(border2, 20.0);

popup2.IsOpen = true;
var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.Children.Add(canvas1);
grid.Children.Add(canvas2);
Grid.SetColumn(canvas1, 0);
Grid.SetColumn(canvas2, 1);
```

9.1.3. Target Origin Point and Popup Alignment Point

Target origin point and popup alignment point are points on the target area and popup window respectively, which are used to position popup windows with respect to target area. In order to indicate offset of one point relative to another one, use the System.Windows.Controls.Primitives.Popup.

HorizontalOffset and System.Windows.Controls.Primitives.Popup.**VerticalOffset** properties, which set horizontal and vertical offset respectively. These values describe offset of the popup alignment point with respect to the target origin point. Accurate position of these points is set by the System.Windows.Controls.Primitives.Popup.**Placement** property.

The below markup snippet shows creation of a popup window and its alignment point offset (full example is in the folder Wpf.Controls.Popup.Offsets.Xaml):

XAML

```
<Grid>
    <Canvas Background="Green" Margin="10,10,10,100">
        <Popup IsOpen="True" HorizontalOffset="40"
               Placement="Bottom" VerticalOffset="50">
            <TextBlock Background="LightBlue"
                       Padding="3" Text="Popup Text"/>
        </Popup>
    </Canvas>
</Grid>
```

Figure 33 shows result of the above markup.

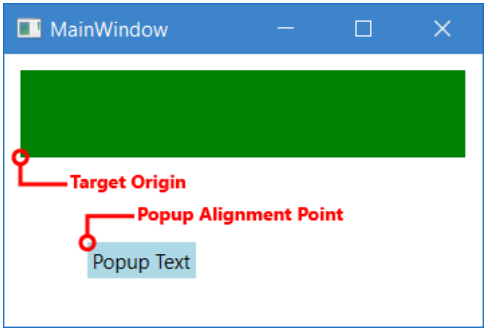

Figure 33. The Popup element with the offset of popup alignment point being specified

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Popup.Offsets.CSharp):

C#

```
var textBlock = new TextBlock
{
    Background = Brushes.LightBlue,
    Padding = new Thickness(3.0),
    Text = "Popup Text"
};

var popup = new Popup
{
    Child = textBlock,
    HorizontalOffset = 40.0,
    Placement = PlacementMode.Bottom,
    VerticalOffset = 50.0
};

var canvas = new Canvas
{
    Background = Brushes.Green,
    Margin = new Thickness(10.0, 10.0, 10.0, 100.0)
};

canvas.Children.Add(popup);
popup.IsOpen = true;
var grid = new Grid();
grid.Children.Add(canvas);
```

9.2. Interaction Between Placement Properties

In order to define correct target area, target origin point, and offset of the popup alignment point, the following properties must be considered: System.Windows.Controls.Primitives.Popup.**Placement**, System.Windows.Controls.

Primitives.Popup.**PlacementRectangle** and System.Windows.Controls.Primitives.Popup.**PlacementTarget**.

Below, there is description of the above concepts for all values of the System.Windows.Controls.Primitives.PlacementMode enumeration:

PlacementMode	Target Object	Target Area	Target Origin Point	Popup Alignment Point
Absolute	Not applied. The PlacementTarget property is ignored.	Screen area or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the screen.	Upper left corner of the target area.	Upper left corner of a popup window.
AbsolutePoint	Not applied. The PlacementTarget property is ignored.	Screen area or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the screen.	Upper left corner of the target area.	Upper left corner of a popup window.
Bottom	Parent element or the PlacementTarget property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Bottom left corner of the target area.	Upper left corner of the popup window.
Center	Parent element or the PlacementTarget property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Center of the target area.	Center of the popup window.
Custom	Parent element or the PlacementTarget property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Is defined using CustomPopupPlacementCallback.	Is defined using CustomPopupPlacementCallback.

Place-mentMode	Target Object	Target Area	Target Origin Point	Popup Alignment Point
Left	Parent element or the Placement-Target property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Upper left corner of the target area.	Upper right corner of the pop-up window.
Mouse	Not applied. The Placement-Target property is ignored.	Mouse pointer area. PlacementRectangle is ignored.	Bottom left corner of the target area.	Upper left corner of the pop-up window.
Mouse-Point	Not applied. The PlacementTar-get property is ignored.	Mouse pointer area. PlacementRectangle is ignored.	Upper left corner of the target area.	Upper left corner of the pop-up window.
Relative	Parent element or the Placement-Target property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Upper left corner of the target area.	Upper left corner of the pop-up window.
Relative-Point	Parent element or the Placement-Target property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Upper left corner of the target area.	Upper left corner of the pop-up window.
Right	Parent element or the Placement-Target property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Upper right corner of the target area.	Upper left corner of the pop-up window.
Top	Parent element or the Placement-Target property value, if set.	Target object or the PlacementRectangle property value, if set. PlacementRectangle describes area in relation to the upper left corner of the target object.	Upper right corner of the target area.	Upper left corner of the pop-up window.

9.3. 2.18.5 Overlapping of a Popup Window by Screen Borders

Part of a popup window cannot be overlapped by the screen border. In case such situation occurs, one of the following happens:

- Popup window is aligned with the screen border that overlaps it.
- Popup window uses another alignment point.
- Popup window uses another alignment point and another origin target.

Accurate behavior of a popup window when collapsing one of the screen borders depends on the value set for the System.Windows.Controls.Primitives.Popup.**Placement** property.

Below, there is a description of possible types of collapses with screen borders for all values of the System.Windows.Controls.Primitives.PlacementMode enumeration:

Placement-Mode	Upper Border	Bottom Border	Left Border	Right Border
Absolute	Aligns to the upper screen border.	Aligns to the bottom screen border.	Aligns to the left screen border.	Aligns to the right screen border.
Absolute-Point	Aligns to the upper screen border.	Popup alignment point shifts to the bottom left corner of the popup window.	Aligns to the left screen border.	Popup alignment point shifts to the upper right corner of the popup window.
Bottom	Aligns to the upper screen border.	Origin target shifts to the upper left corner of the target area, and the popup alignment window shifts to the bottom left corner of the popup window.	Aligns to the left screen border.	Aligns to the right screen border.

Placement-Mode	Upper Border	Bottom Border	Left Border	Right Border
Center	Aligns to the upper screen border.	Aligns to the bottom screen border.	Aligns to the left screen border.	Aligns to the right screen border.
Left	Aligns to the upper screen border.	Aligns to the bottom screen border.	Origin target shifts to the upper right corner of the target area, and the popup alignment window shifts to the upper left corner of the popup window.	Aligns to the right screen border.
Mouse	Aligns to the upper screen border.	Origin target shifts to the upper left corner of the target area, and the popup alignment window shifts to the bottom left corner of the popup window.	Aligns to the left screen border.	Aligns to the right screen border.
Mouse-Point	Aligns to the upper screen border.	The popup alignment window shifts to the bottom left corner of the popup window.	Aligns to the left screen border.	The popup alignment window shifts to the upper right corner of the popup window.
Relative	Aligns to the upper screen border.	Aligns to the bottom screen border.	Aligns to the left screen border.	Aligns to the right screen border.
Relative-Point	Aligns to the upper screen border.	The popup alignment window shifts to the bottom left corner of the popup window.	Aligns to the left screen border.	The popup alignment window shifts to the upper right corner of the popup window.
Right	Aligns to the upper screen border.	Aligns to the bottom screen border.	Aligns to the left screen border.	Origin target shifts to the upper left corner of the target area, and the popup alignment window shifts to the upper right corner of the popup window.

Placement-Mode	Upper Border	Bottom Border	Left Border	Right Border
Top	Origin target shifts to the bottom left corner of the target area, and the popup alignment window shifts to the upper left corner of the popup window.	Aligns to the bottom screen border.	Aligns to the left screen border.	Aligns to the right screen border.

9.4. Popup Positioning Configuration

WPF allows configuring the origin target point and popup alignment point with respect to its own algorithm. For this, set the System.Windows.Controls.Primitives.PlacementMode.**Custom** value to the System.Windows.Controls.Primitives.Popup.**Placement** property and set a delegate that returns a set of valid positioning points by setting a value to the System.Windows.Controls.Primitives.Popup.**CustomPopupPlacementCallback** property.

10. Menu

WPF menu is a variation of items controls (Fig. 34). If you look at the menu from the programmer point of view, you can realize that menu is almost a complete copy of the control that displays a list of elements in hierarchy (System.Windows.Controls.TreeView).

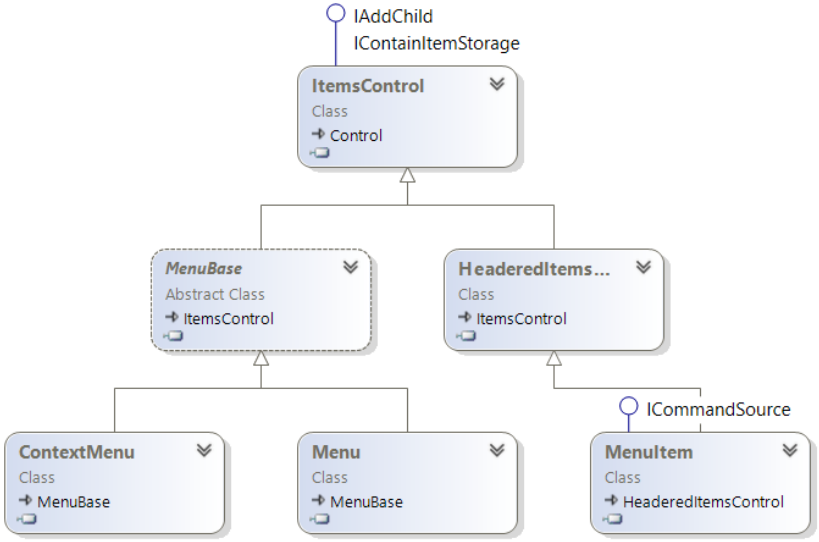

Figure 34. Class diagram that describes menu

10.1. Menu Control

The System.Windows.Controls.Menu control displays menu with hierarchical structure.

Properties of the System.Windows.Controls.Menu class:

- **IsMainMenu** (System.Boolean). Gets or sets a value, which indicates whether the current menu should get messages

addressed to the main menu of the window, such as pressing Alt and F10 keys. True if the current menu should get these messages, otherwise false. Its default value is false.

The System.Windows.Controls.MenuItem class is applied to the System.Windows.Controls.Menu control as an element container.

Properties of the System.Windows.Controls.MenuItem class:

- **Icon** (System.Object). Gets or sets an icon of a menu item. Its default value is null.
- **InputGestureText** (System.String). Gets or sets a string describing gesture of the input, which activates the menu item, for example, *Ctrl+C*. Its default value is System.String.**Empty**.
- **IsCheckable** (System.Boolean). Gets or sets a value, which indicates whether a menu item can be enabled/disabled. True if it can, otherwise false. Its default value is false.
- **IsChecked** (System.Boolean). Get or sets a value, which indicates whether the menu item is enabled. True if it is, otherwise false. Its default value is false.
- **IsHighlighted** (System.Boolean). Gets or sets a value, which indicates whether a menu item is highlighted. True if it is, otherwise false. Its default value is false.
- **IsPressed** (System.Boolean). Gets or sets a value, which indicates whether the menu item is pressed. True if it is, otherwise false. Its default value is false.
- **IsSubmenuOpen** (System.Boolean). Gets or sets a value, which indicates whether the submenu of the menu item is opened. True if it is, otherwise false. Its default value is false.

- **Role** (System.Windows.Controls.MenuItemRole). Gets or sets a menu item role. Its default value is System.Windows.Controls.MenuItemRole.**TopLevelItem**.
- **StaysOpenOnClick** (System.Boolean). Gets or sets a value, which indicates whether the submenu, where the current menu item is, must be closed after pressing the current menu item. True if it must not be closed upon pressing the current menu item, otherwise false. Its default value is false.
- Events of the System.Windows.Controls.MenuItem class:
- **Checked** (Systen.Windows.RoutedEventHandler). Occurs when the menu item is enabled.
- **Click** (System.Windows.RoutedEventHandler). Occurred when the menu item is clicked.
- **SubmenuClosed** (System.Windows.RoutedEventHandler). Occurs when the submenu closes.
- **SubmenuOpened** (System.Windows.RoutedEventHandler). Occurs when the submenu opens.
- **Unchecked** (System.Windows.RoutedEventHandler). Occurs when the menu item is disabled.

In order to specify menu item role, use the System.Windows.Controls.MenuItemRole enumeration, which contains the following options:

- **SubmenuHeader**. A submenu header.
- **SubmenuItem**. A submenu menu item (can execute commands).
- **TopLevelHeader**. Header of the top level menu.
- **TopLevelItem**. Item of the top level menu (can execute commands).

When creating a menu, you can easily fall in a situation, when some submenu contains too many menu items. In order to interact with such a menu in a more convenient way, you can divide menu items into logically connected groups and separate them from each other using a special separator described by the System.Windows.Controls.Separator class.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.Menu.Xaml):

XAML

```
<Menu IsMainMenu="True">
    <MenuItem Header="File">
        <MenuItem Header="New">
            <MenuItem Header="Project..."
                      InputGestureText="Ctrl+Shift+N"/>
            <MenuItem Header="File..."
                      InputGestureText="Ctrl+N"/>
        </MenuItem>
        <MenuItem Header="Start Page"/>
        <Separator/>
        <MenuItem Header="Add">
            <MenuItem Header="New Project..."/>
            <MenuItem Header="Existing Project..."/>
        </MenuItem>
        <MenuItem Header="Exit"
                  InputGestureText="Alt+F4"/>
    </MenuItem>
    <MenuItem Header="Window">
        <MenuItem Header="1 MainWindow.xaml"
                  IsCheckable="True"
                  IsChecked="True"/>
        <MenuItem Header="2 MainWindow.xaml.cs"
                  IsCheckable="True"/>
        <MenuItem Header="3 App.xaml"
                  IsCheckable="True"/>
```

```
        <MenuItem Header="4 App.xaml.cs"
                  IsCheckable="True"/>
    </MenuItem>
</Menu>
```

Figure 35 shows result of t he above markup.

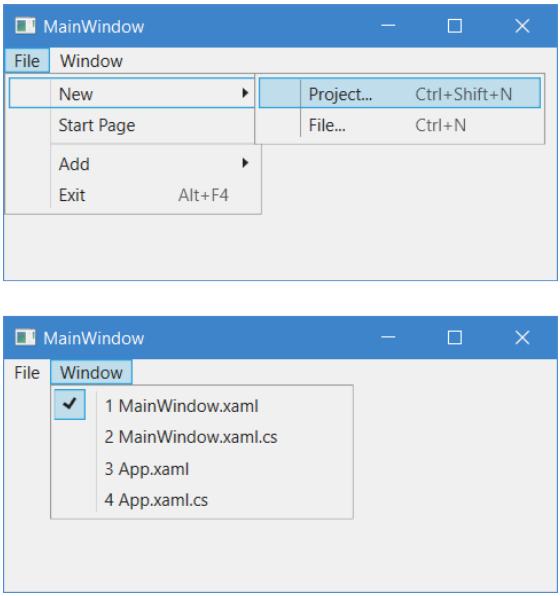

Figure 35. The Menu control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.Menu.CSharp):

C#

```
var separator = new Separator();
var menuItem31 = new MenuItem { Header = "Project...",
                 InputGestureText = "Ctrl+Shift+N" };
var menuItem32 = new MenuItem { Header = "File...",
                 InputGestureText = "Ctrl+N" };
```

```
var menuItem33 = new MenuItem { Header =
                 "New Project..." };
var menuItem34 = new MenuItem { Header =
                 "Existing Project..." };

var menuItem21 = new MenuItem { Header = "New" };
menuItem21.Items.Add(menuItem31);
menuItem21.Items.Add(menuItem32);
var menuItem22 = new MenuItem { Header = "Start Page" };
var menuItem23 = new MenuItem { Header = "Add" };
menuItem23.Items.Add(menuItem33);
menuItem23.Items.Add(menuItem34);
var menuItem24 = new MenuItem { Header = "Exit",
                 InputGestureText = "Alt+F4" };
var menuItem25 = new MenuItem
{
    Header = "1 MainWindow.xaml",
    IsCheckable = true,
    IsChecked = true
};
var menuItem26 = new MenuItem { Header =
                 "2 MainWindow.xaml.cs",
                 IsCheckable = true };
var menuItem27 = new MenuItem { Header = "3 App.xaml",
                 IsCheckable = true };
var menuItem28 = new MenuItem { Header = "4 App.xaml.cs",
                 IsCheckable = true };

var menuItem11 = new MenuItem { Header = "File" };
menuItem11.Items.Add(menuItem21);
menuItem11.Items.Add(menuItem22);
menuItem11.Items.Add(separator);
menuItem11.Items.Add(menuItem23);
menuItem11.Items.Add(menuItem24);
var menuItem12 = new MenuItem { Header = "Window" };
menuItem12.Items.Add(menuItem25);
menuItem12.Items.Add(menuItem26);
```

```
menuItem12.Items.Add(menuItem27);
menuItem12.Items.Add(menuItem28);

var menu = new Menu { IsMainMenu = true };
menu.Items.Add(menuItem11);
menu.Items.Add(menuItem12);
```

10.2. ContextMenu Control

The System.Windows.Controls.ContextMenu control describes a popup menu, which allows controls to provide additional functionality depending on the context.

Properties of the System.Windows.Controls.ContextMenu class:

- **CustomPopupPlacementCallback** (System.Windows.Controls.Primitives.CustomPopupPlacementCallback). **Gets or sets callback method, which is used to specify positioning of a context menu if the** System.Windows.Controls.ContextMenu**.Placement property value equals** System.Windows.Controls.Primitives.PlacementMode**.Custom.**
- **HasDropShadow** (System.Boolean). Gets or sets a value, which indicates, whether the context menu must drop shadow. True if it must, otherwise false. Its default value is false.
- **HorizontalOffset** (System.Double). Gets or sets horizontal offset relative to target area. Its default value is 0.0.
- **IsOpen** (System.Boolean). Gets or sets a value, which indicates whether the context menu is displayed. True if it is, otherwise false. Its default value is false.

- **Placement** (System.Windows.Controls.Primitives.PlacementMode). Gets or sets placement mode of the context menu. Its default value is System.Windows.Controls.Primitives.PlacementMode.**MousePoint**.
- **PlacementRectangle** (System.Windows.Rect). Gets or sets an area, relative to which the context menu is positioned when displayed. Its default value is System.Windows.Rect.**Empty**.
- **PlacementTarget** (System.Windows.UIElement). Gets or sets an element, relative to which the context menu is positioned when displayed. Its default value is null.
- **StaysOpen** (System.Boolean). Gets or sets a value, which indicates whether the menu must be closed automatically. True if the menu must stay open until this property is set to false, otherwise false. Its default value is false.
- **VerticalOffset** (System.Double). Gets or sets vertical offset relative to the target area. Its default value is 0.0.

Events of the System.Windows.Controls.ContextMenu class:

- **Closed** (System.Windows.RoutedEventArgs). Occurs when context menu closes.
- **Opened** (System.Windows.RoutedEventArgs). Occurs when the context menu opens.

The System.Windows.Controls.MenuItem class is applied as a container for the System.Windows.Controls.ContextMenu control.

Context menu can be set to any element, which type is derived from System.Windows.Controls.FrameworkElement,

since the System.Windows.Controls.FrameworkElement. **ContextMenu** property is declared in it.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.ContextMenu.Xaml):

XAML

```
<ContextMenu>
    <MenuItem Header="Edit">
        <MenuItem Header="Cut"
                  InputGestureText="Ctrl+X"/>
        <MenuItem Header="Copy"
                  InputGestureText="Ctrl+C"/>
        <MenuItem Header="Paste"
                  InputGestureText="Ctrl+V"/>
    </MenuItem>
    <MenuItem Header="Select All"
              InputGestureText="Ctrl+A"/>
</ContextMenu>
```

Figure 36 shows result of the above markup.

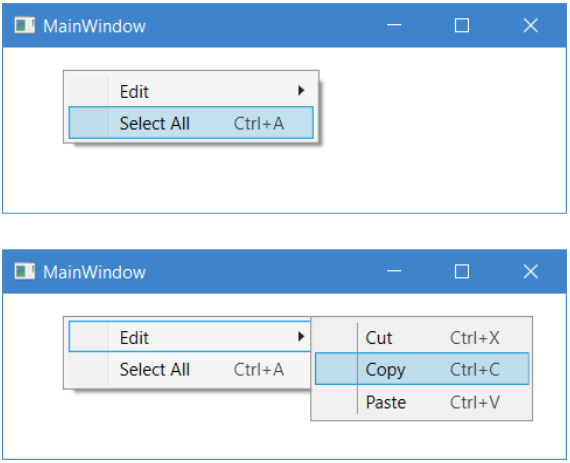

Figure 36. ContextMenu Control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.ContextMenu.CSharp):

C#

```
var menuItem21 = new MenuItem { Header = "Cut",
                 InputGestureText = "Ctrl+X" };
var menuItem22 = new MenuItem { Header = "Copy",
                 InputGestureText = "Ctrl+C" };
var menuItem23 = new MenuItem { Header = "Paste",
                 InputGestureText = "Ctrl+V" };

var menuItem11 = new MenuItem { Header = "Edit" };
menuItem11.Items.Add(menuItem21);
menuItem11.Items.Add(menuItem22);
menuItem11.Items.Add(menuItem23);
var menuItem12 = new MenuItem { Header = "Select All",
                 InputGestureText = "Ctrl+A" };

var contextMenu = new ContextMenu();
contextMenu.Items.Add(menuItem11);
contextMenu.Items.Add(menuItem12);
```

11. Tooltips

The System.Windows.Controls.ToolTip control displays tooltips to the user about this or that control or a part of the application interface.

In order to set a tooltip to the control, use its System.Windows.FrameworkElement.**ToolTip** property.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.ToolTip.Simple.Xaml):

XAML

```
<Button Height="23" ToolTip="Button Tooltip"
        Width="75">OK</Button>
```

Figure 37 shows result of the above markup.

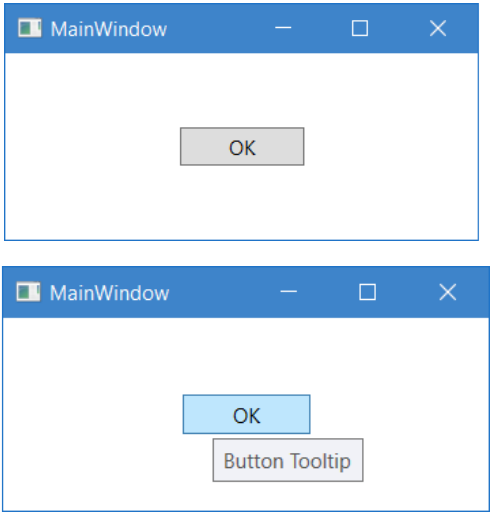

Figure 37. The Button control that has a simple tooltip

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.ToolTip.Simple.CSharp):

C#

```
var button = new Button
{
    Content = "OK",
    Height = 23.0,
    ToolTip = "Button Tooltip",
    Width = 75.0
};
```

The System.Windows.FrameworkElement.**ToolTip** property has the System.Object type, which allows it to write any object in it. Everything set as a value will be used as content for a popup window, which has a tooltip. This allows describing virtually any tooltip structure.

The below markup snippet shows a tooltip with a more complicated structure (full example is in the folder Wpf.Controls.ToolTip.Advanced.Xaml):

XAML

```
<Button Content="OK" Height="23" Width="75">
    <Button.ToolTip>
        <StackPanel Orientation="Horizontal">
            <TextBlock Foreground="Red" Text="Red "/>
            <TextBlock Foreground="Green"
                       Text="Green "/>
            <TextBlock Foreground="Blue"
                       Text="Blue"/>
        </StackPanel>
    </Button.ToolTip>
</Button>
```

Figure 38 shows result of the above markup.

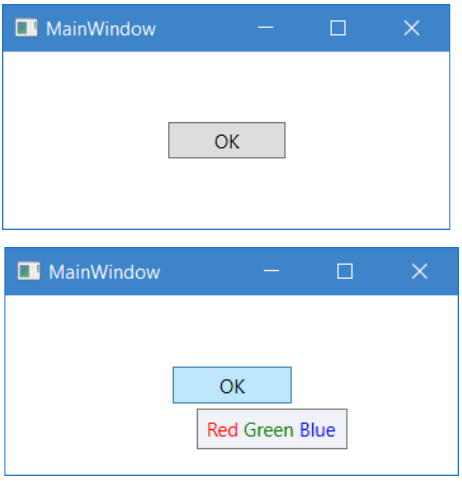

Figure 38. The Button control that has a more complicated tooltip

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.ToolTip.Advanced.CSharp):

C#

```
var textBlock1 = new TextBlock { Foreground =
                 Brushes.Red, Text = "Red " };
var textBlock2 = new TextBlock { Foreground =
                 Brushes.Green, Text = "Green " };
var textBlock3 = new TextBlock { Foreground =
                 Brushes.Blue, Text = "Blue" };

var stackPanel = new StackPanel { Orientation =
                 Orientation.Horizontal };
stackPanel.Children.Add(textBlock1);
stackPanel.Children.Add(textBlock2);
stackPanel.Children.Add(textBlock3);
```

```
var button = new Button
{
    Content = "OK",
    Height = 23.0,
    ToolTip = stackPanel,
    Width = 75.0
};
```

In order to configure various tooltip parameters, use attached properties or events of the System.Windows.Controls.ToolTipService class on an element, which requires configuration.

Attached properties of the System.Windows.Controls.ToolTipService class:

- **BetweenShowDelay** (System.Int32). Gets or sets maximum amount of time in milliseconds between showing of two tooltips in case when the second tooltip is shown without initial delay. Its default value is 100.0.
- **HasDropShadow** (System.Boolean). Gets or sets a value, which indicates whether the tooltip must drop shadow. True if it must, otherwise false. Its default value is false.
- **HorizontalOffset** (System.Double). Gets or sets horizontal offset relative to target area. Its default value is 0.0.
- **InitialShowDelay** (System.Int32). Gets or sets amount of time in milliseconds, which must pass before the tooltips is shown. Its default value is System.Windows.SystemParameters.**MouseHoverTime**
- **IsEnabled** (System.Boolean). Gets or sets a value, which indicates whether the tooltip must be shown. True if it must, otherwise false. Its default value is false.

- **IsOpen** (System.Boolean). Gets or sets a value, which indicates whether the tooltip is displayed. True if it is, otherwise false. Its default value is false.
- **Placement** (System.Windows.Controls.Primitives.PlacementMode). Gets or sets tooltip placement mode. Its default value is System.Windows.Controls.Primitives.PlacementMode.**Mouse**.
- **PlacementRectangle** (System.Windows.Rect). Gets or sets an area, relative to which tooltip is placed at display. Its default value is System.Windows.Rect.**Empty**.
- **PlacementTarget** (System.Windows.UIElement). Gets or sets an element, relative to which tooltip is positioned at display. Its default value is null.
- **ShowDuration** (System.Int32). Gets or sets amount of time in milliseconds, during which the tooltip is invisible. Its default value is 5000.0.
- **ShowOnDisabled** (System.Boolean). Gets or sets a value, which indicates whether the tooltip must be shown if the element, to which it belongs, is disabled. True if the tooltip must be shown even though the element is disabled, otherwise false. Its default value is false.
- **VerticalOffset** (System.Double). Gets or sets vertical offset relative to the target area. Its default value is 0.0.

Attached events of the System.Windows.Controls.ToolTipService class:

- **ToolTipClosing** (System.Windows.Controls.ToolTipEventArgs). Occurs when the tooltip closes.
- **ToolTipOpening** (System.Windows.Controls.ToolTipEventArgs). Occurs when the tooltip opens.

Usually, when the user moves the mouse pointer to the area above the control, which has the tooltip, a certain amount of time must pass before the tooltip is shown. Value of such initial delay is described by the System.Windows.Controls.ToolTipService.**InitialShowDelay** property. However, after the tooltip is shown, there is a certain time span, when the next tooltip can be shown without initial delay. This time span is indicated by the System.Windows.Controls.ToolTipService.**BetweenShowDelay** property. When the user moves the mouse pointer within the range of this time span from one element, which has an opened tooltip, to another one, which can also display tooltips, the second tooltip is displayed at once, without the initial delay.

12. Draggable Controls

WPF has controls, the main task of which is to provide "drag and drop" functionality to the user. The base class for them is the System.Windows.Controls.Primitives.Thumb class (Fig. 39).

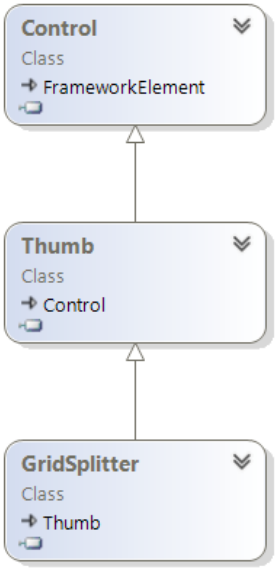

Figure 39. Class diagram that describes draggable controls

Properties of the System.Windows.Controls.Primitives.Thumb class:

- **IsDragging** (System.Boolean). Gets a value, which indicates whether the element has a logical focus, mouse capture, and whether the left mouse button is pressed. True if the element has logical focus, mouse capture, and

if the left mouse button is pressed, otherwise false. Its default value is false.

Events of the System.Windows.Controls.Primitives.Thumb class:

- **DragCompleted** (System.Windows.Controls.Primitives.DragCompletedEventArgs). Occurs when the element loses mouse capture.
- **DragDelta** (System.Windows.Controls.Primitives.DragDeltaEventArgs). Occurs when the mouse pointer is shifted which the element has a logical focus and mouse capture.
- **DragStarted** (System.Windows.Controls.Primitives.DragStartedEventArgs). Occurs when the element gets logical focus and mouse capture.

Methods of the System.Windows.Controls.Primitives.Thumb class:

- **CancelDrag**(). Cancels dragging.

12.1. GridSplitter Control

The System.Windows.Controls.GridSplitter control redistributes space between rows or columns of the System.Windows.Controls.Grid control.

Properties of the System.Windows.Controls.GridSplitter class:

- **DragIncrement** (System.Double). Gets or sets minimum distance the mouse pointer has to cover to change size of rows or columns. Its default value is 1.0.
- **KeyboardIncrement** (System.Double). Gets or sets offset size for each arrow button press. Its default value is 10.0.

- **ResizeBehavior** (System.Windows.Controls.GridResizeBehavior). Gets or sets which rows or columns relative to the one, where the element is, will change their sizes. The default value is System.Windows.Controls.GridResizeBehavior.**BasedOnAlignment**.
- **ResizeDirection** (System.Windows.Controls.GridResizeDirection). Gets or sets which size (rows or columns) will change using this element. Its default value is System.Windows.Controls.GridResizeDirection.**Auto**.
- **ShowsPreview** (System.Boolean). Gets or sets a value, which indicates whether the string or row sizes must be updated during the dragging. True if row or column sizes must be updated during dragging, otherwise false. Its default value is false.

In order to specify behavior when dragging an element, use the System.Windows.Controls.GridResizeBehavior enumeration, which has the following options:

- **BasedOnAlignment**. Space is distributed based on values of the System.Windows.FrameworkElement.**HorizontalAlignment** and System.Windows.FrameworkElement.**VerticalAlignment** properties.
- **CurrentAndNext**. If the element is oriented horizontally, then space is distributed between the row, where it was placed, and the next row (the one below). If the element is oriented vertically, the space is distributed between the column, where it is placed, and the next column (the one to the right).
- **PreviousAndCurrent**. If the element is oriented horizontally, then space is distributed between the row, where it is

placed, and the next row (the one above). If the element is oriented vertically, then space is distributed between the column, in which it is placed, and the previous column (the one to the left).

- **PreviousAndNext**. If the element is oriented horizontally, then space is distributed between the previous and next row. If the element is oriented vertically, then space is distributed between the previous and next column.

In order to specify the element dragging direction, use the System.Windows.Controls.GridResizeDirection enumeration, which has the following options:

- **Auto**. Space is distributed based on the System.Windows.FrameworkElement.**HorizontalAlignment**, System.Windows.FrameworkElement.**VerticalAlignment**, System.Windows.FrameworkElement.**ActualHeight** and System.Windows.FrameworkElement.**ActualWidth** properties of the element.
- **Columns**. Space is distributed between columns.
- **Rows**. Space is distributed between rows.

Usually, when using the element that distributes space between rows and columns, a separate row or column respectively is separated for it.

The below markup snippet shows this control (full example is in the folder Wpf.Controls.GridSplitter.Xaml):

XAML

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

```
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Border Background="Aqua"
            Grid.Column="0"
            Grid.Row="0"/>
    <Border Background="Brown"
            Grid.Column="2"
            Grid.Row="0"/>
    <Border Background="Bisque"
            Grid.Column="3"
            Grid.Row="0"/>
    <Border Background="DarkViolet"
            Grid.Column="0"
            Grid.Row="2"/>
    <Border Background="Khaki"
            Grid.Column="2"
            Grid.Row="2"/>
    <Border Background="Orange"
            Grid.Column="3"
            Grid.Row="2"/>
    <GridSplitter Grid.Column="1"
                  Grid.Row="0"
                  Grid.RowSpan="3"
                  HorizontalAlignment="Stretch"
                  Width="3"/>
    <GridSplitter Height="3"
                  Grid.Column="0"
                  Grid.ColumnSpan="4"
                  Grid.Row="1"
                  HorizontalAlignment="Stretch"/>
</Grid>
```

Figure 40 shows result of the above markup.

Figure 40. The GridSplitter control

The following code corresponds to the above markup (full example is in the folder Wpf.Controls.GridSplitter.CSharp):

C#

```
var border1 = new Border { Background = Brushes.Aqua };
var border2 = new Border { Background = Brushes.Brown };
var border3 = new Border { Background = Brushes.Bisque };
var border4 = new Border { Background = Brushes.
              DarkViolet };
var border5 = new Border { Background = Brushes.Khaki };
var border6 = new Border { Background = Brushes.
              Orange };

var gridSplitter1 = new GridSplitter
{
    HorizontalAlignment = HorizontalAlignment.
                          Stretch, Width = 3.0
};

var gridSplitter2 = new GridSplitter
{
    Height = 3.0,
    HorizontalAlignment = HorizontalAlignment.Stretch
};
```

```
var grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition {
     Width = GridLength.Auto });
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition {
     Height = GridLength.Auto });
grid.RowDefinitions.Add(new RowDefinition());

grid.Children.Add(border1);
grid.Children.Add(border2);
grid.Children.Add(border3);
grid.Children.Add(border4);
grid.Children.Add(border5);
grid.Children.Add(border6);
grid.Children.Add(gridSplitter1);
grid.Children.Add(gridSplitter2);

Grid.SetColumn(border1, 0);
Grid.SetRow(border1, 0);
Grid.SetColumn(border2, 2);
Grid.SetRow(border2, 0);
Grid.SetColumn(border3, 3);
Grid.SetRow(border3, 0);
Grid.SetColumn(border4, 0);
Grid.SetRow(border4, 2);
Grid.SetColumn(border5, 2);
Grid.SetRow(border5, 2);
Grid.SetColumn(border6, 3);
Grid.SetRow(border6, 2);
Grid.SetColumn(gridSplitter1, 1);
Grid.SetRow(gridSplitter1, 0);
Grid.SetRowSpan(gridSplitter1, 3);
Grid.SetColumn(gridSplitter2, 0);
Grid.SetColumnSpan(gridSplitter2, 4);
Grid.SetRow(gridSplitter2, 1);
```

13. Events Routing

As it was said before, WPF has two types of events routing: tunneling and bubbling. Routed events are usually represented in pairs: preview event, which is tunneled, and main events, which bubble. Preview events are called in the same way the main events, which is are their pair, are as a rule; but it has an additional prefix — **Preview**. For example, there is the main event System.Windows.UIElement.**MouseDown** and its preview version System.Windows.UIElement.**PreviewMouseDown**.

In order to better understand tunneling and bubbling, let's look at the following markup snippet (full example is in the folder Wpf.Events.Routing.Xaml):

XAML

```
<Window Height="300" Width="300">
    <Grid Background="Red">
        <Border Background="Green" Margin="20">
            <Rectangle Fill="Blue" Margin="20"/>
        </Border>
    </Grid>
</Window>
```

Figure 41 shows result of the above markup.

If one presses mouse button on the blue rectangle (the <Rectangle> element), then the following happens. First the preview event will be generated, which describes mouse button press: System.Windows.UIElement.**PreviewMouseDown**. This event will be tunneled, i.e., it will be directed from the root element (<Window>) to the one, to whom it is addressed (event

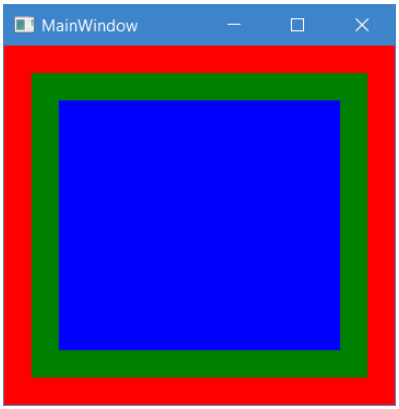

Figure 41. Tunneling and bubbling

source). In this case, preview event of mouse button press will pass all elements in the following order: <Window>, <Grid>, <Border>, <Rectangle>. After this, the mouse button press event will be generated: System.Windows.UIElement.**MouseDown**, which will bubble on the element visual tree, starting with the source element, in the following order: <Rectangle>, <Border>, <Grid>, <Window>.

13.1. Direct Routing

Actually, there is another, third, type of routing: direct routing. As a matter of fact, in this case no routing is executed since event is generated for the source element only and it doesn't move along the element tree. These "direct" events are required in the situations when event bubbling is pointless. For example, event of mouse pointer movement to the area of the new element — System.Windows.UIElement.**MouseEnter**. If one attaches this event handler at the level of a window, it will make sense to wait for this handler to work each time the mouse pointer moves to the window area from

the outside. This is what happens in reality. However, if such event was routed and bubbled each time when mouse pointer moves within the bounds of the window from one element to another one, both element and window would be informed. Even more, this event would occur for all elements in a chain of parent elements for the one, to which mouse pointer was moved. And this is not what is expected from such an event. This is why the described event uses direct routing.

13.2. Hit Testing in the Visual Layer

When the user manipulates a mouse, the application must indicate, to which element exactly the event corresponding to the performed actions must be addressed. Element under the mouse pointer is the target one, so-called event source. In order to do this, the visual tree of elements is traversed taking into account the mouse pointer coordinates. However, there is a set of rules that can change the result of this process, specifically, make an element "invisible" for such a test. Visual tree elements have the System.Windows.UIElement.**IsHitTestVisible** property, which is responsible for whether the element (and its child elements) is taken into account in the hit test algorithm.

In the below markup snippet, the <Border> element is marked as invisible for hit test (full example is in the folder Wpf.Properties.IsHitTestVisible.Xaml):

XAML

```
<Window Height="300" Width="300">
    <Grid Background="Red">
        <Border Background="Green"
                IsHitTestVisible="False"
                Margin="20">
```

```
            <Rectangle Fill="Blue" Margin="20"/>
        </Border>
    </Grid>
</Window>
```

If one clicks a mouse button at the blue rectangle (the <Rectangle> element), the following will happen. First the System.Windows.UIElement.**PreviewMouseDown** event will be tunneled as follows: <Window>, <Grid>. The <Border> element and its child element won't be considered. After this, the System.Windows.UIElement.**MouseDown** event will bubble in the following order: <Grid>, <Window>. In fact, the event was triggered as if there was no <Border> element in the visual tree.

You can get a similar result in another way, usually, unintentionally even if the System.Windows.UIElement.**IsHitTestVisible** property will indicate that the element must be considered at the hit test. If the background brush is not set for an element, the transparent area of the client area of the element won't be taken into account. But if the element has a border of non-zero thickness with the set brush, the border area will be taken into account. If it is necessary for the element to be transparent, but if it also must be taken into account at the hit test, then the transparent brush must be indicated. In other words, to make the element be taken into account, the property describing the background brush must contain value different from null. Transparent brush can be set only in one of the following ways:

XAML

```
<Border Background="Transparent" Height="100"
        Width="100"/>
```

C#

```
var border = new Border
{
    Background = Brushes.Transparent,
    Height = 100.0,
    Width = 100.0
};
```

Properties of the System.Windows.UIElement class, connected with the hit test:

- **IsHitTestVisible** (System.Boolean). Gets or sets a value which indicates whether the element can be returned as the hit test result of the mouse pointer over an element. True if it can, otherwise false. Its default value is true.

Events of the System.Windows.UIElement class connected with the hit test:

- **IsHitTestVisibleChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when value of the System.Windows.UIElement.**IsHitTestVisible** property is changed.

Methods of the System.Windows.UIElement class connected to hit test:

- **InputHitTest**(point). Returns an element which is at the specified coordinates relative to the current element.

13.3. Parameters of the Routed Events

When creating events in .NET Framework, it is common practice to use delegates of a specific type for them. Such delegates must describe methods, which return nothing and take two parameters: the first parameter describes an object,

to which event handler is attached and for which it is called, and the second one stores additional information specific for the event (for example, mouse pointer coordinates or a code of the pressed key). It is adopted to use the System.EventArgs class if the event doesn't require additional information and a class derived from it if no additional information is required. WPF has an intermediate class for routed events — System.Windows.RoutedEventArgs, which, in turn, is inherited from the System.EventArgs class. This is needed because even if routed event requires no additional information specific for it, it must store routing information.

Properties of the System.Windows.RoutedEventArgs class:

- **Handled** (System.Boolean). Gets or sets a value, which indicated the current state of the routed event. True if the event is handled, otherwise false. Its default value is false.
- **OriginalSource** (System.Object). Gets an object source of an event. In this case, visual tree elements are taken into account, i.e., if the event works on some component, for example, on a text box, then this property will contain exactly this part.
- **RoutedEvent** (System.Windows.RoutedEvent). Gets or sets an object associated with this event.
- **Source** (System.Object). Gets or sets an object source of an event. In this case, elements from the logical tree are taken into account, i.e., if the event works on some component, for example, on a text box, then this property will contain a text box.

Delegates that are applied for routed events are arranged according to the same principle, only the type of the second parameter is changed:

C#

```
delegate void RoutedEventHandler(object sender,
         RoutedEventArgs e);
```

14. Input Handling

This section considers API provided by WPF to get input from various devices. The most common of them are keyboard and mouse. The majority of functionality connected with input handling is in the System.Windows.UIElement base class.

14.1. Mouse

Input handling with mouse in majority of cases comes down to handling events required of the mouse, which occur when clicking and releasing buttons, moving the mouse pointer, and scrolling wheel.

Properties of the System.Windows.UIElement class related to the mouse:

- **IsMouseCaptured** (System.Boolean). Gets a value, which indicates whether the mouse is captured by the element. True if it is, otherwise false.
- **IsMouseCaptureWithin** (System.Boolean). Gets a value, which indicates, whether the mouse is captured by an element or one of the child elements. True if is, otherwise false.
- **IsMouseDirectlyOver** (System.Boolean). Gets a value, which indicates whether the mouse pointer is above the element (excluding child elements of the visual tree). True if the mouse pointer is above the element, otherwise false.
- **IsMouseOver** (System.Boolean). Gets a value, which indicates whether the mouse pointer is above the element (including child elements of the visual tree). True if it is, otherwise false.

Events of the System.Windows.UIElement class related to the mouse:

- **GotMouseCapture** (System.Windows.Input.MouseEventHandler). Occurs when element captures the mouse.
- **IsMouseCapturedChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when the System.Windows.UIElement.**IsMouseCaptured** property value is changed.
- **IsMouseCaptureChangedWithin** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when the System.Windows.UIElement.**IsMouseCaptureWithin** property value is changed.
- **IsMouseDirectlyOverChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when the System.Windows.UIElement.**IsMouseDirectlyOver** property value is changed.
- **LostMouseCapture** (System.Windows.Input.MouseEventHandler). Occurs when the element releases mouse capture.
- **MouseDown** (System.Windows.Input.MouseButtonEventHandler). Occurs when any of the mouse buttons is pressed as long as the mouse pointer is above the element.
- **MouseEnter** (System.Windows.Input.MouseEventHandler). Occurs when the mouse pointer moves to the element area from another element or from outside.
- **MouseLeave** (System.Windows.Input.MouseEventHandler). Occurs when the mouse pointer leaves the element area.
- **MouseLeftButtonDown** (System.Windows.Input.MouseButtonEventHandler). Occurs when left mouse button is pressed as long as mouse pointer is above the element.

- **MouseLeftButtonUp** (System.Windows.Input.MouseButtonEventHandler). Occurs when left mouse button is released as long as mouse pointer is above the element.
- **MouseMove** (System.Windows.Input.MouseEventHandler). Occurs when the mouse pointer moves above the element area.
- **MouseRightButtonDown** (System.Windows.Input.MouseButtonEventHandler). Occurs when the right mouse button is pressed as long as the mouse pointer is above the element.
- **MouseRightButtonUp** (System.Windows.Input.MouseButtonEventHandler). Occurs when the right mouse button is released as long as the mouse pointer is above the element.
- **MouseUp** (System.Windows.Input.MouseButtonEventHandler). Occurs when any of the mouse buttons is released as long as mouse pointer is above the element.
- **MouseWheel** (System.Windows.Input.MouseWheelEventHandler). Occurs when the mouse wheel is scrolled as long as the mouse pointer is above the element.
- **PreviewMouseDown** (System.Windows.Input.MouseButtonEventHandler). Preview event for the System.Windows.UIElement.**MouseDown** event.
- **PreviewMouseLeftButtonDown** (System.Windows.Input.MouseButtonEventHandler). Preview event for the System.Windows.UIElement.**MouseLeftButtonDown** event.
- **PreviewMouseLeftButtonUp** (System.Windows.Input.MouseButtonEventHandler). Preview event for the System.Windows.UIElement.**MouseLeftButtonUp** event.
- **PreviewMouseMove** (System.Windows.Input.MouseEventHandler). Preview event for the System.Windows.UIElement.**MouseMove** event.

- **PreviewMouseRightButtonDown** (System.Windows.Input.MouseButtonEventHandler). Preview event for the System.Windows.UIElement.**MouseRightButtonDown** event.
- **PreviewMouseRightButtonUp** (System.Windows.Input.MouseButtonEventHandler). Preview event for the System.Windows.UIElement.**MouseRightButtonUp** event.
- **PreviewMouseUp** (System.Windows.Input.MouseButtonEventHandler). Preview event for the System.Windows.UIElement.**MouseUp** event.
- **PreviewMouseWheel** (System.Windows.Input.MouseWheelEventHandler). Preview event for the System.Windows.UIElement.**MouseWheel** event.

Methods of the System.Windows.UIElement class related to the mouse:

- **CaptureMouse**(). Tries to capture the mouse. Returns true if the operation was successful, otherwise false.
- **ReleaseMouseCapture**(). Releases mouse capture if the current element captured it.

Events of the System.Windows.Controls.Control class related to the mouse:

- **MouseDoubleClick** (System.Windows.Input.MouseButtonEventHandler). Occurs at double click with the mouse as long as the mouse pointer is above the element.
- **PreviewMouseDoubleClick** (System.Windows.Input.MouseButtonEventHandler). Preview event for the System.Windows.Controls.Control.**MouseDoubleClick** event.

In addition to the mouse interaction API placed in the base classes of elements, there is the System.Windows.Input.Mouse

static class representing a full set of useful properties and methods.

Static properties of the System.Windows.Input.Mouse class:

- **Captured** (System.Windows.IInputElement). Gets an element, which captured mouse at the current moment.
- **DirectlyOver** (System.Windows.IInputElement). Gets an element, above which there is the mouse pointer.
- **LeftButton** (System.Windows.Input.MouseButtonState). Gets the left mouse button state.
- **MiddleButton** (System.Windows.Input.MouseButtonState). Gets the middle mouse button state.
- **RightButton** (System.Windows.Input.MouseButtonState). Gets the right mouse button state.
- **XButton1** (System.Windows.Input.MouseButtonState). Gets the first additional mouse button state.
- **XButton2** (Syste.Windows.Input.MouseButtonState). Gets the second additional mouse button state.
- Static methods of the System.Windows.Input.Mouse class:
- **GetPosition**(relativeTo). Returns position of the mouse pointer relative to the specified element.
- **SetCursor**(cursor). Sets the specified mouse pointer.

In order to find out the mouse button state, use the System.Windows.Input.MouseButtonState enumeration, which has the following options:

- **Pressed**. The button is pressed.
- **Released**. The button is released.

14.2. Keyboard

Events of the System.Windows.UIElement class related to the keyboard:

- **KeyDown** (System.Windows.Input.KeyEventHandler). Occurs when a keyboard key is pressed as long as the element has keyboard focus.
- **KeyUp** (System.Windows.Input.KeyEventHandler). Occurs when a keyboard key is released as long as the element has keyboard focus.
- **PreviewKeyDown** (System.Windows.Input.KeyEventHandler). Preview event for the System.Windows.UIElement.**KeyDown** event.
- **PreviewKeyUp** (System.Windows.Input.KeyEventHandler). Preview event for the System.Windows.UIElement.**KeyUp** event.

Just like the mouse, the keyboard has a static class that contains additional functionality as a separate API — System.Windows.Input.Keyboard.

Static properties of the System.Windows.Input.Keyboard class:

- **FocusedElement** (System.Windows.Input.IInputElement). Gets an element, which has a keyboard focus.
- **Modifiers** (System.Windows.Input.ModifierKeys). Gets a set of the currently pressed control keys.

Static methods of the System.Windows.Input.Keyboard class:

- **ClearFocus**(). Clears keyboard focus from the element which has it.
- **Focus**(element). Sets keyboard focus to the specified element and returns the element, which has a keyboard focus.
- **GetKeyStates**(key). Gets a set of states for the specified key.

- **IsKeyDown**(key). Checks whether the specified key is pressed. Returns true if the specified key is pressed, otherwise false.
- **IsKeyToggled**(key). Checks whether the specified key is enabled. Returns true if the specified key is enabled, otherwise false.
- **IsKeyUp**(key). Checks, whether the specified key is released. Returns true if it is released, otherwise false.

In order to find out which control keys are pressed, use the System.Windows.Input.ModifierKeys enumeration, which contains the following options:

- **Alt** key.
- **Control** key.
- **Shift** key.
- **Windows** key.

In order to find out key state on the keyboard, use the System.Windows.Input.KeyStates enumeration, which has the following options:

- **None**. Key is released.
- **Pressed**. Key is pressed.
- **Toggled**. Key is enabled.

In order to indicate a specific virtual key, use the System.Windows.Input.Key enumeration, which has options for all available keys.

14.2.1. Focus

WPF has two focus-related concepts: keyboard focus and logical focus. Keyboard focus is related to the element, which gets keyboard input, and logical is related to the element in

the focus scope, which has focus. Element, which has a keyboard focus, also has logical focus, but the element, which has logical focus, doesn't always has keyboard focus.

Properties of the System.Windows.UIElement class related to the focus:

- **Focusable** (System.Boolean). Gets or sets a value, which indicates whether the element can get focus. True if the element can get focus, otherwise false. Its default value depends on the specific derived class.
- **IsFocused** (System.Boolean). Gets a value, which indicates whether the element has logical focus. True if the element has logical focus, otherwise false.
- **IsKeyboardFocused** (System.Boolean). Gets a value, which indicates whether the element has a keyboard focus. True if it has, otherwise false.
- **IsKeyboardFocusWithin** (System.Boolean). Gets a value, which indicates whether the element or one of its child elements has a keyboard focus. True if the element or one of its child elements has it, otherwise false.

Events of the System.Windows.UIElement class related to the focus:

- **FocusableChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when value of the System.Windows.UIElement.**Focusable** property is changed.
- **GotFocus** (System.Windows.RoutedEventHandler). Occurs when the element gets a logical focus.
- **GotKeyboardFocus** (System.Windows.Input.KeyboardFocusChangedEventHandler). Occurs when an element gets a keyboard focus.

- **IsKeyboardFocusedChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when value of the System.Windows.UIElement.**IsKeyboardFocused** property is changed.
- **IsKeyboardFocusWithinChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when value of the System.Windows.UIElement.**IsKeyboardFocused** property is changed.
- **LostFocus** (System.Windows.RoutedEventHandler). Occurs when the element loses logical focus.
- **LostKeyboardFocus** (System.Windows.Input.KeyboardFocusChangedEventHandler). Occurs when the element loses the keyboard focus.
- **PreviewGotKeyboardFocus** (System.Windows.Input.KeyboardFocusChangedEventHandler). Preview event for the System.Windows.UIElement.**GotKeyboardFocus** event.
- **PreviewLostKeyboardFocus** (System.Windows.Input.KeyboardFocusChangedEventHandler). Preview event for the System.Windows.UIElement.**LostKeyboardFocus** event.

Methods of the System.Windows.UIElement class related to focus:

- **Focus**(). Tries to set focus to an element. Returns true if the specified element got both logical and keyboard focus, false if the specified element got only logical focus or an attempt to change focus was not successful.

Keyboard Focus

As it was said above, if the element has a keyboard focus, then it gets a keyboard input. Only one element within the whole

desktop can have a keyboard focus. In WPF, to find out whether the element has such a focus, there is the System.Windows.UIElement.**IsKeyboardFocused** property. Or the element having a keyboard focus can be obtained by using the System.Windows.Input.Keyboard.**FocusedElement** property.

In order to make element have an opportunity to get a keyboard focus, its properties System.Windows.UIElement.**Focusable** and System.Windows.UIElement.**IsVisible** must return true.

Keyboard focus can be got by an element through specific user manipulations. For example, by pressing Tab or clicking mouse button on some element. Keyboard focus can be programmatically moved to an element using the System.Windows.Input.Keyboard.**Focus** static method. This method tries to set focus to a specified element and returns an element, which has keyboard focus, which may differ from the specified one (for example, a new or old element blocked request for new focus to be got).

Controls of base class have two properties responsible for whether the element has a keyboard focus: System.Windows.UIElement.**IsKeyboardFocused** and System.Windows.UIElement.**IsKeyboardFocusWithin**. The first one indicates whether the element has a focus, while the second one indicates whether an element or one of its child elements has a focus.

In order to set the initial keyboard focus when starting an application, the element, which must have focus, must be in a visual tree of the initial application window and its properties System.Windows.UIElement.**Focusable** and System.Windows.UIElement.**IsVisible** must return true. Recommend-

ed loaded place of the initial focus is the System.Windows.FrameworkElement.**Loaded** event of the initial window:

C#

```
private void Window_Loaded(object sender,
        RoutedEventArgs e)
{
    Keyboard.Focus(submitButton);
}
```

Logical Focus

There can be several so-called focus scopes within an application interface. In other words, elements can be divided into several areas, within each of them there will be their own element that has focus. In this case it is a logical focus. When the keyboard focus is moved from one focus scope to another one, the element, which lost the keyboard focus, will preserve logical focus. When the keyboard focus moves to another focus scope, it is set to an element that has a logical focus. This allows us to implement switching between several groups of logically divided elements, remembering an element that has a focus in each group and restoring it at return.

There can be several elements within an application, which has a logical focus, but only one within the focus scope. Element that has a keyboard focus also has a logical focus within the focus scope it belongs to.

In order to turn an element into focus scope, use the System.Windows.Input.FocusManager.**IsFocusScope** attached property.

Example of creating focus scope from a panel, using markup:

XAML

```
<StackPanel FocusManager.IsFocusScope="True">
    <Button>Submit</Button>
    <Button>Cancel</Button>
</StackPanel>
```

The following code corresponds to the above markup:

C#

```
var submitButton = new Button { Content = "Submit" };
var cancelButton = new Button { Content = "Cancel" };
var stackPanel = new StackPanel();
stackPanel.Children.Add(submitButton);
stackPanel.Children.Add(cancelButton);

FocusManager.SetIsFocusScope(stackPanel, true);
```

In order to get a focus scope, use the System.Windows.Input.FocusManager.**GetFocusScope** static method by specifying an element, for which you need to get a scope.

In order to get or set a focus element within a focus scope, there are the corresponding static methods: System.Windows.Input.FocusManager.**GetFocusElement** and System.Windows.Input.FocusManager.**SetFocusElement**. The last one is usually used to set the initial logical focus.

14.2.2. Access Key

All elements that represent this or that type of buttons (or menu items) support interaction with access keys, which works like the System.Windows.Controls.Label mnemonics. The same mechanism of indicating the underscore symbol before the required symbol is used for this.

15. Review of Basic Classes of Visual Tree Elements

As you have already noticed, WPF has several base classes that contain lots of useful functionality, from which all panels and controls discussed up to this point are inherited. This section reviews functionality of some of these classes that wasn't reviewed before.

15.1. UIElement Class

Properties of the System.Windows.UIElement class:

- **IsEnabled** (System.Boolean). Gets or sets a value, which indicates whether the element is available for user interaction. True if it is, otherwise false. Its default value is true.
- **IsVisible** (System.Boolean). Gets a value, which indicated whether the element is displayed. True if it is visible, otherwise false.
- **Opacity** (System.Double). Gets or sets opacity factor of an element. This property can take values from 0.0 (transparent) to 1.0 (opaque). Its default value is 1.0.
- **SnapsToDevicePixels** (System.Boolean). Gets or sets a value, which indicates whether device-specific pixels must be used when displaying an element. True if the element must be displayed in accordance with device-specific pixels, otherwise false. Its default value is false. When displaying an element with pixel density greater than 96 pixels per inch, inclusion of this property allows avoiding unwanted anti-aliasing, which occurs if the calculated element size doesn't match device pixels.

- **Visibility** (System.Windows.Visibility). Gets or sets element visibility mode. Its default value is System.Windows.Visibility.**Visible**.

Events of the System.Windows.UIElement class:

- **IsEnabledChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when the System.Windows.UIElement.**IsEnabled** property value is changed.
- **IsVisibleChanged** (System.Windows.DependencyPropertyChangedEventHandler). Occurs when the System.Windows.UIElement.**IsVisible** property value is changed.

Methods of the System.Windows.UIElement class:

- **TranslatePoint**(point, relativeTo). Translates coordinates relative to the current element to the coordinates that are relative to another element.

In order to specify the element visibility mode, use the System.Windows.Visibility enumeration, which has the following options:

- **Collapsed**. Element is not displayed, and it doesn't reserve space in the interface.
- **Hidden**. Element is not displayed, but it reserves space in the interface.
- **Visible**. Element is displayed.

15.2. FrameworkElement Class

Properties of the System.Windows.FrameworkElement class:

- **ContextMenu** (System.Windows.Controls.ContextMenu). Gets or sets a context menu of an element.

- **IsInitialized** (System.Boolean). Gets a value, which indicates whether the object is initialized. True if it is, otherwise false.
- **IsLoaded** (System.Boolean). Gets a value, which indicates whether the element is loaded in a visual tree. True if the element is loaded in the visual tree, otherwise false.
- **Name** (System.String). Gets or sets an element name. Its default value is System.String.**Empty**.
- **Parent** (System.Windows.DependencyObject). Gets a parent element.
- **Tag** (System.Object). Gets or sets a random object associated with this element, which can be used to store additional information describing the element.
- **ToolTip** (System.Object). Gets or sets a tooltip of an element.
- **UseLayoutRounding** (System.Boolean). Gets or sets a value, which indicates whether the calculated size and coordinates of the element must be rounded at the layout stage. True if the rounding is required, otherwise false. Its default value is false.

Events of the System.Windows.FrameworkElement class:

- **ContextMenuClosing** (System.Windows.Controls.ContextMenuEventHandler). Occurs right before the context menu of an element closes.
- **ContextMenuOpening** (System.Windows.Controls.ContextMenuEventHandler). Occurs right before the context menu of an element opens.
- **Initialized** (System.EventHandler). Occurs when an element is initialized.

- **Loaded** (System.Windows.RoutedEventHandler). Occurs when an element is loaded to a visual tree, its size and position are calculated and it is ready to be displayed.
- **SizeChanged** (System.Windows.SizeChangedEventHandler). Occurs when the System.Windows.FrameworkElement.**ActialHeight** or System.Windows.FrameworkElement.**ActialWidth** property value is changed.
- **ToolTipClosing** (System.Windows.Controls.ToolTipEventHandler). Occurs right before the tooltip closes.
- **ToolTipOpening** (System.Windows.Controls.ToolTipEventHandler). Occurs right before the tooltip opens.
- **Unloaded** (System.Windows.RoutedEventHandler). Occurs when the element is unloaded from the visual tree.

Methods of the System.Windows.FrameworkElement class:

- **FindName**(name). Gets an element with the indicated name or null if the required element is not found.

15.3. Control Class

Properties of the System.Windows.Controls.Control class:

- **Background** (System.Windows.Media.Brush). Gets or sets a brush, which is used to fill the set element background. Its default value is System.Windows.Media.Brushes.**Transparent**.
- **BorderBrush** (System.Windows.Media.Brush). Gets or sets a brush, which is used to fill the element border. Its default value is System.Windows.Media.Brushes.**Transparent**.
- **BorderThickness** (System.Windiws.Thickness). Gets or sets border thickness. Its default value is the System.

Windows.Thickness object, which describes zero thickness on all sides.

- **FontFamily** (System.Windows.Media.FontFamily). Gets or sets a font family.
- **FontSize** (System.Double). Gets or sets a font size. This property can take only positive values. Its default value is System.Windows.SystemFonts.**MessageFontSize**.
- **FontStretch** (System.Windows.FontStretch). Gets or sets font stretch mode. Its default value is System.Windows.FontStretches.**Normal**.
- **FontStyle** (System.Windows.FontStyle). Gets or sets a font style. Its default value is System.Windows.FontStyles.**Normal**.
- **FontWeight** (System.Windows.FontWeight). Gets or sets a font weight. Its default value is System.Windows.FontWeights.**Normal**.
- **Foreground** (System.Windows.Media.Brush). Gets or sets a brush, which is used to fill the element foreground.
- **IsTabStop** (System.Boolean). Gets or sets a value, which indicates whether the element must be taken into account when navigating with the Tab key. True if it must, otherwise false. Its default value is true.
- **TabIndex** (System.Int32). Gets or sets the order of getting element focus when navigating with the Tab key. Its default value is System.Int32.**MaxValue**.

16. Home Task

16.1. Task 1

Develop the *Keyboard Trainer* application (Fig. 42). Main window must display a keyboard with non-interactive keys required to help the user to feel confident with the keyboard without looking at it. When clicking each of the keys, it must be highlighted on the screen. After the training session starts, an input line must be generated for the user, which takes into account the selected complexity level.

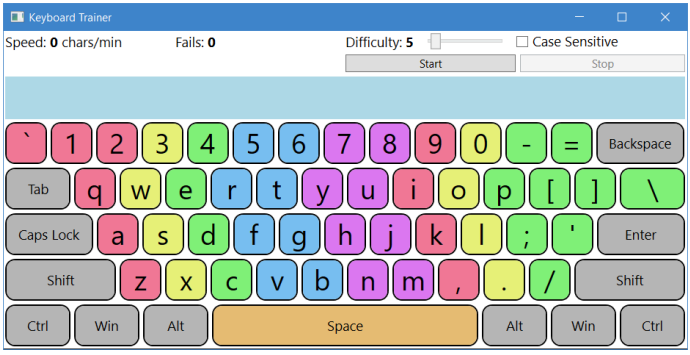

Figure 42. Task 1

The upper part of a window must display static information: type speed of correct text and the amount of mistakes. The upper part also must have controls, which allow configuring the complexity of the generated line. Using the slider, the user can choose number of characters, which must be used for the generated line. At that the user can indicate whether a case sensitive line must be generated. As used symbols, all symbols shown on the Figures 43 and 44 can be used.

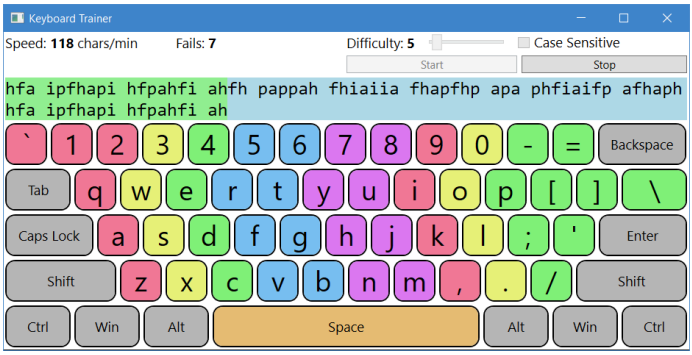

Figure 43. Task 1

After clicking the Start button, a line of characters must be generated including all configurations set by the user. After this the keys pressed by the user must be taken into account and displayed as correctly typed characters or as mistakes.

When clicking Shift and Caps Lock, the application keyboard must change the displayed characters, considering the actually typed ones (Fig. 44).

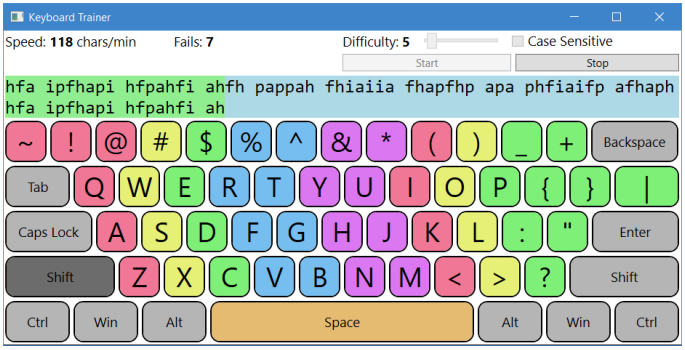

Figure 44. Task 1

You should also provide stopping of the controls that cannot be used in the current state of an application. For

example, you cannot click *Stop* if the user didn't press *Start* before this.

If you have some XAML problems when executing the task, you can use the sample markup, which is in the folder KeyboardTrainer.

Lesson 2

Controls