The Canvas control

The Canvas is probably the simplest Panel of them all. It doesn't really do anything by default, it just allows you to put controls in it and then position them yourself using explicit coordinates.

If you have ever used another UI library like WinForms, this will probably make you feel right at home, but while it can be tempting to have absolute control of all the child controls, this also means that the Panel won't do anything for you once the user starts resizing your window, if you localize absolutely positioned text or if the content is scaled.

<Window x:Class="WpfTutorialSamples.Panels.Canvas"

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

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

Title="Canvas" Height="200" Width="200">

<Canvas>

<Button>Button 1</Button>

<Button>Button 2</Button>

</Canvas>

</Window>

<Window x:Class="WpfTutorialSamples.Panels.Canvas"

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

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

Title="Canvas" Height="200" Width="200">

<Canvas>

<Button Canvas.Left="10">Top left</Button>

<Button Canvas.Right="10">Top right</Button>

<Button Canvas.Left="10" Canvas.Bottom="10">Bottom left</Button>

<Button Canvas.Right="10" Canvas.Bottom="10">Bottom right</Button>

</Canvas>

</Window>

## Z-Index

In the next example, we'll use a couple of the shape related controls of WPF to illustrate another very important concept when using the Canvas: Z-Index. Normally, if two controls within a Canvas overlaps, the one defined last in the markup will take precedence and overlap the other(s). However, by using the attached ZIndex property on the Panel class, this can easily be changed.

First, an example where we don't use z-index at all:

<Window x:Class="WpfTutorialSamples.Panels.CanvasZIndex"

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

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

Title="CanvasZIndex" Height="275" Width="260">

<Canvas>

<Ellipse Fill="Gainsboro" Canvas.Left="25" Canvas.Top="25" Width="200" Height="200" />

<Rectangle Fill="LightBlue" Canvas.Left="25" Canvas.Top="25" Width="50" Height="50" />

<Rectangle Fill="LightCoral" Canvas.Left="50" Canvas.Top="50" Width="50" Height="50" />

<Rectangle Fill="LightCyan" Canvas.Left="75" Canvas.Top="75" Width="50" Height="50" />

</Canvas>

</Window>

# The WrapPanel control

The **WrapPanel** will position each of its child controls next to the other, horizontally (default) or vertically, until there is no more room, where it will wrap to the next line and then continue. Use it when you want a vertical or horizontal list controls that automatically wraps when there's no more room.

When the WrapPanel uses the Horizontal orientation, the child controls will be given the same height, based on the tallest item. When the WrapPanel is the Vertical orientation, the child controls will be given the same width, based on the widest item.

<Window x:Class="WpfTutorialSamples.Panels.WrapPanel"

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

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

Title="WrapPanel" Height="300" Width="300">

<WrapPanel>

<Button>Test button 1</Button>

<Button>Test button 2</Button>

<Button>Test button 3</Button>

<Button Height="40">Test button 4</Button>

<Button>Test button 5</Button>

<Button>Test button 6</Button>

</WrapPanel>

</Window>

<Window x:Class="WpfTutorialSamples.Panels.WrapPanel"

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

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

Title="WrapPanel" Height="120" Width="300">

<WrapPanel Orientation="Vertical">

<Button>Test button 1</Button>

<Button>Test button 2</Button>

<Button>Test button 3</Button>

<Button Width="140">Test button 4</Button>

<Button>Test button 5</Button>

<Button>Test button 6</Button>

</WrapPanel>

</Window>

The StackPanel control

The **StackPanel** is very similar to the WrapPanel, but with at least one important difference: The StackPanel doesn't wrap the content. Instead it stretches it content in one direction, allowing you to stack item after item on top of each other. Let's first try a very simple example, much like we did with the WrapPanel:

<Window x:Class="WpfTutorialSamples.Panels.StackPanel"

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

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

Title="StackPanel" Height="160" Width="300">

<StackPanel>

<Button>Button 1</Button>

<Button>Button 2</Button>

<Button>Button 3</Button>

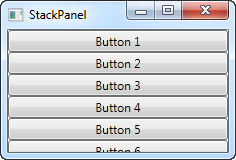
<Button>Button 4</Button>

<Button>Button 5</Button>

<Button>Button 6</Button>

</StackPanel>

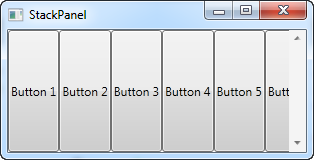
</Window>



The first thing you should notice is how the StackPanel doesn't really care whether or not there's enough room for the content. It doesn't wrap the content in any way and it doesn't automatically provide you with the ability to scroll (you can use a ScrollViewer control for that though - more on that in a later chapter).

You might also notice that the default orientation of the StackPanel is Vertical, unlike the WrapPanel where the default orientation is Horizontal. But just like for the WrapPanel, this can easily be changed, using the Orientation property:

<StackPanel Orientation="Horizontal">



Another thing you will likely notice is that the StackPanel stretches its child control by default. On a vertically aligned StackPanel, like the one in the first example, all child controls get stretched horizontally. On a horizontally aligned StackPanel, all child controls get stretched vertically, as seen above. The StackPanel does this by setting the HorizontalAlignment or VerticalAlignment property on its child controls to Stretch, but you can easily override this if you want to. Have a look at the next example, where we use the same markup as we did in the previous example, but this time we assign values to the VerticalAlignment property for all the child controls:

<Window x:Class="WpfTutorialSamples.Panels.StackPanel"

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

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

Title="StackPanel" Height="160" Width="300">

<StackPanel Orientation="Horizontal">

<Button VerticalAlignment="Top">Button 1</Button>

<Button VerticalAlignment="Center">Button 2</Button>

<Button VerticalAlignment="Bottom">Button 3</Button>

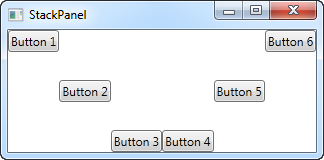
<Button VerticalAlignment="Bottom">Button 4</Button>

<Button VerticalAlignment="Center">Button 5</Button>

<Button VerticalAlignment="Top">Button 6</Button>

</StackPanel>

</Window>



We use the Top, Center and Bottom values to place the buttons in a nice pattern, just for kicks. The same can of course be done for a vertically aligned StackPanel, where you would use the HorizontalAlignment on the child controls:

<Window x:Class="WpfTutorialSamples.Panels.StackPanel"

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

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

Title="StackPanel" Height="160" Width="300">

<StackPanel Orientation="Vertical">

<Button HorizontalAlignment="Left">Button 1</Button>

<Button HorizontalAlignment="Center">Button 2</Button>

<Button HorizontalAlignment="Right">Button 3</Button>

<Button HorizontalAlignment="Right">Button 4</Button>

<Button HorizontalAlignment="Center">Button 5</Button>

<Button HorizontalAlignment="Left">Button 6</Button>

</StackPanel>

</Window>



As you can see, the controls still go from top to bottom, but instead of having the same width, each control is aligned to the left, the right or center.

# The DockPanel control

The **DockPanel** makes it easy to dock content in all four directions (top, bottom, left and right). This makes it a great choice in many situations, where you want to divide the window into specific areas, especially because by default, the last element inside the DockPanel, unless this feature is specifically disabled, will automatically fill the rest of the space (center).

As we've seen with many of the other panels in WPF, you start taking advantage of the panel possibilities by using an attached property of it, in this case the DockPanel.Dock property, which decides in which direction you want the child control to dock to. If you don't use this, the first control(s) will be docked to the left, with the last one taking up the remaining space. Here's an example on how you use it:

<Window x:Class="WpfTutorialSamples.Panels.DockPanel"

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

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

Title="DockPanel" Height="250" Width="250">

<DockPanel>

<Button DockPanel.Dock="Left">Left</Button>

<Button DockPanel.Dock="Top">Top</Button>

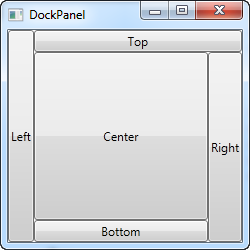
<Button DockPanel.Dock="Right">Right</Button>

<Button DockPanel.Dock="Bottom">Bottom</Button>

<Button>Center</Button>

</DockPanel>

</Window>



As already mentioned, we don't assign a dock position for the last child, because it automatically centers the control, allowing it to fill the remaining space. You will also notice that the controls around the center only takes up the amount of space that they need - everything else is left for the center position. That is also why you will see the Right button take up a bit more space than the Left button - the extra character in the text simply requires more pixels.

The last thing that you will likely notice, is how the space is divided. For instance, the Top button doesn't get all of the top space, because the Left button takes a part of it. The DockPanel decides which control to favor by looking at their position in the markup. In this case, the Left button gets precedence because it's placed first in the markup. Fortunately, this also means that it's very easy to change, as we'll see in the next example, where we have also evened out the space a bit by assigning widths/heights to the child controls:

<Window x:Class="WpfTutorialSamples.Panels.DockPanel"

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

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

Title="DockPanel" Height="250" Width="250">

<DockPanel>

<Button DockPanel.Dock="Top" Height="50">Top</Button>

<Button DockPanel.Dock="Bottom" Height="50">Bottom</Button>

<Button DockPanel.Dock="Left" Width="50">Left</Button>

<Button DockPanel.Dock="Right" Width="50">Right</Button>

<Button>Center</Button>

</DockPanel>

</Window>



The top and bottom controls now take precedence over the left and right controls, and they're all taking up 50 pixels in either height or width. If you make the window bigger or smaller, you will also see that this static width/height remains the same no matter what - only the center area increases or decreases in size as you resize the window.

## LastChildFill

As already mentioned, the default behavior is that the last child of the DockPanel takes up the rest of the space, but this can be disabled using the LastChildFill. Here's an example where we disable it, and at the same time we'll show the ability to dock more than one control to the same side:

<Window x:Class="WpfTutorialSamples.Panels.DockPanel"

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

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

Title="DockPanel" Height="300" Width="300">

<DockPanel LastChildFill="False">

<Button DockPanel.Dock="Top" Height="50">Top</Button>

<Button DockPanel.Dock="Bottom" Height="50">Bottom</Button>

<Button DockPanel.Dock="Left" Width="50">Left</Button>

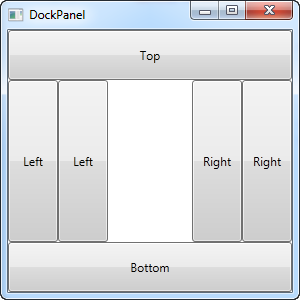
<Button DockPanel.Dock="Left" Width="50">Left</Button>

<Button DockPanel.Dock="Right" Width="50">Right</Button>

<Button DockPanel.Dock="Right" Width="50">Right</Button>

</DockPanel>

</Window>



In this example, we dock two controls to the left and two controls to the right, and at the same time, we turn off the LastChildFill property. This leaves us with empty space in the center, which may be preferable in some cases.

# The Grid Control

The Grid is probably the most complex of the panel types. A Grid can contain multiple rows and columns. You define a height for each of the rows and a width for each of the columns, in either an absolute amount of pixels, in a percentage of the available space or as auto, where the row or column will automatically adjust its size depending on the content. Use the Grid when the other panels doesn't do the job, e.g. when you need multiple columns and often in combination with the other panels.

In its most basic form, the Grid will simply take all of the controls you put into it, stretch them to use the maximum available space and place it on top of each other:

<Window x:Class="WpfTutorialSamples.Panels.Grid"

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

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

Title="Grid" Height="300" Width="300">

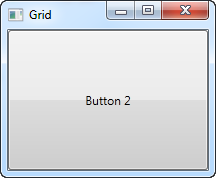
<Grid>

<Button>Button 1</Button>

<Button>Button 2</Button>

</Grid>

</Window>



As you can see, the last control gets the top position, which in this case means that you can't even see the first button. Not terribly useful for most situations though, so let's try dividing the space, which is what the grid does so well. We do that by using ColumnDefinitions and RowDefinitions. In the first example, we'll stick to columns:

<Window x:Class="WpfTutorialSamples.Panels.Grid"

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

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

Title="Grid" Height="300" Width="300">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<Button>Button 1</Button>

<Button Grid.Column="1">Button 2</Button>

</Grid>

</Window>



In this example, we have simply divided the available space into two columns, which will share the space equally, using a "star width" (this will be explained later). On the second button, I use a so-called Attached property to place the button in the second column (0 is the first column, 1 is the second and so on). I could have used this property on the first button as well, but it automatically gets assigned to the first column and the first row, which is exactly what we want here.

As you can see, the controls take up all the available space, which is the default behavior when the grid arranges its child controls. It does this by setting the HorizontalAlignment and VerticalAlignment on its child controls to Stretch.

In some situations you may want them to only take up the space they need though and/or control how they are placed in the Grid. The easiest way to do this is to set the HorizontalAlignment and VerticalAlignment directly on the controls you wish to manipulate. Here's a modified version of the above example:

<Window x:Class="WpfTutorialSamples.Panels.Grid"

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

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

Title="Grid" Height="300" Width="300">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="\*" />

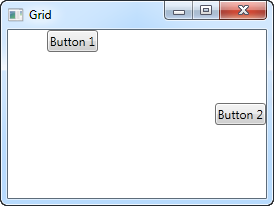
</Grid.ColumnDefinitions>

<Button VerticalAlignment="Top" HorizontalAlignment="Center">Button 1</Button>

<Button Grid.Column="1" VerticalAlignment="Center" HorizontalAlignment="Right">Button 2</Button>

</Grid>

</Window>



As you can see from the resulting screenshot, the first button is now placed in the top and centered. The second button is placed in the middle, aligned to the right.

The Grid - Rows & columns

In the last chapter, we introduced you to the great Grid panel and showed you a couple of basic examples on how to use it. In this chapter we will do some more advanced layouts, as this is where the Grid really shines. First of all, let's throw in more columns and even some rows, for a true tabular layout:

<Window x:Class="WpfTutorialSamples.Panels.TabularGrid"

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

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

Title="TabularGrid" Height="300" Width="300">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="2\*" />

<ColumnDefinition Width="1\*" />

<ColumnDefinition Width="1\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="2\*" />

<RowDefinition Height="1\*" />

<RowDefinition Height="1\*" />

</Grid.RowDefinitions>

<Button>Button 1</Button>

<Button Grid.Column="1">Button 2</Button>

<Button Grid.Column="2">Button 3</Button>

<Button Grid.Row="1">Button 4</Button>

<Button Grid.Column="1" Grid.Row="1">Button 5</Button>

<Button Grid.Column="2" Grid.Row="1">Button 6</Button>

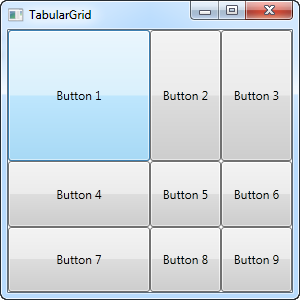
<Button Grid.Row="2">Button 7</Button>

<Button Grid.Column="1" Grid.Row="2">Button 8</Button>

<Button Grid.Column="2" Grid.Row="2">Button 9</Button>

</Grid>

</Window>



A total of nine buttons, each placed in their own cell in a grid containing three rows and three columns. We once again use a star based width, but this time we assign a number as well - the first row and the first column has a width of 2\*, which basically means that it uses twice the amount of space as the rows and columns with a width of 1\* (or just \* - that's the same).

You will also notice that I use the Attached properties Grid.Row and Grid.Column to place the controls in the grid, and once again you will notice that I have omitted these properties on the controls where I want to use either the first row or the first column (or both). This is essentially the same as specifying a zero. This saves a bit of typing, but you might prefer to assign them anyway for a better overview - that's totally up to you!

The Grid - Units

So far we have mostly used the star width/height, which specifies that a row or a column should take up a certain percentage of the combined space. However, there are two other ways of specifying the width or height of a column or a row: Absolute units and the Auto width/height. Let's try creating a Grid where we mix these:

<Window x:Class="WpfTutorialSamples.Panels.GridUnits"

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

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

Title="GridUnits" Height="200" Width="400">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="1\*" />

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="100" />

</Grid.ColumnDefinitions>

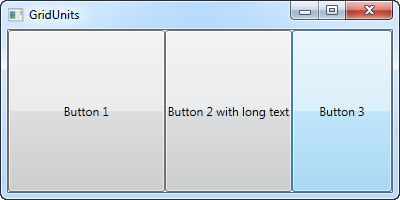
<Button>Button 1</Button>

<Button Grid.Column="1">Button 2 with long text</Button>

<Button Grid.Column="2">Button 3</Button>

</Grid>

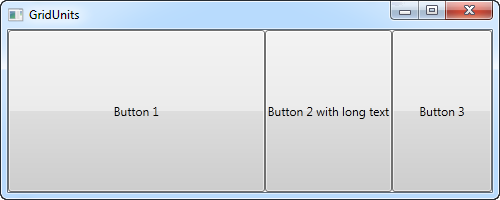
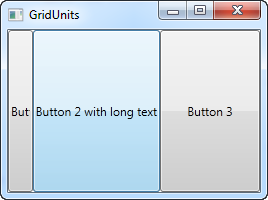
</Window>



In this example, the first button has a star width, the second one has its width set to Auto and the last one has a static width of 100 pixels.

The result can be seen on the screenshot, where the second button only takes exactly the amount of space it needs to render its longer text, the third button takes exactly the 100 pixels it was promised and the first button, with the variable width, takes the rest.

In a Grid where one or several columns (or rows) have a variable (star) width, they automatically get to share the width/height not already used by the columns/rows which uses an absolute or Auto width/height. This becomes more obvious when we resize the window:



On the first screenshot, you will see that the Grid reserves the space for the last two buttons, even though it means that the first one doesn't get all the space it needs to render properly. On the second screenshot, you will see the last two buttons keeping the exact same amount of space, leaving the surplus space to the first button.

This can be a very useful technique when designing a wide range of dialogs. For instance, consider a simple contact form where the user enters a name, an e-mail address and a comment. The first two fields will usually have a fixed height, while the last one might as well take up as much space as possible, leaving room to type a longer comment. In one of the next chapters, we will try building a contact form, using the grid and rows and columns of different heights and widths.

# The Grid - Spanning

The default Grid behavior is that each control takes up one cell, but sometimes you want a certain control to take up more rows or columns. Fortunately the Grid makes this very easy, with the Attached properties ColumnSpan and RowSpan. The default value for this property is obviously 1, but you can specify a bigger number to make the control span more rows or columns.

Here's a very simple example, where we use the ColumnSpan property:

<Window x:Class="WpfTutorialSamples.Panels.GridColRowSpan"

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

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

Title="GridColRowSpan" Height="110" Width="300">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="1\*" />

<ColumnDefinition Width="1\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="\*" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

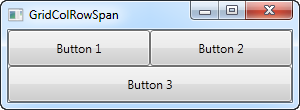
<Button>Button 1</Button>

<Button Grid.Column="1">Button 2</Button>

<Button Grid.Row="1" Grid.ColumnSpan="2">Button 3</Button>

</Grid>

</Window>



We just define two columns and two rows, all of them taking up their equal share of the place. The first two buttons just use the columns normally, but with the third button, we make it take up two columns of space on the second row, using the ColumnSpan attribute.

This is all so simple that we could have just used a combination of panels to achieve the same effect, but for just slightly more advanced cases, this is really useful. Let's try something which better shows how powerful this is:

<Window x:Class="WpfTutorialSamples.Panels.GridColRowSpanAdvanced"

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

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

Title="GridColRowSpanAdvanced" Height="300" Width="300">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="\*" />

<RowDefinition Height="\*" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<Button Grid.ColumnSpan="2">Button 1</Button>

<Button Grid.Column="3">Button 2</Button>

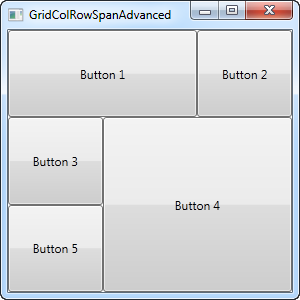
<Button Grid.Row="1">Button 3</Button>

<Button Grid.Column="1" Grid.Row="1" Grid.RowSpan="2" Grid.ColumnSpan="2">Button 4</Button>

<Button Grid.Column="0" Grid.Row="2">Button 5</Button>

</Grid>

</Window>



With three columns and three rows we would normally have nine cells, but in this example, we use a combination of row and column spanning to fill all the available space with just five buttons. As you can see, a control can span either extra columns, extra rows or in the case of button 4: both.

So as you can see, spanning multiple columns and/or rows in a Grid is very easy. In a later article, we will use the spanning, along with all the other Grid techniques in a more practical example.

# The GridSplitter

As you saw in the previous articles, the Grid panel makes it very easy to divide up the available space into individual cells. Using column and row definitions, you can easily decide how much space each row or column should take up, but what if you want to allow the user to change this? This is where the GridSplitter control comes into play.

The GridSplitter is used simply by adding it to a column or a row in a Grid, with the proper amount of space for it, e.g. 5 pixels. It will then allow the user to drag it from side to side or up and down, while changing the size of the column or row on each of the sides of it. Here's an example:

<Window x:Class="WpfTutorialSamples.Panels.GridSplitterSample"

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

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

Title="GridSplitterSample" Height="300" Width="300">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="5" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

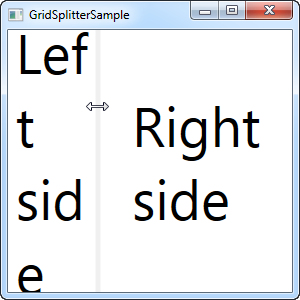
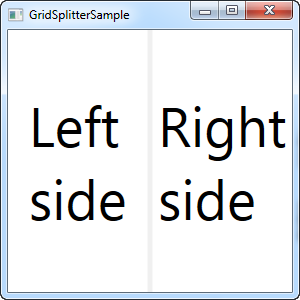
<TextBlock FontSize="55" HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap">Left side</TextBlock>

<GridSplitter Grid.Column="1" Width="5" HorizontalAlignment="Stretch" />

<TextBlock Grid.Column="2" FontSize="55" HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap">Right side</TextBlock>

</Grid>

</Window>



As you can see, I've simply created a Grid with two equally wide columns, with a 5 pixel column in the middle. Each of the sides are just a TextBlock control to illustrate the point. As you can see from the screenshots, the GridSplitter is rendered as a dividing line between the two columns and as soon as the mouse is over it, the cursor is changed to reflect that it can be resized.

## Horizontal GridSplitter

The GridSplitter is very easy to use and of course it supports horizontal splits as well. In fact, you hardly have to change anything to make it work horizontally instead of vertically, as the next example will show:

<Window x:Class="WpfTutorialSamples.Panels.GridSplitterHorizontalSample"

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

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

Title="GridSplitterHorizontalSample" Height="300" Width="300">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*" />

<RowDefinition Height="5" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

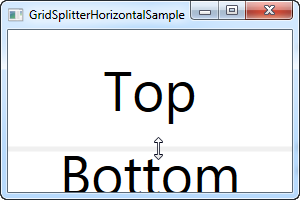
<TextBlock FontSize="55" HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap">Top</TextBlock>

<GridSplitter Grid.Row="1" Height="5" HorizontalAlignment="Stretch" />

<TextBlock Grid.Row="2" FontSize="55" HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap">Bottom</TextBlock>

</Grid>

</Window>



As you can see, I simply changed the columns into rows and on the GridSplitter, I defined a Height instead of a Width. The GridSplitter figures out the rest on its own, but in case it doesn't, you can use the **ResizeDirection** property on it to force it into either Rows or Columns mode.

Using the Grid: A contact form

In the last couple of chapters we went through a lot of theoretic information, each with some very theoretic examples. In this chapter we will combine what we have learned about the Grid so far, into an example that can be used in the real world: A simple contact form.

The good thing about the contact form is that it's just an example of a commonly used dialog - you can take the techniques used and apply them to almost any type of dialog that you need to create.

The first take on this task is very simple and will show you a very basic contact form. It uses three rows, two of them with Auto heights and the last one with star height, so it consumes the rest of the available space:

<Window x:Class="WpfTutorialSamples.Panels.GridContactForm"

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

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

Title="GridContactForm" Height="300" Width="300">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

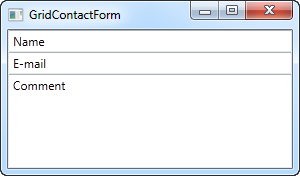
<TextBox>Name</TextBox>

<TextBox Grid.Row="1">E-mail</TextBox>

<TextBox Grid.Row="2" AcceptsReturn="True">Comment</TextBox>

</Grid>

</Window>



As you can see, the last TextBox simply takes up the remaining space, while the first two only takes up the space they require. Try resizing the window and you will see the comment TextBox resize with it.

In this very simple example, there are no labels to designate what each of the fields are for. Instead, the explanatory text is inside the TextBox, but this is not generally how a Windows dialog looks. Let's try improving the look and usability a bit:

<Window x:Class="WpfTutorialSamples.Panels.GridContactFormTake2"

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

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

Title="GridContactFormTake2" Height="300" Width="300">

<Grid Margin="10">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<Label>Name: </Label>

<TextBox Grid.Column="1" Margin="0,0,0,10" />

<Label Grid.Row="1">E-mail: </Label>

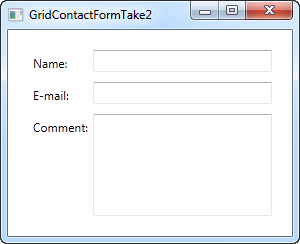
<TextBox Grid.Row="1" Grid.Column="1" Margin="0,0,0,10" />

<Label Grid.Row="2">Comment: </Label>

<TextBox Grid.Row="2" Grid.Column="1" AcceptsReturn="True" />

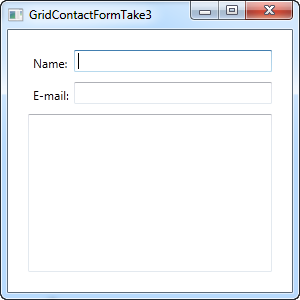
</Grid>

</Window>



But perhaps you're in a situation where the comment field is pretty self-explanatory? In that case, let's skip the label and use ColumnSpan to get even more space for the comment TextBox:

<TextBox Grid.ColumnSpan="2" Grid.Row="2" AcceptsReturn="True" />



So as you can see, the Grid is a very powerful panel. Hopefully you can use all of these techniques when designing your own dialogs.

WPF Resources:-

# Resources

WPF introduces a very handy concept: The ability to store data as a resource, either locally for a control, locally for the entire window or globally for the entire application. The data can be pretty much whatever you want, from actual information to a hierarchy of WPF controls. This allows you to place data in one place and then use it from or several other places, which is very useful.

The concept is used a lot for styles and templates, which we'll discuss later on in this tutorial, but as it will be illustrated in this chapter, you can use it for many other things as well. Allow me to demonstrate it with a simple example:

<Window x:Class="WpfTutorialSamples.WPF\_Application.ResourceSample"

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

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

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

Title="ResourceSample" Height="150" Width="350">

<Window.Resources>

<sys:String x:Key="strHelloWorld">Hello, world!</sys:String>

</Window.Resources>

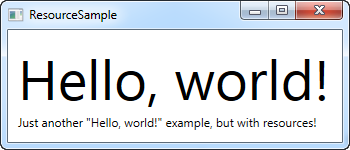
<StackPanel Margin="10">

<TextBlock Text="{StaticResource strHelloWorld}" FontSize="56" />

<TextBlock>Just another "<TextBlock Text="{StaticResource strHelloWorld}" />" example, but with resources!</TextBlock>

</StackPanel>

</Window>



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

## StaticResource vs. DynamicResource

In the examples so far, I have used the StaticResource markup extension to reference a resource. However, an alternative exists, in form of the DynamicResource.

The main difference is that a static resource is resolved only once, which is at the point where the XAML is loaded. If the resource is then changed later on, this change will not be reflected where you have used the StaticResource.

A DynamicResource on the other hand, is resolved once it's actually needed, and then again if the resource changes. Think of it as binding to a static value vs. binding to a function that monitors this value and sends it to you each time it's changed - it's not exactly how it works, but it should give you a better idea of when to use what. Dynamic resources also allows you to use resources which are not even there during design time, e.g. if you add them from Code-behind during the startup of the application.

## More resource types

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

<Window x:Class="WpfTutorialSamples.WPF\_Application.ExtendedResourceSample"

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

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

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

Title="ExtendedResourceSample" Height="160" Width="300"

Background="{DynamicResource WindowBackgroundBrush}">

<Window.Resources>

<sys:String x:Key="ComboBoxTitle">Items:</sys:String>

<x:Array x:Key="ComboBoxItems" Type="sys:String">

<sys:String>Item #1</sys:String>

<sys:String>Item #2</sys:String>

<sys:String>Item #3</sys:String>

</x:Array>

<LinearGradientBrush x:Key="WindowBackgroundBrush">

<GradientStop Offset="0" Color="Silver"/>

<GradientStop Offset="1" Color="Gray"/>

</LinearGradientBrush>

</Window.Resources>

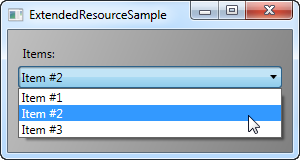
<StackPanel Margin="10">

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

<ComboBox ItemsSource="{StaticResource ComboBoxItems}" />

</StackPanel>

</Window>



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

## Local and application wide resources

For now, we have stored resources on a window-level, which means that you can access them from all over the window.

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

<StackPanel Margin="10">

<StackPanel.Resources>

<sys:String x:Key="ComboBoxTitle">Items:</sys:String>

</StackPanel.Resources>

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

</StackPanel>

In this case, we add the resource to the StackPanel and then use it from its child control, the Label. Other controls inside of the StackPanel could have used it as well, just like children of these child controls would have been able to access it. Controls outside of this particular StackPanel wouldn't have access to it, though.

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

<Application x:Class="WpfTutorialSamples.App"

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

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

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

StartupUri="WPF application/ExtendedResourceSample.xaml">

<Application.Resources>

<sys:String x:Key="ComboBoxTitle">Items:</sys:String>

</Application.Resources>

</Application>

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

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

## Resources from Code-behind

So far, we've accessed all of our resources directly from XAML, using a markup extension. However, you can of course access your resources from Code-behind as well, which can be useful in several situations. In the previous example, we saw how we could store resources in several different places, so in this example, we'll be accessing three different resources from Code-behind, each stored in a different scope:

**App.xaml:**

<Application x:Class="WpfTutorialSamples.App"

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

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

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

StartupUri="WPF application/ResourcesFromCodeBehindSample.xaml">

<Application.Resources>

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

</Application.Resources>

</Application>

**Window:**

<Window x:Class="WpfTutorialSamples.WPF\_Application.ResourcesFromCodeBehindSample"

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

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

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

Title="ResourcesFromCodeBehindSample" Height="175" Width="250">

<Window.Resources>

<sys:String x:Key="strWindow">Hello, Window world!</sys:String>

</Window.Resources>

<DockPanel Margin="10" Name="pnlMain">

<DockPanel.Resources>

<sys:String x:Key="strPanel">Hello, Panel world!</sys:String>

</DockPanel.Resources>

<WrapPanel DockPanel.Dock="Top" HorizontalAlignment="Center" Margin="10">

<Button Name="btnClickMe" Click="btnClickMe\_Click">Click me!</Button>

</WrapPanel>

<ListBox Name="lbResult" />

</DockPanel>

</Window>

**Code-behind:**

using System;

using System.Windows;

namespace WpfTutorialSamples.WPF\_Application

{

public partial class ResourcesFromCodeBehindSample : Window

{

public ResourcesFromCodeBehindSample()

{

InitializeComponent();

}

private void btnClickMe\_Click(object sender, RoutedEventArgs e)

{

lbResult.Items.Add(pnlMain.FindResource("strPanel").ToString());

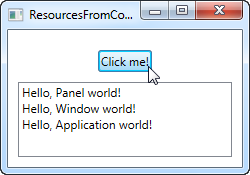
lbResult.Items.Add(this.FindResource("strWindow").ToString());

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

}

}

}



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

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

Notice how we use the FindResource() method on different scopes - first on the panel, then on the window and then on the current **Application** object. It makes sense to look for the resource where we know it is, but as already mentioned, if a resource is not found, the search progresses up the hierarchy, so in principal, we could have used the FindResource() method on the panel in all three cases, since it would have continued up to the window and later on up to the application level, if not found.

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

Basic Control:-

# The TextBlock control

*TextBlock is not a control, per se, since it doesn't inherit from the Control class, but it's used much like any other control in the WPF framework, so we'll call it a control to keep things simple.*

The **TextBlock** control is one of the most fundamental controls in WPF, yet it's very useful. It allows you to put text on the screen, much like a Label control does, but in a simpler and less resource demanding way. A common understanding is that a Label is for short, one-line texts (but may include e.g. an image), while the TextBlock works very well for multiline strings as well, but can only contain text (strings). Both the Label and the TextBlock offers their own unique advantages, so what you should use very much depends on the situation.

We already used a TextBlock control in the "Hello, WPF!" article, but for now, let's have a look at the TextBlock in its simplest form:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBlockSample"

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

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

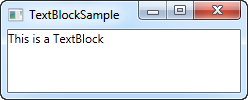
Title="TextBlockSample" Height="100" Width="200">

<Grid>

<TextBlock>This is a TextBlock</TextBlock>

</Grid>

</Window>



That's as simple as it comes and if you have read the previous chapters of this tutorial, then there should be nothing new here. The text between the TextBlock is simply a shortcut for setting the Text property of the TextBlock.

For the next example, let's try a longer text to show how the TextBlock deals with that. I've also added a bit of margin, to make it look just a bit better:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBlockSample"

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

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

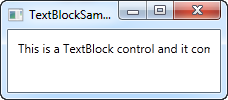
Title="TextBlockSample" Height="100" Width="200">

<Grid>

<TextBlock Margin="10">This is a TextBlock control and it comes with a very long text</TextBlock>

</Grid>

</Window>



## Dealing with long strings

As you will soon realize from the screenshot, the TextBlock is perfectly capable of dealing with long, multiline texts, but it will not do anything by default. In this case the text is too long to be rendered inside the window, so WPF renders as much of the text as possible and then just stops.

Fortunately, there are several ways of dealing with this. In the next example I'll show you all of them, and then I'll explain each of them afterwards:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBlockSample"

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

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

Title="TextBlockSample" Height="200" Width="250">

<StackPanel>

<TextBlock Margin="10" Foreground="Red">

This is a TextBlock control<LineBreak />

with multiple lines of text.

</TextBlock>

<TextBlock Margin="10" TextTrimming="CharacterEllipsis" Foreground="Green">

This is a TextBlock control with text that may not be rendered completely, which will be indicated with an ellipsis.

</TextBlock>

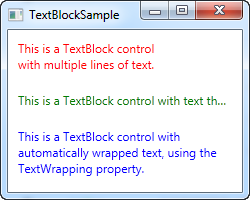
<TextBlock Margin="10" TextWrapping="Wrap" Foreground="Blue">

This is a TextBlock control with automatically wrapped text, using the TextWrapping property.

</TextBlock>

</StackPanel>

</Window>



So, we have three TextBlock controls, each with a different color (using the Foreground property) for an easier overview. They all handle the fact that their text content is too long in different ways:

The red TextBlock uses a **LineBreak** tag to manually break the line at a designated location. This gives you absolute control over where you want the text to break onto a new line, but it's not very flexible for most situations. If the user makes the window bigger, the text will still wrap at the same position, even though there may now be room enough to fit the entire text onto one line.

The green TextBlock uses the **TextTrimming** property with the value **CharacterEllipsis** to make the TextBlock show an ellipsis (...) when it can't fit any more text into the control. This is a common way of showing that there's more text, but not enough room to show it. This is great when you have text that might be too long but you absolutely don't want it to use more than one line. As an alternative to **CharacterEllipsis** you may use **WordEllipsis**, which will trim the text at the end of the last possible word instead of the last possible character, preventing that a word is only shown in part.

The blue TextBlock uses the **TextWrapping** property with the value **Wrap**, to make the TextBlock wrap to the next line whenever it can't fit anymore text into the previous line. Contrary to the first TextBlock, where we manually define where to wrap the text, this happens completely automatic and even better: It's also automatically adjusted as soon as the TextBlock get more or less space available. Try making the window in the example bigger or smaller and you will see how the wrapping is updated to match the situation.

This was all about dealing with simple strings in the TextBlock. In the next chapter, we'll look into some of the more advanced functionality of the TextBlock, which allows us to create text of various styles within the TextBlock and much more.

# The TextBlock control - Inline formatting

In the last article we looked at the core functionality of the TextBlock control: Displaying a simple string and wrapping it if necessary. We even used another color than the default for rendering the text, but what if you wanted to do more than just define a static color for all the text in the TextBlock?

Luckily the TextBlock control supports inline content. These small control-like constructs all inherit from the Inline class, which means that they can be rendered inline, as a part of a larger text. As of writing, the supported elements include AnchoredBlock, Bold, Hyperlink, InlineUIContainer, Italic, LineBreak, Run, Span, and Underline. In the following examples, we'll have a look at most of them.

## Bold, Italic and Underline

These are probably the simplest types of inline elements. The names should tell you a lot about what they do, but we'll still give you a quick example on how to use them:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBlockInlineSample"

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

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

Title="TextBlockInlineSample" Height="100" Width="300">

<Grid>

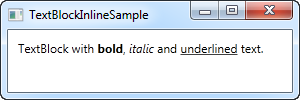
<TextBlock Margin="10" TextWrapping="Wrap">

TextBlock with <Bold>bold</Bold>, <Italic>italic</Italic> and <Underline>underlined</Underline> text.

</TextBlock>

</Grid>

</Window>



Much like with HTML, you just surround your text with a Bold tag to get bold text and so on. This makes it very easy to create and display diverse text in your applications.

All three of these tags are just child classes of the Span element, each setting a specific property on the Span element to create the desired effect. For instance, the Bold tag just sets the FontWeight property on the underlying Span element, the Italic element sets the FontStyle and so on.

## LineBreak

Simply inserts a line break into the text. Please see the previous chapter for an example where we use the LineBreak element.

## Hyperlink

The Hyperlink element allows you to have links in your text. It's rendered with a style that suits your current Windows theme, which will usually be some sort of underlined blue text with a red hover effect and a hand mouse cursor. You can use the NavigateUri property to define the URL that you wish to navigate to. Here's an example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBlockHyperlinkSample"

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

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

Title="TextBlockHyperlinkSample" Height="100" Width="300">

<Grid>

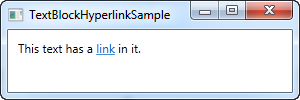
<TextBlock Margin="10" TextWrapping="Wrap">

This text has a <Hyperlink RequestNavigate="Hyperlink\_RequestNavigate" NavigateUri="https://www.google.com">link</Hyperlink> in it.

</TextBlock>

</Grid>

</Window>



The Hyperlink is also used inside of WPF Page's, where it can be used to navigate between pages. In that case, you won't have to specifically handle the RequestNavigate event, like we do in the example, but for launching external URL's from a regular WPF application, we need a bit of help from this event and the Process class. We subscribe to the RequestNavigate event, which allows us to launch the linked URL in the users default browser with a simple event handler like this one in the code behind file:

private void Hyperlink\_RequestNavigate(object sender, System.Windows.Navigation.RequestNavigateEventArgs e)

{

System.Diagnostics.Process.Start(e.Uri.AbsoluteUri);

}

## Run

The Run element allows you to style a string using all the available properties of the Span element, but while the Span element may contain other inline elements, a Run element may only contain plain text. This makes the Span element more flexible and therefore the logical choice in most cases.

## Span

The Span element doesn't have any specific rendering by default, but allows you to set almost any kind of specific rendering, including font size, style and weight, background and foreground colors and so on. The great thing about the Span element is that it allows for other inline elements inside of it, making it easy to do even advanced combinations of text and style. In the following example, I have used many Span elements to show you some of the many possibilities when using inline Span elements:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBlockSpanSample"

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

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

Title="TextBlockSpanSample" Height="100" Width="300">

<Grid>

<TextBlock Margin="10" TextWrapping="Wrap">

This <Span FontWeight="Bold">is</Span> a

<Span Background="Silver" Foreground="Maroon">TextBlock</Span>

with <Span TextDecorations="Underline">several</Span>

<Span FontStyle="Italic">Span</Span> elements,

<Span Foreground="Blue">

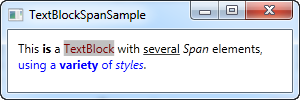
using a <Bold>variety</Bold> of <Italic>styles</Italic>

</Span>.

</TextBlock>

</Grid>

</Window>



So as you can see, if none of the other elements make sense in your situation or if you just want a blank canvas when starting to format your text, the Span element is a great choice.

## Formatting text from C#/Code-Behind

As you can see, formatting text through XAML is very easy, but in some cases, you might prefer or even need to do it from your C#/Code-Behind file. This is a bit more cumbersome, but here's an example on how you may do it:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBlockCodeBehindSample"

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

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

Title="TextBlockCodeBehindSample" Height="100" Width="300">

<Grid></Grid>

</Window>

using System;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Documents;

using System.Windows.Media;

namespace WpfTutorialSamples.Basic\_controls

{

public partial class TextBlockCodeBehindSample : Window

{

public TextBlockCodeBehindSample()

{

InitializeComponent();

TextBlock tb = new TextBlock();

tb.TextWrapping = TextWrapping.Wrap;

tb.Margin = new Thickness(10);

tb.Inlines.Add("An example on ");

tb.Inlines.Add(new Run("the TextBlock control ") { FontWeight = FontWeights.Bold });

tb.Inlines.Add("using ");

tb.Inlines.Add(new Run("inline ") { FontStyle = FontStyles.Italic });

tb.Inlines.Add(new Run("text formatting ") { Foreground = Brushes.Blue });

tb.Inlines.Add("from ");

tb.Inlines.Add(new Run("Code-Behind") { TextDecorations = TextDecorations.Underline });

tb.Inlines.Add(".");

this.Content = tb;

}

}

}

# The Label control

The Label control, in its most simple form, will look very much like the TextBlock which we used in another article. You will quickly notice though that instead of a Text property, the Label has a Content property. The reason for that is that the Label can host any kind of control directly inside of it, instead of just text. This content can be a string as well though, as you will see in this first and very basic example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.LabelControlSample"

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

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

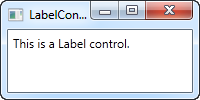
Title="LabelControlSample" Height="100" Width="200">

<Grid>

<Label Content="This is a Label control." />

</Grid>

</Window>



Another thing you might notice is the fact that the Label, by default, has a bit of padding, allowing the text to be rendered a few pixels away from the top, left corner. This is not the case for the TextBlock control, where you will have to specify it manually.

In a simple case like this, where the content is simply a string, the Label will actually create a TextBlock internally and show your string in that.

## The Label control vs. the TextBlock control

So why use a Label at all then? Well, there are a few important differences between the Label and the TextBlock. The TextBlock only allows you to render a text string, while the Label also allows you to:

* Specify a border
* Render other controls, e.g. an image
* Use templated content through the ContentTemplate property
* **Use access keys to give focus to related controls**

The last bullet point is actually one of the main reasons for using a Label over the TextBlock control. Whenever you just want to render simple text, you should use the TextBlock control, since it's lighter and performs better than the Label in most cases.

## Label and Access keys (mnemonics)

In Windows and other operating systems as well, it's common practice that you can access controls in a dialog by holding down the [Alt] key and then pressing a character which corresponds to the control that you wish to access. The character to press will be highlighted when you hold down the [Alt] key. TextBlock controls doesn't support this functionality, but the Label does, so for control labels, the Label control is usually an excellent choice. Let's look at an example of it in action:

<Window x:Class="WpfTutorialSamples.Basic\_controls.LabelControlSample"

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

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

Title="LabelControlSample" Height="180" Width="250">

<StackPanel Margin="10">

<Label Content="\_Name:" Target="{Binding ElementName=txtName}" />

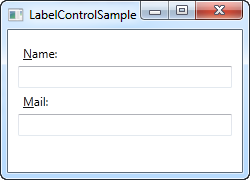
<TextBox Name="txtName" />

<Label Content="\_Mail:" Target="{Binding ElementName=txtMail}" />

<TextBox Name="txtMail" />

</StackPanel>

</Window>



The screenshot shows our sample dialog as it looks when the Alt key is pressed. Try running it, holding down the [Alt] key and then pressing N and M. You will see how focus is moved between the two textboxes.

So, there's several new concepts here. First of all, we define the access key by placing an underscore (\_) before the character. It doesn't have to be the first character, it can be before any of the characters in your label content. The common practice is to use the first character that's not already used as an access key for another control.

We use the **Target** property to connect the Label and the designated control. We use a standard WPF binding for this, using the **ElementName** property, all of which we will describe later on in this tutorial. The binding is based on the name of the control, so if you change this name, you will also have to remember to change the binding.

## Using controls as Label content

As already mentioned, the Label control allows you to host other controls, while still keeping the other benefits. Let's try an example where we have both an image and a piece of text inside the Label, while also having an access key for each of the labels:

<Window x:Class="WpfTutorialSamples.Basic\_controls.LabelControlAdvancedSample"

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

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

Title="LabelControlAdvancedSample" Height="180" Width="250">

<StackPanel Margin="10">

<Label Target="{Binding ElementName=txtName}">

<StackPanel Orientation="Horizontal">

<Image Source="http://cdn1.iconfinder.com/data/icons/fatcow/16/bullet\_green.png" />

<AccessText Text="\_Name:" />

</StackPanel>

</Label>

<TextBox Name="txtName" />

<Label Target="{Binding ElementName=txtMail}">

<StackPanel Orientation="Horizontal">

<Image Source="http://cdn1.iconfinder.com/data/icons/fatcow/16/bullet\_blue.png" />

<AccessText Text="\_Mail:" />

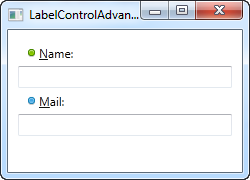
</StackPanel>

</Label>

<TextBox Name="txtMail" />

</StackPanel>

</Window>



This is just an extended version of the previous example - instead of a simple text string, our Label will now host both an image and a piece of text (inside the AccessText control, which allows us to still use an access key for the label). Both controls are inside a horizontal StackPanel, since the Label, just like any other ContentControl derivate, can only host one direct child control.

*The Image control, described later in this tutorial, uses a remote image - this is ONLY for demonstrational purposes and is NOT a good idea for most real life applications.*

# The TextBox control

The TextBox control is the most basic text-input control found in WPF, allowing the end-user to write plain text, either on a single line, for dialog input, or in multiple lines, like an editor.

## Single-line TextBox

The TextBox control is such a commonly used thing that you actually don't have to use any properties on it, to have a full-blown editable text field. Here's a barebone example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBoxSample"

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

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

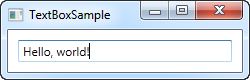
Title="TextBoxSample" Height="80" Width="250">

<StackPanel Margin="10">

<TextBox />

</StackPanel>

</Window>



That's all you need to get a text field. I added the text after running the sample and before taking the screenshot, but you can do it through markup as well, to pre-fill the textbox, using the Text property:

<TextBox Text="Hello, world!" />

Try right-clicking in the TextBox. You will get a menu of options, allowing you to use the TextBox with the Windows Clipboard. The default keyboard shortcuts for undoing and redoing (Ctrl+Z and Ctrl+Y) should also work, and all of this functionality you get for free!

## Multi-line TextBox

If you run the above example, you will notice that the TextBox control by default is a single-line control. Nothing happens when you press Enter and if you add more text than what can fit on a single line, the control just scrolls. However, making the TextBox control into a multi-line editor is very simple:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBoxSample"

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

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

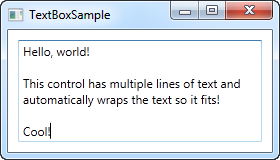
Title="TextBoxSample" Height="160" Width="280">

<Grid Margin="10">

<TextBox AcceptsReturn="True" TextWrapping="Wrap" />

</Grid>

</Window>



I have added two properties: The AcceptsReturn makes the TextBox into a multi-line control by allowing the use of the Enter/Return key to go to the next line, and the TextWrapping property, which will make the text wrap automatically when the end of a line is reached.

## Spellcheck with TextBox

As an added bonus, the TextBox control actually comes with automatic spell checking for English and a couple of other languages (as of writing, English, French, German, and Spanish languages are supported).

It works much like in Microsoft Word, where spelling errors are underlined and you can right-click it for suggested alternatives. Enabling spell checking is very easy:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBoxSample"

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

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

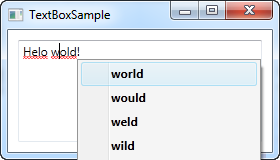
Title="TextBoxSample" Height="160" Width="280">

<Grid Margin="10">

<TextBox AcceptsReturn="True" TextWrapping="Wrap" SpellCheck.IsEnabled="True" Language="en-US" />

</Grid>

</Window>



We have used the previous, multi-line textbox example as the basis and then I have added two new properties: The attached property from the SpellCheck class called IsEnabled, which simply enables spell checking on the parent control, and the Language property, which instructs the spell checker which language to use.

## Working with TextBox selections

Just like any other editable control in Windows, the TextBox allows for selection of text, e.g. to delete an entire word at once or to copy a piece of the text to the clipboard. The WPF TextBox has several properties for working with selected text, all of them which you can read or even modify. In the next example, we will be reading these properties:

<Window x:Class="WpfTutorialSamples.Basic\_controls.TextBoxSelectionSample"

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

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

Title="TextBoxSelectionSample" Height="150" Width="300">

<DockPanel Margin="10">

<TextBox SelectionChanged="TextBox\_SelectionChanged" DockPanel.Dock="Top" />

<TextBox Name="txtStatus" AcceptsReturn="True" TextWrapping="Wrap" IsReadOnly="True" />

</DockPanel>

</Window>

The example consists of two TextBox controls: One for editing and one for outputting the current selection status to. For this, we set the IsReadOnly property to true, to prevent editing of the status TextBox. We subscribe the SelectionChanged event on the first TextBox, which we handle in the Code-behind:

using System;

using System.Text;

using System.Windows;

using System.Windows.Controls;

namespace WpfTutorialSamples.Basic\_controls

{

public partial class TextBoxSelectionSample : Window

{

public TextBoxSelectionSample()

{

InitializeComponent();

}

private void TextBox\_SelectionChanged(object sender, RoutedEventArgs e)

{

TextBox textBox = sender as TextBox;

txtStatus.Text = "Selection starts at character #" + textBox.SelectionStart + Environment.NewLine;

txtStatus.Text += "Selection is " + textBox.SelectionLength + " character(s) long" + Environment.NewLine;

txtStatus.Text += "Selected text: '" + textBox.SelectedText + "'";

}

}

}



We use three interesting properties to accomplish this:

**SelectionStart** , which gives us the current cursor position or if there's a selection: Where it starts.

**SelectionLength** , which gives us the length of the current selection, if any. Otherwise it will just return 0.

**SelectedText** , which gives us the currently selected string if there's a selection. Otherwise an empty string is returned.

## Modifying the selection

All of these properties are both readable and writable, which means that you can modify them as well. For instance, you can set the SelectionStart and SelectionLength properties to select a custom range of text, or you can use the SelectedText property to insert and select a string. Just remember that the TextBox has to have focus, e.g. by calling the Focus() method first, for this to work.

**Basic controls:**

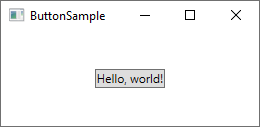
# The Button control

No GUI framework would be complete without a Button control, so of course WPF has a nice one included, and just like the rest of the framework controls, it's very flexible and will allow you to accomplish almost anything. But let's start out with some basic examples.

## A simple Button

Just like many other WPF controls, a Button can be displayed simply by adding a Button tag to your Window. If you put text between the tags (or another control), it will act as the content of the Button:

<Button>Hello, world!</Button>



Pretty simple, right? Of course, the Button doesn't actually do anything yet, but if you point to it, you will find that it comes with a nice hover effect right out of the box. But let's make the Button do something, by subscribing to its **Click** event (more information about this process can be found in the article on subscribing to events in XAML):

<Button Click="HelloWorldButton\_Click">Hello, World!</Button>

In Code-behind, you will need a matching method to handle the click:

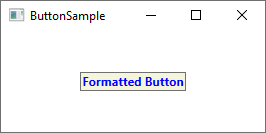
private void HelloWorldButton\_Click(object sender, RoutedEventArgs e)  
{  
    MessageBox.Show("Hello, world!");  
}

You now have a very basic button and when you click on it, a message will be displayed!

### Formatted content

Internally, simple text inside the Content of the Button is turned into a TextBlock control, which also means that you can control the same aspects of the text formatting. You will find several properties on the Button control for doing this, including (but not limited to) **Foreground**, **Background**, **FontWeight** and so on. In other words, it's very easy to change the formatting of the text inside a Button control:

<Button Background="Beige" Foreground="Blue" FontWeight="Bold">Formatted Button</Button>

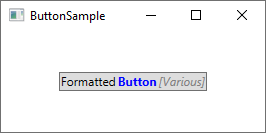


By setting these properties directly on the Button, you are of course limited to applying the same formatting for all of the content, but if that's not good enough, just read on for even more advanced content formatting.

## Buttons with advanced content

We have already talked about this several times, but one of the very cool things about WPF is the ability to replace simple text inside a control with other WPF controls. This also means that you don't have to limit your buttons to simple text, formatted in the same way - you can just add several text controls with different formatting. The WPF Button only supports one direct child control, but you can just make that a Panel, which will then host as many controls as you need to. You can use this to create buttons with various types of formatting:

<Button>  
    <StackPanel Orientation="Horizontal">  
 <TextBlock>Formatted </TextBlock>  
 <TextBlock Foreground="Blue" FontWeight="Bold" Margin="2,0">Button</TextBlock>  
 <TextBlock Foreground="Gray" FontStyle="Italic">[Various]</TextBlock>  
    </StackPanel>  
</Button>

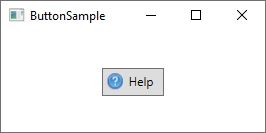


But of course, you are not limited to just text - you can put whatever you want inside your buttons, which leads us to a subject that I know many people will ask for. Buttons with images!

### Buttons with Images (ImageButton)

In many UI frameworks, you will find a regular Button and then one or several other variants, which will offer extra features. One of the most commonly used variants is the **ImageButton**, which, as the name implies, is a Button which will usually allow you to include an image before the text. But in WPF, there's no need for a separate control to accomplish this - as you just saw, we can put several controls inside a Button, so you can just as easily add an Image control to it, like this:

<Button Padding="5">    
    <StackPanel Orientation="Horizontal">    
 <Image Source="/WpfTutorialSamples;component/Images/help.png" />    
 <TextBlock Margin="5,0">Help</TextBlock>    
    </StackPanel>    
</Button>



It's really that simple to create an ImageButton in WPF, and you are of course free to move things around, e.g. if you want the image after the text instead of before etc.

## Button Padding

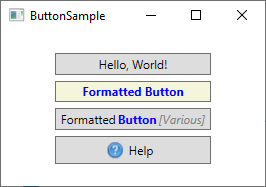
You may have noticed that buttons in the WPF framework doesn't come with any padding by default. This means that the text is very close to the borders, which might look a little bit strange, because most buttons found elsewhere (web, other applications etc.) do have at least some padding in the sides. No worries, because the Button comes with a **Padding** property:

<Button Padding="5,2">Hello, World!</Button>

This will apply a padding of 5 pixels on the sides, and 2 pixels in the top and bottom. But having to apply padding to all of your buttons might get a bit tiresome at a certain point, so here's a small tip: You can apply the padding globally, either across the entire application or just this specific Window, using a Style (more on styles later). Here's an example where we apply it to the Window, using the Window.Resources property:

<Window.Resources>  
    <Style TargetType="{x:Type Button}">  
 <Setter Property="Padding" Value="5,2"/>  
    </Style>  
</Window.Resources>

This padding will now be applied to all your buttons, but you can of course override it by specifically defining the Padding property on a Button. Here's how all the buttons of this example look with the common padding:



**Basic controls:**

# The CheckBox control

The CheckBox control allows the end-user to toggle an option on or off, usually reflecting a Boolean value in the Code-behind. Let's jump straight into an example, in case you're not sure how a CheckBox looks:

<Window x:Class="WpfTutorialSamples.Basic\_controls.CheckBoxSample"

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

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

Title="CheckBoxSample" Height="140" Width="250">

<StackPanel Margin="10">

<Label FontWeight="Bold">Application Options</Label>

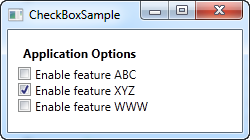
<CheckBox>Enable feature ABC</CheckBox>

<CheckBox IsChecked="True">Enable feature XYZ</CheckBox>

<CheckBox>Enable feature WWW</CheckBox>

</StackPanel>

</Window>



As you can see, the CheckBox is very easy to use. On the second CheckBox, I use the IsChecked property to have it checked by default, but other than that, no properties are needed to use it. The IsChecked property should also be used from Code-behind if you want to check whether a certain CheckBox is checked or not.

## Custom content

The CheckBox control inherits from the ContentControl class, which means that it can take custom content and display next to it. If you just specify a piece of text, like I did in the example above, WPF will put it inside a TextBlock control and display it, but this is just a shortcut to make things easier for you. You can use any type of control inside of it, as we'll see in the next example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.CheckBoxSample"

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

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

Title="CheckBoxSample" Height="140" Width="250">

<StackPanel Margin="10">

<Label FontWeight="Bold">Application Options</Label>

<CheckBox>

<TextBlock>

Enable feature <Run Foreground="Green" FontWeight="Bold">ABC</Run>

</TextBlock>

</CheckBox>

<CheckBox IsChecked="True">

<WrapPanel>

<TextBlock>

Enable feature <Run FontWeight="Bold">XYZ</Run>

</TextBlock>

<Image Source="/WpfTutorialSamples;component/Images/question.png" Width="16" Height="16" Margin="5,0" />

</WrapPanel>

</CheckBox>

<CheckBox>

<TextBlock>

Enable feature <Run Foreground="Blue" TextDecorations="Underline" FontWeight="Bold">WWW</Run>

</TextBlock>

</CheckBox>

</StackPanel>

</Window>



As you can see from the sample markup, you can do pretty much whatever you want with the content. On all three check boxes, I do something differently with the text, and on the middle one I even throw in an Image control. By specifying a control as the content, instead of just text, we get much more control of the appearance, and the cool thing is that no matter which part of the content you click on, it will activate the CheckBox and toggle it on or off.

## The IsThreeState property

As mentioned, the CheckBox usually corresponds to a boolean value, which means that it only has two states: true or false (on or off). However, since a boolean data type might be nullable, effectively allowing for a third option (true, false or null), the CheckBox control can also support this case. By setting the IsThreeState property to true, the CheckBox will get a third state called "the indeterminate state".

A common usage for this is to have a "Enable all" CheckBox, which can control a set of child checkboxes, as well as show their collective state. Our example shows how you may create a list of features that can be toggled on and off, with a common "Enable all" CheckBox in the top:

<Window x:Class="WpfTutorialSamples.Basic\_controls.CheckBoxThreeStateSample"

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

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

Title="CheckBoxThreeStateSample" Height="170" Width="300">

<StackPanel Margin="10">

<Label FontWeight="Bold">Application Options</Label>

<StackPanel Margin="10,5">

<CheckBox IsThreeState="True" Name="cbAllFeatures" Checked="cbAllFeatures\_CheckedChanged" Unchecked="cbAllFeatures\_CheckedChanged">Enable all</CheckBox>

<StackPanel Margin="20,5">

<CheckBox Name="cbFeatureAbc" Checked="cbFeature\_CheckedChanged" Unchecked="cbFeature\_CheckedChanged">Enable feature ABC</CheckBox>

<CheckBox Name="cbFeatureXyz" IsChecked="True" Checked="cbFeature\_CheckedChanged" Unchecked="cbFeature\_CheckedChanged">Enable feature XYZ</CheckBox>

<CheckBox Name="cbFeatureWww" Checked="cbFeature\_CheckedChanged" Unchecked="cbFeature\_CheckedChanged">Enable feature WWW</CheckBox>

</StackPanel>

</StackPanel>

</StackPanel>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.Basic\_controls

{

public partial class CheckBoxThreeStateSample : Window

{

public CheckBoxThreeStateSample()

{

InitializeComponent();

}

private void cbAllFeatures\_CheckedChanged(object sender, RoutedEventArgs e)

{

bool newVal = (cbAllFeatures.IsChecked == true);

cbFeatureAbc.IsChecked = newVal;

cbFeatureXyz.IsChecked = newVal;

cbFeatureWww.IsChecked = newVal;

}

private void cbFeature\_CheckedChanged(object sender, RoutedEventArgs e)

{

cbAllFeatures.IsChecked = null;

if((cbFeatureAbc.IsChecked == true) && (cbFeatureXyz.IsChecked == true) && (cbFeatureWww.IsChecked == true))

cbAllFeatures.IsChecked = true;

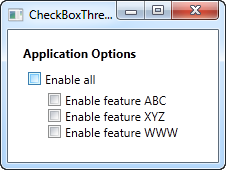
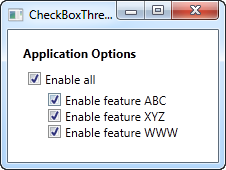
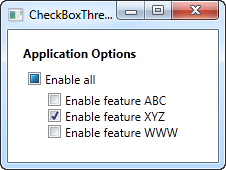
if((cbFeatureAbc.IsChecked == false) && (cbFeatureXyz.IsChecked == false) && (cbFeatureWww.IsChecked == false))

cbAllFeatures.IsChecked = false;

}

}

}



This example works from two different angles: If you check or uncheck the "Enable a

# The RadioButton control

The RadioButton control allows you to give your user a list of possible options, with only one of them selected at the same time. You can achieve the same effect, using less space, with the ComboBox control, but a set of radio buttons tend to give the user a better overview of the options they have.

<Window x:Class="WpfTutorialSamples.Basic\_controls.RadioButtonSample"

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

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

Title="RadioButtonSample" Height="150" Width="250">

<StackPanel Margin="10">

<Label FontWeight="Bold">Are you ready?</Label>

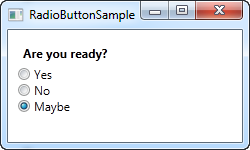
<RadioButton>Yes</RadioButton>

<RadioButton>No</RadioButton>

<RadioButton IsChecked="True">Maybe</RadioButton>

</StackPanel>

</Window>



All we do is add a Label with a question, and then three radio buttons, each with a possible answer. We define a default option by using the IsChecked property on the last RadioButton, which the user can change simply by clicking on one of the other radio buttons. **This is also the property you would want to use from Code-behind to check if a RadioButton is checked or not.**

## RadioButton groups

If you try running the example above, you will see that, as promised, only one RadioButton can be checked at the same time. But what if you want several groups of radio buttons, each with their own, individual selection? This is what the **GroupName** property comes into play, which allows you to specify which radio buttons belong together. Here's an example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.RadioButtonSample"

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

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

Title="RadioButtonSample" Height="230" Width="250">

<StackPanel Margin="10">

<Label FontWeight="Bold">Are you ready?</Label>

<RadioButton GroupName="ready">Yes</RadioButton>

<RadioButton GroupName="ready">No</RadioButton>

<RadioButton GroupName="ready" IsChecked="True">Maybe</RadioButton>

<Label FontWeight="Bold">Male or female?</Label>

<RadioButton GroupName="sex">Male</RadioButton>

<RadioButton GroupName="sex">Female</RadioButton>

<RadioButton GroupName="sex" IsChecked="True">Not sure</RadioButton>

</StackPanel>

</Window>



With the GroupName property set on each of the radio buttons, a selection can now be made for each of the two groups. Without this, only one selection for all six radio buttons would be possible.

## Custom content

The RadioButton inherits from the ContentControl class, which means that it can take custom content and display next to it. If you just specify a piece of text, like I did in the example above, WPF will put it inside a TextBlock control and display it, but this is just a shortcut to make things easier for you. You can use any type of control inside of it, as we'll see in the next example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.RadioButtonCustomContentSample"

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

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

Title="RadioButtonCustomContentSample" Height="150" Width="250">

<StackPanel Margin="10">

<Label FontWeight="Bold">Are you ready?</Label>

<RadioButton>

<WrapPanel>

<Image Source="/WpfTutorialSamples;component/Images/accept.png" Width="16" Height="16" Margin="0,0,5,0" />

<TextBlock Text="Yes" Foreground="Green" />

</WrapPanel>

</RadioButton>

<RadioButton Margin="0,5">

<WrapPanel>

<Image Source="/WpfTutorialSamples;component/Images/cancel.png" Width="16" Height="16" Margin="0,0,5,0" />

<TextBlock Text="No" Foreground="Red" />

</WrapPanel>

</RadioButton>

<RadioButton IsChecked="True">

<WrapPanel>

<Image Source="/WpfTutorialSamples;component/Images/question.png" Width="16" Height="16" Margin="0,0,5,0" />

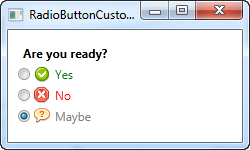
<TextBlock Text="Maybe" Foreground="Gray" />

</WrapPanel>

</RadioButton>

</StackPanel>

</Window>



Markup-wise, this example gets a bit heavy, but the concept is pretty simple. For each RadioButton, we have a WrapPanel with an image and a piece of text inside of it. Since we now take control of the text using a TextBlock control, this also allows us to format the text in any way we want to. For this example, I have changed the text color to match the choice. An Image control (read more about those later) is used to display an image for each choice.

Notice how you can click anywhere on the RadioButton, even on the image or the text, to toggle it on, because we have specified it as content of the RadioButton. If you had placed it as a separate panel, next to the RadioButton, the user would have to click directly on the round circle of the RadioButton to activate it, which is less practical.

# The PasswordBox control

For editing regular text in WPF we have the TextBox, but what about editing passwords? The functionality is very much the same, but we want WPF to display something else than the actual characters when typing in a password, to shield it from nosy people looking over your shoulder. For this purpose, WPF has the **PasswordBox** control, which is just as easy to use as the TextBox. Allow me to illustrate with an example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.PasswordBoxSample"

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

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

Title="PasswordBoxSample" Height="160" Width="300">

<StackPanel Margin="10">

<Label>Text:</Label>

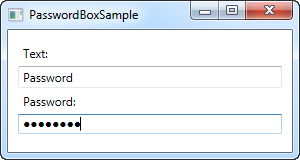
<TextBox />

<Label>Password:</Label>

<PasswordBox />

</StackPanel>

</Window>



In the screenshot, I have entered the exact same text into the two text boxes, but in the password version, the characters are replaced with dots. You can actually control which character is used instead of the real characters, using the **PasswordChar** property:

<PasswordBox PasswordChar="X" />

In this case, the character X will be used instead of the dots. In case you need to control the length of the password, there's a **MaxLength** property for you:

<PasswordBox MaxLength="6" />

I have used both properties in this updated example:

<Window x:Class="WpfTutorialSamples.Basic\_controls.PasswordBoxSample"

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

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

Title="PasswordBoxSample" Height="160" Width="300">

<StackPanel Margin="10">

<Label>Text:</Label>

<TextBox />

<Label>Password:</Label>

<PasswordBox MaxLength="6" PasswordChar="X" />

</StackPanel>

</Window>



Notice how the characters are now X's instead, and that I was only allowed to enter 6 characters in the box.

## PasswordBox and binding

When you need to obtain the password from the PasswordBox, you can use the **Password** property from Code-behind. However, for security reasons, the Password property is not implemented as a dependency property, which means that you can't bind to it.

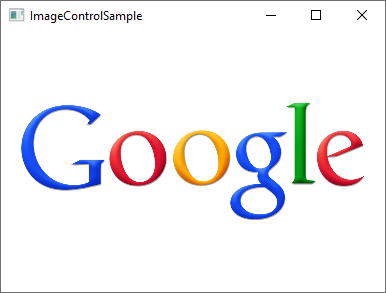
This may or may not be important to you - as already stated, you can still read the password from Code-behind, but for MVVM implementations or if you just love data bindings, a workaround has been developed. You can read much more about it here: <http://blog.functionalfun.net/2008/06/wpf-passwordbox-and-data-binding.html>

# The Image control

The WPF **Image** control will allow you to display images inside your applications. It's a very versatile control, with many useful options and methods, as you will learn in this article. But first, let's see the most basic example of including an image inside a Window:

<Image Source="https://upload.wikimedia.org/wikipedia/commons/3/30/Googlelogo.png" />

The result will look like this:



The **Source** property, which we used in this example to specify the image that should be displayed, is probably the most important property of this control, so let's dig into that subject to begin with.

## The Source property

As you can see from our first example, the **Source** property makes it easy to specify which image should be displayed inside the Image control - in this specific example, we used a remote image, which the Image control will just automatically fetch and display as soon as it becomes visible. That's a fine example of how versatile the Image control is, but in a lot of situations, you likely want to bundle the images with your application, instead of loading it from a remote source. This can be accomplished just as easily!

As you probably know, you can add resource files to your project - they can exist inside your current Visual Studio project and be seen in the Solution Explorer just like any other WPF-related file (Windows, User Controls etc.). A relevant example of a resource file is an image, which you can simply copy into a relevant folder of your project, to have it included. It will then be compiled into your application (unless you specifically ask VS not to do that) and can then be accessed using the URL format for resources. So, if you have an image called "google.png" inside a folder called "Images", the syntax could look like this:

<Image Source="/WpfTutorialSamples;component/Images/google.png" />

These URI's, often referred to as "**Pack URI's**", are a heavy topic with a lot more details, but for now, just notice that it's essentially made up of two parts:

* The first part (/WpfTutorialSamples;component), where the assembly name (**WpfTutorialSamples** in my application) is combined with the word "component"
* The second part, where the relative path of the resource is specified: /Images/google.png

Using this syntax, you can easily reference resources included in your application. To simplify things, **the WPF framework will also accept a simple, relative URL** - this will suffice in most cases, unless you're doing something more complicated in your application, in regards to resources. Using a simple relative URL, it would look like this:

<Image Source="/Images/google.png" />

### Loading images dynamically (Code-behind)

Specifying the Image Source directly in your XAML will work out for a lot of cases, but sometimes you need to load an image dynamically, e.g. based on a user choice. This is possible to do from Code-behind. Here's how you can load an image found on the user's computer, based on their selection from an OpenFileDialog:

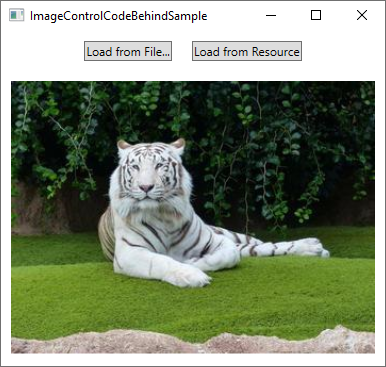
private void BtnLoadFromFile\_Click(object sender, RoutedEventArgs e)  
{  
    OpenFileDialog openFileDialog = new OpenFileDialog();  
    if(openFileDialog.ShowDialog() == true)  
    {  
 Uri fileUri = new Uri(openFileDialog.FileName);  
 imgDynamic.Source = new BitmapImage(fileUri);  
    }  
}

Notice how I create a **BitmapImage** instance, which I pass a **Uri** object to, based on the selected path from the dialog. We can use the exact same technique to load an image included in the application as a resource:

private void BtnLoadFromResource\_Click(object sender, RoutedEventArgs e)  
{  
    Uri resourceUri = new Uri("/Images/white\_bengal\_tiger.jpg", UriKind.Relative);  
    imgDynamic.Source = new BitmapImage(resourceUri);      
}

We use the same relative path as we used in one of the previous examples - just be sure to pass in the **UriKind.Relative** value when you create the **Uri** instance, so it knows that the path supplied is not an absolute path. Here's the XAML source, as well as a screenshot, of our Code-behind sample:

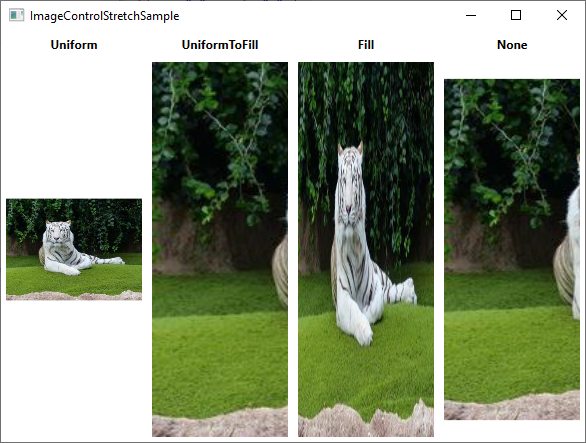
<Window x:Class="WpfTutorialSamples.Basic\_controls.ImageControlCodeBehindSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 xmlns:local="clr-namespace:WpfTutorialSamples.Basic\_controls"  
 mc:Ignorable="d"  
 Title="ImageControlCodeBehindSample" Height="300" Width="400">  
    <StackPanel>  
 <WrapPanel Margin="10" HorizontalAlignment="Center">  
     <Button Name="btnLoadFromFile" Margin="0,0,20,0" Click="BtnLoadFromFile\_Click">Load from File...</Button>  
     <Button Name="btnLoadFromResource" Click="BtnLoadFromResource\_Click">Load from Resource</Button>  
 </WrapPanel>  
 <Image Name="imgDynamic" Margin="10"  />  
    </StackPanel>  
</Window>



## The Stretch property

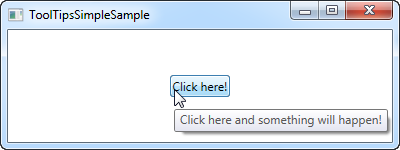
After the Source property, which is important for obvious reasons, I think the second most interesting property of the Image control might be the **Stretch** property. It controls what happens when the dimensions of the image loaded doesn't completely match the dimensions of the **Image** control. This will happen all the time, since the size of your Windows can be controlled by the user and unless your layout is very static, this means that the size of the Image control(s) will also change.

As you can see from this next example, the Stretch property can make quite a bit of difference in how an image is displayed:



<Window x:Class="WpfTutorialSamples.Basic\_controls.ImageControlStretchSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 xmlns:local="clr-namespace:WpfTutorialSamples.Basic\_controls"  
 mc:Ignorable="d"  
 Title="ImageControlStretchSample" Height="450" Width="600">  
    <Grid>  
 <Grid.ColumnDefinitions>  
     <ColumnDefinition Width="\*" />  
     <ColumnDefinition Width="\*" />  
     <ColumnDefinition Width="\*" />  
     <ColumnDefinition Width="\*" />  
 </Grid.ColumnDefinitions>  
 <Grid.RowDefinitions>  
     <RowDefinition Height="Auto" />  
     <RowDefinition Height="\*" />  
 </Grid.RowDefinitions>  
 <Label Grid.Column="0" HorizontalAlignment="Center" FontWeight="Bold">Uniform</Label>  
 <Label Grid.Column="1" HorizontalAlignment="Center" FontWeight="Bold">UniformToFill</Label>  
 <Label Grid.Column="2" HorizontalAlignment="Center" FontWeight="Bold">Fill</Label>  
 <Label Grid.Column="3" HorizontalAlignment="Center" FontWeight="Bold">None</Label>  
 <Image Source="/Images/white\_bengal\_tiger.jpg" Stretch="Uniform" Grid.Column="0" Grid.Row="1" Margin="5" />  
 <Image Source="/Images/white\_bengal\_tiger.jpg" Stretch="UniformToFill" Grid.Column="1" Grid.Row="1" Margin="5" />  
 <Image Source="/Images/white\_bengal\_tiger.jpg" Stretch="Fill" Grid.Column="2" Grid.Row="1" Margin="5" />  
 <Image Source="/Images/white\_bengal\_tiger.jpg" Stretch="None" Grid.Column="3" Grid.Row="1" Margin="5" />  
    </Grid>  
</Window>

It can be a bit hard to tell, but all four Image controls display the same image, but with different values for the Stretch property. Here's how the various modes work:

* **Uniform:** This is the default mode. The image will be automatically scaled so that it fits within the Image area. The [Aspect ratio](https://en.wikipedia.org/wiki/Aspect_ratio_(image)) of the image will be preserved.
* **UniformToFill:** The image will be scaled so that it completely fills the Image area. The Aspect ratio of the image will be preserved.
* **Fill:** The image will be scaled to fit the area of the Image control. Aspect ratio might NOT be preserved, because the height and width of the image are scaled independently.
* **None:** If the image is smaller than the Image control, nothing is done. If it's bigger than the Image control, the image will simply be cropped to fit into the Image control, meaning that only part of it will be visible.
* Control ToolTips
* Tooltips, infotips or hints - various names, but the concept remains the same: The ability to get extra information about a specific control or link by hovering the mouse over it. WPF obviously supports this concept as well, and by using the **ToolTip** property found on the **FrameworkElement** class, which almost any WPF control inherits from.
* Specifying a tooltip for a control is very easy, as you will see in this first and very basic example:
* <Window x:Class="WpfTutorialSamples.Control\_concepts.ToolTipsSimpleSample"
* xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
* xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
* Title="ToolTipsSimpleSample" Height="150" Width="400">
* <Grid VerticalAlignment="Center" HorizontalAlignment="Center">
* <Button ToolTip="Click here and something will happen!">Click here!</Button>
* </Grid>
* </Window>
* 
* As you can see on the screenshots, this results in a floating box with the specified string, once the mouse hovers over the button. This is what most UI frameworks offers - the display of a text string and nothing more.
* However, in WPF, the **ToolTip** property is actually not a string type, but instead an object type, meaning that we can put whatever we want in there. This opens up for some pretty cool possibilities, where we can provide the user with much richer and more helpful tooltips. For instance, consider this example and compare it to the first one:
* <Window x:Class="WpfTutorialSamples.Control\_concepts.ToolTipsAdvancedSample"
* xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
* xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
* Title="ToolTipsAdvancedSample" Height="200" Width="400" UseLayoutRounding="True">
* <DockPanel>
* <ToolBar DockPanel.Dock="Top">
* <Button ToolTip="Create a new file">
* <Button.Content>
* <Image Source="/WpfTutorialSamples;component/Images/page\_white.png" Width="16" Height="16" />
* </Button.Content>
* </Button>
* <Button>
* <Button.Content>
* <Image Source="/WpfTutorialSamples;component/Images/folder.png" Width="16" Height="16" />
* </Button.Content>
* <Button.ToolTip>
* <StackPanel>
* <TextBlock FontWeight="Bold" FontSize="14" Margin="0,0,0,5">Open file</TextBlock>
* <TextBlock>
* Search your computer or local network
* <LineBreak />
* for a file and open it for editing.
* </TextBlock>
* <Border BorderBrush="Silver" BorderThickness="0,1,0,0" Margin="0,8" />
* <WrapPanel>
* <Image Source="/WpfTutorialSamples;component/Images/help.png" Margin="0,0,5,0" />
* <TextBlock FontStyle="Italic">Press F1 for more help</TextBlock>
* </WrapPanel>
* </StackPanel>
* </Button.ToolTip>
* </Button>
* </ToolBar>
* <TextBox>
* Editor area...
* </TextBox>
* </DockPanel>
* </Window>
* 
* Notice how this example uses a simple string tooltip for the first button and then a much more advanced one for the second button. In the advanced case, we use a panel as the root control and then we're free to add controls to that as we please. The result is pretty cool, with a header, a description text and a hint that you can press F1 for more help, including a help icon.

## Advanced options

* The ToolTipService class has a bunch of interesting properties that will affect the behavior of your tooltips. You set them directly on the control that has the tooltip, for instance like here, where we extend the time a tooltip is shown using the **ShowDuration** property (we set it to 5.000 milliseconds or 5 seconds):
* <Button ToolTip="Create a new file" ToolTipService.ShowDuration="5000" Content="Open" />
* You can also control whether or not the popup should have a shadow, using the **HasDropShadow** property, or whether tooltips should be displayed for disabled controls as well, using the **ShowOnDisabled** property. There are several other interesting properties, so for a complete list, please consult the

# WPF text rendering

In this article, we'll be discussing why text is sometimes rendered more blurry with WPF, how this was later fixed and how you can control text rendering yourself.

As already mentioned in this tutorial, WPF does a lot more things on its own when compared to other UI frameworks like WinForms, which will use the Windows API for many, many things. This is also clear when it comes to the rendering of text - WinForms uses the GDI API from Windows, while WPF has its own text rendering implementation, to better support animations as well as the device independent nature of WPF.

Unfortunately, this led to text being rendered a bit blurry, especially in small font sizes. This was a rather big problem for WPF programmers for some time, but luckily, Microsoft made a lot of improvements in the WPF text rendering engine in .NET framework version 4.0. This means that if you're using this version or higher, your text should be almost as good as pixel perfect.

## Controlling text rendering

With .NET framework 4.0, Microsoft also decided to give more control of text rendering to the programmer, by introducing the **TextOptions** class with the **TextFormattingMode** and **TextRenderingMode** properties. This allows you to specifically decide how text should be formatted and rendered on a control level. This is probably best illustrated with an example, so have a look at the code and the screenshots below to see how you can affect text rendering with these properties.

### TextFormattingMode

Using the TextFormattingMode property, you get to decide which algorithm should be used when formatting the text. You can choose between **Ideal** (the default value) and **Display**. You would normally want to leave this property untouched, since the Ideal setting will be best for most situations, but in cases where you need to render very small text, the Display setting can sometimes yield a better result. Here's an example where you can see the difference (although it's very subtle):

<Window x:Class="WpfTutorialSamples.Control\_concepts.TextFormattingModeSample"

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

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

Title="TextFormattingModeSample" Height="200" Width="400">

<StackPanel Margin="10">

<Label TextOptions.TextFormattingMode="Ideal" FontSize="9">TextFormattingMode.Ideal, small text</Label>

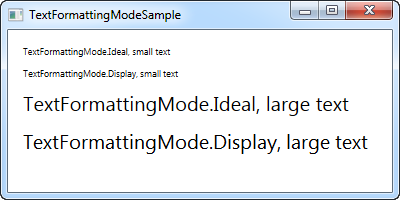
<Label TextOptions.TextFormattingMode="Display" FontSize="9">TextFormattingMode.Display, small text</Label>

<Label TextOptions.TextFormattingMode="Ideal" FontSize="20">TextFormattingMode.Ideal, large text</Label>

<Label TextOptions.TextFormattingMode="Display" FontSize="20">TextFormattingMode.Display, large text</Label>

</StackPanel>

</Window>



### TextRenderingMode

The **TextRenderingMode** property gives you control of which antialiasing algorithm is used when rendering text. It has the biggest effect in combination with the **Display** setting for the **TextFormattingMode** property, which we'll use in this example to illustrate the differences:

<Window x:Class="WpfTutorialSamples.Control\_concepts.TextRenderingModeSample"

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

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

Title="TextRenderingModeSample" Height="300" Width="400">

<StackPanel Margin="10" TextOptions.TextFormattingMode="Display">

<Label TextOptions.TextRenderingMode="Auto" FontSize="9">TextRenderingMode.Auto, small text</Label>

<Label TextOptions.TextRenderingMode="Aliased" FontSize="9">TextRenderingMode.Aliased, small text</Label>

<Label TextOptions.TextRenderingMode="ClearType" FontSize="9">TextRenderingMode.ClearType, small text</Label>

<Label TextOptions.TextRenderingMode="Grayscale" FontSize="9">TextRenderingMode.Grayscale, small text</Label>

<Label TextOptions.TextRenderingMode="Auto" FontSize="18">TextRenderingMode.Auto, large text</Label>

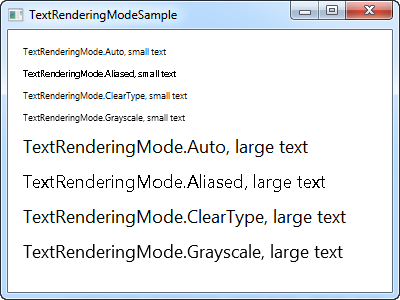
<Label TextOptions.TextRenderingMode="Aliased" FontSize="18">TextRenderingMode.Aliased, large text</Label>

<Label TextOptions.TextRenderingMode="ClearType" FontSize="18">TextRenderingMode.ClearType, large text</Label>

<Label TextOptions.TextRenderingMode="Grayscale" FontSize="18">TextRenderingMode.Grayscale, large text</Label>

</StackPanel>

</Window>

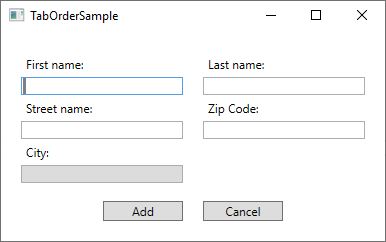


As you can see, the resulting text differs quite a bit in how it looks and once again, you should mainly change this in special circumstances.

# Tab Order

If you have worked with a computer long enough to want to learn programming, you probably also know that you can use the Tab key on the keyboard to navigate through a window/dialog. This allows you to keep your hands on the keyboard when filling out a form or something similar, instead of having to use the mouse to select the next field/control.

WPF supports this behavior straight out of the box, and even better: It will automatically establish the order used when moving from one field to another, so in general, you don't have to worry about this at all. However, sometimes the design of your Window/dialog cause WPF to use a tab order that you might not agree with, for various reasons. Also, you may decide that certain controls should not be a part of the tabbing order. Allow me to illustrate this with an example:



This dialog consists of a Grid, split in the middle, with StackPanel's on each side, containing labels and textboxes. The default tab order behavior is to start with the first control of the Window and then tab through each of the child controls found within it, before moving to the next control. Since the dialog consists of vertically oriented StackPanels, that would mean that we would start in the First name field and then move to the Street name field and then the City field, before moving to StackPanel number two, containing the fields for Last name and Zip code. When tabbing out of the second StackPanel, the two buttons would finally be reached.

However, for this dialog, that's not the behavior I want. Instead I want to tab from First name to Last name (so basically moving horizontally instead of vertically), and on top of that, I don't want to enter the City field when tabbing through the form, because that will be automatically filled based on the Zip code in this imaginary dialog and has therefore been made readonly. To accomplish all of this, I will use two properties: **TabIndex** and **IsTabStop**. TabIndex is used to define the order, while the IsTabStop property will force WPF to skip a control when tabbing through the Window. Here's the markup used to create the dialog:

<Window x:Class="WpfTutorialSamples.Control\_concepts.TabOrderSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 xmlns:local="clr-namespace:WpfTutorialSamples.Control\_concepts"  
 mc:Ignorable="d"  
 Title="TabOrderSample" Height="250" Width="400">  
    <Grid Margin="20">  
 <Grid.ColumnDefinitions>  
     <ColumnDefinition Width="\*" />  
     <ColumnDefinition Width="20" />  
     <ColumnDefinition Width="\*" />  
 </Grid.ColumnDefinitions>  
 <Grid.RowDefinitions>  
     <RowDefinition Height="\*" />  
     <RowDefinition Height="Auto" />  
 </Grid.RowDefinitions>  
 <StackPanel>  
     <Label>First name:</Label>  
     <TextBox TabIndex="0" />  
     <Label>Street name:</Label>  
     <TextBox TabIndex="2" />  
     <Label>City:</Label>  
     <TextBox TabIndex="5" IsReadOnly="True" IsTabStop="False" Background="Gainsboro" />  
 </StackPanel>  
 <StackPanel Grid.Column="2">  
     <Label>Last name:</Label>  
     <TextBox TabIndex="1" />  
     <Label>Zip Code:</Label>  
     <TextBox TabIndex="4" />  
 </StackPanel>  
 <Button Grid.Row="1" HorizontalAlignment="Right" Width="80">Add</Button>  
 <Button Grid.Row="1" Grid.Column="2" HorizontalAlignment="Left" Width="80">Cancel</Button>  
    </Grid>  
</Window>

Notice how I simply give each relevant control a number in the **TabIndex** property, and then use the **IsTabStop** for the TextBox used for the City - it's that simple to control the tab order in a dialog!

# Access Keys

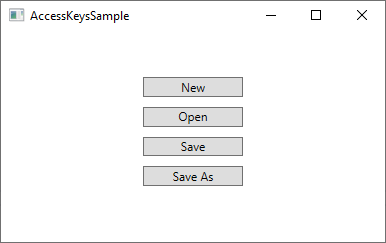
The concept of **Access Keys**, sometimes referred to as Accelerator Keys or Keyboard Accelerators, allows you to reach a specific control inside a window by holding down the Alt key and then pressing another key on the keyboard. This enhances the usability of your windows, because it allows the user to use their keyboard to navigate the window, instead of having to use the mouse.

## Defining Access Keys

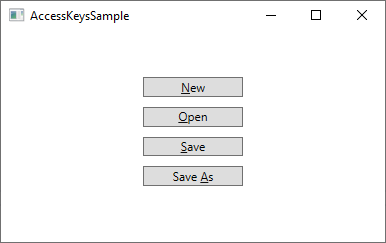
Defining access keys for your WPF control is very easy, but the method might surprise you a bit. Normally, there would be a property for this, but not for Access Keys. Instead, you define the Access Key by prefixing the letter with an underscore in the Text/Content property of the control. For instance, like this:

<Button Content="\_New"></Button>

Notice the underscore (\_) just before the N character - this will turn the N key into the designated Access Key for this Button control. By default, the look of your control(s) doesn't change, as you can see from this example where I have defined Access Keys for all the buttons:

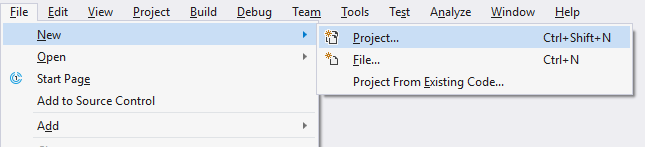


However, as soon as you press the **Alt** key on your Keyboard, the available Access Keys are highlighted by underlining them:



While holding the **Alt** key down, you can now press one of the Access Keys (e.g. N, O or S) to activate the specific button. It will react as if it was clicked with the mouse.

Access Keys are fine for single elements in a dialog/window, but they are even more useful in the traditional Windows Menus, where you will usually need to click your way through a hierarchy of menu items before reaching the one you need. Here's an example from Visual Studio:



In this case, instead of having to navigate through the menu with several mouse moves and clicks when I want to start a new Project, I can hold down the **Alt** key and then press **F** (for File), then **N** (for New) and then **P** (for Project). Sure, this could also have been accomplished with the regular keyboard shortcut (Ctrl+Shift+N), but that shortcut is not visible until you reach the last level of the menu hierarchy, so unless you have it memorized already, it might be easier to use the Access Keys, since they are visually highlighted as soon as you press the **Alt** key.

### Which character(s) should be used as Access Keys?

You might be tempted to just use any of the characters found in the control text/content, but there are actually guidelines for picking the right character. The most important rule is of course to pick a character not used by another control already, but in addition to that, you should use the following guidelines:

* Use the **first character** of the **first word**
* If that's not possible, use the first character of the second or third word (e.g. the **A** in Save As)
* If that's not possible, use the second character of the first word (e.g. **P** in Open)
* If that's not possible, use the second character of the second or third word (e.g. the **l** in Save All)
* In general, you may want to avoid narrow characters like i and l, and go for the wider characters like m, s, w etc.

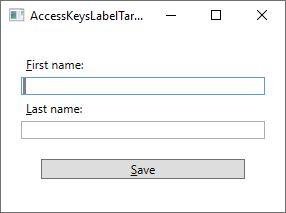
## Tying together two controls

In the examples we have seen so far, we have been able to define the Access Key directly on the control we want to reach. But there's at least one example where this isn't directly possible: When you have an input control, e.g. a **TextBox**, the text that indicate its purpose doesn't exist within the actual TextBox control. Instead, you would usually use a second control to indicate, with text, the purpose of the TextBox control. This would usually be a **Label** control.

So, in this example, the Label control would then hold the descriptive text, and therefore also the Access Key, but the control you want to give attention to would be the TextBox control. No problem - we can use the Target property of the Label to tie it together with the TextBox (or any other control), like this:

<StackPanel Margin="20">  
    <Label Content="\_First name:" Target="{Binding ElementName=txtFirstName}" />  
    <TextBox Name="txtFirstName" />  
    <Label Content="\_Last name:" Target="{Binding ElementName=txtLastName}" />  
    <TextBox Name="txtLastName" />  
    <Button Content="\_Save" Margin="20"></Button>  
</StackPanel>

Notice how the Access Key is specified for the Label controls and then tied to the relevant **TextBox** control using the **Target** property, where we use an **ElementName** based **Binding** to do the actual work. Now we can access the two TextBox controls using Alt+F and Alt+L, and the Button with Alt+S. Here's how it looks:



# Introduction to WPF styles

If you come from the world of developing for the web, using [HTML](http://www.html5-tutorials.org/introduction-to-html/what-is-html/) and [CSS](http://www.css3-tutorial.net/introduction/what-is-css/), you'll quickly realize that XAML is much like HTML: Using tags, you define a structural layout of your application. You can even make your elements look a certain way, using inline properties like Foreground, FontSize and so on, just like you can locally style your HTML tags.

But what happens when you want to use the exact same font size and color on three different TextBlock controls? You can copy/paste the desired properties to each of them, but what happens when three controls becomes 50 controls, spread out over several windows? And what happens when you realize that the font size should be 14 instead of 12?

## Basic style example

We'll talk much more about all the details, but for this introduction chapter, I want to show you a very basic example on how to use styling:

<Window x:Class="WpfTutorialSamples.Styles.SimpleStyleSample"

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

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

Title="SimpleStyleSample" Height="200" Width="250">

<StackPanel Margin="10">

<StackPanel.Resources>

<Style TargetType="TextBlock">

<Setter Property="Foreground" Value="Gray" />

<Setter Property="FontSize" Value="24" />

</Style>

</StackPanel.Resources>

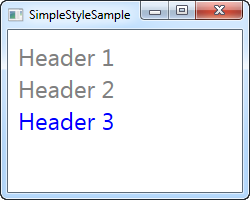
<TextBlock>Header 1</TextBlock>

<TextBlock>Header 2</TextBlock>

<TextBlock Foreground="Blue">Header 3</TextBlock>

</StackPanel>

</Window>



For the resources of my StackPanel, I define a **Style**. I use the TargetType property to tell WPF that this style should be applied towards ALL TextBlock controls within the scope (the StackPanel), and then I add two Setter elements to the style. The Setter elements are used to set specific properties for the target controls, in this case **Foreground** and **FontSize** properties. The **Property** property tells WPF which property we want to target, and the **Value** property defines the desired value.

Notice that the last TextBlock is blue instead of gray. I did that to show you that while a control might get styling from a designated style, you are completely free to override this locally on the control - values defined directly on the control will always take precedence over style values.

# Using WPF styles

In the previous chapter, where we introduced the concept of styles, we used a very basic example of a locally defined style, which targeted a specific type of controls - the TextBlock. However, styles can be defined in several different scopes, depending on where and how you want to use them, and you can even restrict styles to only be used on controls where you explicitly want it. In this chapter, I'll show you all the different ways in which a style can be defined.

## Local control specific style

You can actually define a style directly on a control, like this:

<Window x:Class="WpfTutorialSamples.Styles.ControlSpecificStyleSample"

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

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

Title="ControlSpecificStyleSample" Height="100" Width="300">

<Grid Margin="10">

<TextBlock Text="Style test">

<TextBlock.Style>

<Style>

<Setter Property="TextBlock.FontSize" Value="36" />

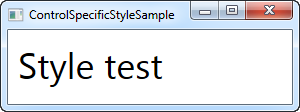
</Style>

</TextBlock.Style>

</TextBlock>

</Grid>

</Window>



In this example, the style only affects this specific TextBlock control, so why bother? Well, in this case, it makes no sense at all. I could have replaced all that extra markup with a single FontSize property on the TextBlock control, but as we'll see later, styles can do a bit more than just set properties, for instance, style triggers could make the above example useful in a real life application. However, most of the styles you'll define will likely be in a higher scope.

## Local child control style

Using the **Resources** section of a control, you can target child controls of this control (and child controls of those child controls and so on). This is basically what we did in the introduction example in the last chapter, which looked like this:

<Window x:Class="WpfTutorialSamples.Styles.SimpleStyleSample"

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

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

Title="SimpleStyleSample" Height="200" Width="250">

<StackPanel Margin="10">

<StackPanel.Resources>

<Style TargetType="TextBlock">

<Setter Property="Foreground" Value="Gray" />

<Setter Property="FontSize" Value="24" />

</Style>

</StackPanel.Resources>

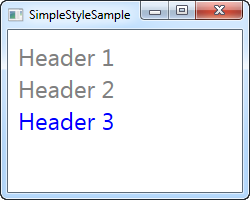
<TextBlock>Header 1</TextBlock>

<TextBlock>Header 2</TextBlock>

<TextBlock Foreground="Blue">Header 3</TextBlock>

</StackPanel>

</Window>



This is great for the more local styling needs. For instance, it would make perfect sense to do this in a dialog where you simply needed a set of controls to look the same, instead of setting the individual properties on each of them.

## Window-wide styles

The next step up in the scope hierarchy is to define the style(s) within the Window resources. This is done in exactly the same way as above for the StackPanel, but it's useful in those situations where you want a specific style to apply to all controls within a window (or a UserControl for that matter) and not just locally within a specific control. Here's a modified example:

<Window x:Class="WpfTutorialSamples.Styles.WindowWideStyleSample"

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

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

Title="WindowWideStyleSample" Height="200" Width="300">

<Window.Resources>

<Style TargetType="TextBlock">

<Setter Property="Foreground" Value="Gray" />

<Setter Property="FontSize" Value="24" />

</Style>

</Window.Resources>

<StackPanel Margin="10">

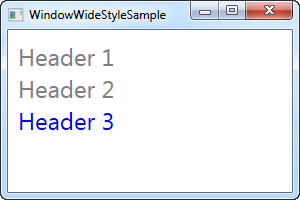
<TextBlock>Header 1</TextBlock>

<TextBlock>Header 2</TextBlock>

<TextBlock Foreground="Blue">Header 3</TextBlock>

</StackPanel>

</Window>



As you can see, the result is exactly the same, but it does mean that you could have controls placed everywhere within the window and the style would still apply.

## Application-wide styles

If you want your styles to be used all over the application, across different windows, you can define it for the entire application. This is done in the App.xaml file that Visual Studio has likely created for you, and it's done just like in the window-wide example:

**App.xaml**

<Application x:Class="WpfTutorialSamples.App"

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

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

StartupUri="Styles/WindowWideStyleSample.xaml">

<Application.Resources>

<Style TargetType="TextBlock">

<Setter Property="Foreground" Value="Gray" />

<Setter Property="FontSize" Value="24" />

</Style>

</Application.Resources>

</Application>

**Window**

<Window x:Class="WpfTutorialSamples.Styles.WindowWideStyleSample"

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

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

Title="ApplicationWideStyleSample" Height="200" Width="300">

<StackPanel Margin="10">

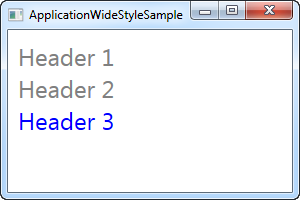
<TextBlock>Header 1</TextBlock>

<TextBlock>Header 2</TextBlock>

<TextBlock Foreground="Blue">Header 3</TextBlock>

</StackPanel>

</Window>



## Explicitly using styles

You have a lot of control over how and where to apply styling to your controls, from local styles and right up to the application-wide styles, that can help you get a consistent look all over your application, but so far, all of our styles have targeted a specific control type, and then ALL of these controls have used it. This doesn't have to be the case though.

By setting the **x:Key** property on a style, you are telling WPF that you only want to use this style when you explicitly reference it on a specific control. Let's try an example where this is the case:

<Window x:Class="WpfTutorialSamples.Styles.ExplicitStyleSample"

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

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

Title="ExplicitStyleSample" Height="150" Width="300">

<Window.Resources>

<Style x:Key="HeaderStyle" TargetType="TextBlock">

<Setter Property="Foreground" Value="Gray" />

<Setter Property="FontSize" Value="24" />

</Style>

</Window.Resources>

<StackPanel Margin="10">

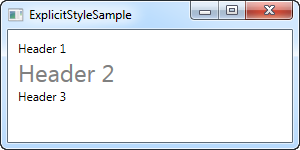
<TextBlock>Header 1</TextBlock>

<TextBlock Style="{StaticResource HeaderStyle}">Header 2</TextBlock>

<TextBlock>Header 3</TextBlock>

</StackPanel>

</Window>



Notice how even though the TargetType is set to TextBlock, and the style is defined for the entire window, only the TextBlock in the middle, where I explicitly reference the **HeaderStyle** style, uses the style. This allows you to define styles that target a specific control type, but only use it in the places where you need it.

# Trigger, DataTrigger & EventTrigger

So far, we worked with styles by setting a static value for a specific property. However, using triggers, you can change the value of a given property, once a certain condition changes. Triggers come in multiple flavors: Property triggers, event triggers and data triggers. They allow you to do stuff that would normally be done in code-behind completely in markup instead, which is all a part of the ongoing process of separating style and code.

## Property trigger

The most common trigger is the property trigger, which in markup is simply defined with a <Trigger> element. It watches a specific property on the owner control and when that property has a value that matches the specified value, properties can change. In theory this might sound a bit complicated, but it's actually quite simple once we turn theory into an example:

<Window x:Class="WpfTutorialSamples.Styles.StyleTriggersSample"

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

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

Title="StyleTriggersSample" Height="100" Width="300">

<Grid>

<TextBlock Text="Hello, styled world!" FontSize="28" HorizontalAlignment="Center" VerticalAlignment="Center">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Setter Property="Foreground" Value="Blue"></Setter>

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="Red" />

<Setter Property="TextDecorations" Value="Underline" />

</Trigger>

</Style.Triggers>

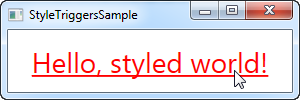
</Style>

</TextBlock.Style>

</TextBlock>

</Grid>

</Window>



In this style, we set the **Foreground** property to blue, to make it look like a hyperlink. We then add a trigger, which listens to the**IsMouseOver** property - once this property changes to **True**, we apply two setters: We change the **Foreground** to red and then we make it underlined. This is a great example on how easy it is to use triggers to apply design changes, completely without any code-behind code.

We define a local style for this specific TextBlock, but as shown in the previous articles, the style could have been globally defined as well, if we wanted it to apply to all TextBlock controls in the application.

## Data triggers

Data triggers, represented by the <DataTrigger> element, are used for properties that are not necessarily dependency properties. They work by creating a binding to a regular property, which is then monitored for changes. This also opens up for binding your trigger to a property on a different control. For instance, consider the following example:

<Window x:Class="WpfTutorialSamples.Styles.StyleDataTriggerSample"

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

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

Title="StyleDataTriggerSample" Height="200" Width="200">

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">

<CheckBox Name="cbSample" Content="Hello, world?" />

<TextBlock HorizontalAlignment="Center" Margin="0,20,0,0" FontSize="48">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Setter Property="Text" Value="No" />

<Setter Property="Foreground" Value="Red" />

<Style.Triggers>

<DataTrigger Binding="{Binding ElementName=cbSample, Path=IsChecked}" Value="True">

<Setter Property="Text" Value="Yes!" />

<Setter Property="Foreground" Value="Green" />

</DataTrigger>

</Style.Triggers>

</Style>

</TextBlock.Style>

</TextBlock>

</StackPanel>

</Window>



In this example, we have a **CheckBox** and a **TextBlock**. Using a **DataTrigger**, we bind the TextBlock to the **IsChecked** property of the CheckBox. We then supply a default style, where the text is "No" and the foreground color is red, and then, using a DataTrigger, we supply a style for when the IsChecked property of the CheckBox is changed to True, in which case we make it green with a text saying "Yes!" (as seen on the screenshot).

## Event triggers

Event triggers, represented by the <EventTrigger> element, are mostly used to trigger an animation, in response to an event being called. We haven't discussed animations yet, but to demonstrate how an event trigger works, we'll use them anyway. Have a look on the chapter about animations for more details. Here's the example:

<Window x:Class="WpfTutorialSamples.Styles.StyleEventTriggerSample"

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

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

Title="StyleEventTriggerSample" Height="100" Width="300">

<Grid>

<TextBlock Name="lblStyled" Text="Hello, styled world!" FontSize="18" HorizontalAlignment="Center" VerticalAlignment="Center">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Style.Triggers>

<EventTrigger RoutedEvent="MouseEnter">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation Duration="0:0:0.300" Storyboard.TargetProperty="FontSize" To="28" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

<EventTrigger RoutedEvent="MouseLeave">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation Duration="0:0:0.800" Storyboard.TargetProperty="FontSize" To="18" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

</Style.Triggers>

</Style>

</TextBlock.Style>

</TextBlock>

</Grid>

</Window>



The markup might look a bit overwhelming, but if you run this sample and look at the result, you'll see that we've actually accomplished a pretty cool animation, going both ways, in ~20 lines of XAML. As you can see, I use an EventTrigger to subscribe to two events: **MouseEnter** and **MouseLeave**. When the mouse enters, I make a smooth and animated transition to a FontSize of 28 pixels in 300 milliseconds. When the mouse leaves, I change the FontSize back to 18 pixels but I do it a bit slower, just because it looks kind of cool.

# WPF MultiTrigger and MultiDataTrigger

In the previous chapter, we worked with triggers to get dynamic styles. So far they have all been based on a single property, but WPF also supports multi triggers, which can monitor two or more property conditions and only trigger once all of them are satisfied.

There are two types of multi triggers: The **MultiTrigger**, which just like the regular Trigger works on dependency properties, and then the **MultiDataTrigger**, which works by binding to any kind of property. Let's start with a quick example on how to use the MultiTrigger.

## MultiTrigger

<Window x:Class="WpfTutorialSamples.Styles.StyleMultiTriggerSample"

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

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

Title="StyleMultiTriggerSample" Height="100" Width="250">

<Grid>

<TextBox VerticalAlignment="Center" HorizontalAlignment="Center" Text="Hover and focus here" Width="150">

<TextBox.Style>

<Style TargetType="TextBox">

<Style.Triggers>

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="IsKeyboardFocused" Value="True" />

<Condition Property="IsMouseOver" Value="True" />

</MultiTrigger.Conditions>

<MultiTrigger.Setters>

<Setter Property="Background" Value="LightGreen" />

</MultiTrigger.Setters>

</MultiTrigger>

</Style.Triggers>

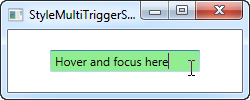
</Style>

</TextBox.Style>

</TextBox>

</Grid>

</Window>



In this example, we use a trigger to change the background color of the TextBox once it has keyboard focus AND the mouse cursor is over it, as seen on the screenshot. This trigger has two conditions, but we could easily have added more if needed. In the Setters section, we define the properties we wish to change when all the conditions are met - in this case, just the one (background color).

## MultiDataTrigger

Just like a regular DataTrigger, the MultiDataTrigger is cool because it uses bindings to monitor a property. This means that you can use all of the cool WPF binding techniques, including binding to the property of another control etc. Let me show you how easy it is:

<Window x:Class="WpfTutorialSamples.Styles.StyleMultiDataTriggerSample"

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

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

Title="StyleMultiDataTriggerSample" Height="150" Width="200">

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">

<CheckBox Name="cbSampleYes" Content="Yes" />

<CheckBox Name="cbSampleSure" Content="I'm sure" />

<TextBlock HorizontalAlignment="Center" Margin="0,20,0,0" FontSize="28">

<TextBlock.Style>

<Style TargetType="TextBlock">

<Setter Property="Text" Value="Unverified" />

<Setter Property="Foreground" Value="Red" />

<Style.Triggers>

<MultiDataTrigger>

<MultiDataTrigger.Conditions>

<Condition Binding="{Binding ElementName=cbSampleYes, Path=IsChecked}" Value="True" />

<Condition Binding="{Binding ElementName=cbSampleSure, Path=IsChecked}" Value="True" />

</MultiDataTrigger.Conditions>

<Setter Property="Text" Value="Verified" />

<Setter Property="Foreground" Value="Green" />

</MultiDataTrigger>

</Style.Triggers>

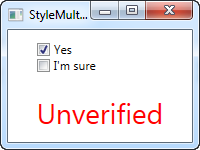
</Style>

</TextBlock.Style>

</TextBlock>

</StackPanel>

</Window>



In this example, I've re-created the example we used with the regular DataTrigger, but instead of binding to just one property, I bind to the same property (IsChecked) but on two different controls. This allows us to trigger the style only once both checkboxes are checked - if you remove a check from either one of them, the default style will be applied instead.

# Trigger animations

One of the things that became a LOT easier with WPF, compared to previous frameworks like WinForms, is animation. Triggers have direct support for using animations in response to the trigger being fired, instead of just switching between two static values.

For this, we use the **EnterActions** and **ExitActions** properties, which are present in all of the trigger types already discussed (except for the EventTrigger), both single and multiple. Here's an example:

<Window x:Class="WpfTutorialSamples.Styles.StyleTriggerEnterExitActions"

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

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

Title="StyleTriggerEnterExitActions" Height="200" Width="200" UseLayoutRounding="True">

<Grid>

<Border Background="LightGreen" Width="100" Height="100" BorderBrush="Green">

<Border.Style>

<Style TargetType="Border">

<Style.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Trigger.EnterActions>

<BeginStoryboard>

<Storyboard>

<ThicknessAnimation Duration="0:0:0.400" To="3" Storyboard.TargetProperty="BorderThickness" />

<DoubleAnimation Duration="0:0:0.300" To="125" Storyboard.TargetProperty="Height" />

<DoubleAnimation Duration="0:0:0.300" To="125" Storyboard.TargetProperty="Width" />

</Storyboard>

</BeginStoryboard>

</Trigger.EnterActions>

<Trigger.ExitActions>

<BeginStoryboard>

<Storyboard>

<ThicknessAnimation Duration="0:0:0.250" To="0" Storyboard.TargetProperty="BorderThickness" />

<DoubleAnimation Duration="0:0:0.150" To="100" Storyboard.TargetProperty="Height" />

<DoubleAnimation Duration="0:0:0.150" To="100" Storyboard.TargetProperty="Width" />

</Storyboard>

</BeginStoryboard>

</Trigger.ExitActions>

</Trigger>

</Style.Triggers>

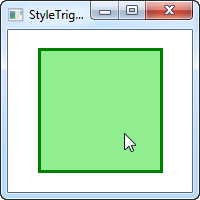
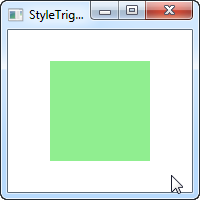
</Style>

</Border.Style>

</Border>

</Grid>

</Window>



In this example, we have a green square. It has a trigger that fires once the mouse is over, in which case it fires of several animations, all defined in the **EnterActions** part of the trigger. In there, we animate the thickness of the border from its default 0 to a thickness of 3, and then we animate the width and height from 100 to 125. This all happens simultaneously, because they are a part of the same **StoryBoard**, and even at slightly different speeds, since we have full control of how long each animation should run.

We use the ExitActions to reverse the changes we made, with animations that goes back to the default values. We run the reversing animations slightly faster, because we can and because it looks cool.

The two states are represented on the two screenshots, but to fully appreciate the effect, you should try running the example on your own machine, using the source code above.

**UserControls & CustomControls:**

# Creating & using a UserControl

User controls, in WPF represented by the User Control class, is the concept of grouping markup and code into a reusable container, so that the same interface, with the same functionality, can be used in several different places and even across several applications.

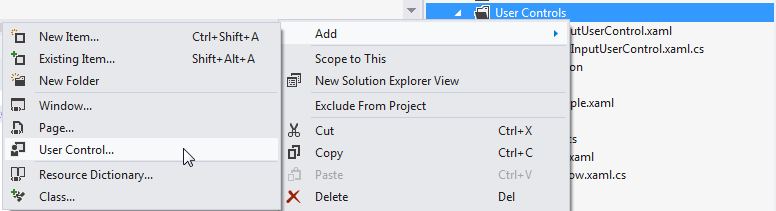
A user control acts much like a WPF Window - an area where you can place other controls, and then a Code-behind file where you can interact with these controls. The file that contains the user control also ends with .xaml, and the Code-behind ends with .xaml.cs - just like a Window. The starting markup looks a bit different though:

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

Nothing too strange though - a root UserControl element instead of the Window element, and then the DesignHeight and DesignWidth properties, which controls the size of the user control in design-time (in runtime, the size will be decided by the container that holds the user control). You will notice the same thing in Code-behind, where it simply inherits UserControl instead of Window.

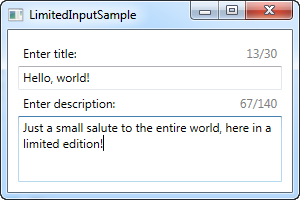
## Creating a User Control

Add a user control to your project just like you would add another Window, by right-clicking on the project or folder name where you want to add it, as illustrated on this screenshot (things might look a bit different, depending on the version of Visual Studio you're using):



For this article, we'll be creating a useful User control with the ability to limit the amount of text in a TextBox to a specific number of characters, while showing the user how many characters have been used and how many may be used in total. This is very simple to do, and used in a lot of web applications like Twitter. It would be easy to just add this functionality to your regular Window, but since it could be useful to do in several places in your application, it makes sense to wrap it in an easily reusable UserControl.

Before we dive into the code, let's have a look at the end result that we're going for:



Here's the code for the user control itself:

<UserControl x:Class="WpfTutorialSamples.User\_Controls.LimitedInputUserControl"  
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"   
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"   
      mc:Ignorable="d"   
      d:DesignHeight="300" d:DesignWidth="300">  
    <Grid>  
 <Grid.RowDefinitions>  
     <RowDefinition Height="Auto" />  
     <RowDefinition Height="\*" />  
 </Grid.RowDefinitions>        
 <Grid.ColumnDefinitions>  
     <ColumnDefinition Width="\*" />  
     <ColumnDefinition Width="Auto" />  
 </Grid.ColumnDefinitions>  
 <Label Content="{Binding Title}" />  
 <Label Grid.Column="1">  
     <StackPanel Orientation="Horizontal">  
 <TextBlock Text="{Binding ElementName=txtLimitedInput, Path=Text.Length}" />  
 <TextBlock Text="/" />  
 <TextBlock Text="{Binding MaxLength}" />  
     </StackPanel>  
 </Label>  
 <TextBox MaxLength="{Binding MaxLength}" Grid.Row="1" Grid.ColumnSpan="2" Name="txtLimitedInput" ScrollViewer.VerticalScrollBarVisibility="Auto" TextWrapping="Wrap" />  
    </Grid>  
</UserControl>

using System;  
using System.Windows.Controls;  
  
namespace WpfTutorialSamples.User\_Controls  
{  
    public partial class LimitedInputUserControl : UserControl  
    {  
 public LimitedInputUserControl()  
 {  
     InitializeComponent();  
     this.DataContext = this;  
 }  
  
 public string Title { get; set; }  
  
 public int MaxLength { get; set; }  
    }  
}

The markup is pretty straight forward: A Grid, with two columns and two rows. The upper part of the Grid contains two labels, one showing the title and the other one showing the stats. Each of them use data binding for all of the information needed - the **Title** and **MaxLength** comes from the Code-behind properties, which we have defined in as regular properties on a regular class.

The current character count is obtained by binding to the Text.Length property directly on the TextBox control, which uses the lower part of the user control. The result can be seen on the screenshot above. Notice that because of all these bindings, we don't need any C# code to update the labels or set the MaxLength property on the TextBox - instead, we just bind directly to the properties.

## Consuming/using the User Control

With the above code in place, all we need is to consume (use) the User control within our Window. We'll do that by adding a reference to the namespace the UserControl lives in, in the top of the XAML code of your Window:

xmlns:uc="clr-namespace:WpfTutorialSamples.User\_Controls"

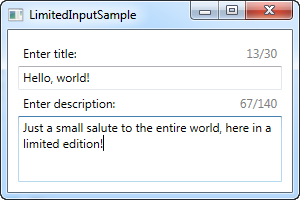
After that, we can use the uc prefix to add the control to our Window like it was any other WPF control:

<uc:LimitedInputUserControl Title="Enter title:" MaxLength="30" Height="50" />

Notice how we use the **Title** and **MaxLength** properties directly in the XAML. Here's the full code sample for our window:

<Window x:Class="WpfTutorialSamples.User\_Controls.LimitedInputSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:uc="clr-namespace:WpfTutorialSamples.User\_Controls"  
 Title="LimitedInputSample" Height="200" Width="300">  
    <Grid Margin="10">  
 <Grid.RowDefinitions>  
     <RowDefinition Height="Auto" />  
     <RowDefinition Height="\*" />  
 </Grid.RowDefinitions>  
   
 <uc:LimitedInputUserControl Title="Enter title:" MaxLength="30" Height="50" />  
 <uc:LimitedInputUserControl Title="Enter description:" MaxLength="140" Grid.Row="1" />  
   
    </Grid>  
</Window>

With that, we can reuse this entire piece of functionality in a single line of code, as illustrated in this example where we have the limited text input control two times. As already shown, the final result looks like this:



# The Border control

The Border control is a Decorator control that you may use to draw a border, a background, or even both, around another element. Since the WPF panels don't support drawing a border around its edges, the Border control can help you achieve just that, simply by surrounding e.g. a Panel with the Border control.

A simple example on using the Border as described above could look like this:

<Window x:Class="WpfTutorialSamples.Misc\_controls.BorderSample"

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

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

Title="BorderSample" Height="170" Width="200">

<Grid Margin="10">

<Border Background="GhostWhite" BorderBrush="Gainsboro" BorderThickness="1">

<StackPanel Margin="10">

<Button>Button 1</Button>

<Button Margin="0,10">Button 2</Button>

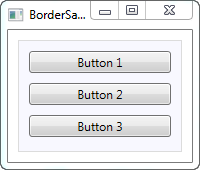
<Button>Button 3</Button>

</StackPanel>

</Border>

</Grid>

</Window>



The Border is completely lookless until you define either a background or a border brush and thickness, so that's what I've done here, using the **Background**, **BorderBrush** and **BorderThickness**properties.

## Border with round corners

One of the features I really appreciate about the Border is the fact that it's so easy to get round corners. Just look at this slightly modified example, where the corners are now rounded:

<Window x:Class="WpfTutorialSamples.Misc\_controls.BorderSample"

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

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

Title="BorderSample" Height="175" Width="200">

<Grid Margin="10">

<Border Background="GhostWhite" BorderBrush="Silver" BorderThickness="1" CornerRadius="8,8,3,3">

<StackPanel Margin="10">

<Button>Button 1</Button>

<Button Margin="0,10">Button 2</Button>

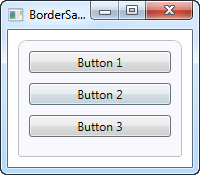
<Button>Button 3</Button>

</StackPanel>

</Border>

</Grid>

</Window>



All I've done is adding the **CornerRadius** property. It can be specified with a single value, which will be used for all four corners, or like I did in the example here, where I specify separate values for the top right and left followed by the bottom right and left.

## Border color/thickness

The above border is very discrete, but this can easily be changed by regulating the color and/or thickness. Because the BorderThickness property is of the **Thickness** type, you can even manipulate each of the border widths individually or by giving a value for the left and right and one for the top and bottom borders.

<Window x:Class="WpfTutorialSamples.Misc\_controls.BorderSample"

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

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

Title="BorderSample" Height="175" Width="200">

<Grid Margin="10">

<Border Background="GhostWhite" BorderBrush="DodgerBlue" BorderThickness="1,3,1,5">

<StackPanel Margin="10">

<Button>Button 1</Button>

<Button Margin="0,10">Button 2</Button>

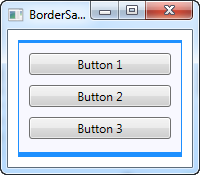
<Button>Button 3</Button>

</StackPanel>

</Border>

</Grid>

</Window>



## Border background

The Background property is of the type Brush, which opens up a lot of cool possibilities. As seen in the initial examples, it's very easy to just use a simple color as the background, but you can actually use gradients as well, and it's not even that hard to do:

<Window x:Class="WpfTutorialSamples.Misc\_controls.BorderSample"

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

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

Title="BorderSample" Height="175" Width="200">

<Grid Margin="10">

<Border BorderBrush="Navy" BorderThickness="1,3,1,5">

<Border.Background>

<LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">

<GradientStop Color="LightCyan" Offset="0.0" />

<GradientStop Color="LightBlue" Offset="0.5" />

<GradientStop Color="DarkTurquoise" Offset="1.0" />

</LinearGradientBrush>

</Border.Background>

<StackPanel Margin="10">

<Button>Button 1</Button>

<Button Margin="0,10">Button 2</Button>

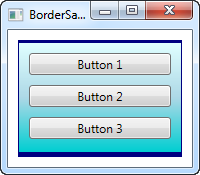
<Button>Button 3</Button>

</StackPanel>

</Border>

</Grid>

</Window>



In this case, I've specified a **LinearGradientBrush** to be used for the background of the Border and then a more fitting border color. The LinearGradientBrush might not have the most obvious syntax, so I will explain that in a later chapter, including other brush types, but for now, you can try my example and change the values to see the result.

# The Slider control

The Slider control allows you to pick a numeric value by dragging a thumb along a horizontal or vertical line. You see it in a lot of user interfaces, but it can still be a bit hard to recognize from the description alone, so here's a very basic example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.SliderSample"

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

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

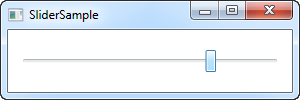
Title="SliderSample" Height="100" Width="300">

<StackPanel VerticalAlignment="Center" Margin="10">

<Slider Maximum="100" />

</StackPanel>

</Window>



This will allow the end-user to select a value between 0 and 100 by dragging the button (referred to as the thumb) along the line.

## Ticks

In the example, I have dragged the thumb beyond the middle, but it's obviously hard to see the exact value. One way to remedy this is to turn on ticks, which are small markers shown on the line to give a better indication on how far the thumb is. Here's an example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.SliderSample"

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

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

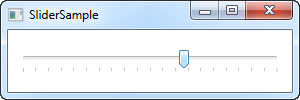
Title="SliderSample" Height="100" Width="300">

<StackPanel VerticalAlignment="Center" Margin="10">

<Slider Maximum="100" TickPlacement="BottomRight" TickFrequency="5" />

</StackPanel>

</Window>



I turn on the tick markers by giving the **TickPlacement** property another value than None, which is the default. In my example, I want the ticks placed below the line, but you can use **TopLeft** or even **Both** as possible values, to change this.

Also notice my use of the **TickFrequency** property. It defaults to 1, but in an example where the range of possible values goes from 0 to 100, this will result in 100 tick markers, which will have to be fitted into the limited space. In a case like this, it makes sense to raise the TickFrequency to something that will make it look less crowded.

## Snapping to ticks

If you have a look at the screenshot above, you will see that the thumb is between ticks. This makes sense, since there are five values between each tick, as specified by the TickFrequency property. Also, the value of the Slider control is in fact by default a double, meaning that the value can (and will likely) be a non-integer. We can change this by using the **IsSnapToTickEnabled** property, like in the below example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.SliderSnapToTickSample"

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

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

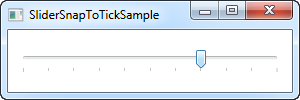
Title="SliderSnapToTickSample" Height="100" Width="300">

<StackPanel VerticalAlignment="Center" Margin="10">

<Slider Maximum="100" TickPlacement="BottomRight" TickFrequency="10" IsSnapToTickEnabled="True" />

</StackPanel>

</Window>



Notice that I've changed the TickFrequency to 10, and then enabled the IsSnapToTickEnabled property. This ensures that the thumb can only be placed directly on a tick value, so for this example, it can only be 0, 10, 20, 30, 40 and so on.

## Slider value

So far, we've just used the Slider illustratively, but of course, the actual purpose is to read its current value and use it for something. The Slider has a Value property for that, which you can of course read from Code-behind, or even bind to.

A common scenario in using the Slider is to combine it with a TextBox, which will allow the user to see the currently selected value, as well as changing it by entering a number instead of dragging the Slider thumb. Normally, you would have to subscribe to change events on both the Slider and the TextBox and then update accordingly, but a simple binding can do all of that for us:

<Window x:Class="WpfTutorialSamples.Misc\_controls.SliderBoundValueSample"

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

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

Title="SliderBoundValueSample" Height="100" Width="300">

<DockPanel VerticalAlignment="Center" Margin="10">

<TextBox Text="{Binding ElementName=slValue, Path=Value, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

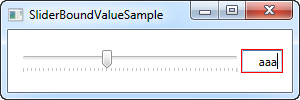
<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" IsSnapToTickEnabled="True" Name="slValue" />

</DockPanel>

</Window>



Now you can change the value by using either the Slider or by entering a value in the TextBox, and it will be immediately reflected in the other control. As an added bonus, we get simple validation as well, without any extra work, like if we try to enter a non-numeric value in the TextBox:



## Responding to changed values

Of course, while bindings are very cool for a lot of purposes, you still may want to respond to changes in the Slider value from your Code-behind. Fortunately for us, the Slider comes with a ValueChanged event which will help us with that. To illustrate this, I've created a more complex sample with three sliders, where we change the Red, Green and Blue (RGB) values of a color:

<Window x:Class="WpfTutorialSamples.Misc\_controls.SliderValueChangedSample"

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

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

Title="SliderValueChangedSample" Height="200" Width="300">

<StackPanel Margin="10" VerticalAlignment="Center">

<DockPanel VerticalAlignment="Center" Margin="10">

<Label DockPanel.Dock="Left" FontWeight="Bold">R:</Label>

<TextBox Text="{Binding ElementName=slColorR, Path=Value, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" IsSnapToTickEnabled="True" Name="slColorR" ValueChanged="ColorSlider\_ValueChanged" />

</DockPanel>

<DockPanel VerticalAlignment="Center" Margin="10">

<Label DockPanel.Dock="Left" FontWeight="Bold">G:</Label>

<TextBox Text="{Binding ElementName=slColorG, Path=Value, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" IsSnapToTickEnabled="True" Name="slColorG" ValueChanged="ColorSlider\_ValueChanged" />

</DockPanel>

<DockPanel VerticalAlignment="Center" Margin="10">

<Label DockPanel.Dock="Left" FontWeight="Bold">B:</Label>

<TextBox Text="{Binding ElementName=slColorB, Path=Value, UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right" TextAlignment="Right" Width="40" />

<Slider Maximum="255" TickPlacement="BottomRight" TickFrequency="5" IsSnapToTickEnabled="True" Name="slColorB" ValueChanged="ColorSlider\_ValueChanged" />

</DockPanel>

</StackPanel>

</Window>

using System;

using System.Windows;

using System.Windows.Media;

namespace WpfTutorialSamples.Misc\_controls

{

public partial class SliderValueChangedSample : Window

{

public SliderValueChangedSample()

{

InitializeComponent();

}

private void ColorSlider\_ValueChanged(object sender, RoutedPropertyChangedEventArgs<double> e)

{

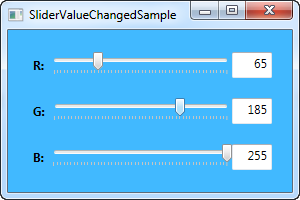
Color color = Color.FromRgb((byte)slColorR.Value, (byte)slColorG.Value, (byte)slColorB.Value);

this.Background = new SolidColorBrush(color);

}

}

}



In the XAML part of the code, we have three DockPanels, each with a Label, a Slider and a TextBox control. Just like before, the Text property of the TextBox controls have been bound to the Value of the Slider.

Each slider subscribes to the same **ValueChanged** event, in which we create a new Color instance, based on the currently selected values and then uses this color to create a new SolidColorBrush for the Background property of the Window.

All in all, this is a pretty good example of what the Slider control can be used for.

# The ProgressBar control

WPF comes with a handy control for displaying progress, called the **ProgressBar**. It works by setting a minimum and maximum value and then incrementing a value, which will give a visual indication on how far in the process you currently are. Here's a very basic example to demonstrate it with:

<Window x:Class="WpfTutorialSamples.Misc\_controls.ProgressBarSample"

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

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

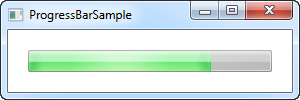
Title="ProgressBarSample" Height="100" Width="300">

<Grid Margin="20">

<ProgressBar Minimum="0" Maximum="100" Value="75" />

</Grid>

</Window>



In this case, I've used a pretty standard approach of showing progress as a percentage (between 0 and 100%), giving it an initial value of 75. Another approach is to use actual minimum and maximum values from a list of tasks you're performing. For instance, if you loop through a collected list of files while checking each of them, you can set the Minimum property to 0, the Maximum to the amount of files in your list, and then just increment as you loop through it.

The ProgressBar is, just like other standard WPF controls, rendered to match the visual style of the operating system. Here on Windows 7, it has a nice animated gradient, as seen on the screenshot.

## Showing progress while performing a lengthy task

The above example illustrates how simple it is to use a ProgressBar, but normally you would of course want to show the progress of some actual work and not just a static value.

In most situations you will use the ProgressBar to show progress for some heavy/lengthy task, and this this is where most new programmers run into a very common problem: If you do a piece of heavy work on the UI thread, while trying to simultaneously update e.g. a ProgressBar control, you will soon realize that you can't do both, at the same time, on the same thread. Or to be more clear, you can, but the ProgressBar won't actually show each update to the progress before the task is completed, which pretty much renders it useless.

To illustrate, you can try the following example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.ProgressBarTaskOnUiThread"

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

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

Title="ProgressBarTaskOnUiThread" Height="100" Width="300"

ContentRendered="Window\_ContentRendered">

<Grid Margin="20">

<ProgressBar Minimum="0" Maximum="100" Name="pbStatus" />

</Grid>

</Window>

using System;

using System.Threading;

using System.Windows;

namespace WpfTutorialSamples.Misc\_controls

{

public partial class ProgressBarTaskOnUiThread : Window

{

public ProgressBarTaskOnUiThread()

{

InitializeComponent();

}

private void Window\_ContentRendered(object sender, EventArgs e)

{

for(int i = 0; i < 100; i++)

{

pbStatus.Value++;

Thread.Sleep(100);

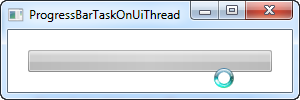
}

}

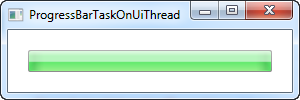
}

}

A very basic example, where, as soon as the window is ready, we do a loop from 0 to 100 and in each iteration, we increment the value of the ProgressBar. Any modern computer can do this faster than you can blink your eyes, so I've added a delay to each iteration of 100 milliseconds. Unfortunately, as I already described, nothing will happen. This is how it looks in the middle of the process:



Notice that the cursor indicates that something is happening, yet the ProgressBar still looks like it did at the start (empty). As soon as the loop, which represents our lengthy task, is done, the ProgressBar will look like this:



That really didn't help your users see the progress! Instead, we have to perform the task on a worker thread and then push updates to the UI thread, which will then be able to immediately process and visually show these updates. An excellent tool for handling this job is the BackgroundWorker class, which we talk much more about elsewhere in this tutorial. Here's the same example as above, but this time using a BackgroundWorker:

<Window x:Class="WpfTutorialSamples.Misc\_controls.ProgressBarTaskOnWorkerThread"

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

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

Title="ProgressBarTaskOnWorkerThread" Height="100" Width="300"

ContentRendered="Window\_ContentRendered">

<Grid Margin="20">

<ProgressBar Minimum="0" Maximum="100" Name="pbStatus" />

</Grid>

</Window>

using System;

using System.ComponentModel;

using System.Threading;

using System.Windows;

namespace WpfTutorialSamples.Misc\_controls

{

public partial class ProgressBarTaskOnWorkerThread : Window

{

public ProgressBarTaskOnWorkerThread()

{

InitializeComponent();

}

private void Window\_ContentRendered(object sender, EventArgs e)

{

BackgroundWorker worker = new BackgroundWorker();

worker.WorkerReportsProgress = true;

worker.DoWork += worker\_DoWork;

worker.ProgressChanged += worker\_ProgressChanged;

worker.RunWorkerAsync();

}

void worker\_DoWork(object sender, DoWorkEventArgs e)

{

for(int i = 0; i < 100; i++)

{

(sender as BackgroundWorker).ReportProgress(i);

Thread.Sleep(100);

}

}

void worker\_ProgressChanged(object sender, ProgressChangedEventArgs e)

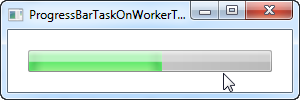
{

pbStatus.Value = e.ProgressPercentage;

}

}

}



As you can see on the screenshot, the progress is now updated all the way through the task, and as the cursor indicates, no hard work is being performed on the UI thread, which means that you can still interact with the rest of the interface.

Please be aware that while the BackgroundWorker does help a lot with multithreading related problems, there are still some things you should be aware of, so please have a look at the BackgroundWorker articles in this tutorial before doing anything more advanced than a scenario like the one above.

## Indeterminate

For some tasks, expressing the progress as a percentage is not possible or you simply don't know how long it will take. For those situations, the indeterminate progress bar has been invented, where an animation lets the user know that something is happening, while indicating that the running time can't be determined.

The WPF ProgressBar supports this mode through the use of the IsIndeterminate property, which we'll show you in the next example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.ProgressBarIndeterminateSample"

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

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

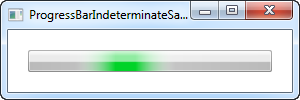
Title="ProgressBarIndeterminateSample" Height="100" Width="300">

<Grid Margin="20">

<ProgressBar Minimum="0" Maximum="100" Name="pbStatus" IsIndeterminate="True" />

</Grid>

</Window>



Notice that the green progress indicator is not anchored to either of the sides - instead it floats freely from start to finish and then it starts all over again.

## ProgressBar with text

One thing that I really missed from the standard WPF ProgressBar is the ability to show a text representation of the progress as well as the progress bar. Fortunately for us, the flexibility of WPF makes this really easy for us to accomplish. Here's an example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.ProgressBarTextSample"

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

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

Title="ProgressBarTextSample" Height="100" Width="300">

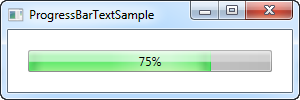
<Grid Margin="20">

<ProgressBar Minimum="0" Maximum="100" Value="75" Name="pbStatus" />

<TextBlock Text="{Binding ElementName=pbStatus, Path=Value, StringFormat={}{0:0}%}" HorizontalAlignment="Center" VerticalAlignment="Center" />

</Grid>

</Window>



We accomplish the above by putting the ProgressBar and the TextBlock showing the percentage inside of the same Grid, without specifying any rows or columns. This will render the TextBlock on top of the ProgressBar, which is exactly what we want here, because the TextBlock has a transparent background by default.

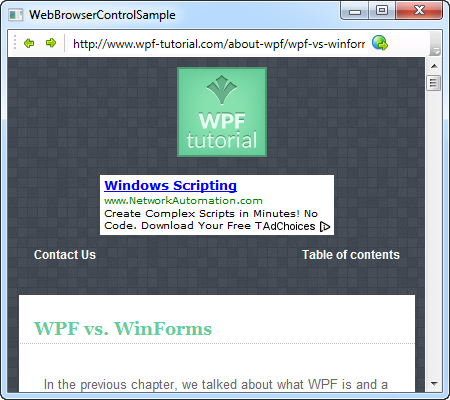
We use a binding to make sure that the TextBlock show the same value as the ProgressBar. Notice the special **StringFormat** syntax, which allows us to show the value with a percentage sign postfix - it might look a bit strange, but please see the **StringFormat** article of this tutorial for more information on it.

**Misc. controls:**

# The WebBrowser control

WPF comes with a ready to use WebBrowser control, which allows you to host a complete web browser within your application. The WebBrowser control is really just a shell around an ActiveX version of Internet Explorer, but since this is an integrated part of Windows, your application should work on all Windows machines without requiring the installation of additional components.

I've done things a bit differently in this article: Instead of starting off with a very limited example and then adding to it, I've create just one but more complex example. It illustrates how easy you can get a small web browser up and running. It's very basic in its functionality, but you can easily extend it if you want to. Here's how it looks:



So let's have a look at the code:

<Window x:Class="WpfTutorialSamples.Misc\_controls.WebBrowserControlSample"

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

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

Title="WebBrowserControlSample" Height="300" Width="450">

<Window.CommandBindings>

<CommandBinding Command="NavigationCommands.BrowseBack" CanExecute="BrowseBack\_CanExecute" Executed="BrowseBack\_Executed" />

<CommandBinding Command="NavigationCommands.BrowseForward" CanExecute="BrowseForward\_CanExecute" Executed="BrowseForward\_Executed" />

<CommandBinding Command="NavigationCommands.GoToPage" CanExecute="GoToPage\_CanExecute" Executed="GoToPage\_Executed" />

</Window.CommandBindings>

<DockPanel>

<ToolBar DockPanel.Dock="Top">

<Button Command="NavigationCommands.BrowseBack">

<Image Source="/WpfTutorialSamples;component/Images/arrow\_left.png" Width="16" Height="16" />

</Button>

<Button Command="NavigationCommands.BrowseForward">

<Image Source="/WpfTutorialSamples;component/Images/arrow\_right.png" Width="16" Height="16" />

</Button>

<Separator />

<TextBox Name="txtUrl" Width="300" KeyUp="txtUrl\_KeyUp" />

<Button Command="NavigationCommands.GoToPage">

<Image Source="/WpfTutorialSamples;component/Images/world\_go.png" Width="16" Height="16" />

</Button>

</ToolBar>

<WebBrowser Name="wbSample" Navigating="wbSample\_Navigating"></WebBrowser>

</DockPanel>

</Window>

using System;

using System.Windows;

using System.Windows.Input;

namespace WpfTutorialSamples.Misc\_controls

{

public partial class WebBrowserControlSample : Window

{

public WebBrowserControlSample()

{

InitializeComponent();

wbSample.Navigate("http://www.wpf-tutorial.com");

}

private void txtUrl\_KeyUp(object sender, KeyEventArgs e)

{

if(e.Key == Key.Enter)

wbSample.Navigate(txtUrl.Text);

}

private void wbSample\_Navigating(object sender, System.Windows.Navigation.NavigatingCancelEventArgs e)

{

txtUrl.Text = e.Uri.OriginalString;

}

private void BrowseBack\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = ((wbSample != null) && (wbSample.CanGoBack));

}

private void BrowseBack\_Executed(object sender, ExecutedRoutedEventArgs e)

{

wbSample.GoBack();

}

private void BrowseForward\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = ((wbSample != null) && (wbSample.CanGoForward));

}

private void BrowseForward\_Executed(object sender, ExecutedRoutedEventArgs e)

{

wbSample.GoForward();

}

private void GoToPage\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = true;

}

private void GoToPage\_Executed(object sender, ExecutedRoutedEventArgs e)

{

wbSample.Navigate(txtUrl.Text);

}

}

}

The code might seem a bit overwhelming at first, but if you take a second look, you'll realize that there's a lot of repetition in it.

Let's start off by talking about the **XAML** part. Notice that I'm using several concepts discussed elsewhere in this tutorial, including the ToolBar control and WPF commands. The ToolBar is used to host a couple of buttons for going backward and forward. After that, we have an address bar for entering and showing the current URL, along with a button for navigating to the entered URL.

Below the toolbar, we have the actual WebBrowser control. As you can see, using it only requires a single line of XAML - in this case we subscribe to the **Navigating** event, which occurs as soon as the WebBrowser starts navigating to a URL.

In **Code-behind**, we start off by navigating to a URL already in the constructor of the Window, to have something to show immediately instead of a blank control. We then have the **txtUrl\_KeyUp** event, in which we check to see if the user has hit Enter inside of the address bar - if so, we start navigating to the entered URL.

The **wbSample\_Navigating** event makes sure that the address bar is updated each time a new navigation starts. This is important because we want it to show the current URL no matter if the user initiated the navigation by entering a new URL or by clicking a link on the webpage.

The last part of the Code-behind is simple handling of our commands: Two for the back and forward buttons, where we use the CanGoBack and CanGoForward to decide whether they can execute, and the GoBack and GoForward methods to do the actual work. This is very standard when dealing with WPF commands, as described in the commands section of this tutorial.

For the last command, we allow it to always execute and when it does, we use the Navigate() method once again.

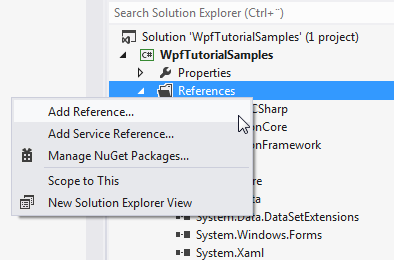
# The WindowsFormsHost control

WPF and WinForms are two distinct UI frameworks, both created by Microsoft. WPF is meant as a more modern alternative to WinForms, which was the first .NET UI framework. To lighten the transition between the two, Microsoft has made sure that WinForms controls may still be used inside of a WPF application. This is done with the WindowsFormsHost, which we'll discuss in this article.

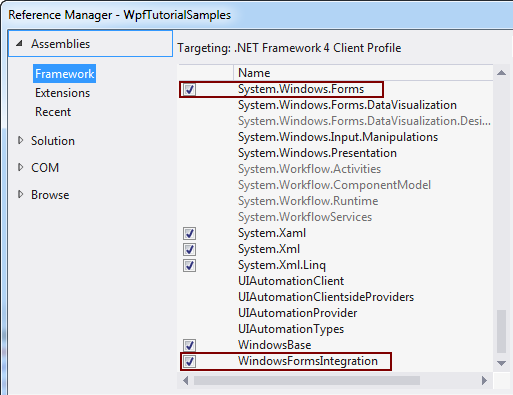
To use the WindowsFormsHost and controls from WinForms, you need to add a reference to the following assemblies in your application:

* WindowsFormsIntegration
* System.Windows.Forms

In Visual Studio, this is done by right-clicking the "**References**" node in your project and selecting "**Add reference**":



In the dialog that pops up, you should select "**Assemblies**" and then check the two assemblies that we need to add:



## Using the WinForms WebBrowser control

In a previous article, we used the WPF WebBrowser control to create a small web browser. However, as stated in that article, the WPF WebBrowser control is a bit limited when compared to the WinForms version. There are many examples on things easily done with the WinForms version, which are either harder or impossible to do with the WPF version.

A small example is the **DocumentTitle** property and corresponding **DocumentTitleChanged** event, which makes it easy to get and update the title of the window to match the title of the current webpage. We'll use this as an excuse to test out the WinForms version right here in our WPF application:

<Window x:Class="WpfTutorialSamples.Misc\_controls.WindowsFormsHostSample"

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

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

xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"

Title="WindowsFormsHostSample" Height="350" Width="450">

<Grid>

<WindowsFormsHost Name="wfhSample">

<WindowsFormsHost.Child>

<wf:WebBrowser DocumentTitleChanged="wbWinForms\_DocumentTitleChanged" />

</WindowsFormsHost.Child>

</WindowsFormsHost>

</Grid>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.Misc\_controls

{

public partial class WindowsFormsHostSample : Window

{

public WindowsFormsHostSample()

{

InitializeComponent();

(wfhSample.Child as System.Windows.Forms.WebBrowser).Navigate("http://www.wpf-tutorial.com");

}

private void wbWinForms\_DocumentTitleChanged(object sender, EventArgs e)

{

this.Title = (sender as System.Windows.Forms.WebBrowser).DocumentTitle;

}

}

}

Pay special attention to the line where we add the WinForms namespace to the window, so that we may reference controls from it:

xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"

This will allow us to reference WinForms controls using the wf: prefix.

The WindowsFormsHost is fairly simple to use, as you can see. It has a Child property, in which you can define a single WinForms control, much like the WPF Window only holds a single root control. If you need more controls from WinForms inside of your WindowsFormsHost, you can use the **Panel** control from WinForms or any of the other container controls.

The WinForms WebBrowser control is used by referencing the System.Windows.Forms assembly, using the wf prefix, as explained above.

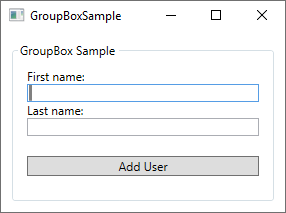
In **Code-behind**, we do an initial call to Navigate, to have a visible webpage instead of the empty control on startup. We then handle the**DocumentTitleChanged** event, in which we update the Title property of the Window in accordance with the current **DocumentTitle** value of the WebBrowser control.

Congratulations, you now have a WPF application with a WinForms WebBrowser hosted inside of it.

**Misc. controls:**

# The GroupBox control

The GroupBox control will allow you to visually group a set of controls together. This could obviously be done using one of the many panels as well, but the GroupBox adds a special type of header and border, which has historically been used a lot within in the Windows operating system. Here's a screenshot of how it might look when you use the **GroupBox** control:



Notice the border around the controls, with the text "GroupBox Sample" placed inside the border line - this is how a GroupBox looks and acts. Using a GroupBox is as simple as adding the tag to your Window and writing something relevant in the **Header** property:

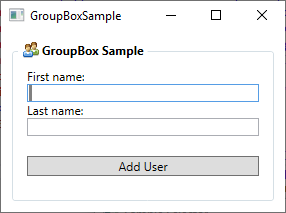
<GroupBox Header="GroupBox Sample">  
  
</GroupBox>

The GroupBox can only contain a single child element, but that's no problem - just make this one control a Panel, and you are free to add multiple controls to the panel, e.g. to create a dialog like the one displayed above. Here's the full XAML code listing for my example dialog:

<Window x:Class="WpfTutorialSamples.Misc\_controls.GroupBoxSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 xmlns:local="clr-namespace:WpfTutorialSamples.Misc\_controls"  
 mc:Ignorable="d"  
 Title="GroupBoxSample" Height="220" Width="300">  
    <Grid>  
 <GroupBox Header="GroupBox Sample" Margin="10" Padding="10">  
     <StackPanel>  
 <TextBlock>First name:</TextBlock>  
 <TextBox />  
 <TextBlock>Last name:</TextBlock>  
 <TextBox />  
 <Button Margin="0,20">Add User</Button>  
     </StackPanel>  
 </GroupBox>  
    </Grid>  
</Window>

## GroupBox with custom Header

The Header of a GroupBox is normally just plain, unformatted text, but perhaps you're looking to make it a bit more fancy? No problem, because just like pretty much anything found in the WPF framework, you can just replace the text with one or several other controls. So you can just add a TextBlock control and then change the formatting, e.g. the color of the text. You can even add an image, if you want to, like I have done in this next example:



Now the Header has an image and bold text, and it's so easy to do:

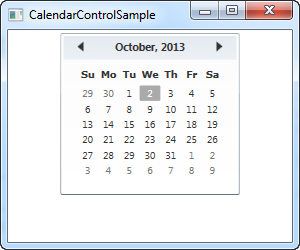
<Window x:Class="WpfTutorialSamples.Misc\_controls.GroupBoxSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 xmlns:local="clr-namespace:WpfTutorialSamples.Misc\_controls"  
 mc:Ignorable="d"  
 Title="GroupBoxSample" Height="220" Width="300">  
    <Grid>  
 <GroupBox Margin="10" Padding="10">  
     <GroupBox.Header>  
 <StackPanel Orientation="Horizontal">  
     <Image Source="/WpfTutorialSamples;component/Images/group.png" Margin="3,0" />  
     <TextBlock FontWeight="Bold">GroupBox Sample</TextBlock>  
 </StackPanel>  
     </GroupBox.Header>  
     <StackPanel>  
 <TextBlock>First name:</TextBlock>  
 <TextBox />  
 <TextBlock>Last name:</TextBlock>  
 <TextBox />  
 <Button Margin="0,20">Add User</Button>  
     </StackPanel>  
 </GroupBox>  
    </Grid>  
</Window>

Notice how I have simply replaced the Header property with a **GroupBox.Header** tag, which then hosts a StackPanel to contain an Image and a TextBlock - with that in place, you have full control of how the Header should look!

# The Calendar control

WPF comes with a control for displaying a full calendar right out of the box. It's so simple that you only have to drop it inside your window for a full calendar view, like this:

<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarControlSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarControlSample" Height="250" Width="300">  
    <Grid>  
 <Calendar />  
    </Grid>  
</Window>



Notice how you now get a full list of the dates within the selected month, including the possibility to jump to previous and next months using the arrows in the top of the control. Unless you set a specific date, the current month will be shown and the current date will be marked as selected.

## Calendar size

You will probably notice from our first example that the Calendar doesn't take up all the available space. In fact, even if you give it a large width and height, the actual calendar part will still only take up the amount of space you see on the screenshot, and if you set either of the values very low, the calendar will only be partially visible.

This fixed size behavior is not very typical WPF, where things usually stretch to fill out available space, and it can be a bit annoying to work with if you have a designated amount of space available for the calendar which you want it to fill out. Fortunately for us, everything in WPF is scalable but in the case of the Calendar control, it needs a bit of help. We'll use the Viewbox control for this purpose:

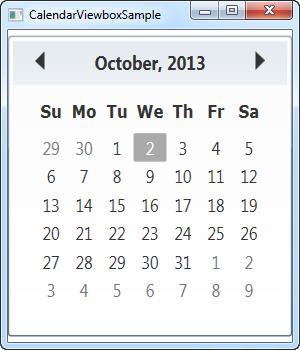
<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarViewboxSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarViewboxSample" Height="350" Width="300">  
    <Viewbox>  
 <Calendar />  
    </Viewbox>  
</Window>



Notice how the Calendar control now scales up to the point where it uses all the available space in the width. The scaling is performed on all parts of the control, including font sizes and border widths.

You will probably also notice that the Calendar control doesn't use up all the available height space. This is noticeable because the window is higher than it is wide and by default, the Viewbox will stretch while maintaining the original aspect ratio. You can easily make it stretch to fill all space in both directions though - simply change the **Stretch** property from its default **Uniform** value to **Fill**:

<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarViewboxSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarViewboxSample" Height="350" Width="300">  
    <Viewbox Stretch="Fill" StretchDirection="UpOnly">  
 <Calendar />  
    </Viewbox>  
</Window>



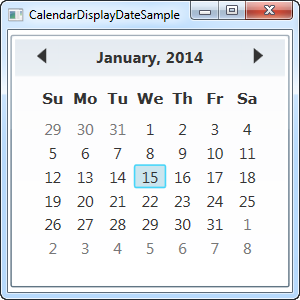
Now it takes up all the available space, in both directions. This is generally not preferable though, since most controls, and this one in particular, will look strange if it gets an abnormal set of dimensions, e.g. 800 pixels high and 300 pixels wide. A **Stretch** mode set to **Uniform** (or left out, as it is the default) is usually the way to go.

I would recommend including the **StretchDirection** property though, as seen in this example. It allows us to specify that the contents should only be scaled up or down, which can be useful. For instance, the Calendar control becomes quite useless below a certain size, where you can no longer see what it is, and to avoid that, you can set the **StretchDirection** to **UpOnly** - the Calendar control will then no longer be scaled below its default size.

## Setting the initial view using DisplayDate

The Calendar control will by default show the current month, but you can change this by using the **DisplayDate** property. Simply set it to a date within the month you wish to start with and it will be reflected in the control:

<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarDisplayDateSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarDisplayDateSample" Height="300" Width="300">  
    <Viewbox>  
 <Calendar DisplayDate="01.01.2014" />  
    </Viewbox>  
</Window>



## Calendar SelectionMode

The **SelectionMode** property is interesting. By changing it from its default value, **SingleDate**, you can select multiple dates or ranges of dates. Here's an example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarSelectionModeSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarSelectionModeSample" Height="300" Width="300">  
    <Viewbox>  
 <Calendar SelectionMode="SingleRange" />  
    </Viewbox>  
</Window>



In the **SingleRange** SelectionMode, you can select an entire range of dates, either by holding down the left mouse button and dragging from one date to another or by holding down the Ctrl or Shift keys while clicking several dates, much like multi selection works in all parts of Windows. On the screenshot, I've selected an entire week, from Sunday to Monday, but you can just as easily select dates in the middle of the week and ranges which expands a single week.

SingleRange mode only allows a single range of dates to be selected though, much like the name suggests. This means that you can't select two dates which are not next to each other, and you can't select more than one range. If you want this, you should switch to **MultipleRange** selection:

<Calendar SelectionMode="MultipleRange" />



With this property, there are really no limits to the dates you can select. In this case, I've selected all the Saturdays, all the Sundays and a couple of week days in between.

Of course, if you don't want the ability to select one or several dates, you can set the **SelectionMode** to **None**.

Now let's discuss how we can work with the selected date(s) of the Calendar control.

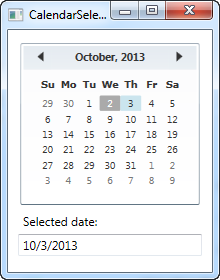
## Working with the selected date

The **SelectedDate** property is all you need if you're only allowing single selections (see the above explanation on selection modes). It allows you to both set and get a currently selected date, from Code-behind as well as through a data binding.

Here's an example where we set the selected date to tomorrow from Code-behind and then use a data binding to read out the selected date to a TextBox control:

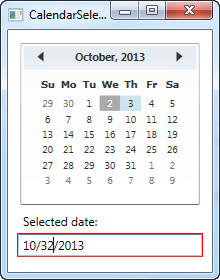
<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarSelectionSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarSelectionSample" Height="280" Width="220">  
    <StackPanel Margin="10">  
 <Calendar Name="cldSample" SelectionMode="MultipleRange" SelectedDate="10.10.2013" />  
 <Label>Selected date:</Label>  
 <TextBox Text="{Binding ElementName=cldSample, Path=SelectedDate, StringFormat=d, UpdateSourceTrigger=PropertyChanged}" />  
    </StackPanel>  
</Window>

using System;  
using System.Windows;  
  
namespace WpfTutorialSamples.Misc\_controls  
{  
    public partial class CalendarSelectionSample : Window  
    {  
 public CalendarSelectionSample()  
 {  
     InitializeComponent();  
     cldSample.SelectedDate = DateTime.Now.AddDays(1);  
 }  
    }  
}



In Code-behind, we simply set the **SelectedDate** property to the current date plus one day, meaning tomorrow. The user can then change this selection by clicking in the Calendar control, and through the data binding established in Text property of the TextBox, this change will automatically be reflected there.

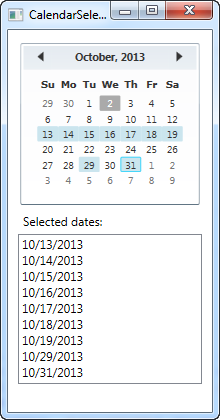
As an added bonus, through the magic of data binding, you can also change the value from the TextBox - just input a valid date and the change will be immediately reflected in the Calendar control. Should you enter a bad date, the automatic binding validation notifies you of the problem:



## Working with multiple selected dates

If you allow more than one selected date at the time, you won't find the SelectedDate property that useful. Instead, you should use the SelectedDates, which is a collection of currently selected dates in the Calendar control. This property can be accessed from Code-behind or used with a binding, like we do here:

<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarSelectedDatesSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarSelectedDatesSample" Height="420" Width="220">  
    <StackPanel Margin="10">  
 <Calendar Name="cldSample" SelectionMode="MultipleRange" />  
 <Label>Selected dates:</Label>  
 <ListBox ItemsSource="{Binding ElementName=cldSample, Path=SelectedDates}" MinHeight="150" />  
    </StackPanel>  
</Window>



With a simple binding like that, we're now able to display a list of the currently selected dates.

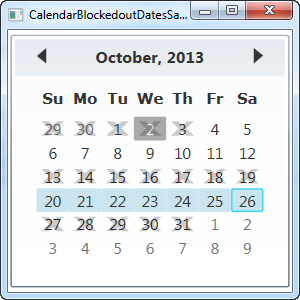
If you want to react to dates being changed from Code-behind, you can subscribe to the **SelectedDatesChanged** event of the Calendar control.

## Blackout dates

Depending on what you use the Calendar control for, you may want to black out certain dates. This could be relevant e.g. in a booking application, where you want to prevent already reserved dates from being selected. The Calendar control supports this right out of the box through the use of the **BlackoutDates** collection, which you can of course use from both XAML and Code-behind:

<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarBlockedoutDatesSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarBlockedoutDatesSample" Height="300" Width="300">  
    <Viewbox>  
 <Calendar Name="cldSample" SelectionMode="MultipleRange">  
     <Calendar.BlackoutDates>  
 <CalendarDateRange Start="10.13.2013" End="10.19.2013" />  
 <CalendarDateRange Start="10.27.2013" End="10.31.2013" />  
     </Calendar.BlackoutDates>  
 </Calendar>  
    </Viewbox>  
</Window>

using System;  
using System.Windows;  
using System.Windows.Controls;  
  
namespace WpfTutorialSamples.Misc\_controls  
{  
    public partial class CalendarBlockedoutDatesSample : Window  
    {  
 public CalendarBlockedoutDatesSample()  
 {  
     InitializeComponent();  
     cldSample.BlackoutDates.AddDatesInPast();  
     cldSample.BlackoutDates.Add(new CalendarDateRange(DateTime.Today, DateTime.Today.AddDays(1)));  
 }  
    }  
}



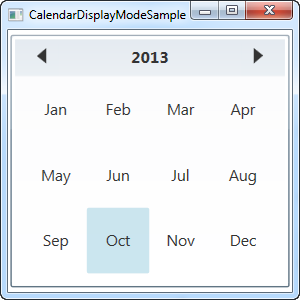
In this example, I demonstrate both ways of adding blacked out dates - through XAML and through Code-behind. Both ways works by adding instances of **CalendarDateRange** to the **BlackedoutDates** collection.

In XAML, I'm hardcoding the date ranges (mostly to show you it can be done that way too), while I do something a bit more clever in Code-behind, by first adding all past dates to the collection with a single call to the **AddDatesInPast()** method and then adding a range consisting of today and tomorrow.

## DisplayMode - showing months or years

The **DisplayMode** property can change the Calendar control from a place where you can select a date to a place where you can select a month or even a year. This is done through the DisplayMode property, which defaults to Month, which we've used in all the previous examples. Here's how it looks if we change it:

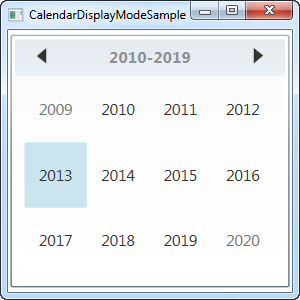
<Window x:Class="WpfTutorialSamples.Misc\_controls.CalendarDisplayModeSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CalendarDisplayModeSample" Height="300" Width="300">  
    <Viewbox>  
 <Calendar DisplayMode="Year" />  
    </Viewbox>  
</Window>



By setting the **DisplayMode** to **Year**, we can now select a month of a given year. You can change the year in the top, by using the arrows.

The Calendar control also allows for selecting an entire year, by using the **Decade** value for the **DisplayMode** property:

<Calendar DisplayMode="Decade" />

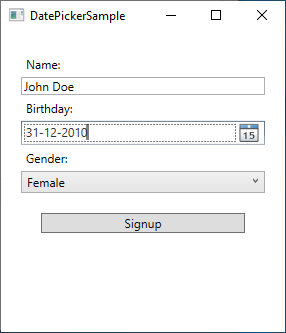


# The DatePicker control

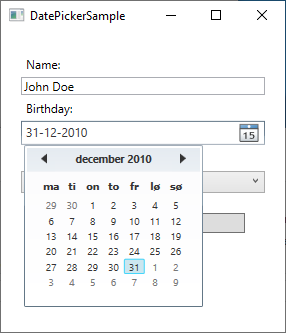
Dealing with dates in general can be cumbersome. There are many ways of writing a date, depending on where in the world your user(s) are, so allowing them to enter a date freely in a TextBox is almost never a good idea. Fortunately for us, WPF comes with several controls for dealing with dates.

We already looked at one of these controls, the Calendar control, which is great if selecting the date is the primary task of your dialog. However, often you will need to collect a date along with a lot of other information, in a form with multiple input controls like TextBox's, ComboBox's and so on. For a situation like that, you need a date-input control which can blend in with the rest and fit into the layout of a form - in other words, you need the **DatePicker** control!

The DatePicker control will be displayed pretty much like a regular TextBox, but with a small button which will bring up a Calendar-view when clicked, allowing your user to select the date. Here's an example of how it could look:



You can then write the date manually or click the small button to select the date from a calendar control:



## Adding a DatePicker control

The DatePicker control works straight out of the box - just add it anywhere in your Window and you're good to go:

<DatePicker></DatePicker>

Here's the full code listing used to create the example dialog above:

<Window x:Class="WpfTutorialSamples.Misc\_controls.DatePickerSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 xmlns:local="clr-namespace:WpfTutorialSamples.Misc\_controls"  
 mc:Ignorable="d"  
 Title="DatePickerSample" Height="300" Width="300">  
    <StackPanel Margin="20">  
 <Label>Name:</Label>  
 <TextBox />  
 <Label>Birthday:</Label>  
 <DatePicker></DatePicker>  
 <Label>Gender:</Label>  
 <ComboBox>  
     <ComboBoxItem>Female</ComboBoxItem>  
     <ComboBoxItem>Male</ComboBoxItem>  
 </ComboBox>  
 <Button Margin="20">Signup</Button>  
    </StackPanel>  
</Window>

## DisplayDate and SelectedDate

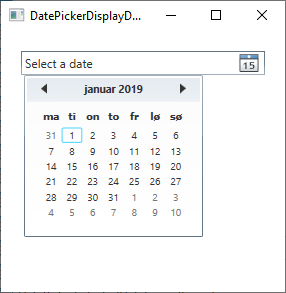
By default, the DatePicker control will not have a date selected - that is left to the user. However, if you need to prefill the control with a date, just use the **SelectedDate** property, like this:

<DatePicker SelectedDate="2000-12-31"></DatePicker>

The DatePicker will now have a pre-selected date, which the user can choose to override by selecting/entering another date. The **SelectedDate** can also be set from Code-behind, and perhaps more importantly, it can also be read from Code-behind, or you can bind it's value to your Model or another control.

Sometimes you might need to start the calendar at a specific date, without actually selecting one for the user. For that, we have the **DisplayDate** property. The default value is the current date, but you can easily change that:

<DatePicker Name="dp1" DisplayDate="2019-01-01" />

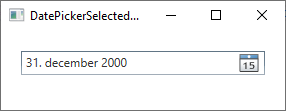


Notice how, when we use the DisplayDate property, the calendar starts at the specified date (and highlights it), but no date is actually selected (as indicated by the "Select a date" text).

## SelectedDateFormat

Another interesting property is the **SelectedDateFormat**. The default value is **Short**, but if you change it to **Long**, it will be formatted in a slightly more verbose way:

<DatePicker SelectedDate="2000-12-31" SelectedDateFormat="Long"></DatePicker>



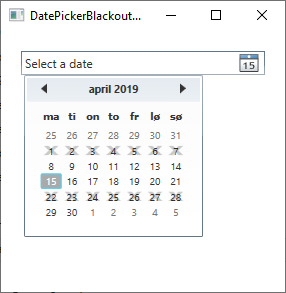
Whether the Short or the Long format is used, the actual format of the date is decided by the culture of your application. If you don't specifically define a culture for your application, the system settings are used. You will notice from the screenshots of this article that on this computer, the date format is DMY (date-month-year), but this can easily be changed by setting a specific culture. We'll discuss that elsewhere in this tutorial.

## Blackout dates

Depending on what you use the DatePicker control for, you may want to black out certain dates. This will prevent the selection of these dates, which will be visually indicated, and could be relevant e.g. in a booking application, where you want to prevent already reserved dates from being selected. The DatePicker control supports this right out of the box through the use of the BlackoutDates collection, which you can of course use from both XAML and Code-behind. Here's how to do it with XAML:

<DatePicker Name="dp1">  
    <DatePicker.BlackoutDates>  
 <CalendarDateRange Start="2019-04-01" End="2019-04-07" />  
 <CalendarDateRange Start="2019-04-22" End="2019-04-28" />  
    </DatePicker.BlackoutDates>  
</DatePicker>

The result will look like this:



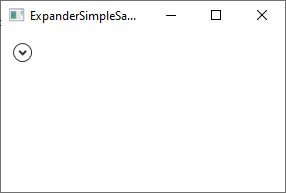
Doing it from Code-behind is just as easy and it has two added benefits: First of all, you can create the date range dynamically, e.g. based on the current date. You can also use the **AddDatesInPast()** method to automatically exclude all dates in the past. Here's an example:

dp1.BlackoutDates.AddDatesInPast();  
dp1.BlackoutDates.Add(new CalendarDateRange(DateTime.Now, DateTime.Now.AddDays(7)));

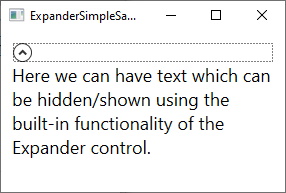
With that in place, all past dates, as well as the next week, will be unavailable for selection.

# The Expander control

The **Expander** control will provide you with the ability to hide/show a piece of content. This would usually be a piece of text, but thanks to the flexibility of WPF, it can be used for any type of mixed content like texts, images and even other WPF controls. To see what I'm talking about, here's an example:



Notice the arrow part - as soon as you click it, the Expander control will expand and reveal its content:



The code for it is of course very simple:

<Expander>    
    <TextBlock TextWrapping="Wrap" FontSize="18">    
 Here we can have text which can be hidden/shown using the built-in functionality of the Expander control.    
    </TextBlock>    
</Expander>

By default, the Expander is NOT expanded and therefore looks like it does on the first screenshot. The user can expand it by clicking it or you can make it initially expanded by using the **IsExpanded** property:

<Expander IsExpanded="True">

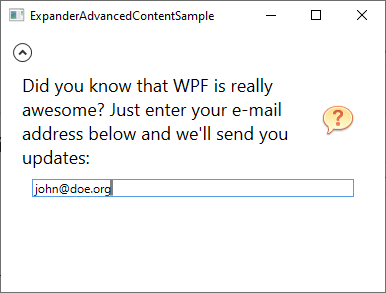
You can of course also read this property at runtime, if you need to know about the current state of the Expander control.

## Advanced content

The Content of the Expander can only be one control, like in our first example where we use a TextBlock control, but nothing prevents you from making this e.g. a Panel, which can then hold as many child controls as you want it to. This allows you to host rich content inside your Expander, from text and images to e.g. a ListView or any other WPF control.

Here's an example of more advanced content, where we use several panels, text and an image and even a TextBox control:

<Expander Margin="10">  
    <StackPanel Margin="10">  
 <DockPanel>  
     <Image Source="/WpfTutorialSamples;component/Images/question32.png" Width="32" Height="32" DockPanel.Dock="Right" Margin="10"></Image>  
     <TextBlock TextWrapping="Wrap" FontSize="18">  
     Did you know that WPF is really awesome? Just enter your e-mail address below and we'll send you updates:  
     </TextBlock>  
 </DockPanel>  
 <TextBox Margin="10">john@doe.org</TextBox>  
    </StackPanel>  
</Expander>

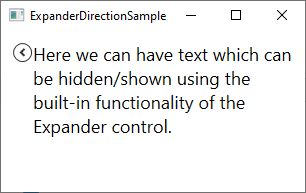


## ExpandDirection

By default, the Expander control will occupy all available space inside its container control and then expand in accordance with the value of the **ExpandDirection** property, which is set to **Down** as default. You can see this indicated on the screenshots above because the arrow is placed in top of the control and it points up or down based on whether the control has been expanded or not.

If you change the value of the **ExpandDirection** property, it will affect how the Expander control acts and looks. For instance, if you change the value to **Right**, the arrow will be placed on the left side and point to the left/right instead of up/down. Here's an example:

<Expander Margin="10" ExpandDirection="Right">  
    <TextBlock TextWrapping="Wrap" FontSize="18">  
     Here we can have text which can be hidden/shown using the built-in functionality of the Expander control.  
    </TextBlock>  
</Expander>



You can of course set this property to **Up** or **Left** as well - if you do so, the button will be placed at the bottom or to the right.

## Custom header

In all the examples so far, the Expander control is almost look-less, except for the button which is used to show/hide the content - it's drawn as a circular button with an arrow inside. You can easily customize the header-area of the control though, using the **Header** property. Here's an example where we use this property to add an explanatory text next to the button:

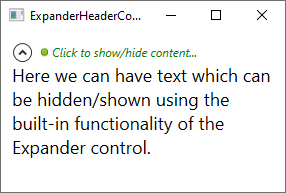
<Expander Margin="10" Header="Click to show/hide content...">  
    <TextBlock TextWrapping="Wrap" FontSize="18">  
     Here we can have text which can be hidden/shown using the built-in functionality of the Expander control.  
    </TextBlock>  
</Expander>



But you don't have to settle for a simple piece of text - the **Header** property will allow you to add controls to it, to create an even more customized look:

<Expander Margin="10">  
    <Expander.Header>  
 <DockPanel VerticalAlignment="Stretch">  
     <Image Source="/WpfTutorialSamples;component/Images/bullet\_green.png" Height="16" DockPanel.Dock="Left" />  
     <TextBlock FontStyle="Italic" Foreground="Green">Click to show/hide content...</TextBlock>  
 </DockPanel>  
    </Expander.Header>  
    <TextBlock TextWrapping="Wrap" FontSize="18">  
     Here we can have text which can be hidden/shown using the built-in functionality of the Expander control.  
    </TextBlock>  
</Expander>

Notice how I simply add a Panel as the content of the **Header** property and inside of that, I can do whatever I want, like adding an Image and a TextBlock control with custom formatting:



**The TabControl:**

# Using the WPF TabControl

The WPF TabControl allows you to split your interface up into different areas, each accessible by clicking on the tab header, usually positioned at the top of the control. Tab controls are commonly used in Windows applications and even within Windows' own interfaces, like the properties dialog for files/folders etc.

Just like with most other WPF controls, the TabControl is very easy to get started with. Here's a very basic example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.TabControlSample"

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

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

Title="TabControlSample" Height="200" Width="250">

<Grid>

<TabControl>

<TabItem Header="General">

<Label Content="Content goes here..." />

</TabItem>

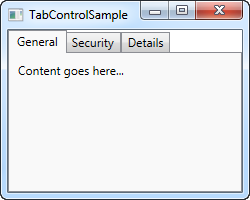
<TabItem Header="Security" />

<TabItem Header="Details" />

</TabControl>

</Grid>

</Window>



As you can see, each tab is represented with a **TabItem** element, where the text shown on it is controlled by the **Header** property. The TabItem element comes from the ContentControl class, which means that you may define a single element inside of it that will be shown if the tab is active (like on the screenshot). I used a Label in this example, but if you want to place more than one control inside of the tab, just use one of the panels with child controls inside of it.

## Customized headers

Once again, WPF proves to be extremely flexible when you want to customize the look of your tabs. Obviously the content can be rendered any way you like it, but so can the tab headers! The Header property can be filled with anything you like, which we'll take advantage of in the next example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.TabControlWithCustomHeadersSample"

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

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

Title="TabControlWithCustomHeadersSample" Height="200" Width="250">

<Grid>

<Grid>

<TabControl>

<TabItem>

<TabItem.Header>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_blue.png" />

<TextBlock Text="Blue" Foreground="Blue" />

</StackPanel>

</TabItem.Header>

<Label Content="Content goes here..." />

</TabItem>

<TabItem>

<TabItem.Header>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_red.png" />

<TextBlock Text="Red" Foreground="Red" />

</StackPanel>

</TabItem.Header>

</TabItem>

<TabItem>

<TabItem.Header>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_green.png" />

<TextBlock Text="Green" Foreground="Green" />

</StackPanel>

</TabItem.Header>

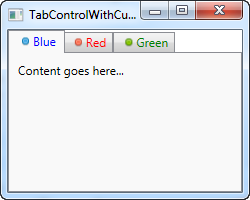
</TabItem>

</TabControl>

</Grid>

</Grid>

</Window>



The amount of markup might be a bit overwhelming, but as you can probably see once you dig into it, it's all very simple. Each of the tabs now has a TabControl.Header element, which contains a StackPanel, which in turn contains an Image and a TextBlock control. This allows us to have an image on each of the tabs as well as customize the color of the text (we could have made it bold, italic or another size as well).

## Controlling the TabControl

Sometimes you may wish to control which tab is selected programmatically or perhaps get some information about the selected tab. The WPF TabControl has several properties which makes this possible, including SelectedIndex and SelectedItem. In the next example, I've added a couple of buttons to the first example which allows us to control the TabControl:

<Window x:Class="WpfTutorialSamples.Misc\_controls.ControllingTheTabControlSample"

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

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

Title="ControllingTheTabControlSample" Height="300" Width="350">

<DockPanel>

<StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom" Margin="2,5">

<Button Name="btnPreviousTab" Click="btnPreviousTab\_Click">Prev.</Button>

<Button Name="btnNextTab" Click="btnNextTab\_Click">Next</Button>

<Button Name="btnSelectedTab" Click="btnSelectedTab\_Click">Selected</Button>

</StackPanel>

<TabControl Name="tcSample">

<TabItem Header="General">

<Label Content="Content goes here..." />

</TabItem>

<TabItem Header="Security" />

<TabItem Header="Details" />

</TabControl>

</DockPanel>

</Window>

using System;

using System.Windows;

using System.Windows.Controls;

namespace WpfTutorialSamples.Misc\_controls

{

public partial class ControllingTheTabControlSample : Window

{

public ControllingTheTabControlSample()

{

InitializeComponent();

}

private void btnPreviousTab\_Click(object sender, RoutedEventArgs e)

{

int newIndex = tcSample.SelectedIndex - 1;

if(newIndex < 0)

newIndex = tcSample.Items.Count - 1;

tcSample.SelectedIndex = newIndex;

}

private void btnNextTab\_Click(object sender, RoutedEventArgs e)

{

int newIndex = tcSample.SelectedIndex + 1;

if(newIndex >= tcSample.Items.Count)

newIndex = 0;

tcSample.SelectedIndex = newIndex;

}

private void btnSelectedTab\_Click(object sender, RoutedEventArgs e)

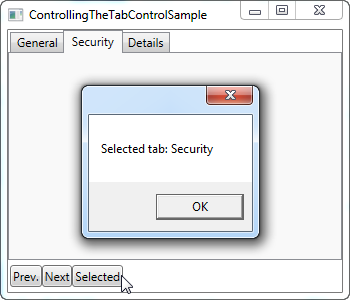
{

MessageBox.Show("Selected tab: " + (tcSample.SelectedItem as TabItem).Header);

}

}

}



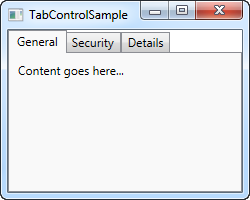
As you can see, I've simply added a set of buttons in the lower part of the interface. The first two allows will select the previous or next tab on the control, while the last one will display information about the currently selected tab, as demonstrated on the screenshot.

The first two buttons uses the **SelectedIndex** property to determine where we are and then either subtracts or adds one to that value, making sure that the new index doesn't fall below or above the amount of available items. The third button uses the **SelectedItem** property to get a reference to the selected tab. As you can see, I have to typecast it into the TabItem class to get a hold of the header property, since the SelectedProperty is of the object type by default.

**The TabControl:**

# WPF TabControl: Tab positions

The tabs of a TabControl is usually placed on top of the control, which is also how it will look by default when using the WPF TabControl:



However, using the **TabStripPlacement**property, we can very easily change this:

<Window x:Class="WpfTutorialSamples.Misc\_controls.TabStripPlacementSample"

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

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

Title="TabStripPlacementSample" Height="200" Width="250">

<Grid>

<TabControl TabStripPlacement="Bottom">

<TabItem Header="General">

<Label Content="Content goes here..." />

</TabItem>

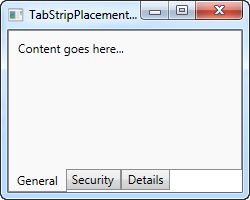
<TabItem Header="Security" />

<TabItem Header="Details" />

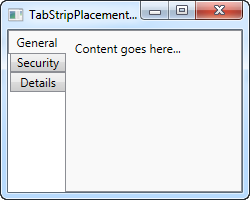
</TabControl>

</Grid>

</Window>



The TabStripPlacement can be set to Top, Bottom, Left and Right. However, if we set it to Left or Right, we get a result like this:



I personally would expect that the tabs to be rotated when placed on one of the sides, so that the tab text becomes vertical instead of horizontal, but the WPF TabControl doesn't do this. Fortunately, we can accomplish this behavior with a small hack:

<Window x:Class="WpfTutorialSamples.Misc\_controls.TabStripPlacementSample"

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

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

Title="TabStripPlacementSample" Height="200" Width="250" UseLayoutRounding="True">

<Grid>

<TabControl TabStripPlacement="Left">

<TabControl.Resources>

<Style TargetType="{x:Type TabItem}">

<Setter Property="HeaderTemplate">

<Setter.Value>

<DataTemplate>

<ContentPresenter Content="{TemplateBinding Content}">

<ContentPresenter.LayoutTransform>

<RotateTransform Angle="270" />

</ContentPresenter.LayoutTransform>

</ContentPresenter>

</DataTemplate>

</Setter.Value>

</Setter>

<Setter Property="Padding" Value="3" />

</Style>

</TabControl.Resources>

<TabItem Header="General">

<Label Content="Content goes here..." />

</TabItem>

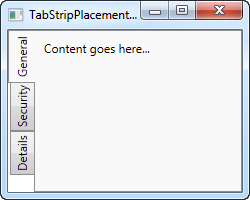
<TabItem Header="Security" />

<TabItem Header="Details" />

</TabControl>

</Grid>

</Window>



If you haven't yet read the chapters on templates or styles, this might seem a bit confusing, but what we do is using a style targeted at the TabItem elements, where we override the HeaderTemplate and then apply a rotate transform to the tabs. For tabs placed on the left side, we rotate 270 degrees - if placed on the right, you should only rotate 90 degrees, to make it look correct.

**The TabControl:**

# WPF TabControl: Styling the TabItems

In one of the previous articles, we discovered how easy it was to customize the tab headers of the WPF TabControl, for instance to add an image or color the text. However, if you wish to go beyond that and directly influence how the tab looks, including shape and borders, you need to override the control template of the TabItem element, and while this is not as straight forward as most other areas of WPF, it's still manageable.

So, if you would like to get full control of how the tabs of your TabControl looks, check out the next example:

<Window x:Class="WpfTutorialSamples.Misc\_controls.StyledTabItemsSample"

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

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

Title="StyledTabItemsSample" Height="150" Width="250">

<Grid>

<TabControl Margin="10" BorderThickness="0" Background="LightGray">

<TabControl.Resources>

<Style TargetType="TabItem">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="TabItem">

<Grid Name="Panel">

<ContentPresenter x:Name="ContentSite"

VerticalAlignment="Center"

HorizontalAlignment="Center"

ContentSource="Header"

Margin="10,2"/>

</Grid>

<ControlTemplate.Triggers>

<Trigger Property="IsSelected" Value="True">

<Setter TargetName="Panel" Property="Background" Value="LightSkyBlue" />

</Trigger>

<Trigger Property="IsSelected" Value="False">

<Setter TargetName="Panel" Property="Background" Value="White" />

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</TabControl.Resources>

<TabItem Header="General">

<Label Content="Content goes here..." />

</TabItem>

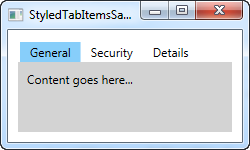
<TabItem Header="Security" />

<TabItem Header="Details" />

</TabControl>

</Grid>

</Window>



As you can see, this makes the TabControl looks a bit Windows 8'ish, with no borders and a less subtle color to mark the selected tab and no background for the unselected tabs. All of this is accomplished by changing the ControlTemplate, using a Style. By adding a **ContentPresenter**control, we specify where the content of the TabItem should be placed. We also have a couple of triggers, which controls the background color of the tabs based on the **IsSelected** property.

In case you want a less subtle look, it's as easy as changing the template. For instance, you might want a border, but with round corners and a gradient background - no problem! Check out this next example, where we accomplish just that:

<Window x:Class="WpfTutorialSamples.Misc\_controls.StyledTabItemsWithBorderSample"

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

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

Title="StyledTabItemsWithBorderSample" Height="150" Width="250">

<Grid>

<TabControl Margin="10" BorderBrush="Gainsboro">

<TabControl.Resources>

<Style TargetType="TabItem">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="TabItem">

<Border Name="Border" BorderThickness="1,1,1,0" BorderBrush="Gainsboro" CornerRadius="4,4,0,0" Margin="2,0">

<ContentPresenter x:Name="ContentSite"

VerticalAlignment="Center"

HorizontalAlignment="Center"

ContentSource="Header"

Margin="10,2"/>

</Border>

<ControlTemplate.Triggers>

<Trigger Property="IsSelected" Value="True">

<Setter TargetName="Border" Property="Background" Value="LightSkyBlue" />

</Trigger>

<Trigger Property="IsSelected" Value="False">

<Setter TargetName="Border" Property="Background" Value="GhostWhite" />

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</TabControl.Resources>

<TabItem Header="General">

<Label Content="Content goes here..." />

</TabItem>

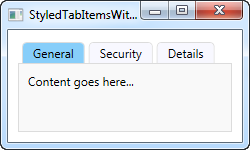
<TabItem Header="Security" />

<TabItem Header="Details" />

</TabControl>

</Grid>

</Window>



As you can see, I pretty much just added a Border control around the ContentPresenter to achieve this changed look. Hopefully this should demonstrate just how easy it is to get custom styled tabs and how many possibilities there are in this technique.

**List controls:**

# The ItemsControl

WPF has a wide range of controls for displaying a list of data. They come in several shapes and forms and vary in how complex they are and how much work they perform for you. The simplest variant is the ItemsControl, which is pretty much just a markup-based loop - you need to apply all the styling and templating, but in many cases, that's just what you need.

## A simple ItemsControl example

Let's kick off with a very simple example, where we hand-feed the ItemsControl with a set of items. This should show you just how simple the ItemsControl is:

<Window x:Class="WpfTutorialSamples.ItemsControl.ItemsControlSample"

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

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

xmlns:system="clr-namespace:System;assembly=mscorlib"

Title="ItemsControlSample" Height="150" Width="200">

<Grid Margin="10">

<ItemsControl>

<system:String>ItemsControl Item #1</system:String>

<system:String>ItemsControl Item #2</system:String>

<system:String>ItemsControl Item #3</system:String>

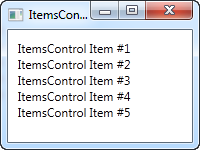
<system:String>ItemsControl Item #4</system:String>

<system:String>ItemsControl Item #5</system:String>

</ItemsControl>

</Grid>

</Window>



As you can see, there is nothing that shows that we're using a control for repeating the items instead of just manually adding e.g. 5 TextBlock controls - the ItemsControl is completely lookless by default. If you click on one of the items, nothing happens, because there's no concept of selected item(s) or anything like that.

## ItemsControl with data binding

Of course the ItemsControl is not meant to be used with items defined in the markup, like we did in the first example. Like pretty much any other control in WPF, the ItemsControl is made for data binding, where we use a template to define how our code-behind classes should be presented to the user.

To demonstrate that, I've whipped up an example where we display a TODO list to the user, and to show you just how flexible everything gets once you define your own templates, I've used a ProgressBar control to show you the current completion percentage. First some code, then a screenshot and then an explanation of it all:

<Window x:Class="WpfTutorialSamples.ItemsControl.ItemsControlDataBindingSample"

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

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

Title="ItemsControlDataBindingSample" Height="150" Width="300">

<Grid Margin="10">

<ItemsControl Name="icTodoList">

<ItemsControl.ItemTemplate>

<DataTemplate>

<Grid Margin="0,0,0,5">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="100" />

</Grid.ColumnDefinitions>

<TextBlock Text="{Binding Title}" />

<ProgressBar Grid.Column="1" Minimum="0" Maximum="100" Value="{Binding Completion}" />

</Grid>

</DataTemplate>

</ItemsControl.ItemTemplate>

</ItemsControl>

</Grid>

</Window>

using System;

using System.Windows;

using System.Collections.Generic;

namespace WpfTutorialSamples.ItemsControl

{

public partial class ItemsControlDataBindingSample : Window

{

public ItemsControlDataBindingSample()

{

InitializeComponent();

List<TodoItem> items = new List<TodoItem>();

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

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

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

icTodoList.ItemsSource = items;

}

}

public class TodoItem

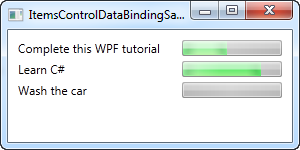
{

public string Title { get; set; }

public int Completion { get; set; }

}

}



The most important part of this example is the template that we specify inside of the ItemsControl, using a DataTemplate tag inside of the ItemsControl.ItemTemplate. We add a Grid panel, to get two columns: In the first we have a TextBlock, which will show the title of the TODO item, and in the second column we have a ProgressBar control, which value we bind to the Completion property.

The template now represents a TodoItem, which we declare in the Code-behind file, where we also instantiate a number of them and add them to a list. In the end, this list is assigned to the **ItemsSource** property of our ItemsControl, which then does the rest of the job for us. Each item in the list is displayed by using our template, as you can see from the resulting screenshot.

## The ItemsPanelTemplate property

In the above examples, all items are rendered from top to bottom, with each item taking up the full row. This happens because the ItemsControl throw all of our items into a vertically aligned StackPanel by default. It's very easy to change though, since the ItemsControl allows you to change which panel type is used to hold all the items. Here's an example:

<Window x:Class="WpfTutorialSamples.ItemsControl.ItemsControlPanelSample"

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

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

xmlns:system="clr-namespace:System;assembly=mscorlib"

Title="ItemsControlPanelSample" Height="150" Width="250">

<Grid Margin="10">

<ItemsControl>

<ItemsControl.ItemsPanel>

<ItemsPanelTemplate>

<WrapPanel />

</ItemsPanelTemplate>

</ItemsControl.ItemsPanel>

<ItemsControl.ItemTemplate>

<DataTemplate>

<Button Content="{Binding}" Margin="0,0,5,5" />

</DataTemplate>

</ItemsControl.ItemTemplate>

<system:String>Item #1</system:String>

<system:String>Item #2</system:String>

<system:String>Item #3</system:String>

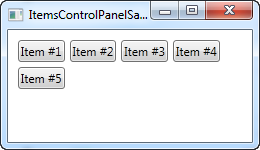
<system:String>Item #4</system:String>

<system:String>Item #5</system:String>

</ItemsControl>

</Grid>

</Window>



We specify that the ItemsControl should use a WrapPanel as its template by declaring one in the **ItemsPanelTemplate** property and just for fun, we throw in an ItemTemplate that causes the strings to be rendered as buttons. You can use any of the WPF panels, but some are more useful than others.

Another good example is the UniformGrid panel, where we can define a number of columns and then have our items neatly shown in equally-wide columns:

<Window x:Class="WpfTutorialSamples.ItemsControl.ItemsControlPanelSample"

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

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

xmlns:system="clr-namespace:System;assembly=mscorlib"

Title="ItemsControlPanelSample" Height="150" Width="250">

<Grid Margin="10">

<ItemsControl>

<ItemsControl.ItemsPanel>

<ItemsPanelTemplate>

<UniformGrid Columns="2" />

</ItemsPanelTemplate>

</ItemsControl.ItemsPanel>

<ItemsControl.ItemTemplate>

<DataTemplate>

<Button Content="{Binding}" Margin="0,0,5,5" />

</DataTemplate>

</ItemsControl.ItemTemplate>

<system:String>Item #1</system:String>

<system:String>Item #2</system:String>

<system:String>Item #3</system:String>

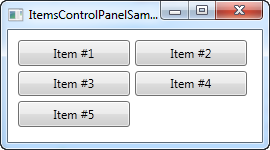
<system:String>Item #4</system:String>

<system:String>Item #5</system:String>

</ItemsControl>

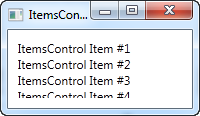
</Grid>

</Window>



## ItemsControl with scrollbars

Once you start using the ItemsControl, you might run into a very common problem: By default, the ItemsControl doesn't have any scrollbars, which means that if the content doesn't fit, it's just clipped. This can be seen by taking our first example from this article and resizing the window:



WPF makes this very easy to solve though. There are a number of possible solutions, for instance you can alter the template used by the ItemsControl to include a ScrollViewer control, but the easiest solution is to simply throw a ScrollViewer around the ItemsControl. Here's an example:

<Window x:Class="WpfTutorialSamples.ItemsControl.ItemsControlSample"

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

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

xmlns:system="clr-namespace:System;assembly=mscorlib"

Title="ItemsControlSample" Height="150" Width="200">

<Grid Margin="10">

<ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto">

<ItemsControl>

<system:String>ItemsControl Item #1</system:String>

<system:String>ItemsControl Item #2</system:String>

<system:String>ItemsControl Item #3</system:String>

<system:String>ItemsControl Item #4</system:String>

<system:String>ItemsControl Item #5</system:String>

</ItemsControl>

</ScrollViewer>

</Grid>

</Window>



I set the two visibility options to Auto, to make them only visible when needed. As you can see from the screenshot, you can now scroll through the list of items.

**List controls:**

# The ListBox control

In the last article, we had a look at the ItemsControl, which is probably the simplest list in WPF. The ListBox control is the next control in line, which adds a bit more functionality. One of the main differences is the fact that the ListBox control actually deals with selections, allowing the end-user to select one or several items from the list and automatically giving visual feedback for it.

Here's an example of a very simple ListBox control:

<Window x:Class="WpfTutorialSamples.ListBox\_control.ListBoxSample"

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

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

Title="ListBoxSample" Height="120" Width="200">

<Grid Margin="10">

<ListBox>

<ListBoxItem>ListBox Item #1</ListBoxItem>

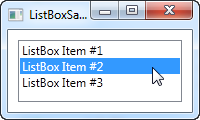
<ListBoxItem>ListBox Item #2</ListBoxItem>

<ListBoxItem>ListBox Item #3</ListBoxItem>

</ListBox>

</Grid>

</Window>



This is as simple as it gets: We declare a ListBox control, and inside of it, we declare three ListBoxItem's, each with its own text. However, since the ListBoxItem is actually a ContentControl, we can define custom content for it:

<Window x:Class="WpfTutorialSamples.ListBox\_control.ListBoxSample"

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

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

Title="ListBoxSample" Height="120" Width="200">

<Grid Margin="10">

<ListBox>

<ListBoxItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_blue.png" />

<TextBlock>ListBox Item #1</TextBlock>

</StackPanel>

</ListBoxItem>

<ListBoxItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_green.png" />

<TextBlock>ListBox Item #2</TextBlock>

</StackPanel>

</ListBoxItem>

<ListBoxItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_red.png" />

<TextBlock>ListBox Item #3</TextBlock>

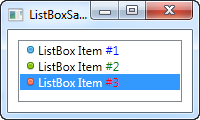
</StackPanel>

</ListBoxItem>

</ListBox>

</Grid>

</Window>



For each of the ListBoxItem's we now add a StackPanel, in which we add an Image and a TextBlock. This gives us full control of the content as well as the text rendering, as you can see from the screenshot, where different colors have been used for each of the numbers.

From the screenshot you might also notice another difference when comparing the ItemsControl to the ListBox: By default, a border is shown around the control, making it look like an actual control instead of just output.

## Data binding the ListBox

Manually defining items for the ListBox makes for a fine first example, but most of the times, your ListBox controls will be filled with items from a data source using data binding. By default, if you bind a list of items to the ListBox, their ToString() method will be used to represent each item. This is rarely what you want, but fortunately, we can easily declare a template that will be used to render each item.

I have re-used the TODO based example from the ItemsControl article, where we build a cool TODO list using a simple Code-behind class and, in this case, a ListBox control for the visual representation. Here's the example:

<Window x:Class="WpfTutorialSamples.ListBox\_control.ListBoxDataBindingSample"

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

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

Title="ListBoxDataBindingSample" Height="150" Width="300">

<Grid Margin="10">

<ListBox Name="lbTodoList" HorizontalContentAlignment="Stretch">

<ListBox.ItemTemplate>

<DataTemplate>

<Grid Margin="0,2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="100" />

</Grid.ColumnDefinitions>

<TextBlock Text="{Binding Title}" />

<ProgressBar Grid.Column="1" Minimum="0" Maximum="100" Value="{Binding Completion}" />

</Grid>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

</Grid>

</Window>

using System;

using System.Windows;

using System.Collections.Generic;

namespace WpfTutorialSamples.ListBox\_control

{

public partial class ListBoxDataBindingSample : Window

{

public ListBoxDataBindingSample()

{

InitializeComponent();

List<TodoItem> items = new List<TodoItem>();

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

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

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

lbTodoList.ItemsSource = items;

}

}

public class TodoItem

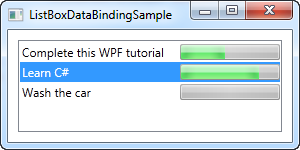
{

public string Title { get; set; }

public int Completion { get; set; }

}

}



All the magic happens in the ItemTemplate that we have defined for the ListBox. In there, we specify that each ListBox item should consist of a Grid, divided into two columns, with a TextBlock showing the title in the first and a ProgressBar showing the completion status in the second column. To get the values out, we use some very simple data binding, which is all explained in the data binding part of this tutorial.

In the Code-behind file, we have declared a very simple TodoItem class to hold each of our TODO items. In the constructor of the window, we initialize a list, add three TODO items to it and then assign it to the ItemsSource of the ListBox. The combination of the ItemsSource and the ItemTemplate we specified in the XAML part, this is all WPF need to render all of the items as a TODO list.

Please notice the **HorizontalContentAlignment** property that I set to **Stretch** on the ListBox. The default content alignment for a ListBox item is **Left**, which means that each item only takes up as much horizontal space as it needs. The result? Well, not quite what we want:



By using the Stretch alignment, each item is stretched to take up the full amount of available space, as you can see from the previous screenshot.

## Working with ListBox selection

As mentioned, a key difference between the ItemsControl and the ListBox is that the ListBox handles and displays user selection for you. Therefore, a lot of ListBox question revolves around somehow working with the selection. To help with some of these questions, I have created a bigger example, showing you some selection related tricks:

<Window x:Class="WpfTutorialSamples.ListBox\_control.ListBoxSelectionSample"

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

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

Title="ListBoxSelectionSample" Height="250" Width="450">

<DockPanel Margin="10">

<StackPanel DockPanel.Dock="Right" Margin="10,0">

<StackPanel.Resources>

<Style TargetType="Button">

<Setter Property="Margin" Value="0,0,0,5" />

</Style>

</StackPanel.Resources>

<TextBlock FontWeight="Bold" Margin="0,0,0,10">ListBox selection</TextBlock>

<Button Name="btnShowSelectedItem" Click="btnShowSelectedItem\_Click">Show selected</Button>

<Button Name="btnSelectLast" Click="btnSelectLast\_Click">Select last</Button>

<Button Name="btnSelectNext" Click="btnSelectNext\_Click">Select next</Button>

<Button Name="btnSelectCSharp" Click="btnSelectCSharp\_Click">Select C#</Button>

<Button Name="btnSelectAll" Click="btnSelectAll\_Click">Select all</Button>

</StackPanel>

<ListBox Name="lbTodoList" HorizontalContentAlignment="Stretch" SelectionMode="Extended" SelectionChanged="lbTodoList\_SelectionChanged">

<ListBox.ItemTemplate>

<DataTemplate>

<Grid Margin="0,2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="100" />

</Grid.ColumnDefinitions>

<TextBlock Text="{Binding Title}" />

<ProgressBar Grid.Column="1" Minimum="0" Maximum="100" Value="{Binding Completion}" />

</Grid>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

</DockPanel>

</Window>

using System;

using System.Windows;

using System.Collections.Generic;

namespace WpfTutorialSamples.ListBox\_control

{

public partial class ListBoxSelectionSample : Window

{

public ListBoxSelectionSample()

{

InitializeComponent();

List<TodoItem> items = new List<TodoItem>();

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

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

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

lbTodoList.ItemsSource = items;

}

private void lbTodoList\_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e)

{

if(lbTodoList.SelectedItem != null)

this.Title = (lbTodoList.SelectedItem as TodoItem).Title;

}

private void btnShowSelectedItem\_Click(object sender, RoutedEventArgs e)

{

foreach(object o in lbTodoList.SelectedItems)

MessageBox.Show((o as TodoItem).Title);

}

private void btnSelectLast\_Click(object sender, RoutedEventArgs e)

{

lbTodoList.SelectedIndex = lbTodoList.Items.Count - 1;

}

private void btnSelectNext\_Click(object sender, RoutedEventArgs e)

{

int nextIndex = 0;

if((lbTodoList.SelectedIndex >= 0) && (lbTodoList.SelectedIndex < (lbTodoList.Items.Count - 1)))

nextIndex = lbTodoList.SelectedIndex + 1;

lbTodoList.SelectedIndex = nextIndex;

}

private void btnSelectCSharp\_Click(object sender, RoutedEventArgs e)

{

foreach(object o in lbTodoList.Items)

{

if((o is TodoItem) && ((o as TodoItem).Title.Contains("C#")))

{

lbTodoList.SelectedItem = o;

break;

}

}

}

private void btnSelectAll\_Click(object sender, RoutedEventArgs e)

{

foreach(object o in lbTodoList.Items)

lbTodoList.SelectedItems.Add(o);

}

}

public class TodoItem

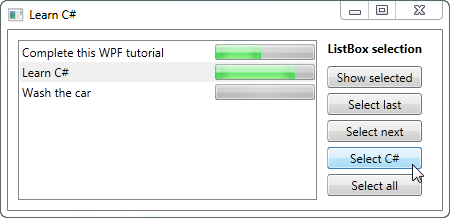
{

public string Title { get; set; }

public int Completion { get; set; }

}

}



As you can see, I have defined a range of buttons to the right of the ListBox, to either get or manipulate the selection. I've also changed the **SelectionMode** to **Extended**, to allow for the selection of multiple items. This can be done either programmatically, as I do in the example, or by the end-user, by holding down **[Ctrl]** or **[Shift]** while clicking on the items.

For each of the buttons, I have defined a click handler in the Code-behind. Each action should be pretty self-explanatory and the C# code used is fairly simple, but if you're still in doubt, try running the example on your own machine and test out the various possibilities in the example.

**List controls:**

# The ComboBox control

The ComboBox control is in many ways like the ListBox control, but takes up a lot less space, because the list of items is hidden when not needed. The ComboBox control is used many places in Windows, but to make sure that everyone knows how it looks and works, we'll jump straight into a simple example:

<Window x:Class="WpfTutorialSamples.ComboBox\_control.ComboBoxSample"

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

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

Title="ComboBoxSample" Height="150" Width="200">

<StackPanel Margin="10">

<ComboBox>

<ComboBoxItem>ComboBox Item #1</ComboBoxItem>

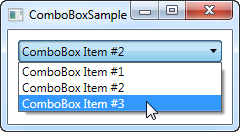
<ComboBoxItem IsSelected="True">ComboBox Item #2</ComboBoxItem>

<ComboBoxItem>ComboBox Item #3</ComboBoxItem>

</ComboBox>

</StackPanel>

</Window>



In the screenshot, I have activated the control by clicking it, causing the list of items to be displayed. As you can see from the code, the ComboBox, in its simple form, is very easy to use. All I've done here is manually add some items, making one of them the default selected item by setting the IsSelected property on it.

## Custom content

In the first example we only showed text in the items, which is pretty common for the ComboBox control, but since the ComboBoxItem is a ContentControl, we can actually use pretty much anything as content. Let's try making a slightly more sophisticated list of items:

<Window x:Class="WpfTutorialSamples.ComboBox\_control.ComboBoxCustomContentSample"

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

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

Title="ComboBoxCustomContentSample" Height="150" Width="200">

<StackPanel Margin="10">

<ComboBox>

<ComboBoxItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_red.png" />

<TextBlock Foreground="Red">Red</TextBlock>

</StackPanel>

</ComboBoxItem>

<ComboBoxItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_green.png" />

<TextBlock Foreground="Green">Green</TextBlock>

</StackPanel>

</ComboBoxItem>

<ComboBoxItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_blue.png" />

<TextBlock Foreground="Blue">Blue</TextBlock>

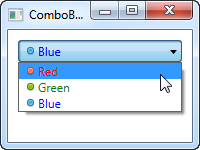
</StackPanel>

</ComboBoxItem>

</ComboBox>

</StackPanel>

</Window>



For each of the ComboBoxItem's we now add a StackPanel, in which we add an Image and a TextBlock. This gives us full control of the content as well as the text rendering, as you can see from the screenshot, where both text color and image indicates a color value.

## Data binding the ComboBox

As you can see from the first examples, manually defining the items of a ComboBox control is easy using XAML, but you will likely soon run into a situation where you need the items to come from some kind of data source, like a database or just an in-memory list. Using WPF data binding and a custom template, we can easily render a list of colors, including a preview of the color:

<Window x:Class="WpfTutorialSamples.ComboBox\_control.ComboBoxDataBindingSample"

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

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

Title="ComboBoxDataBindingSample" Height="200" Width="200">

<StackPanel Margin="10">

<ComboBox Name="cmbColors">

<ComboBox.ItemTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal">

<Rectangle Fill="{Binding Name}" Width="16" Height="16" Margin="0,2,5,2" />

<TextBlock Text="{Binding Name}" />

</StackPanel>

</DataTemplate>

</ComboBox.ItemTemplate>

</ComboBox>

</StackPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Media;

namespace WpfTutorialSamples.ComboBox\_control

{

public partial class ComboBoxDataBindingSample : Window

{

public ComboBoxDataBindingSample()

{

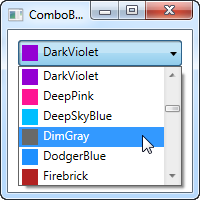
InitializeComponent();

cmbColors.ItemsSource = typeof(Colors).GetProperties();

}

}

}



It's actually quite simple: In the Code-behind, I obtain a list of all the colors using a Reflection based approach with the Colors class. I assign it to the **ItemsSource** property of the ComboBox, which then renders each color using the template I have defined in the XAML part.

Each item, as defined by the ItemTemplate, consists of a StackPanel with a Rectangle and a TextBlock, each bound to the color value. This gives us a complete list of colors, with minimal effort - and it looks pretty good too, right?

## IsEditable

In the first examples, the user was only able to select from our list of items, but one of the cool things about the ComboBox is that it supports the possibility of letting the user both select from a list of items or enter their own value. This is extremely useful in situations where you want to help the user by giving them a pre-defined set of options, while still giving them the option to manually enter the desired value. This is all controlled by the **IsEditable** property, which changes the behavior and look of the ComboBox quite a bit:

<Window x:Class="WpfTutorialSamples.ComboBox\_control.ComboBoxEditableSample"

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

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

Title="ComboBoxEditableSample" Height="150" Width="200">

<StackPanel Margin="10">

<ComboBox IsEditable="True">

<ComboBoxItem>ComboBox Item #1</ComboBoxItem>

<ComboBoxItem>ComboBox Item #2</ComboBoxItem>

<ComboBoxItem>ComboBox Item #3</ComboBoxItem>

</ComboBox>

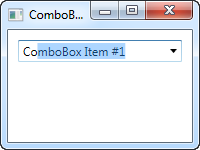
</StackPanel>

</Window>



As you can see, I can enter a completely different value or pick one from the list. If picked from the list, it simply overwrites the text of the ComboBox.

As a lovely little bonus, the ComboBox will automatically try to help the user select an existing value when the user starts typing, as you can see from the next screenshot, where I just started typing "Co":



By default, the matching is not case-sensitive but you can make it so by setting the **IsTextSearchCaseSensitive** to True. If you don't want this auto complete behavior at all, you can disable it by setting the **IsTextSearchEnabled** to False.

## Working with ComboBox selection

A key part of using the ComboBox control is to be able to read the user selection, and even control it with code. In the next example, I've re-used the data bound ComboBox example, but added some buttons for controlling the selection. I've also used the **SelectionChanged** event to capture when the selected item is changed, either by code or by the user, and act on it.

Here's the sample:

<Window x:Class="WpfTutorialSamples.ComboBox\_control.ComboBoxSelectionSample"

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

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

Title="ComboBoxSelectionSample" Height="125" Width="250">

<StackPanel Margin="10">

<ComboBox Name="cmbColors" SelectionChanged="cmbColors\_SelectionChanged">

<ComboBox.ItemTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal">

<Rectangle Fill="{Binding Name}" Width="16" Height="16" Margin="0,2,5,2" />

<TextBlock Text="{Binding Name}" />

</StackPanel>

</DataTemplate>

</ComboBox.ItemTemplate>

</ComboBox>

<WrapPanel Margin="15" HorizontalAlignment="Center">

<Button Name="btnPrevious" Click="btnPrevious\_Click" Width="55">Previous</Button>

<Button Name="btnNext" Click="btnNext\_Click" Margin="5,0" Width="55">Next</Button>

<Button Name="btnBlue" Click="btnBlue\_Click" Width="55">Blue</Button>

</WrapPanel>

</StackPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Reflection;

using System.Windows;

using System.Windows.Media;

namespace WpfTutorialSamples.ComboBox\_control

{

public partial class ComboBoxSelectionSample : Window

{

public ComboBoxSelectionSample()

{

InitializeComponent();

cmbColors.ItemsSource = typeof(Colors).GetProperties();

}

private void btnPrevious\_Click(object sender, RoutedEventArgs e)

{

if(cmbColors.SelectedIndex > 0)

cmbColors.SelectedIndex = cmbColors.SelectedIndex - 1;

}

private void btnNext\_Click(object sender, RoutedEventArgs e)

{

if(cmbColors.SelectedIndex < cmbColors.Items.Count-1)

cmbColors.SelectedIndex = cmbColors.SelectedIndex + 1;

}

private void btnBlue\_Click(object sender, RoutedEventArgs e)

{

cmbColors.SelectedItem = typeof(Colors).GetProperty("Blue");

}

private void cmbColors\_SelectionChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e)

{

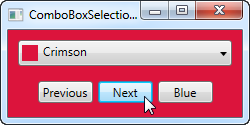
Color selectedColor = (Color)(cmbColors.SelectedItem as PropertyInfo).GetValue(null, null);

this.Background = new SolidColorBrush(selectedColor);

}

}

}



The interesting part of this example is the three event handlers for our three buttons, as well as the **SelectionChanged** event handler. In the first two, we select the previous or the next item by reading the **SelectedIndex** property and then subtracting or adding one to it. Pretty simple and easy to work with.

In the third event handler, we use the SelectedItem to select a specific item based on the value. I do a bit of extra work here (using .NET reflection), because the ComboBox is bound to a list of properties, each being a color, instead of a simple list of colors, but basically it's all about giving the value contained by one of the items to the **SelectedItem** property.

In the fourth and last event handler, I respond to the selected item being changed. When that happens, I read the selected color (once again using Reflection, as described above) and then use the selected color to create a new background brush for the Window. The effect can be seen on the screenshot.

If you're working with an editable ComboBox (IsEditable property set to true), you can read the **Text** property to know the value the user has entered or selected.

# A simple ListView example

The WPF ListView control is very bare minimum in its most simple form. In fact, it will look a whole lot like the WPF ListBox, until you start adding specialized views to it. That's not so strange, since a ListView inherits directly from the ListBox control. So, a default ListView is actually just a ListBox, with a different selection mode (more on that later).

Let's try creating a ListView in its most simple form:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewBasicSample"

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

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

Title="ListViewBasicSample" Height="200" Width="200">

<Grid>

<ListView Margin="10">

<ListViewItem>A ListView</ListViewItem>

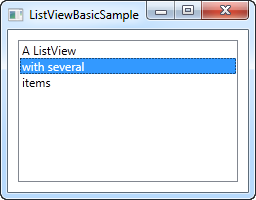
<ListViewItem IsSelected="True">with several</ListViewItem>

<ListViewItem>items</ListViewItem>

</ListView>

</Grid>

</Window>



This is pretty much as simple as it gets, using manually specified ListViewItem to fill the list and with nothing but a text label representing each item - a bare minimum WPF ListView control.

## ListViewItem with an image

Because of the look-less nature of WPF, specifying an image for a ListViewItem isn't just about assigning an image ID or key to a property. Instead, you take full control of it and specify the controls needed to render both image and text in the ListViewItem. Here's an example:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewBasicSample"

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

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

Title="ListViewBasicSample" Height="200" Width="200">

<Grid>

<ListView Margin="10">

<ListViewItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_green.png" Margin="0,0,5,0" />

<TextBlock>Green</TextBlock>

</StackPanel>

</ListViewItem>

<ListViewItem>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_blue.png" Margin="0,0,5,0" />

<TextBlock>Blue</TextBlock>

</StackPanel>

</ListViewItem>

<ListViewItem IsSelected="True">

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_red.png" Margin="0,0,5,0" />

<TextBlock>Red</TextBlock>

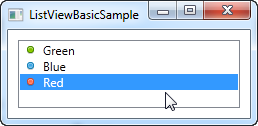
</StackPanel>

</ListViewItem>

</ListView>

</Grid>

</Window>



What we do here is very simple. Because the ListViewItem derives from the ContentControl class, we can specify a WPF control as its content. In this case, we use a StackPanel, which has an Image and a TextBlock as its child controls.

**The ListView control:**

# ListView, data binding and ItemTemplate

In the previous article, we manually populated a ListView control through XAML code, but in WPF, it's all about data binding. The concept of data binding is explained in detail in another part of this tutorial, but generally speaking it's about separating data from layout. So, let's try binding some data to a ListView:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewDataBindingSample"

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

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

Title="ListViewDataBindingSample" Height="300" Width="300">

<Grid>

<ListView Margin="10" Name="lvDataBinding"></ListView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

namespace WpfTutorialSamples.ListView\_control

{

public partial class ListViewDataBindingSample : Window

{

public ListViewDataBindingSample()

{

InitializeComponent();

List<User> items = new List<User>();

items.Add(new User() { Name = "John Doe", Age = 42 });

items.Add(new User() { Name = "Jane Doe", Age = 39 });

items.Add(new User() { Name = "Sammy Doe", Age = 13 });

lvDataBinding.ItemsSource = items;

}

}

public class User

{

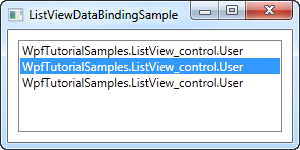
public string Name { get; set; }

public int Age { get; set; }

}

}

We populate a list of our own User objects, each user having a name and an age. The data binding process happens automatically as soon as we assign the list to the ItemsSource property of the ListView, but the result is a bit discouraging:



Each user is represented by their type name in the ListView. This is to be expected, because .NET doesn't have a clue about how you want your data to be displayed, so it just calls the ToString() method on each object and uses that to represent the item.

We can use that to our advantage and override the ToString() method, to get a more meaningful output. Try replacing the User class with this version:

public class User

{

public string Name { get; set; }

public int Age { get; set; }

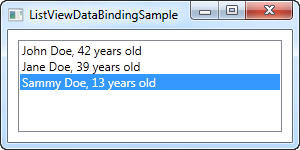
public override string ToString()

{

return this.Name + ", " + this.Age + " years old";

}

}



This is a much more user friendly display and will do just fine in some cases, but relying on a simple string is not that flexible. Perhaps you want a part of the text to be bold or another color? Perhaps you want an image? Fortunately, WPF makes all of this very simple using templates.

## ListView with an ItemTemplate

WPF is all about templating, so specifying a data template for the ListView is very easy. In this example, we'll do a bunch of custom formatting in each item, just to show you how flexible this makes the WPF ListView.

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewItemTemplateSample"

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

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

Title="ListViewItemTemplateSample" Height="150" Width="350">

<Grid>

<ListView Margin="10" Name="lvDataBinding">

<ListView.ItemTemplate>

<DataTemplate>

<WrapPanel>

<TextBlock Text="Name: " />

<TextBlock Text="{Binding Name}" FontWeight="Bold" />

<TextBlock Text=", " />

<TextBlock Text="Age: " />

<TextBlock Text="{Binding Age}" FontWeight="Bold" />

<TextBlock Text=" (" />

<TextBlock Text="{Binding Mail}" TextDecorations="Underline" Foreground="Blue" Cursor="Hand" />

<TextBlock Text=")" />

</WrapPanel>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

namespace WpfTutorialSamples.ListView\_control

{

public partial class ListViewItemTemplateSample : Window

{

public ListViewItemTemplateSample()

{

InitializeComponent();

List<User> items = new List<User>();

items.Add(new User() { Name = "John Doe", Age = 42, Mail = "john@doe-family.com" });

items.Add(new User() { Name = "Jane Doe", Age = 39, Mail = "jane@doe-family.com" });

items.Add(new User() { Name = "Sammy Doe", Age = 13, Mail = "sammy.doe@gmail.com" });

lvDataBinding.ItemsSource = items;

}

}

public class User

{

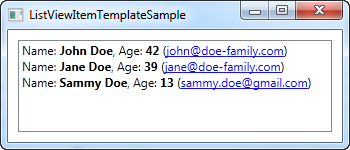
public string Name { get; set; }

public int Age { get; set; }

public string Mail { get; set; }

}

}



We use a bunch of TextBlock controls to build each item, where we put part of the text in bold. For the e-mail address, which we added to this example, we underline it, give it a blue color and change the mouse cursor, to make it behave like a hyperlink.

# ListView with a GridView

In the previous ListView articles, we have used the most basic version of the WPF ListView, which is the one without a custom View specified. This results in a ListView that acts very much like the WPF ListBox, with some subtle differences. The real power lies in the views though and WPF comes with one specialized view built-in: The GridView.

By using the GridView, you can get several columns of data in your ListView, much like you see it in Windows Explorer. Just to make sure that everyone can visualize it, we'll start off with a basic example:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewGridViewSample"

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

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

Title="ListViewGridViewSample" Height="200" Width="400">

<Grid>

<ListView Margin="10" Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

<GridViewColumn Header="Mail" Width="150" DisplayMemberBinding="{Binding Mail}" />

</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

namespace WpfTutorialSamples.ListView\_control

{

public partial class ListViewGridViewSample : Window

{

public ListViewGridViewSample()

{

InitializeComponent();

List<User> items = new List<User>();

items.Add(new User() { Name = "John Doe", Age = 42, Mail = "john@doe-family.com" });

items.Add(new User() { Name = "Jane Doe", Age = 39, Mail = "jane@doe-family.com" });

items.Add(new User() { Name = "Sammy Doe", Age = 7, Mail = "sammy.doe@gmail.com" });

lvUsers.ItemsSource = items;

}

}

public class User

{

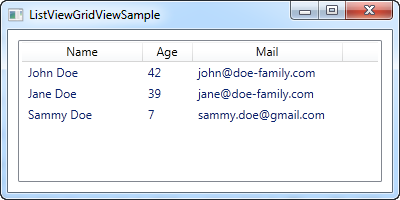
public string Name { get; set; }

public int Age { get; set; }

public string Mail { get; set; }

}

}



So, we use the same User class as previously, for test data, which we then bind to the ListView. This is all the same as we saw in previous chapters, but as you can see from the screenshot, the layout is very different. This is the power of data binding - the same data, but presented in a completely different way, just by changing the markup.

In the markup (XAML), we define a View for the ListView, using the ListView.View property. We set it to a GridView, which is currently the only included view type in WPF (you can easily create your own though!). The GridView is what gives us the column-based view that you see on the screenshot.

Inside of the GridView, we define three columns, one for each of the pieces of data that we wish to show. The **Header** property is used to specify the text that we would like to show for the column and then we use the **DisplayMemberBinding** property to bind the value to a property from our User class.

## Templated cell content

Using the **DisplayMemberBinding**property is pretty much limited to outputting simple strings, with no custom formatting at all, but the WPF ListView is much more flexible than that. By specifying a **CellTemplate**, we take full control of how the content is rendered within the specific column cell.

The GridViewColumn will use the DisplayMemberBinding as its first priority, if it's present. The second choice will be the CellTemplate property, which we'll use for this example:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewGridViewCellTemplateSample"

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

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

Title="ListViewGridViewCellTemplateSample" Height="200" Width="400">

<Grid>

<ListView Margin="10" Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

<GridViewColumn Header="Mail" Width="150">

<GridViewColumn.CellTemplate>

<DataTemplate>

<TextBlock Text="{Binding Mail}" TextDecorations="Underline" Foreground="Blue" Cursor="Hand" />

</DataTemplate>

</GridViewColumn.CellTemplate>

</GridViewColumn>

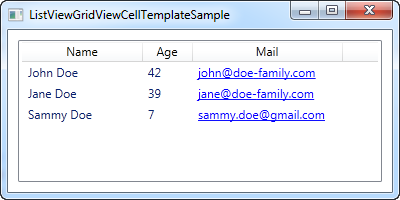
</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

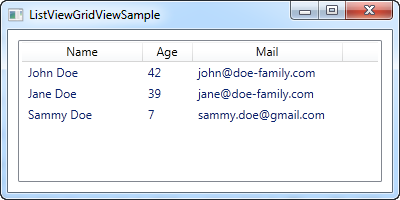


Please notice: The Code-behind code for this example is the same as the one used for the first example in this article.

We specify a custom **CellTemplate** for the last column, where we would like to do some special formatting for the e-mail addresses. For the other columns, where we just want basic text output, we stick with the **DisplayMemberBinding**, simply because it requires way less markup.

# How-to: ListView with left aligned column names

In a normal ListView, the column names are left aligned, but for some reason, Microsoft decided to center the names by default in the WPF ListView. In many cases this will make your application look out-of-style compared to other Windows applications. This is how the ListView will look in WPF by **default**:



Let's try changing that to left aligned column names. Unfortunately, there are no direct properties on the GridViewColumn to control this, but fortunately that doesn't mean that it can't be changed.

Using a Style, targeted at the GridViewColumHeader, which is the element used to show the header of a GridViewColumn, we can change the HorizontalAlignment property. In this case it defaults to Center, but we can change it to Left, to accomplish what we want:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewGridViewSample"

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

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

Title="ListViewGridViewSample" Height="200" Width="400">

<Grid>

<ListView Margin="10" Name="lvUsers">

<ListView.Resources>

<Style TargetType="{x:Type GridViewColumnHeader}">

<Setter Property="HorizontalContentAlignment" Value="Left" />

</Style>

</ListView.Resources>

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

<GridViewColumn Header="Mail" Width="150" DisplayMemberBinding="{Binding Mail}" />

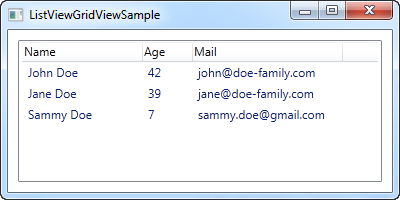
</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>



The part that does all the work for us, is the Style defined in the Resources of the ListView:

<Style TargetType="{x:Type GridViewColumnHeader}">

<Setter Property="HorizontalContentAlignment" Value="Left" />

</Style>

## Local or global style

By defining the Style within the control itself, it only applies to this particular ListView. In many cases you might like to make it apply to all the ListViews within the same Window/Page or perhaps even globally across the application. You can do this by either copying the style to the Window resources or the Application resources. Here's the same example, where we have applied the style to the entire Window instead of just the particular ListView:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewGridViewSample"

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

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

Title="ListViewGridViewSample" Height="200" Width="400">

<Window.Resources>

<Style TargetType="{x:Type GridViewColumnHeader}">

<Setter Property="HorizontalContentAlignment" Value="Left" />

</Style>

</Window.Resources>

<Grid>

<ListView Margin="10" Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

<GridViewColumn Header="Mail" Width="150" DisplayMemberBinding="{Binding Mail}" />

</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

In case you want another alignment, e.g. right alignment, you just change the value of the style like this:

<Setter Property="HorizontalContentAlignment" Value="Right" />

**The ListView control:**

# ListView grouping

As we already talked about earlier, the WPF ListView is very flexible. Grouping is yet another thing that it supports out of the box, and it's both easy to use and extremely customizable. Let's jump straight into the first example, then I'll explain it and afterwards we can use the standard WPF tricks to customize the appearance even further.

For this article, I've borrowed the sample code from a previous article and then expanded on it to support grouping. It looks like this:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewGroupSample"

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

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

Title="ListViewGroupSample" Height="300" Width="300">

<Grid Margin="10">

<ListView Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

</GridView>

</ListView.View>

<ListView.GroupStyle>

<GroupStyle>

<GroupStyle.HeaderTemplate>

<DataTemplate>

<TextBlock FontWeight="Bold" FontSize="14" Text="{Binding Name}"/>

</DataTemplate>

</GroupStyle.HeaderTemplate>

</GroupStyle>

</ListView.GroupStyle>

</ListView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Data;

namespace WpfTutorialSamples.ListView\_control

{

public partial class ListViewGroupSample : Window

{

public ListViewGroupSample()

{

InitializeComponent();

List<User> items = new List<User>();

items.Add(new User() { Name = "John Doe", Age = 42, Sex = SexType.Male });

items.Add(new User() { Name = "Jane Doe", Age = 39, Sex = SexType.Female });

items.Add(new User() { Name = "Sammy Doe", Age = 13, Sex = SexType.Male });

lvUsers.ItemsSource = items;

CollectionView view = (CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource);

PropertyGroupDescription groupDescription = new PropertyGroupDescription("Sex");

view.GroupDescriptions.Add(groupDescription);

}

}

public enum SexType { Male, Female };

public class User

{

public string Name { get; set; }

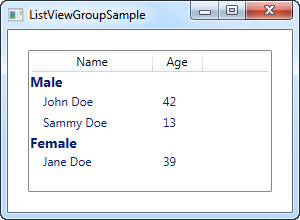
public int Age { get; set; }

public string Mail { get; set; }

public SexType Sex { get; set; }

}

}



In XAML, I have added a GroupStyle to the ListView, in which I define a template for the header of each group. It consists of a TextBlock control, where I've used a slightly larger and bold text to show that it's a group - as we'll see later on, this can of course be customized a lot more. The TextBlock Text property is bound to a Name property, **but please be aware that this is not the Name property on the data object (in this case the User class)**. Instead, it is the name of the group, as assigned by WPF, based on the property we use to divide the objects into groups.

In Code-behind, we do the same as we did before: We create a list and add some User objects to it and then we bind the list to the ListView - nothing new there, except for the new Sex property that I've added, which tells whether the user is male or female.

After assigning an ItemsSource, we use this to get a CollectionView that the ListView creates for us. This specialized View instance contains a lot of possibilities, including the ability to group the items. We use this by adding a so-called PropertyGroupDescription to the GroupDescriptions of the view. This basically tells WPF to group by a specific property on the data objects, in this case the Sex property.

## Customizing the group header

The above example was great for showing the basics of ListView grouping, but the look was a tad boring, so let's exploit the fact that WPF lets us define our own templates and spice things up. A common request is to be able to collapse and expand the group, and while WPF doesn't provide this behavior by default, it's somewhat easy to implement yourself. We'll do it by completely re-templating the group container.

It might look a bit cumbersome, but the principles used are somewhat simple and you will see them in other situations when you customize the WPF controls. Here's the code:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewCollapseExpandGroupSample"

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

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

Title="ListViewCollapseExpandGroupSample" Height="300" Width="300">

<Grid Margin="10">

<ListView Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

</GridView>

</ListView.View>

<ListView.GroupStyle>

<GroupStyle>

<GroupStyle.ContainerStyle>

<Style TargetType="{x:Type GroupItem}">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate>

<Expander IsExpanded="True">

<Expander.Header>

<StackPanel Orientation="Horizontal">

<TextBlock Text="{Binding Name}" FontWeight="Bold" Foreground="Gray" FontSize="22" VerticalAlignment="Bottom" />

<TextBlock Text="{Binding ItemCount}" FontSize="22" Foreground="Green" FontWeight="Bold" FontStyle="Italic" Margin="10,0,0,0" VerticalAlignment="Bottom" />

<TextBlock Text=" item(s)" FontSize="22" Foreground="Silver" FontStyle="Italic" VerticalAlignment="Bottom" />

</StackPanel>

</Expander.Header>

<ItemsPresenter />

</Expander>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</GroupStyle.ContainerStyle>

</GroupStyle>

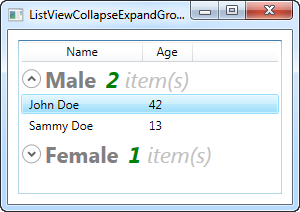
</ListView.GroupStyle>

</ListView>

</Grid>

</Window>

**The Code-behind is exactly the same as used in the first example - feel free to scroll up and grab it.**



Now our groups look a bit more exciting, and they even include an expander button, that will toggle the visibility of the group items when you click it (that's why the single female user is not visible on the screenshot - I collapsed that particular group). By using the ItemCount property that the group exposes, we can even show how many items each group currently consists of.

As you can see, it requires a bit more markup than we're used to, but this example also goes a bit beyond what we usually do, so that seems fair. When you read through the code, you will quickly realize that many of the lines are just common elements like style and template.

**The ListView control:**

# ListView sorting

In the last chapter we saw how we could group items in the WPF ListView by accessing the View instance of the ListView and then adding a group description. Applying sorting to a ListView is just as easy, and most of the process is exactly the same. Let's try a simple example where we sort the user objects by their age:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewSortingSample"

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

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

Title="ListViewSortingSample" Height="200" Width="300">

<Grid Margin="10">

<ListView Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Windows;

using System.Windows.Data;

namespace WpfTutorialSamples.ListView\_control

{

public partial class ListViewSortingSample : Window

{

public ListViewSortingSample()

{

InitializeComponent();

List<User> items = new List<User>();

items.Add(new User() { Name = "John Doe", Age = 42 });

items.Add(new User() { Name = "Jane Doe", Age = 39 });

items.Add(new User() { Name = "Sammy Doe", Age = 13 });

items.Add(new User() { Name = "Donna Doe", Age = 13 });

lvUsers.ItemsSource = items;

CollectionView view = (CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource);

view.SortDescriptions.Add(new SortDescription("Age", ListSortDirection.Ascending));

}

}

public class User

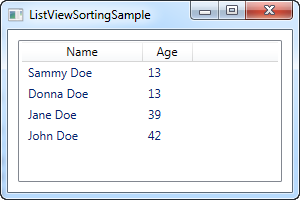
{

public string Name { get; set; }

public int Age { get; set; }

}

}



The XAML looks just like a previous example, where we simply have a couple of columns for displaying information about the user - nothing new here.

In the Code-behind, we once again create a list of User objects, which we then assign as the ItemsSource of the ListView. Once we've done that, we use the ItemsSource property to get the CollectionView instance that the ListView automatically creates for us and which we can use to manipulate how the ListView shows our objects.

With the view object in our hand, we add a new SortDescription to it, specifying that we want our list sorted by the Age property, in ascending order. As you can see from the screenshot, this works perfectly well - the list is sorted by age, instead of being in the same order as the items were added.

## Multiple sort criteria

As shown in the first example, sorting is very easy, but on the screenshot you'll see that Sammy comes before Donna. They have the same age, so in this case, WPF will just use the order in which they were added. Fortunately, WPF lets us specify as many sort criteria as we want. In the example above, try changing the view-related code into something like this:

CollectionView view = (CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource);

view.SortDescriptions.Add(new SortDescription("Age", ListSortDirection.Ascending));

view.SortDescriptions.Add(new SortDescription("Name", ListSortDirection.Ascending));



Now the view will be sorted using age first, and when two identical values are found, the name will be used as a secondary sorting parameter.

## Summary

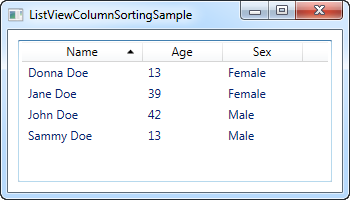
It's very easy to sort the contents of a ListView, as seen in the above examples, but so far, all the sorting is decided by the programmer and not the end-user. In the next article I'll give you a how-to article showing you how to let the user decide the sorting by clicking on the columns, as seen in Windows.

# How-to: ListView with column sorting

In the last chapter we saw how we could easily sort a ListView from Code-behind, and while this will suffice for some cases, it doesn't allow the end-user to decide on the sorting. Besides that, there was no indication on which column the ListView was sorted by. In Windows, and in many user interfaces in general, it's common to illustrate sort directions in a list by drawing a triangle next to the column name currently used to sort by.

In this how-to article, I'll give you a practical solution that gives us all of the above, but please bear in mind that some of the code here goes a bit beyond what we have learned so far - that's why it has the "how-to" label.

This article builds upon the previous one, but I'll still explain each part as we go along. Here's our goal - a ListView with column sorting, including visual indication of sort field and direction. The user simply clicks a column to sort by and if the same column is clicked again, the sort direction is reversed. Here's how it looks:



## The XAML

The first thing we need is some XAML to define our user interface. It currently looks like this:

<Window x:Class="WpfTutorialSamples.ListView\_control.ListViewColumnSortingSample"

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

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

Title="ListViewColumnSortingSample" Height="200" Width="350">

<Grid Margin="10">

<ListView Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Width="120" DisplayMemberBinding="{Binding Name}">

<GridViewColumn.Header>

<GridViewColumnHeader Tag="Name" Click="lvUsersColumnHeader\_Click">Name</GridViewColumnHeader>

</GridViewColumn.Header>

</GridViewColumn>

<GridViewColumn Width="80" DisplayMemberBinding="{Binding Age}">

<GridViewColumn.Header>

<GridViewColumnHeader Tag="Age" Click="lvUsersColumnHeader\_Click">Age</GridViewColumnHeader>

</GridViewColumn.Header>

</GridViewColumn>

<GridViewColumn Width="80" DisplayMemberBinding="{Binding Sex}">

<GridViewColumn.Header>

<GridViewColumnHeader Tag="Sex" Click="lvUsersColumnHeader\_Click">Sex</GridViewColumnHeader>

</GridViewColumn.Header>

</GridViewColumn>

</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

Notice how I have specified headers for each of the columns using an actual GridViewColumnHeader element instead of just specifying a string. This is done so that I may set additional properties, in this case the **Tag** property as well as the **Click** event.

The **Tag** property is used to hold the field name that will be used to sort by, if this particular column is clicked. This is done in the lvUsersColumnHeader\_Click event that each of the columns subscribes to.

That was the key concepts of the XAML. Besides that, we bind to our Code-behind properties Name, Age and Sex, which we'll discuss now.

## The Code-behind

In Code-behind, there are quite a few things happening. I use a total of three classes, which you would normally divide up into individual files, but for convenience, I have kept them in the same file, giving us a total of ~100 lines. First the code and then I'll explain how it works:

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Media;

namespace WpfTutorialSamples.ListView\_control

{

public partial class ListViewColumnSortingSample : Window

{

private GridViewColumnHeader listViewSortCol = null;

private SortAdorner listViewSortAdorner = null;

public ListViewColumnSortingSample()

{

InitializeComponent();

List<User> items = new List<User>();

items.Add(new User() { Name = "John Doe", Age = 42, Sex = SexType.Male });

items.Add(new User() { Name = "Jane Doe", Age = 39, Sex = SexType.Female });

items.Add(new User() { Name = "Sammy Doe", Age = 13, Sex = SexType.Male });

items.Add(new User() { Name = "Donna Doe", Age = 13, Sex = SexType.Female });

lvUsers.ItemsSource = items;

}

private void lvUsersColumnHeader\_Click(object sender, RoutedEventArgs e)

{

GridViewColumnHeader column = (sender as GridViewColumnHeader);

string sortBy = column.Tag.ToString();

if(listViewSortCol != null)

{

AdornerLayer.GetAdornerLayer(listViewSortCol).Remove(listViewSortAdorner);

lvUsers.Items.SortDescriptions.Clear();

}

ListSortDirection newDir = ListSortDirection.Ascending;

if(listViewSortCol == column && listViewSortAdorner.Direction == newDir)

newDir = ListSortDirection.Descending;

listViewSortCol = column;

listViewSortAdorner = new SortAdorner(listViewSortCol, newDir);

AdornerLayer.GetAdornerLayer(listViewSortCol).Add(listViewSortAdorner);

lvUsers.Items.SortDescriptions.Add(new SortDescription(sortBy, newDir));

}

}

public enum SexType { Male, Female };

public class User

{

public string Name { get; set; }

public int Age { get; set; }

public string Mail { get; set; }

public SexType Sex { get; set; }

}

public class SortAdorner : Adorner

{

private static Geometry ascGeometry =

Geometry.Parse("M 0 4 L 3.5 0 L 7 4 Z");

private static Geometry descGeometry =

Geometry.Parse("M 0 0 L 3.5 4 L 7 0 Z");

public ListSortDirection Direction { get; private set; }

public SortAdorner(UIElement element, ListSortDirection dir)

: base(element)

{

this.Direction = dir;

}

protected override void OnRender(DrawingContext drawingContext)

{

base.OnRender(drawingContext);

if(AdornedElement.RenderSize.Width < 20)

return;

TranslateTransform transform = new TranslateTransform

(

AdornedElement.RenderSize.Width - 15,

(AdornedElement.RenderSize.Height - 5) / 2

);

drawingContext.PushTransform(transform);

Geometry geometry = ascGeometry;

if(this.Direction == ListSortDirection.Descending)

geometry = descGeometry;

drawingContext.DrawGeometry(Brushes.Black, null, geometry);

drawingContext.Pop();

}

}

}

Allow me to start from the bottom and then work my way up while explaining what happens. The last class in the file is an Adorner class called **SortAdorner**. All this little class does is to draw a triangle, either pointing up or down, depending on the sort direction. WPF uses the concept of adorners to allow you to paint stuff over other controls, and this is exactly what we want here: The ability to draw a sorting triangle on top of our ListView column header.

The **SortAdorner** works by defining two **Geometry** objects, which are basically used to describe 2D shapes - in this case a triangle with the tip pointing up and one with the tip pointing down. The Geometry.Parse() method uses the list of points to draw the triangles, which will be explained more thoroughly in a later article.

The **SortAdorner** is aware of the sort direction, because it needs to draw the proper triangle, but is not aware of the field that we order by - this is handled in the UI layer.

The **User** class is just a basic information class, used to contain information about a user. Some of this information is used in the UI layer, where we bind to the Name, Age and Sex properties.

In the Window class, we have two methods: The constructor where we build a list of users and assign it to the ItemsSource of our ListView, and then the more interesting click event handler that will be hit when the user clicks a column. In the top of the class, we have defined two private variables: listViewSortCol and listViewSortAdorner. These will help us keep track of which column we're currently sorting by and the adorner we placed to indicate it.

In the lvUsersColumnHeader\_Click event handler, we start off by getting a reference to the column that the user clicked. With this, we can decide which property on the User class to sort by, simply by looking at the Tag property that we defined in XAML. We then check if we're already sorting by a column - if that is the case, we remove the adorner and clear the current sort descriptions.

After that, we're ready to decide the direction. The default is ascending, but we do a check to see if we're already sorting by the column that the user clicked - if that is the case, we change the direction to descending.

In the end, we create a new SortAdorner, passing in the column that it should be rendered on, as well as the direction. We add this to the AdornerLayer of the column header, and at the very end, we add a SortDescription to the ListView, to let it know which property to sort by and in which direction.

**The ListView control:**

# ListView filtering

We've already done several different things with the ListView, like grouping and sorting, but another very useful ability is filtering. Obviously, you could just limit the items you add to the ListView in the first place, but often you would need to filter the ListView dynamically, in runtime, usually based on a user entered filter string. Luckily for us, the view mechanisms of the ListView also make it easy to do just that, like we saw it with sorting and grouping.

Filtering is actually quite easy to do, so let's jump straight into an example, and then we'll discuss it afterwards:

<Window x:Class="WpfTutorialSamples.ListView\_control.FilteringSample"

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

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

Title="FilteringSample" Height="200" Width="300">

<DockPanel Margin="10">

<TextBox DockPanel.Dock="Top" Margin="0,0,0,10" Name="txtFilter" TextChanged="txtFilter\_TextChanged" />

<ListView Name="lvUsers">

<ListView.View>

<GridView>

<GridViewColumn Header="Name" Width="120" DisplayMemberBinding="{Binding Name}" />

<GridViewColumn Header="Age" Width="50" DisplayMemberBinding="{Binding Age}" />

</GridView>

</ListView.View>

</ListView>

</DockPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Data;

namespace WpfTutorialSamples.ListView\_control

{

public partial class FilteringSample : Window

{

public FilteringSample()

{

InitializeComponent();

List<User> items = new List<User>();

items.Add(new User() { Name = "John Doe", Age = 42 });

items.Add(new User() { Name = "Jane Doe", Age = 39 });

items.Add(new User() { Name = "Sammy Doe", Age = 13 });

items.Add(new User() { Name = "Donna Doe", Age = 13 });

lvUsers.ItemsSource = items;

CollectionView view = (CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource);

view.Filter = UserFilter;

}

private bool UserFilter(object item)

{

if(String.IsNullOrEmpty(txtFilter.Text))

return true;

else

return ((item as User).Name.IndexOf(txtFilter.Text, StringComparison.OrdinalIgnoreCase) >= 0);

}

private void txtFilter\_TextChanged(object sender, System.Windows.Controls.TextChangedEventArgs e)

{

CollectionViewSource.GetDefaultView(lvUsers.ItemsSource).Refresh();

}

}

public enum SexType { Male, Female };

public class User

{

public string Name { get; set; }

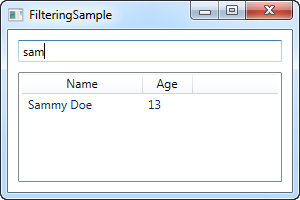
public int Age { get; set; }

public string Mail { get; set; }

public SexType Sex { get; set; }

}

}



The XAML part is pretty simple: We have a TextBox, where the user can enter a search string, and then a ListView to show the result in.

In Code-behind, we start off by adding some User objects to the ListView, just like we did in previous examples. The interesting part happens in the last two lines of the constructor, where we obtain a reference to the **CollectionView** instance for the ListView and then assign a delegate to the **Filter** property. This delegate points to the function called **UserFilter**, which we have implemented just below. It takes each item as the first (and only) parameter and then returns a boolean value that indicates whether or not the given item should be visible on the list.

In the **UserFilter()** method, we take a look at the TextBox control (txtFilter), to see if it contains any text - if it does, we use it to check whether or not the name of the User (which is the property we have decided to filter on) contains the entered string, and then return true or false depending on that. If the TextBox is empty, we return true, because in that case we want all the items to be visible.

The txtFilter\_TextChanged event is also important. Each time the text changes, we get a reference to the View object of the ListView and then call the Refresh() method on it. This ensures that the Filter delegate is called each time the user changes the value of the search/filter string text box.

**The TreeView control:**

# A simple TreeView example

As we talked about in the previous article, the WPF TreeView can be used in a very simple manner, by adding TreeViewItem objects to it, either from Code-behind or simply by declaring them directly in your XAML. This is indeed very easy to get started with, as you can see from the example here:

<Window x:Class="WpfTutorialSamples.TreeView\_control.TreeViewSample"

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

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

Title="TreeViewSample" Height="200" Width="250">

<Grid Margin="10">

<TreeView>

<TreeViewItem Header="Level 1" IsExpanded="True">

<TreeViewItem Header="Level 2.1" />

<TreeViewItem Header="Level 2.2" IsExpanded="True">

<TreeViewItem Header="Level 3.1" />

<TreeViewItem Header="Level 3.2" />

</TreeViewItem>

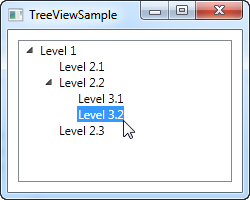
<TreeViewItem Header="Level 2.3" />

</TreeViewItem>

</TreeView>

</Grid>

</Window>



We simply declare the TreeViewItem objects directly in the XAML, in the same structure that we want to display them in, where the first tag is a child of the TreeView control and its child objects are also child tags to its parent object. To specify the text we want displayed for each node, we use the**Header** property. By default, a TreeViewItem is not expanded, but to show you the structure of the example, I have used the **IsExpanded** property to expand the two parent items.

## TreeViewItem's with images and other controls

The **Header** is an interesting property, though. As you can see, I can just specify a text string and then have it rendered directly without doing anything else, but this is WPF being nice to us - internally, it wraps the text inside of a TextBlock control, instead of forcing you to do it. This shows us that we can stuff pretty much whatever we want to into the Header property instead of just a string and then have the TreeView render it - a great example of why it's so easy to customize the look of WPF controls.

One of the common requests from people coming from WinForms or even other UI libraries is the ability to show an image next to the text label of a TreeView item. This is very easy to do with WinForms, because the TreeView is built exactly for this scenario. With the WPF TreeView, it's a bit more complex, but you're rewarded with a lot more flexibility than you could ever get from the WinForms TreeView. Here's an example of it:

<Window x:Class="WpfTutorialSamples.TreeView\_control.TreeViewCustomItemsSample"

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

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

Title="TreeViewCustomItemsSample" Height="200" Width="250">

<Grid Margin="10">

<TreeView>

<TreeViewItem IsExpanded="True">

<TreeViewItem.Header>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_blue.png" />

<TextBlock Text="Level 1 (Blue)" />

</StackPanel>

</TreeViewItem.Header>

<TreeViewItem>

<TreeViewItem.Header>

<StackPanel Orientation="Horizontal">

<TextBlock Text="Level 2.1" Foreground="Blue" />

</StackPanel>

</TreeViewItem.Header>

</TreeViewItem>

<TreeViewItem IsExpanded="True">

<TreeViewItem.Header>

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/bullet\_green.png" />

<TextBlock Text="Level 2.2 (Green)" Foreground="Blue" />

</StackPanel>

</TreeViewItem.Header>

<TreeViewItem>

<TreeViewItem.Header>

<TextBlock Text="Level 3.1" Foreground="Green" />

</TreeViewItem.Header>

</TreeViewItem>

<TreeViewItem>

<TreeViewItem.Header>

<TextBlock Text="Level 3.2" Foreground="Green" />

</TreeViewItem.Header>

</TreeViewItem>

</TreeViewItem>

<TreeViewItem>

<TreeViewItem.Header>

<TextBlock Text="Level 2.3" Foreground="Blue" />

</TreeViewItem.Header>

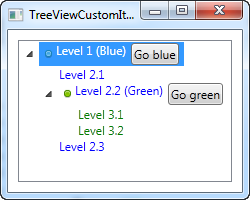
</TreeViewItem>

</TreeViewItem>

</TreeView>

</Grid>

</Window>



I did a whole bunch of things here, just to show you the kind of flexibility you get: I colored the child items and I added images and even buttons to the parent items. Because we're defining the entire thing with simple markup, you can do almost anything, but as you can see from the example code, it does come with a price: Huge amounts of XAML code, for a tree with just six nodes in total!

# TreeView, data binding and multiple templates

The WPF TreeView supports data binding, like pretty much all other WPF controls does, but because the TreeView is hierarchical in nature, a normal DataTemplate often won't suffice. Instead, we use the HierarchicalDataTemplate, which allows us to template both the tree node itself, while controlling which property to use as a source for child items of the node.

## A basic data bound TreeView

In the following example, I'll show you just how easy it is to get started with the HierarchicalDataTemplate:

<Window x:Class="WpfTutorialSamples.TreeView\_control.TreeViewDataBindingSample"

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

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

xmlns:self="clr-namespace:WpfTutorialSamples.TreeView\_control"

Title="TreeViewDataBindingSample" Height="150" Width="200">

<Grid Margin="10">

<TreeView Name="trvMenu">

<TreeView.ItemTemplate>

<HierarchicalDataTemplate DataType="{x:Type self:MenuItem}" ItemsSource="{Binding Items}">

<TextBlock Text="{Binding Title}" />

</HierarchicalDataTemplate>

</TreeView.ItemTemplate>

</TreeView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.IO;

using System.Collections.ObjectModel;

namespace WpfTutorialSamples.TreeView\_control

{

public partial class TreeViewDataBindingSample : Window

{

public TreeViewDataBindingSample()

{

InitializeComponent();

MenuItem root = new MenuItem() { Title = "Menu" };

MenuItem childItem1 = new MenuItem() { Title = "Child item #1" };

childItem1.Items.Add(new MenuItem() { Title = "Child item #1.1" });

childItem1.Items.Add(new MenuItem() { Title = "Child item #1.2" });

root.Items.Add(childItem1);

root.Items.Add(new MenuItem() { Title = "Child item #2" });

trvMenu.Items.Add(root);

}

}

public class MenuItem

{

public MenuItem()

{

this.Items = new ObservableCollection<MenuItem>();

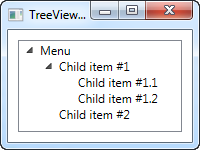
}

public string Title { get; set; }

public ObservableCollection<MenuItem> Items { get; set; }

}

}



In the XAML markup, I have specified a HierarchicalDataTemplate for the **ItemTemplate** of the TreeView. I instruct it to use the **Items** property for finding child items, by setting the **ItemsSource** property of the template, and inside of it I define the actual template, which for now just consists of a TextBlock bound to the **Title** property.

This first example was very simple, in fact so simple that we might as well have just added the TreeView items manually, instead of generating a set of objects and then binding to them. However, as soon as things get a bit more complicated, the advantages of using data bindings gets more obvious.

## Multiple templates for different types

In the next example, I've taken a slightly more complex case, where I want to show a tree of families and their members. A family should be represented in one way, while each of its members should be shown in another way. I achieve this by creating two different templates and specifying them as resources of the tree (or the Window or the Application - that's really up to you), and then allowing the TreeView to pick the correct template based on the underlying type of data.

Here's the code - the explanation of it will follow right after:

<Window x:Class="WpfTutorialSamples.TreeView\_control.TreeViewMultipleTemplatesSample"

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

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

xmlns:self="clr-namespace:WpfTutorialSamples.TreeView\_control"

Title="TreeViewMultipleTemplatesSample" Height="200" Width="250">

<Grid Margin="10">

<TreeView Name="trvFamilies">

<TreeView.Resources>

<HierarchicalDataTemplate DataType="{x:Type self:Family}" ItemsSource="{Binding Members}">

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/group.png" Margin="0,0,5,0" />

<TextBlock Text="{Binding Name}" />

<TextBlock Text=" [" Foreground="Blue" />

<TextBlock Text="{Binding Members.Count}" Foreground="Blue" />

<TextBlock Text="]" Foreground="Blue" />

</StackPanel>

</HierarchicalDataTemplate>

<DataTemplate DataType="{x:Type self:FamilyMember}">

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/user.png" Margin="0,0,5,0" />

<TextBlock Text="{Binding Name}" />

<TextBlock Text=" (" Foreground="Green" />

<TextBlock Text="{Binding Age}" Foreground="Green" />

<TextBlock Text=" years)" Foreground="Green" />

</StackPanel>

</DataTemplate>

</TreeView.Resources>

</TreeView>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Collections.ObjectModel;

namespace WpfTutorialSamples.TreeView\_control

{

public partial class TreeViewMultipleTemplatesSample : Window

{

public TreeViewMultipleTemplatesSample()

{

InitializeComponent();

List<Family> families = new List<Family>();

Family family1 = new Family() { Name = "The Doe's" };

family1.Members.Add(new FamilyMember() { Name = "John Doe", Age = 42 });

family1.Members.Add(new FamilyMember() { Name = "Jane Doe", Age = 39 });

family1.Members.Add(new FamilyMember() { Name = "Sammy Doe", Age = 13 });

families.Add(family1);

Family family2 = new Family() { Name = "The Moe's" };

family2.Members.Add(new FamilyMember() { Name = "Mark Moe", Age = 31 });

family2.Members.Add(new FamilyMember() { Name = "Norma Moe", Age = 28 });

families.Add(family2);

trvFamilies.ItemsSource = families;

}

}

public class Family

{

public Family()

{

this.Members = new ObservableCollection<FamilyMember>();

}

public string Name { get; set; }

public ObservableCollection<FamilyMember> Members { get; set; }

}

public class FamilyMember

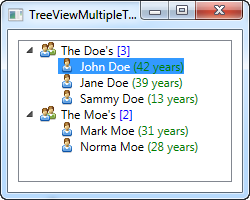
{

public string Name { get; set; }

public int Age { get; set; }

}

}



As mentioned, the two templates are declared as a part of the TreeView resources, allowing the TreeView to select the appropriate template based on the data type that it's about to show. The template defined for the **Family** type is a hierarchical template, using the **Members** property to show its family members.

The template defined for the **FamilyMember** type is a regular DataTemplate, since this type doesn't have any child members. However, if we had wanted each FamilyMember to keep a collection of their children and perhaps their children's children, then we would have used a hierarchical template instead.

In both templates, we use an image representing either a family or a family member, and then we show some interesting data about it as well, like the amount of family members or the person's age.

In the code-behind, we simply create two Family instances, fill each of them with a set of members, and then add each of the families to a list, which is then used as the items source for the TreeView.

# TreeView - Selection/Expansion state

In the previous couple of TreeView articles, we used data binding to display custom objects in a WPF TreeView. This works really well, but it does leave you with one problem: Because each tree node is now represented by your custom class, for instance FamilyMember as we saw in the previous article, you no longer have direct control over TreeView node specific functionality like selection and expansion state. In praxis this means that you can't select or expand/collapse a given node from code-behind.

Lots of solutions exists to handle this, ranging from "hacks" where you use the item generators of the TreeView to get the underlying TreeViewItem, where you can control the IsExpanded and IsSelected properties, to much more advanced MVVM-inspired implementations. In this article I would like to show you a solution that lies somewhere in the middle, making it easy to implement and use, while still not being a complete hack.

## A TreeView selection/expansion solution

The basic principle is to implement two extra properties on your data class: IsExpanded and IsSelected. These two properties are then hooked up to the TreeView, using a couple of styles targeting the TreeViewItem, inside of the **ItemContainerStyle** for the TreeView.

You could easily implement these two properties on all of your objects, but it's much easier to inherit them from a base object. If this is not feasible for your solution, you could create an interface for it and then implement this instead, to establish a common ground. For this example, I've chosen the base class method, because it allows me to very easily get the same functionality for my other objects. Here's the code:

<Window x:Class="WpfTutorialSamples.TreeView\_control.TreeViewSelectionExpansionSample"

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

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

Title="TreeViewSelectionExpansionSample" Height="200" Width="300">

<DockPanel Margin="10">

<WrapPanel Margin="0,10,0,0" DockPanel.Dock="Bottom" HorizontalAlignment="Center">

<Button Name="btnSelectNext" Click="btnSelectNext\_Click" Width="120">Select next</Button>

<Button Name="btnToggleExpansion" Click="btnToggleExpansion\_Click" Width="120" Margin="10,0,0,0">Toggle expansion</Button>

</WrapPanel>

<TreeView Name="trvPersons">

<TreeView.ItemTemplate>

<HierarchicalDataTemplate ItemsSource="{Binding Children}">

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/user.png" Margin="0,0,5,0" />

<TextBlock Text="{Binding Name}" Margin="0,0,4,0" />

</StackPanel>

</HierarchicalDataTemplate>

</TreeView.ItemTemplate>

<TreeView.ItemContainerStyle>

<Style TargetType="TreeViewItem">

<Setter Property="IsSelected" Value="{Binding IsSelected}" />

<Setter Property="IsExpanded" Value="{Binding IsExpanded}" />

</Style>

</TreeView.ItemContainerStyle>

</TreeView>

</DockPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Collections.ObjectModel;

using System.ComponentModel;

using System.Windows.Controls;

namespace WpfTutorialSamples.TreeView\_control

{

public partial class TreeViewSelectionExpansionSample : Window

{

public TreeViewSelectionExpansionSample()

{

InitializeComponent();

List<Person> persons = new List<Person>();

Person person1 = new Person() { Name = "John Doe", Age = 42 };

Person person2 = new Person() { Name = "Jane Doe", Age = 39 };

Person child1 = new Person() { Name = "Sammy Doe", Age = 13 };

person1.Children.Add(child1);

person2.Children.Add(child1);

person2.Children.Add(new Person() { Name = "Jenny Moe", Age = 17 });

Person person3 = new Person() { Name = "Becky Toe", Age = 25 };

persons.Add(person1);

persons.Add(person2);

persons.Add(person3);

person2.IsExpanded = true;

person2.IsSelected = true;

trvPersons.ItemsSource = persons;

}

private void btnSelectNext\_Click(object sender, RoutedEventArgs e)

{

if(trvPersons.SelectedItem != null)

{

var list = (trvPersons.ItemsSource as List<Person>);

int curIndex = list.IndexOf(trvPersons.SelectedItem as Person);

if(curIndex >= 0)

curIndex++;

if(curIndex >= list.Count)

curIndex = 0;

if(curIndex >= 0)

list[curIndex].IsSelected = true;

}

}

private void btnToggleExpansion\_Click(object sender, RoutedEventArgs e)

{

if(trvPersons.SelectedItem != null)

(trvPersons.SelectedItem as Person).IsExpanded = !(trvPersons.SelectedItem as Person).IsExpanded;

}

}

public class Person : TreeViewItemBase

{

public Person()

{

this.Children = new ObservableCollection<Person>();

}

public string Name { get; set; }

public int Age { get; set; }

public ObservableCollection<Person> Children { get; set; }

}

public class TreeViewItemBase : INotifyPropertyChanged

{

private bool isSelected;

public bool IsSelected

{

get { return this.isSelected; }

set

{

if(value != this.isSelected)

{

this.isSelected = value;

NotifyPropertyChanged("IsSelected");

}

}

}

private bool isExpanded;

public bool IsExpanded

{

get { return this.isExpanded; }

set

{

if(value != this.isExpanded)

{

this.isExpanded = value;

NotifyPropertyChanged("IsExpanded");

}

}

}

public event PropertyChangedEventHandler PropertyChanged;

public void NotifyPropertyChanged(string propName)

{

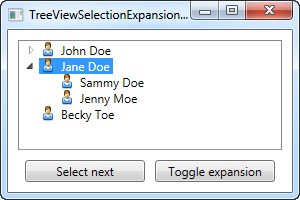
if(this.PropertyChanged != null)

this.PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

}

}



I'm sorry for the rather large amount of code in one place. In a real world solution, it would obviously be spread out over multiple files instead and the data for the tree would likely come from an actual data source, instead of being generated on the fly. Allow me to explain what happens in the example.

## XAML part

I have defined a couple of buttons to be placed in the bottom of the dialog, to use the two new properties. Then we have the TreeView, for which I have defined an ItemTemplate (as demonstrated in a previous chapter) as well as an ItemContainerStyle. If you haven't read the chapters on styling yet, you might not completely understand that part, but it's simply a matter of tying together the properties on our own custom class with the **IsSelected** and **IsExpanded** properties on the TreeViewItems, which is done with Style setters. You can learn more about them elsewhere in this tutorial.

## Code-behind part

In the code-behind, I have defined a **Person** class, with a couple of properties, which inherits our extra properties from the **TreeViewItemBase** class. You should be aware that the TreeViewItemBase class implements the INotifyPropertyChanged interface and uses it to notify of changes to these two essential properties - without this, selection/expansion changes won't be reflected in the UI. The concept of notification changes are explained in the Data binding chapters.

In the main Window class I simply create a range of persons, while adding children to some of them. I add the persons to a list, which I assign as the ItemsSource of the TreeView, which, with a bit of help from the defined template, renders them the way they are shown on the screenshot.

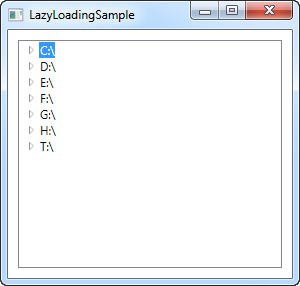
The most interesting part happens when I set the IsExpanded and IsSelected properties on the person2 object. This is what causes the second person (Jane Doe) to be initially selected and expanded, as shown on the screenshot. We also use these two properties on the Person objects (inherited from the TreeViewItemBase class) in the event handlers for the two test buttons (please bear in mind that, to keep the code as small and simple as possible, the selection button only works for the top level items).

# Lazy loading TreeView items

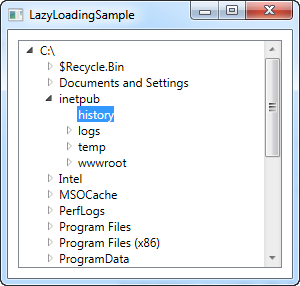
The usual process when using the TreeView is to bind to a collection of items or to manually add each level at the same time. However, in some situations, you want to delay the loading of a nodes child items until they are actually needed. This is especially useful if you have a very deep tree, with lots of levels and child nodes and a great example of this, is the folder structure of your Windows computer.

Each drive on your Windows computer has a range of child folders, and each of those child folders have child folders beneath them and so on. Looping through each drive and each drives child folders could become extremely time consuming and your TreeView would soon consist of a lot of nodes, with a high percentage of them never being needed. This is the perfect task for a lazy-loaded TreeView, where child folders are only loaded on demand.

To achieve this, we simply add a dummy folder to each drive or child folder, and then when the user expands it, we remove the dummy folder and replace it with the actual values. This is how our application looks when it starts - by that time, we have only obtained a list of available drives on the computer:



You can now start expanding the nodes, and the application will automatically load the sub folders. If a folder is empty, it will be shown as empty once you try to expand it, as it can be seen on the next screenshot:



So how is it accomplished? Let's have a look at the code:

<Window x:Class="WpfTutorialSamples.TreeView\_control.LazyLoadingSample"

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

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

Title="LazyLoadingSample" Height="300" Width="300">

<Grid>

<TreeView Name="trvStructure" TreeViewItem.Expanded="TreeViewItem\_Expanded" Margin="10" />

</Grid>

</Window>

using System;

using System.IO;

using System.Windows;

using System.Windows.Controls;

namespace WpfTutorialSamples.TreeView\_control

{

public partial class LazyLoadingSample : Window

{

public LazyLoadingSample()

{

InitializeComponent();

DriveInfo[] drives = DriveInfo.GetDrives();

foreach(DriveInfo driveInfo in drives)

trvStructure.Items.Add(CreateTreeItem(driveInfo));

}

public void TreeViewItem\_Expanded(object sender, RoutedEventArgs e)

{

TreeViewItem item = e.Source as TreeViewItem;

if((item.Items.Count == 1) && (item.Items[0] is string))

{

item.Items.Clear();

DirectoryInfo expandedDir = null;

if(item.Tag is DriveInfo)

expandedDir = (item.Tag as DriveInfo).RootDirectory;

if(item.Tag is DirectoryInfo)

expandedDir = (item.Tag as DirectoryInfo);

try

{

foreach(DirectoryInfo subDir in expandedDir.GetDirectories())

item.Items.Add(CreateTreeItem(subDir));

}

catch { }

}

}

private TreeViewItem CreateTreeItem(object o)

{

TreeViewItem item = new TreeViewItem();

item.Header = o.ToString();

item.Tag = o;

item.Items.Add("Loading...");

return item;

}

}

}

The **XAML** is very simple and only one interesting detail is present: The way we subscribe to the Expanded event of TreeViewItem's. Notice that this is indeed the TreeViewItem and not the TreeView itself, but because the event bubbles up, we are able to just capture it in one place for the entire TreeView, instead of having to subscribe to it for each item we add to the tree. This event gets called each time an item is expanded, which we need to be aware of to load its child items on demand.

In **Code-behind**, we start by adding each drive found on the computer to the TreeView control. We assign the **DriveInfo** instance to the Tag property, so that we can later retrieve it. Notice that we use a custom method to create the TreeViewItem, called **CreateTreeItem()**, since we can use the exact same method when we want to dynamically add a child folder later on. Notice in this method how we add a child item to the Items collection, in the form of a string with the text "Loading...".

Next up is the TreeViewItem\_Expanded event. As already mentioned, this event is raised each time a TreeView item is expanded, so the first thing we do is to check whether this item has already been loaded, by checking if the child items currently consists of only one item, which is a string - if so, we have found the "Loading..." child item, which means that we should now load the actual contents and replace the placeholder item with it.

We now use the items Tag property to get a reference to the **DriveInfo** or **DirectoryInfo** instance that the current item represents, and then we get a list of child directories, which we add to the clicked item, once again using the **CreateTreeItem()** method. Notice that the loop where we add each child folder is in a try..catch block - this is important, because some paths might not be accessible, usually for security reasons. You could grab the exception and use it to reflect this in the interface in one way or another.

# The DataGrid control

The DataGrid control looks a lot like the ListView, when using a GridView, but it offers a lot of additional functionality. For instance, the DataGrid can automatically generate columns, depending on the data you feed it with. The DataGrid is also editable by default, allowing the end-user to change the values of the underlying data source.

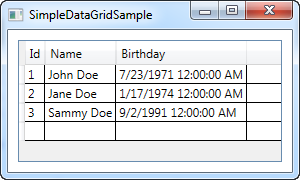
The most common usage for the DataGrid is in combination with a database, but like most WPF controls, it works just as well with an in-memory source, like a list of objects. Since it's a lot easier to demonstrate, we'll mostly be using the latter approach in this tutorial.

## A simple DataGrid

You can start using the DataGrid without setting any properties, because it supports so much out of the box. In this first example, we'll do just that, and then assign a list of our own User objects as the items source:

<Window x:Class="WpfTutorialSamples.DataGrid\_control.SimpleDataGridSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="SimpleDataGridSample" Height="180" Width="300">  
    <Grid Margin="10">  
 <DataGrid Name="dgSimple"></DataGrid>  
 </Grid>  
</Window>

using System;  
using System.Collections.Generic;  
using System.Windows;  
  
namespace WpfTutorialSamples.DataGrid\_control  
{  
 public partial class SimpleDataGridSample : Window  
 {  
 public SimpleDataGridSample()  
 {  
 InitializeComponent();  
  
 List<User> users = new List<User>();  
 users.Add(new User() { Id = 1, Name = "John Doe", Birthday = new DateTime(1971, 7, 23) });  
 users.Add(new User() { Id = 2, Name = "Jane Doe", Birthday = new DateTime(1974, 1, 17) });  
 users.Add(new User() { Id = 3, Name = "Sammy Doe", Birthday = new DateTime(1991, 9, 2) });  
  
 dgSimple.ItemsSource = users;  
 }  
 }  
  
 public class User  
 {  
 public int Id { get; set; }  
  
 public string Name { get; set; }  
  
 public DateTime Birthday { get; set; }  
 }  
}



That's really all you need to start using the DataGrid. The source could just as easily have been a database table/view or even an XML file - the DataGrid is not picky about where it gets its data from.

If you click inside one of the cells, you can see that you're allowed to edit each of the properties by default. As a nice little bonus, you can try clicking one of the column headers - you will see that the DataGrid supports sorting right out of the box!

The last and empty row will let you add to the data source, simply by filling out the cells.

**The DataGrid control:**

# DataGrid columns

In the previous chapter, we had a look at just how easy you could get a WPF DataGrid up and running. One of the reasons why it was so easy is the fact that the DataGrid will automatically generate appropriate columns for you, based on the data source you use.

However, in some situations you might want to manually define the columns shown, either because you don’t want all the properties/columns of the data source, or because you want to be in control of which inline editors are used.

## Manually defined columns

Let's try an example that looks a lot like the one in the previous chapter, but where we define all the columns manually, for maximum control. You can select the column type based on the data that you wish to display/edit. As of writing, the following column types are available:

* DataGridTextColumn
* DataGridCheckBoxColumn
* DataGridComboBoxColumn
* DataGridHyperlinkColumn
* DataGridTemplateColumn

Especially the last one, the DataGridTemplateColumn, is interesting. It allows you to define any kind of content, which opens up the opportunity to use custom controls, either from the WPF library or even your own or 3rd party controls. Here's an example:

<Window x:Class="WpfTutorialSamples.DataGrid\_control.DataGridColumnsSample"

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

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

Title="DataGridColumnsSample" Height="200" Width="300">

<Grid Margin="10">

<DataGrid Name="dgUsers" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Header="Name" Binding="{Binding Name}" />

<DataGridTemplateColumn Header="Birthday">

<DataGridTemplateColumn.CellTemplate>

<DataTemplate>

<DatePicker SelectedDate="{Binding Birthday}" BorderThickness="0" />

</DataTemplate>

</DataGridTemplateColumn.CellTemplate>

</DataGridTemplateColumn>

</DataGrid.Columns>

</DataGrid>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

namespace WpfTutorialSamples.DataGrid\_control

{

public partial class DataGridColumnsSample : Window

{

public DataGridColumnsSample()

{

InitializeComponent();

List<User> users = new List<User>();

users.Add(new User() { Id = 1, Name = "John Doe", Birthday = new DateTime(1971, 7, 23) });

users.Add(new User() { Id = 2, Name = "Jane Doe", Birthday = new DateTime(1974, 1, 17) });

users.Add(new User() { Id = 3, Name = "Sammy Doe", Birthday = new DateTime(1991, 9, 2) });

dgUsers.ItemsSource = users;

}

}

public class User

{

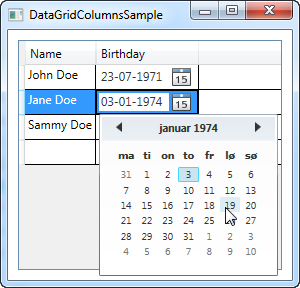
public int Id { get; set; }

public string Name { get; set; }

public DateTime Birthday { get; set; }

}

}



In the markup, I have added the AutoGenerateColumns property on the DataGrid, which I have set to false, to get control of the columns used. As you can see, I have left out the ID column, as I decided that I didn't care for it for this example. For the Name property, I've used a simple text based column, so the most interesting part of this example comes with the Birthday column, where I've used a DataGridTemplateColumn with a DatePicker control inside of it. This allows the end-user to pick the date from a calendar, instead of having to manually enter it, as you can see on the screenshot.

# DataGrid with row details

A very common usage scenario when using a DataGrid control is the ability to show details about each row, typically right below the row itself. The WPF DataGrid control supports this very well, and fortunately it's also very easy to use. Let's start off with an example and then we'll discuss how it works and the options it gives you afterwards:

<Window x:Class="WpfTutorialSamples.DataGrid\_control.DataGridDetailsSample"

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

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

Title="DataGridDetailsSample" Height="200" Width="400">

<Grid Margin="10">

<DataGrid Name="dgUsers" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Header="Name" Binding="{Binding Name}" />

<DataGridTextColumn Header="Birthday" Binding="{Binding Birthday}" />

</DataGrid.Columns>

<DataGrid.RowDetailsTemplate>

<DataTemplate>

<TextBlock Text="{Binding Details}" Margin="10" />

</DataTemplate>

</DataGrid.RowDetailsTemplate>

</DataGrid>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

namespace WpfTutorialSamples.DataGrid\_control

{

public partial class DataGridDetailsSample : Window

{

public DataGridDetailsSample()

{

InitializeComponent();

List<User> users = new List<User>();

users.Add(new User() { Id = 1, Name = "John Doe", Birthday = new DateTime(1971, 7, 23) });

users.Add(new User() { Id = 2, Name = "Jane Doe", Birthday = new DateTime(1974, 1, 17) });

users.Add(new User() { Id = 3, Name = "Sammy Doe", Birthday = new DateTime(1991, 9, 2) });

dgUsers.ItemsSource = users;

}

}

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public DateTime Birthday { get; set; }

public string Details

{

get

{

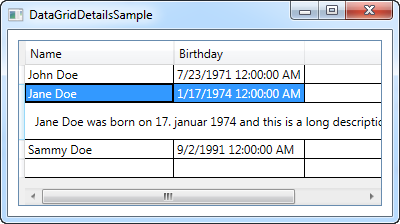
return String.Format("{0} was born on {1} and this is a long description of the person.", this.Name, this.Birthday.ToLongDateString());

}

}

}

}



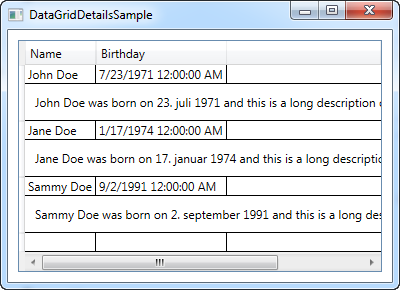
As you can see, I have expanded the example from previous chapters with a new property on the User class: The Description property. It simply returns a bit of information about the user in question, for our details row.

In the markup, I have defined a couple of columns and then I use the **RowDetailsTemplate** to specify a template for the row details. As you can see, it works much like any other WPF template, where I use a DataTemplate with one or several controls inside of it, along with a standard binding against a property on the data source, in this case the Description property.

As you can see from the resulting screenshot, or if you run the sample yourself, the details are now shown below the selected row. As soon as you select another row, the details for that row will be shown and the details for the previously selected row will be hidden.

## Controlling row details visibility

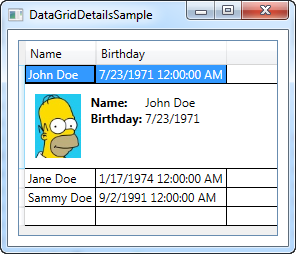
Using the **RowDetailsVisibilityMode** property, you can change the above mentioned behavior though. It defaults to**VisibleWhenSelected**, where details are only visible when its parent row is selected, but you can change it to **Visible** or **Collapsed**. If you set it to Visible, all details rows will be visible all the time, like this:



If you set it to Collapsed, all details will be invisible all the time.

## More details

The first example of this article might have been a tad boring, using just a single, plain TextBlock control. Of course, with this being a DataTemplate, you can do pretty much whatever you want, so I decided to extend the example a bit, to give a better idea of the possibilities. Here's how it looks now:



As you can see from the code listing, it's mostly about expanding the details template into using a panel, which in turn can host more panels and/or controls. Using a Grid panel, we can get the tabular look of the user data, and an Image control allows us to show a picture of the user (which you should preferably load from a locale resource and not a remote one, like I do in the example - and sorry for being too lazy to find a matching image of Jane and Sammy Doe).

<Window x:Class="WpfTutorialSamples.DataGrid\_control.DataGridDetailsSample"

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

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

Title="DataGridDetailsSample" Height="300" Width="300">

<Grid Margin="10">

<DataGrid Name="dgUsers" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Header="Name" Binding="{Binding Name}" />

<DataGridTextColumn Header="Birthday" Binding="{Binding Birthday}" />

</DataGrid.Columns>

<DataGrid.RowDetailsTemplate>

<DataTemplate>

<DockPanel Background="GhostWhite">

<Image DockPanel.Dock="Left" Source="{Binding ImageUrl}" Height="64" Margin="10" />

<Grid Margin="0,10">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<TextBlock Text="ID: " FontWeight="Bold" />

<TextBlock Text="{Binding Id}" Grid.Column="1" />

<TextBlock Text="Name: " FontWeight="Bold" Grid.Row="1" />

<TextBlock Text="{Binding Name}" Grid.Column="1" Grid.Row="1" />

<TextBlock Text="Birthday: " FontWeight="Bold" Grid.Row="2" />

<TextBlock Text="{Binding Birthday, StringFormat=d}" Grid.Column="1" Grid.Row="2" />

</Grid>

</DockPanel>

</DataTemplate>

</DataGrid.RowDetailsTemplate>

</DataGrid>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

namespace WpfTutorialSamples.DataGrid\_control

{

public partial class DataGridDetailsSample : Window

{

public DataGridDetailsSample()

{

InitializeComponent();

List<User> users = new List<User>();

users.Add(new User() { Id = 1, Name = "John Doe", Birthday = new DateTime(1971, 7, 23), ImageUrl = "http://www.wpf-tutorial.com/images/misc/john\_doe.jpg" });

users.Add(new User() { Id = 2, Name = "Jane Doe", Birthday = new DateTime(1974, 1, 17) });

users.Add(new User() { Id = 3, Name = "Sammy Doe", Birthday = new DateTime(1991, 9, 2) });

dgUsers.ItemsSource = users;

}

}

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public DateTime Birthday { get; set; }

public string ImageUrl { get; set; }

}

}

Introduction to WPF data binding

Data binding is general technique that binds two data/information sources together and maintains synchronization of data.

**Data binding:**

## The syntax of a Binding

All the magic happens between the curly braces, which in XAML encapsulates a Markup Extension. For data binding, we use the Binding extension, which allows us to describe the binding relationship for the Text property. In its most simple form, a binding can look like this:

{Binding}

This simply returns the current data context (more about that later). This can definitely be useful, but in the most common situations, you would want to bind a property to another property on the data context. A binding like that would look like this:

{Binding Path=NameOfProperty}

The Path notes the property that you want to bind to, however, since Path is the default property of a binding, you may leave it out if you want to, like this:

{Binding NameOfProperty}

You will see many different examples, some of them where Path is explicitly defined and some where it's left out. In the end it's really up to you though.

A binding has many other properties though, one of them being the ElementName which we use in our example. This allows us to connect directly to another UI element as the source. Each property that we set in the binding is separated by a comma:

{Binding Path=Text, ElementName=txtValue}

# Hello, bound world!

Just like we started this tutorial with the classic "Hello, world!" example, we'll show you how easy it is to use data binding in WPF with a "Hello, bound world!" example. Let's jump straight into it and then I'll explain it afterwards:

<Window x:Class="WpfTutorialSamples.DataBinding.HelloBoundWorldSample"

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

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

Title="HelloBoundWorldSample" Height="110" Width="280">

<StackPanel Margin="10">

<TextBox Name="txtValue" />

<WrapPanel Margin="0,10">

<TextBlock Text="Value: " FontWeight="Bold" />

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

</WrapPanel>

</StackPanel>

</Window>

Using the DataContext

The DataContext property is the default source of your bindings, unless you specifically declare another source, like we did in the previous chapter with the ElementName property. It's defined on the FrameworkElement class, which most UI controls, including the WPF Window, inherits from. Simply put, it allows you to specify a basis for your bindings

There's no default source for the DataContext property (it's simply null from the start), but since a DataContext is inherited down through the control hierarchy, you can set a DataContext for the Window itself and then use it throughout all of the child controls. Let's try illustrating that with a simple example:

<Window x:Class="WpfTutorialSamples.DataBinding.DataContextSample"

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

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

Title="DataContextSample" Height="130" Width="280">

<StackPanel Margin="15">

<WrapPanel>

<TextBlock Text="Window title: " />

<TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" Width="150" />

</WrapPanel>

<WrapPanel Margin="0,10,0,0">

<TextBlock Text="Window dimensions: " />

<TextBox Text="{Binding Width}" Width="50" />

<TextBlock Text=" x " />

<TextBox Text="{Binding Height}" Width="50" />

</WrapPanel>

</StackPanel>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.DataBinding

{

public partial class DataContextSample : Window

{

public DataContextSample()

{

InitializeComponent();

this.DataContext = this;

}

}

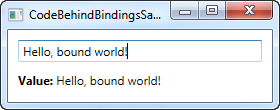
}

# Data binding via Code-behind

As we saw in the previous data binding examples, defining a binding by using XAML is very easy, but for certain cases, you may want to do it from Code-behind instead. This is pretty easy as well and offers the exact same possibilities as when you're using XAML. Let's try the "Hello, bound world" example, but this time create the required binding from Code-behind:

<Window x:Class="WpfTutorialSamples.DataBinding.CodeBehindBindingsSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 Title="CodeBehindBindingsSample" Height="110" Width="280">  
    <StackPanel Margin="10">  
 <TextBox Name="txtValue" />  
 <WrapPanel Margin="0,10">  
     <TextBlock Text="Value: " FontWeight="Bold" />  
     <TextBlock Name="lblValue" />  
 </WrapPanel>  
    </StackPanel>  
</Window>

using System;  
using System.Windows;  
using System.Windows.Controls;  
using System.Windows.Data;  
  
namespace WpfTutorialSamples.DataBinding  
{  
    public partial class CodeBehindBindingsSample : Window  
    {  
 public CodeBehindBindingsSample()  
 {  
     InitializeComponent();  
  
     Binding binding = new Binding("Text");  
     binding.Source = txtValue;  
     lblValue.SetBinding(TextBlock.TextProperty, binding);  
 }  
    }  
}



It works by creating a Binding instance. We specify the path we want directly in the constructor, in this case "Text", since we want to bind to the Text property. We then specify a **Source**, which for this example should be the TextBox control. Now WPF knows that it should use the TextBox as the source control, and that we're specifically looking for the value contained in its Text property.

In the last line, we use the SetBinding method to combine our newly created Binding object with the destination/target control, in this case the TextBlock (lblValue). The **SetBinding()** method takes two parameters, one that tells which dependency property that we want to bind to, and one that holds the binding object that we wish to use.

# The UpdateSourceTrigger property

In the previous article we saw how changes in a TextBox was not immediately sent back to the source. Instead, the source was updated only after focus was lost on the TextBox. This behavior is controlled by a property on the binding called **UpdateSourceTrigger**. It defaults to the value "Default", which basically means that the source is updated based on the property that you bind to. As of writing, all properties except for the Text property, is updated as soon as the property changes (PropertyChanged), while the Text property is updated when focus on the destination element is lost (LostFocus).

Default is, obviously, the default value of the UpdateSourceTrigger. The other options are **PropertyChanged**, **LostFocus** and **Explicit**. The first two has already been described, while the last one simply means that the update has to be pushed manually through to occur, using a call to UpdateSource on the Binding.

To see how all of these options work, I have updated the example from the previous chapter to show you all of them:

<Window x:Class="WpfTutorialSamples.DataBinding.DataContextSample"

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

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

Title="DataContextSample" Height="130" Width="310">

<StackPanel Margin="15">

<WrapPanel>

<TextBlock Text="Window title: " />

<TextBox Name="txtWindowTitle" Text="{Binding Title, UpdateSourceTrigger=Explicit}" Width="150" />

<Button Name="btnUpdateSource" Click="btnUpdateSource\_Click" Margin="5,0" Padding="5,0">\*</Button>

</WrapPanel>

<WrapPanel Margin="0,10,0,0">

<TextBlock Text="Window dimensions: " />

<TextBox Text="{Binding Width, UpdateSourceTrigger=LostFocus}" Width="50" />

<TextBlock Text=" x " />

<TextBox Text="{Binding Height, UpdateSourceTrigger=PropertyChanged}" Width="50" />

</WrapPanel>

</StackPanel>

</Window>

using System;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

namespace WpfTutorialSamples.DataBinding

{

public partial class DataContextSample : Window

{

public DataContextSample()

{

InitializeComponent();

this.DataContext = this;

}

private void btnUpdateSource\_Click(object sender, RoutedEventArgs e)

{

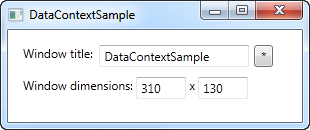
BindingExpression binding = txtWindowTitle.GetBindingExpression(TextBox.TextProperty);

binding.UpdateSource();

}

}

}



As you can see, each of the three textboxes now uses a different **UpdateSourceTrigger**.

The first one is set to **Explicit**, which basically means that the source won't be updated unless you manually do it. For that reason, I have added a button next to the TextBox, which will update the source value on demand. In the Code-behind, you will find the Click handler, where we use a couple of lines of code to get the binding from the destination control and then call the UpdateSource() method on it.

The second TextBox uses the **LostFocus** value, which is actually the default for a Text binding. It means that the source value will be updated each time the destination control loses focus.

The third and last TextBox uses the **PropertyChanged** value, which means that the source value will be updated each time the bound property changes, which it does in this case as soon as the text changes.

Try running the example on your own machine and see how the three textboxes act completely different: The first value doesn't update before you click the button, the second value isn't updated until you leave the TextBox, while the third value updates automatically on each keystroke, text change etc.

Responding to changes

So far in this tutorial, we have mostly created bindings between UI elements and existing classes, but in real life applications, you will obviously be binding to your own data objects. This is just as easy, but once you start doing it, you might discover something that disappoints you: Changes are not automatically reflected, like they were in previous examples. As you will learn in this article, you need just a bit of extra work for this to happen, but fortunately, WPF makes this pretty easy.

There are two different scenarios that you may or may not want to handle when dealing with data source changes: Changes to the list of items and changes in the bound properties in each of the data objects. How to handle them may vary, depending on what you're doing and what you're looking to accomplish, but WPF comes with two very easy solutions that you can use: The **ObservableCollection** and the **INotifyPropertyChanged** interface.

**The following example will show you why we need these two things:**

<Window x:Class="WpfTutorialSamples.DataBinding.ChangeNotificationSample"

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

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

Title="ChangeNotificationSample" Height="150" Width="300">

<DockPanel Margin="10">

<StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">

<Button Name="btnAddUser" Click="btnAddUser\_Click">Add user</Button>

<Button Name="btnChangeUser" Click="btnChangeUser\_Click" Margin="0,5">Change user</Button>

<Button Name="btnDeleteUser" Click="btnDeleteUser\_Click">Delete user</Button>

</StackPanel>

<ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>

</DockPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

namespace WpfTutorialSamples.DataBinding

{

public partial class ChangeNotificationSample : Window

{

private List<User> users = new List<User>();

public ChangeNotificationSample()

{

InitializeComponent();

users.Add(new User() { Name = "John Doe" });

users.Add(new User() { Name = "Jane Doe" });

lbUsers.ItemsSource = users;

}

private void btnAddUser\_Click(object sender, RoutedEventArgs e)

{

users.Add(new User() { Name = "New user" });

}

private void btnChangeUser\_Click(object sender, RoutedEventArgs e)

{

if(lbUsers.SelectedItem != null)

(lbUsers.SelectedItem as User).Name = "Random Name";

}

private void btnDeleteUser\_Click(object sender, RoutedEventArgs e)

{

if(lbUsers.SelectedItem != null)

users.Remove(lbUsers.SelectedItem as User);

}

}

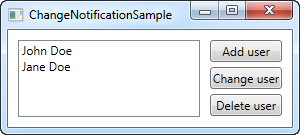
public class User

{

public string Name { get; set; }

}

}



Try running it for yourself and watch how even though you add something to the list or change the name of one of the users, **nothing in the UI is updated**. The example is pretty simple, with a User class that will keep the name of the user, a ListBox to show them in and some buttons to manipulate both the list and its contents. The ItemsSource of the list is assigned to a quick list of a couple of users that we create in the window constructor. The problem is that none of the buttons seems to work. Let's fix that, in two easy steps.

## Reflecting changes in the list data source

The first step is to get the UI to respond to changes in the list source (ItemsSource), like when we add or delete a user. What we need is a list that notifies any destinations of changes to its content, and fortunately, WPF provides a type of list that will do just that. It's called ObservableCollection, and you use it much like a regular List<T>, with only a few differences.

In the final example, which you will find below, we have simply replaced the List<User> with an ObservableCollection<User> - that's all it takes! This will make the Add and Delete button work, but it won't do anything for the "Change name" button, because the change will happen on the bound data object itself and not the source list - the second step will handle that scenario though.

## Reflecting changes in the data objects

The second step is to let our custom User class implement the INotifyPropertyChanged interface. By doing that, our User objects are capable of alerting the UI layer of changes to its properties. This is a bit more cumbersome than just changing the list type, like we did above, but it's still one of the simplest way to accomplish these automatic updates.

## The final and working example

With the two changes described above, we now have an example that WILL reflect changes in the data source. It looks like this:

<Window x:Class="WpfTutorialSamples.DataBinding.ChangeNotificationSample"

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

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

Title="ChangeNotificationSample" Height="135" Width="300">

<DockPanel Margin="10">

<StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">

<Button Name="btnAddUser" Click="btnAddUser\_Click">Add user</Button>

<Button Name="btnChangeUser" Click="btnChangeUser\_Click" Margin="0,5">Change user</Button>

<Button Name="btnDeleteUser" Click="btnDeleteUser\_Click">Delete user</Button>

</StackPanel>

<ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>

</DockPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.ComponentModel;

using System.Collections.ObjectModel;

namespace WpfTutorialSamples.DataBinding

{

public partial class ChangeNotificationSample : Window

{

private ObservableCollection<User> users = new ObservableCollection<User>();

public ChangeNotificationSample()

{

InitializeComponent();

users.Add(new User() { Name = "John Doe" });

users.Add(new User() { Name = "Jane Doe" });

lbUsers.ItemsSource = users;

}

private void btnAddUser\_Click(object sender, RoutedEventArgs e)

{

users.Add(new User() { Name = "New user" });

}

private void btnChangeUser\_Click(object sender, RoutedEventArgs e)

{

if(lbUsers.SelectedItem != null)

(lbUsers.SelectedItem as User).Name = "Random Name";

}

private void btnDeleteUser\_Click(object sender, RoutedEventArgs e)

{

if(lbUsers.SelectedItem != null)

users.Remove(lbUsers.SelectedItem as User);

}

}

public class User : INotifyPropertyChanged

{

private string name;

public string Name {

get { return this.name; }

set

{

if(this.name != value)

{

this.name = value;

this.NotifyPropertyChanged("Name");

}

}

}

public event PropertyChangedEventHandler PropertyChanged;

public void NotifyPropertyChanged(string propName)

{

if(this.PropertyChanged != null)

this.PropertyChanged(this, new PropertyChangedEventArgs(propName));

}

}

}

**Data binding:**

# Value conversion with IValueConverter

So far we have used some simple data bindings, where the sending and receiving property was always compatible. However, you will soon run into situations where you want to use a bound value of one type and then present it slightly differently.

## When to use a value converter

Value converters are very frequently used with data bindings. Here are some basic examples:

* You have a numeric value but you want to show zero values in one way and positive numbers in another way
* You want to check a CheckBox based on a value, but the value is a string like "yes" or "no" instead of a Boolean value
* You have a file size in bytes but you wish to show it as bytes, kilobytes, megabytes or gigabytes based on how big it is

These are some of the simple cases, but there are many more. For instance, you may want to check a checkbox based on a Boolean value, but you want it reversed, so that the CheckBox is checked if the value is false and not checked if the value is true. You can even use a converter to generate an image for an ImageSource, based on the value, like a green sign for true or a red sign for false - the possibilities are pretty much endless!

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

## Implementing a simple value converter

As mentioned, a WPF value converter needs to implement the IValueConverter interface, or alternatively, the IMultiValueConverter interface (more about that one later). Both interfaces just requires you to implement two methods: Convert() and ConvertBack(). As the name implies, these methods will be used to convert the value to the destination format and then back again.

Let's implement a simple converter which takes a string as input and then returns a Boolean value, as well as the other way around. If you're new to WPF, and you likely are since you're reading this tutorial, then you might not know all of the concepts used in the example, but don't worry, they will all be explained after the code listings:

<Window x:Class="WpfTutorialSamples.DataBinding.ConverterSample"

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

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

xmlns:local="clr-namespace:WpfTutorialSamples.DataBinding"

Title="ConverterSample" Height="140" Width="250">

<Window.Resources>

<local:YesNoToBooleanConverter x:Key="YesNoToBooleanConverter" />

</Window.Resources>

<StackPanel Margin="10">

<TextBox Name="txtValue" />

<WrapPanel Margin="0,10">

<TextBlock Text="Current value is: " />

<TextBlock Text="{Binding ElementName=txtValue, Path=Text, Converter={StaticResource YesNoToBooleanConverter}}"></TextBlock>

</WrapPanel>

<CheckBox IsChecked="{Binding ElementName=txtValue, Path=Text, Converter={StaticResource YesNoToBooleanConverter}}" Content="Yes" />

</StackPanel>

</Window>

using System;

using System.Windows;

using System.Windows.Data;

namespace WpfTutorialSamples.DataBinding

{

public partial class ConverterSample : Window

{

public ConverterSample()

{

InitializeComponent();

}

}

public class YesNoToBooleanConverter : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

switch(value.ToString().ToLower())

{

case "yes":

case "oui":

return true;

case "no":

case "non":

return false;

}

return false;

}

public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)

{

if(value is bool)

{

if((bool)value == true)

return "yes";

else

return "no";

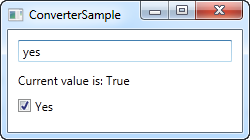
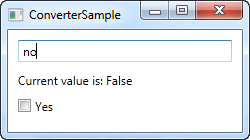
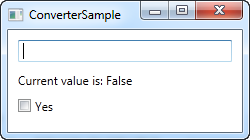
}

return "no";

}

}

}



## Code-behind

So, let's start from the back and then work our way through the example. We have implemented a converter in the Code-behind file called YesNoToBooleanConverter. As advertised, it just implements the two required methods, called Convert() and ConvertBack(). The Convert() methods assumes that it receives a string as the input (the value parameter) and then converts it to a Boolean true or false value, with a fallback value of false. For fun, I added the possibility to do this conversion from French words as well.

The ConvertBack() method obviously does the opposite: It assumes an input value with a Boolean type and then returns the English word "yes" or "no" in return, with a fallback value of "no".

You may wonder about the additional parameters that these two methods take, but they're not needed in this example. We'll use them in one of the next chapters, where they will be explained.

## XAML

In the XAML part of the program, we start off by declaring an instance of our converter as a resource for the window. We then have a TextBox, a couple of TextBlocks and a CheckBox control and this is where the interesting things are happening: We bind the value of the TextBox to the TextBlock and the CheckBox control and using the Converter property and our own converter reference, we juggle the values back and forth between a string and a Boolean value, depending on what's needed.

If you try to run this example, you will be able to change the value in two places: By writing "yes" in the TextBox (or any other value, if you want false) or by checking the CheckBox. No matter what you do, the change will be reflected in the other control as well as in the TextBlock.

# The StringFormat property

As we saw in the previous chapters, the way to manipulate the output of a binding before it is shown is typically through the use of a converter. The cool thing about the converters is that they allow you to convert any data type into a completely different data type. However, for more simple usage scenarios, where you just want to change the way a certain value is shown and not necessarily convert it into a different type, the **StringFormat** property might very well be enough.

Using the StringFormat property of a binding, you lose some of the flexibility you get when using a converter, but in return, it's much simpler to use and doesn't involve the creation of a new class in a new file.

The StringFormat property does exactly what the name implies: It formats the output string, simply by calling the String.Format method. Sometimes an example says more than a thousand words, so before I hit that word count, let's jump straight into an example:

<Window x:Class="WpfTutorialSamples.DataBinding.StringFormatSample"

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

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

xmlns:system="clr-namespace:System;assembly=mscorlib"

Title="StringFormatSample" Height="150" Width="250"

Name="wnd">

<StackPanel Margin="10">

<TextBlock Text="{Binding ElementName=wnd, Path=ActualWidth, StringFormat=Window width: {0:#,#.0}}" />

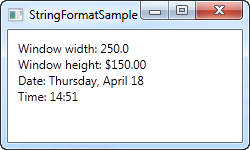
<TextBlock Text="{Binding ElementName=wnd, Path=ActualHeight, StringFormat=Window height: {0:C}}" />

<TextBlock Text="{Binding Source={x:Static system:DateTime.Now}, StringFormat=Date: {0:dddd, MMMM dd}}" />

<TextBlock Text="{Binding Source={x:Static system:DateTime.Now}, StringFormat=Time: {0:HH:mm}}" />

</StackPanel>

</Window>



The first couple of TextBlock's gets their value by binding to the parent Window and getting its width and height. Through the StringFormat property, the values are formatted. For the width, we specify a custom formatting string and for the height, we ask it to use the currency format, just for fun. The value is saved as a double type, so we can use all the same format specifiers as if we had called double.ToString(). You can find a list of them here: <http://msdn.microsoft.com/en-us/library/dwhawy9k.aspx>

Also notice how I can include custom text in the StringFormat - this allows you to pre/post-fix the bound value with text as you please. When referencing the actual value inside the format string, we surround it by a set of curly braces, which includes two values: A reference to the value we want to format (value number 0, which is the first possible value) and the format string, separated by a colon.

For the last two values, we simply bind to the current date (DateTime.Now) and the output it first as a date, in a specific format, and then as the time (hours and minutes), again using our own, pre-defined format. You can read more about DateTime formatting here: <http://msdn.microsoft.com/en-us/library/az4se3k1.aspx>

## Formatting without extra text

Please be aware that if you specify a format string that doesn't include any custom text, which all of the examples above does, then you need to add an extra set of curly braces, when defining it in XAML. The reason is that WPF may otherwise confuse the syntax with the one used for Markup Extensions. Here's an example:

<Window x:Class="WpfTutorialSamples.DataBinding.StringFormatSample"

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

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

xmlns:system="clr-namespace:System;assembly=mscorlib"

Title="StringFormatSample" Height="150" Width="250"

Name="wnd">

<WrapPanel Margin="10">

<TextBlock Text="Width: " />

<TextBlock Text="{Binding ElementName=wnd, Path=ActualWidth, StringFormat={}{0:#,#.0}}" />

</WrapPanel>

</Window>

## Using a specific Culture

If you need to output a bound value in accordance with a specific culture, that's no problem. The Binding will use the language specified for the parent element, or you can specify it directly for the binding, using the ConverterCulture property. Here's an example:

<Window x:Class="WpfTutorialSamples.DataBinding.StringFormatCultureSample"

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

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

xmlns:system="clr-namespace:System;assembly=mscorlib"

Title="StringFormatCultureSample" Height="120" Width="300">

<StackPanel Margin="10">

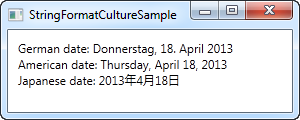
<TextBlock Text="{Binding Source={x:Static system:DateTime.Now}, ConverterCulture='de-DE', StringFormat=German date: {0:D}}" />

<TextBlock Text="{Binding Source={x:Static system:DateTime.Now}, ConverterCulture='en-US', StringFormat=American date: {0:D}}" />

<TextBlock Text="{Binding Source={x:Static system:DateTime.Now}, ConverterCulture='ja-JP', StringFormat=Japanese date: {0:D}}" />

</StackPanel>

</Window>



It's pretty simple: By combining the StringFormat property, which uses the D specifier (Long date pattern) and the ConverterCulture property, we can output the bound values in accordance with a specific culture. Pretty nifty!

# Introduction to WPF Commands

In a previous chapter of this tutorial, we talked about how to handle events, e.g. when the user clicks on a button or a menu item. In a modern user interface, it's typical for a function to be reachable from several places though, invoked by different user actions.

For instance, if you have a typical interface with a main menu and a set of toolbars, an action like New or Open might be available in the menu, on the toolbar, in a context menu (e.g. when right clicking in the main application area) and from a keyboard shortcut like Ctrl+N and Ctrl+O.

Each of these actions needs to perform what is typically the exact same piece of code, so in a WinForms application, you would have to define an event for each of them and then call a common function. With the above example, that would lead to at least three event handlers and some code to handle the keyboard shortcut. Not an ideal situation.

## Commands

With WPF, Microsoft is trying to remedy that with a concept called commands. It allows you to define actions in one place and then refer to them from all your user interface controls like menu items, toolbar buttons and so on. WPF will also listen for keyboard shortcuts and pass them along to the proper command, if any, making it the ideal way to offer keyboard shortcuts in an application.

Commands also solve another hassle when dealing with multiple entrances to the same function. In a WinForms application, you would be responsible for writing code that could disable user interface elements when the action was not available. For instance, if your application was able to use a clipboard command like Cut, but only when text was selected, you would have to manually enable and disable the main menu item, the toolbar button and the context menu item each time text selection changed.

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

## Command bindings

Commands don't actually do anything by them self. At the root, they consist of the ICommand interface, which only defines an event and two methods: Execute() and CanExecute(). The first one is for performing the actual action, while the second one is for determining whether the action is currently available. To perform the actual action of the command, you need a link between the command and your code and this is where the CommandBinding comes into play.

A CommandBinding is usually defined on a Window or a UserControl, and holds a references to the Command that it handles, as well as the actual event handlers for dealing with the Execute() and CanExecute() events of the Command.

## Pre-defined commands

You can of course implement your own commands, which we'll look into in one of the next chapters, but to make it easier for you, the WPF team has defined over 100 commonly used commands that you can use. They have been divided into 5 categories, called ApplicationCommands, NavigationCommands, MediaCommands, EditingCommands and ComponentCommands. Especially ApplicationCommands contains commands for a lot of very frequently used actions like New, Open, Save and Cut, Copy and Paste.

**Commands:**

# Using WPF commands

In the previous article, we discussed a lot of theory about what commands are and how they work. In this chapter, we'll look into how you actually use commands, by assigning them to user interface elements and creating command bindings that links it all together.

We'll start off with a very simple example:

<Window x:Class="WpfTutorialSamples.Commands.UsingCommandsSample"

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

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

Title="UsingCommandsSample" Height="100" Width="200">

<Window.CommandBindings>

<CommandBinding Command="ApplicationCommands.New" Executed="NewCommand\_Executed" CanExecute="NewCommand\_CanExecute" />

</Window.CommandBindings>

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">

<Button Command="ApplicationCommands.New">New</Button>

</StackPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Input;

namespace WpfTutorialSamples.Commands

{

public partial class UsingCommandsSample : Window

{

public UsingCommandsSample()

{

InitializeComponent();

}

private void NewCommand\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = true;

}

private void NewCommand\_Executed(object sender, ExecutedRoutedEventArgs e)

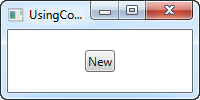
{

MessageBox.Show("The New command was invoked");

}

}

}



We define a command binding on the Window, by adding it to its CommandBindings collection. We specify that Command that we wish to use (the New command from the ApplicationCommands), as well as two event handlers. The visual interface consists of a single button, which we attach the command to using the **Command** property.

In Code-behind, we handle the two events. The **CanExecute** handler, which WPF will call when the application is idle to see if the specific command is currently available, is very simple for this example, as we want this particular command to be available all the time. This is done by setting the **CanExecute** property of the event arguments to true.

The **Executed** handler simply shows a message box when the command is invoked. If you run the sample and press the button, you will see this message. A thing to notice is that this command has a default keyboard shortcut defined, which you get as an added bonus. Instead of clicking the button, you can try to press Ctrl+N on your keyboard - the result is the same.

## Using the CanExecute method

In the first example, we implemented a CanExecute event that simply returned true, so that the button would be available all the time. However, this is of course not true for all buttons - in many cases, you want the button to be enabled or disabled depending on some sort of state in your application.

A very common example of this is the toggling of buttons for using the Windows Clipboard, where you want the Cut and Copy buttons to be enabled only when text is selected, and the Paste button to only be enabled when text is present in the clipboard. This is exactly what we'll accomplish in this example:

<Window x:Class="WpfTutorialSamples.Commands.CommandCanExecuteSample"

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

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

Title="CommandCanExecuteSample" Height="200" Width="250">

<Window.CommandBindings>

<CommandBinding Command="ApplicationCommands.Cut" CanExecute="CutCommand\_CanExecute" Executed="CutCommand\_Executed" />

<CommandBinding Command="ApplicationCommands.Paste" CanExecute="PasteCommand\_CanExecute" Executed="PasteCommand\_Executed" />

</Window.CommandBindings>

<DockPanel>

<WrapPanel DockPanel.Dock="Top" Margin="3">

<Button Command="ApplicationCommands.Cut" Width="60">\_Cut</Button>

<Button Command="ApplicationCommands.Paste" Width="60" Margin="3,0">\_Paste</Button>

</WrapPanel>

<TextBox AcceptsReturn="True" Name="txtEditor" />

</DockPanel>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Input;

namespace WpfTutorialSamples.Commands

{

public partial class CommandCanExecuteSample : Window

{

public CommandCanExecuteSample()

{

InitializeComponent();

}

private void CutCommand\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = (txtEditor != null) && (txtEditor.SelectionLength > 0);

}

private void CutCommand\_Executed(object sender, ExecutedRoutedEventArgs e)

{

txtEditor.Cut();

}

private void PasteCommand\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = Clipboard.ContainsText();

}

private void PasteCommand\_Executed(object sender, ExecutedRoutedEventArgs e)

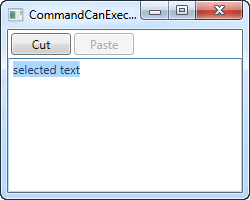
{

txtEditor.Paste();

}

}

}



So, we have this very simple interface with a couple of buttons and a TextBox control. The first button will cut to the clipboard and the second one will paste from it.

In Code-behind, we have two events for each button: One that performs the actual action, which name ends with \_Executed, and then the CanExecute events. In each of them, you will see that I apply some logic to decide whether or not the action can be executed and then assign it to the return value **CanExecute** on the EventArgs.

The cool thing about this is that you don't have to call these methods to have your buttons updated - WPF does it automatically when the application has an idle moment, making sure that you interface remains updated all the time.

## Default command behavior and CommandTarget

As we saw in the previous example, handling a set of commands can lead to quite a bit of code, with a lot of being method declarations and very standard logic. That's probably why the WPF team decided to handle some it for you. In fact, we could have avoided all of the Code-behind in the previous example, because a WPF TextBox can automatically handle common commands like Cut, Copy, Paste, Undo and Redo.

WPF does this by handling the Executed and CanExecute events for you, when a text input control like the TextBox has focus. You are free to override these events, which is basically what we did in the previous example, but if you just want the basic behavior, you can let WPF connect the commands and the TextBox control and do the work for you. Just see how much simpler this example is:

<Window x:Class="WpfTutorialSamples.Commands.CommandsWithCommandTargetSample"

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

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

Title="CommandsWithCommandTargetSample" Height="200" Width="250">

<DockPanel>

<WrapPanel DockPanel.Dock="Top" Margin="3">

<Button Command="ApplicationCommands.Cut" CommandTarget="{Binding ElementName=txtEditor}" Width="60">\_Cut</Button>

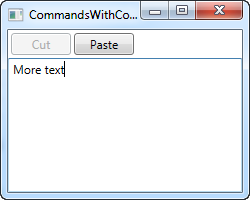
<Button Command="ApplicationCommands.Paste" CommandTarget="{Binding ElementName=txtEditor}" Width="60" Margin="3,0">\_Paste</Button>

</WrapPanel>

<TextBox AcceptsReturn="True" Name="txtEditor" />

</DockPanel>

</Window>



No Code-behind code needed for this example - WPF deals with all of it for us, but only because we want to use these specific commands for this specific control. The TextBox does the work for us.

Notice how I use the **CommandTarget** properties on the buttons, to bind the commands to our TextBox control. This is required in this particular example, because the WrapPanel doesn't handle focus the same way e.g. a Toolbar or a Menu would, but it also makes pretty good sense to give the commands a target.

**ommands:**

# Implementing a custom WPF Command

In the previous chapter, we looked at various ways of using commands already defined in WPF, but of course, you can implement your own commands as well. It's pretty simple, and once you've done it, you can use your own commands just like the ones defined in WPF.

The easiest way to start implementing your own commands is to have a static class that will contain them. Each command is then added to this class as static fields, allowing you to use them in your application. Since WPF, for some strange reason, doesn't implement an Exit/Quit command, I decided to implement one for our custom commands example. It looks like this:

<Window x:Class="WpfTutorialSamples.Commands.CustomCommandSample"

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

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

xmlns:self="clr-namespace:WpfTutorialSamples.Commands"

Title="CustomCommandSample" Height="150" Width="200">

<Window.CommandBindings>

<CommandBinding Command="self:CustomCommands.Exit" CanExecute="ExitCommand\_CanExecute" Executed="ExitCommand\_Executed" />

</Window.CommandBindings>

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<Menu>

<MenuItem Header="File">

<MenuItem Command="self:CustomCommands.Exit" />

</MenuItem>

</Menu>

<StackPanel Grid.Row="1" HorizontalAlignment="Center" VerticalAlignment="Center">

<Button Command="self:CustomCommands.Exit">Exit</Button>

</StackPanel>

</Grid>

</Window>

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Input;

namespace WpfTutorialSamples.Commands

{

public partial class CustomCommandSample : Window

{

public CustomCommandSample()

{

InitializeComponent();

}

private void ExitCommand\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = true;

}

private void ExitCommand\_Executed(object sender, ExecutedRoutedEventArgs e)

{

Application.Current.Shutdown();

}

}

public static class CustomCommands

{

public static readonly RoutedUICommand Exit = new RoutedUICommand

(

"Exit",

"Exit",

typeof(CustomCommands),

new InputGestureCollection()

{

new KeyGesture(Key.F4, ModifierKeys.Alt)

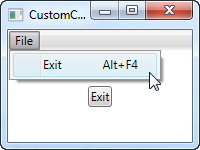
}

);

//Define more commands here, just like the one above

}

}



In the markup, I've defined a very simple interface with a menu and a button, both of them using our new, custom Exit command. This command is defined in Code-behind, in our own **CustomCommands** class, and then referenced in the CommandBindings collection of the window, where we assign the events that it should use to execute/check if it's allowed to execute.

All of this is just like the examples in the previous chapter, except for the fact that we're referencing the command from our own code (using the "self" namespace defined in the top) instead of a built-in command.

In Code-behind, we respond to the two events for our command: One event just allows the command to execute all the time, since that's usually true for an exit/quit command, and the other one calls the **Shutdown** method that will terminate our application. All very simple.

As already explained, we implement our Exit command as a field on a static CustomCommands class. There are several ways of defining and assigning properties on the commands, but I've chosen the more compact approach (it would be even more compact if placed on the same line, but I've added line breaks here for readability) where I assign all of it through the constructor. The parameters are the text/label of the command, the name of the command, the owner type and then an InputGestureCollection, allowing me to define a default shortcut for the command (Alt+F4).

**Dialogs:**

# The MessageBox

WPF offers several dialogs for your application to utilize, but the simplest one is definitely the MessageBox. Its sole purpose is to show a message to the user, and then offer one or several ways for the user to respond to the message.

The MessageBox is used by calling the static Show() method, which can take a range of different parameters, to be able to look and behave the way you want it to. We'll be going through all the various forms in this article, with each variation represented by the MessageBox.Show() line and a screenshot of the result. **In the end of the article, you can find a complete example which lets you test all the variations.**

In its simplest form, the MessageBox just takes a single parameter, which is the message to be displayed:

MessageBox.Show("Hello, world!");



## MessageBox with a title

The above example might be a bit too bare minimum - a title on the window displaying the message would probably help. Fortunately, the second and optional parameter allows us to specify the title:

MessageBox.Show("Hello, world!", "My App");



## MessageBox with extra buttons

By default, the MessageBox only has the one Ok button, but this can be changed, in case you want to ask your user a question and not just show a piece of information. Also notice how I use multiple lines in this message, by using a line break character (\n):

MessageBox.Show("This MessageBox has extra options.\n\nHello, world?", "My App", MessageBoxButton.YesNoCancel);

You control which buttons are displayed by using a value from the MessageBoxButton enumeration - in this case, a Yes, No and Cancel button is included. The following values, which should be self-explanatory, can be used:

* OK
* OKCancel
* YesNoCancel
* YesNo

Now with multiple choices, you need a way to be able to see what the user chose, and fortunately, the MessageBox.Show() method always returns a value from the **MessageBoxResult** enumeration that you can use. Here's an example:

MessageBoxResult result = MessageBox.Show("Would you like to greet the world with a \"Hello, world\"?", "My App", MessageBoxButton.YesNoCancel);

switch(result)

{

case MessageBoxResult.Yes:

MessageBox.Show("Hello to you too!", "My App");

break;

case MessageBoxResult.No:

MessageBox.Show("Oh well, too bad!", "My App");

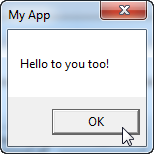
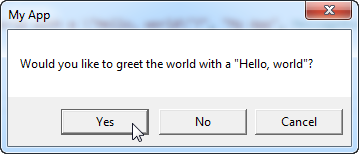
break;

case MessageBoxResult.Cancel:

MessageBox.Show("Nevermind then...", "My App");

break;

}

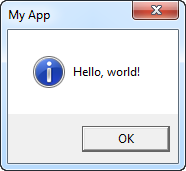


By checking the result value of the MessageBox.Show() method, you can now react to the user choice, as seen in the code example as well as on the screenshots.

## MessageBox with an icon

The MessageBox has the ability to show a pre-defined icon to the left of the text message, by using a fourth parameter:

MessageBox.Show("Hello, world!", "My App", MessageBoxButton.OK, MessageBoxImage.Information);



Using the **MessageBoxImage** enumeration, you can choose between a range of icons for different situations. Here's the complete list:

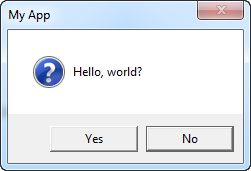
* Asterisk
* Error
* Exclamation
* Hand
* Information
* None
* Question
* Stop
* Warning

The names should say a lot about how they look, but feel free to experiment with the various values or have a look at this MSDN article, where each value is explained and even illustrated: <http://msdn.microsoft.com/en-us/library/system.windows.messageboximage.aspx>

## MessageBox with a default option

The MessageBox will select a button as the default choice, which is then the button invoked in case the user just presses Enter once the dialog is shown. For instance, if you display a MessageBox with a "Yes" and a "No" button, "Yes" will be the default answer. You can change this behavior using a fifth parameter to the MessageBox.Show() method though:

MessageBox.Show("Hello, world?", "My App", MessageBoxButton.YesNo, MessageBoxImage.Question, MessageBoxResult.No);



Notice on the screenshot how the "No" button is slightly elevated, to visually indicate that it is selected and will be invoked if the **Enter** or **Space** button is pressed.

## The complete example

As promised, here's the complete example used in this article:

<Window x:Class="WpfTutorialSamples.Dialogs.MessageBoxSample"

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

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

Title="MessageBoxSample" Height="250" Width="300">

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">

<StackPanel.Resources>

<Style TargetType="Button">

<Setter Property="Margin" Value="0,0,0,10" />

</Style>

</StackPanel.Resources>

<Button Name="btnSimpleMessageBox" Click="btnSimpleMessageBox\_Click">Simple MessageBox</Button>

<Button Name="btnMessageBoxWithTitle" Click="btnMessageBoxWithTitle\_Click">MessageBox with title</Button>

<Button Name="btnMessageBoxWithButtons" Click="btnMessageBoxWithButtons\_Click">MessageBox with buttons</Button>

<Button Name="btnMessageBoxWithResponse" Click="btnMessageBoxWithResponse\_Click">MessageBox with response</Button>

<Button Name="btnMessageBoxWithIcon" Click="btnMessageBoxWithIcon\_Click">MessageBox with icon</Button>

<Button Name="btnMessageBoxWithDefaultChoice" Click="btnMessageBoxWithDefaultChoice\_Click">MessageBox with default choice</Button>

</StackPanel>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.Dialogs

{

public partial class MessageBoxSample : Window

{

public MessageBoxSample()

{

InitializeComponent();

}

private void btnSimpleMessageBox\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("Hello, world!");

}

private void btnMessageBoxWithTitle\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("Hello, world!", "My App");

}

private void btnMessageBoxWithButtons\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("This MessageBox has extra options.\n\nHello, world?", "My App", MessageBoxButton.YesNoCancel);

}

private void btnMessageBoxWithResponse\_Click(object sender, RoutedEventArgs e)

{

MessageBoxResult result = MessageBox.Show("Would you like to greet the world with a \"Hello, world\"?", "My App", MessageBoxButton.YesNoCancel);

switch(result)

{

case MessageBoxResult.Yes:

MessageBox.Show("Hello to you too!", "My App");

break;

case MessageBoxResult.No:

MessageBox.Show("Oh well, too bad!", "My App");

break;

case MessageBoxResult.Cancel:

MessageBox.Show("Nevermind then...", "My App");

break;

}

}

private void btnMessageBoxWithIcon\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("Hello, world!", "My App", MessageBoxButton.OK, MessageBoxImage.Information);

}

private void btnMessageBoxWithDefaultChoice\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("Hello, world?", "My App", MessageBoxButton.YesNo, MessageBoxImage.Question, MessageBoxResult.No);

}

}

}

**Dialogs:**

# The OpenFileDialog

Whenever you open or save a file in almost any Windows application, you will see roughly the same dialogs for doing that. The reason is of course that these dialogs are a part of the Windows API and therefore also accessible to developers on the Windows platform.

For WPF, you will find standard dialogs for both opening and saving files in the **Microsoft.Win32** namespace. In this article we'll focus on the **OpenFileDialog** class, which makes it very easy to display a dialog for opening one or several files.

## Simple OpenFileDialog example

Let's start off by using the OpenFileDialog without any extra options, to load a file to a TextBox control:

<Window x:Class="WpfTutorialSamples.Dialogs.OpenFileDialogSample"

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

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

Title="OpenFileDialogSample" Height="300" Width="300">

<DockPanel Margin="10">

<WrapPanel HorizontalAlignment="Center" DockPanel.Dock="Top" Margin="0,0,0,10">

<Button Name="btnOpenFile" Click="btnOpenFile\_Click">Open file</Button>

</WrapPanel>

<TextBox Name="txtEditor" />

</DockPanel>

</Window>

using System;

using System.IO;

using System.Windows;

using Microsoft.Win32;

namespace WpfTutorialSamples.Dialogs

{

public partial class OpenFileDialogSample : Window

{

public OpenFileDialogSample()

{

InitializeComponent();

}

private void btnOpenFile\_Click(object sender, RoutedEventArgs e)

{

OpenFileDialog openFileDialog = new OpenFileDialog();

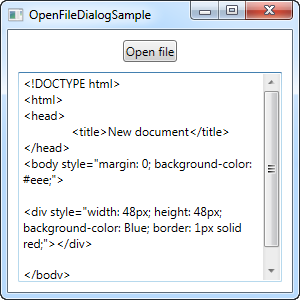
if(openFileDialog.ShowDialog() == true)

txtEditor.Text = File.ReadAllText(openFileDialog.FileName);

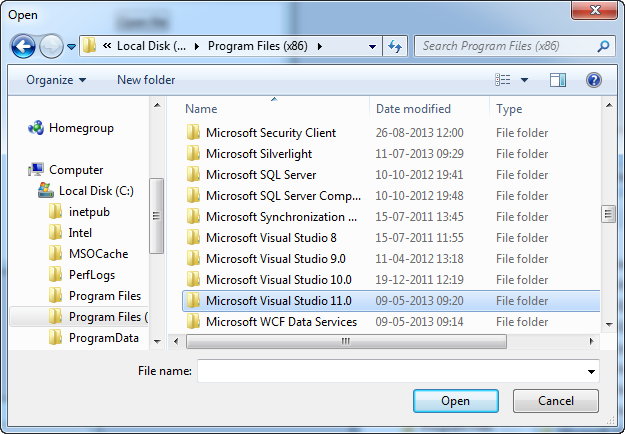
}

}

}



Once you click the Open file button, the OpenFileDialog will be instantiated and shown. Depending on which version of Windows you're using and the theme selected, it will look something like this:



The ShowDialog() will return a nullable boolean value, meaning that it can be either false, true or null. If the user selects a file and presses "Open", the result is True, and in that case, we try to load the file into the TextBox control. We get the complete path of the selected file by using the **FileName** property of the OpenFileDialog.

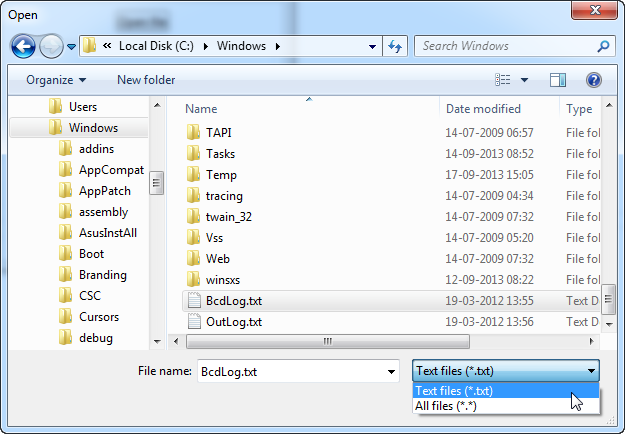
## Filter

Normally when you want your user to open a file in your application, you want to limit it to one or a couple of file types. For instance, Word mostly opens Word file (with the extension .doc or .docx) and Notepad mostly open text files (with the extension .txt).

You can specify a filter for your OpenFileDialog to indicate to the user which types of file they should be opening in your application, as well as limiting the files shown for a better overview. This is done with the Filter property, which we can add to the above example, right after initializing the dialog, like this:

openFileDialog.Filter = "Text files (\*.txt)|\*.txt|All files (\*.\*)|\*.\*";

Here's the result:



Notice how the dialog now has a combo box for selecting the file types, and that the files shown are limited to ones with the extension(s) specified by the selected file type.

The format for specifying the filter might look a bit strange at first sight, but it works by specifying a human-readable version of the desired file extension(s) and then one for the computer to easily parse, separated with a pipe (|) character. If you want more than one file type, as we do in the above example, each set of information are also separated with a pipe character.

So to sum up, the following part means that we want the file type to be named "Text files (\*.txt)" (the extension in the parenthesis is a courtesy to the user, so they know which extension(s) are included) and the second part tells the dialog to show files with a .txt extension:

Text files (\*.txt)|\*.txt

Each file type can of course have multiple extensions. For instance, image files could be specified as both JPEG and PNG files, like this:

openFileDialog.Filter = "Image files (\*.png;\*.jpeg)|\*.png;\*.jpeg|All files (\*.\*)|\*.\*";

Simply separate each extension with a semicolon in the second part (the one for the computer) - in the first part, you can format it the way you want to, but most developers seem to use the same notation for both parts, as seen in the example above.

## Setting the initial directory

The initial directory used by the OpenFileDialog is decided by Windows, but by using the **InitialDirectory**property, you can override it. You will usually set this value to a user specified directory, the application directory or perhaps just to the directory last used. You can set it to a path in a string format, like this:

openFileDialog.InitialDirectory = @"c:\temp\";

If you want to use one of the special folders on Windows, e.g. the Desktop, My Documents or the Program Files directory, you have to take special care, since these may vary from each version of Windows and also be dependent on which user is logged in. The .NET framework can help you though, just use the Environment class and its members for dealing with special folders:

openFileDialog.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

In this case, I get the path for the My Documents folder, but have a look at the SpecialFolder enumeration - it contains values for a lot of interesting paths. For a full list, please see this [MSDN article](http://msdn.microsoft.com/en-us/library/system.environment.specialfolder.aspx).

## Multiple files

If your application supports multiple open files, or you simply want to use the OpenFileDialog to select more than one file at a time, you need to enable the **Multiselect** property. In the next example, we've done just that, and as a courtesy to you, dear reader, we've also applied all the techniques mentioned above, including filtering and setting the initial directory:

<Window x:Class="WpfTutorialSamples.Dialogs.OpenFileDialogMultipleFilesSample"

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

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

Title="OpenFileDialogMultipleFilesSample" Height="300" Width="300">

<DockPanel Margin="10">

<WrapPanel HorizontalAlignment="Center" DockPanel.Dock="Top" Margin="0,0,0,10">

<Button Name="btnOpenFile" Click="btnOpenFiles\_Click">Open files</Button>

</WrapPanel>

<ListBox Name="lbFiles" />

</DockPanel>

</Window>

using System;

using System.IO;

using System.Windows;

using Microsoft.Win32;

namespace WpfTutorialSamples.Dialogs

{

public partial class OpenFileDialogMultipleFilesSample : Window

{

public OpenFileDialogMultipleFilesSample()

{

InitializeComponent();

}

private void btnOpenFiles\_Click(object sender, RoutedEventArgs e)

{

OpenFileDialog openFileDialog = new OpenFileDialog();

openFileDialog.Multiselect = true;

openFileDialog.Filter = "Text files (\*.txt)|\*.txt|All files (\*.\*)|\*.\*";

openFileDialog.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

if(openFileDialog.ShowDialog() == true)

{

foreach(string filename in openFileDialog.FileNames)

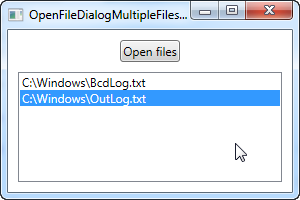
lbFiles.Items.Add(Path.GetFileName(filename));

}

}

}

}



If you test this code, you will see that you can now select multiple files in the same directory, by holding down either **Ctrl** or**Shift** and clicking with the mouse. Once accepted, this example simply adds the filenames to the ListBox control, by looping through the **FileNames** property.

**Dialogs:**

# The SaveFileDialog

The SaveFileDialog will help you select a location and a filename when you wish to save a file. It works and looks much like the OpenFileDialog which we used in the previous article, with a few subtle differences. Just like the OpenFileDialog, the SaveFileDialog is a wrapper around a common Windows dialog, meaning that your users will see roughly the same dialog whether they initiate it in your application or e.g. in Notepad.

## Simple SaveFileDialog example

To kick things off, let's begin with a very simple example on using the SaveFileDialog:

<Window x:Class="WpfTutorialSamples.Dialogs.SaveFileDialogSample"

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

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

Title="SaveFileDialogSample" Height="300" Width="300">

<DockPanel Margin="10">

<WrapPanel HorizontalAlignment="Center" DockPanel.Dock="Top" Margin="0,0,0,10">

<Button Name="btnSaveFile" Click="btnSaveFile\_Click">Save file</Button>

</WrapPanel>

<TextBox Name="txtEditor" TextWrapping="Wrap" AcceptsReturn="True" ScrollViewer.VerticalScrollBarVisibility="Auto" />

</DockPanel>

</Window>

using System;

using System.IO;

using System.Windows;

using Microsoft.Win32;

namespace WpfTutorialSamples.Dialogs

{

public partial class SaveFileDialogSample : Window

{

public SaveFileDialogSample()

{

InitializeComponent();

}

private void btnSaveFile\_Click(object sender, RoutedEventArgs e)

{

SaveFileDialog saveFileDialog = new SaveFileDialog();

if(saveFileDialog.ShowDialog() == true)

File.WriteAllText(saveFileDialog.FileName, txtEditor.Text);

}

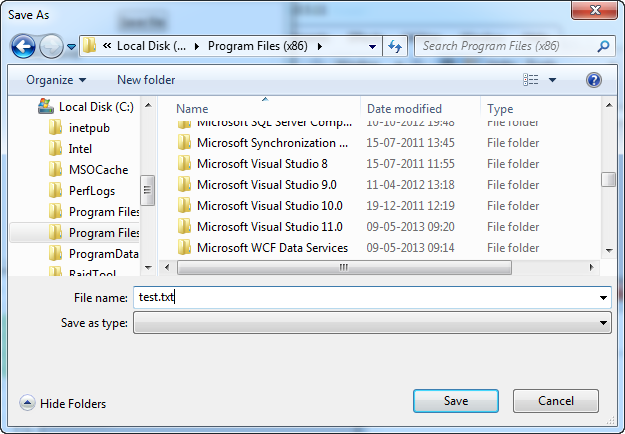
}

}



As you can see, it's mostly about instantiating the **SaveFileDialog** and then calling the **ShowDialog()** method. If it returns true, we use the **FileName** property (which will contain the selected path as well as the user entered file name) as the path to write our contents to.

If you click the save button, you should see a dialog like this, depending on the version of Windows you're using:



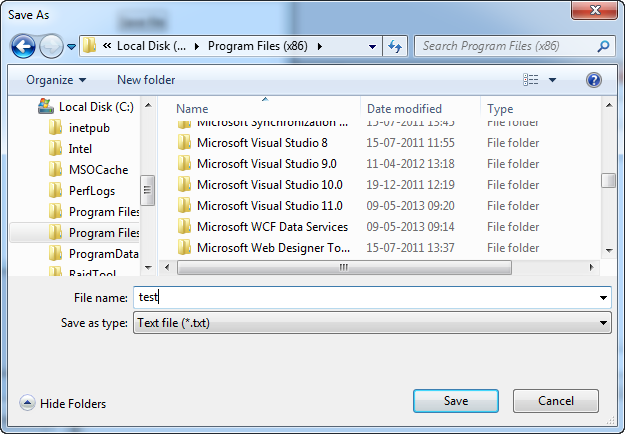
## Filter

As you can see from the first example, I manually added a .txt extension to my desired filename, mainly because the "Save as type" combo box is empty. Just like for the OpenFileDialog, this box is controlled through the **Filter** property, and it's also used in the exact same way.

saveFileDialog.Filter = "Text file (\*.txt)|\*.txt|C# file (\*.cs)|\*.cs";

**For more details about the format of the Filter property, please see the previous article on the OpenFileDialog, where it's explained in details.**

With a filter like the above, the resulting SaveFileDialog will look like this instead:



With that in place, you can write filenames without specifying the extension - it will be taken from the selected file type in the filter combo box instead. This also indicates to the user which file formats your application supports, which is of course important.

## Setting the initial directory

The initial directory used by the SaveFileDialog is decided by Windows, but by using the **InitialDirectory**property, you can override it. You will usually set this value to a user specified directory, the application directory or perhaps just to the directory last used. You can set it to a path in a string format, like this:

saveFileDialog.InitialDirectory = @"c:\temp\";

If you want to use one of the special folders on Windows, e.g. the Desktop, My Documents or the Program Files directory, you have to take special care, since these may vary from each version of Windows and also depend on which user is logged in. The .NET framework can help you though, just use the Environment class and its members for dealing with special folders:

saveFileDialog.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

In this case, I get the path for the My Documents folder, but have a look at the SpecialFolder enumeration - it contains values for a lot of interesting paths. For a full list, please see this [MSDN article](http://msdn.microsoft.com/en-us/library/system.environment.specialfolder.aspx).

## Options

Besides the options already mentioned in this article, I want to draw your attention to the following properties, which will help you tailor the SaveFileDialog to your needs:

**AddExtension** - defaults to true and determines if the SaveFileDialog should automatically append an extension to the filename, if the user omits it. The extension will be based on the selected filter, unless that's not possible, in which case it will fall back to the **DefaultExt** property (if specified). If you want your application to be able to save files without file extensions, you may have to disable this option.

**OverwritePrompt** - defaults to true and determines if the SaveFileDialog should ask for a confirmation if the user enters a file name which will result in an existing file being overwritten. You will normally want to leave this option enabled except in very special situations.

**Title** - you may override this property if you want a custom title on your dialog. It defaults to "Save As" or the localized equivalent and the property is also valid for the OpenFileDialog.

**ValidateNames**- defaults to true and unless it's disabled, it will ensure that the user enters only valid Windows file names before allowing the user to continue.

**Dialogs:**

# The other dialogs

Windows Forms comes with a range of dialogs which we haven't talked about in this tutorial yet, for the simple reason that they don't exist in WPF. The most important one is definitely the [FolderBrowserDialog](http://msdn.microsoft.com/en-us/library/system.windows.forms.folderbrowserdialog.aspx), which lets the user select a folder within the file system, but other dialogs missing in WPF include the [ColorDialog](http://msdn.microsoft.com/en-us/library/system.windows.forms.colordialog.aspx), the[FontDialog](http://msdn.microsoft.com/en-us/library/system.windows.forms.fontdialog.aspx), the[PrintPreviewDialog](http://msdn.microsoft.com/en-us/library/system.windows.forms.printpreviewdialog.aspx) and the [PageSetupDialog](http://msdn.microsoft.com/en-us/library/system.windows.forms.pagesetupdialog.aspx).

This can be a real problem for WPF developers, since re-implementing these dialogs would be a huge task. Fortunately, WPF and WinForms can be mixed, simply by referencing the **System.Windows.Forms** assembly, but since WPF uses different base types for both colors and dialogs, this is not always a viable solution. It is however an easy solution if you just need the FolderBrowserDialog, since it only deals with folder paths as simple strings, but some purists would argue that mixing WPF and WinForms is never the way to go.

A better way to go, if you don't want to reinvent the wheel yourself, might be to use some of the work created by other developers. Here are a couple of links for article which offers a solution to some of the missing dialogs:

- [A FontDialog alternative for WPF](http://www.codeproject.com/Articles/368070/A-WPF-Font-Picker-with-Color)

- [A ColorDialog alternative for WPF](http://www.codeproject.com/Articles/33001/WPF-A-Simple-Color-Picker-With-Preview)

In the end, you should choose the solution which fits the requirements of your application best.

**Dialogs:**

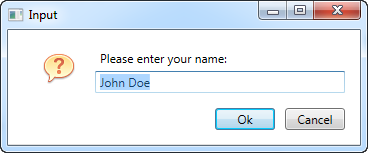
# Creating a custom input dialog

In the last couple of articles, we've looked at using the built-in dialogs of WPF, but creating your own is almost just as easy. In fact, you really just need to create a Window, place the required controls in it and then show it.

However, there are a few things that you should remember when creating dialogs, to ensure that your application acts like other Windows applications. In this article, we'll create a very simple dialog to ask the user a question and then return the answer, while discussing the various good practices that you should follow.

## Designing the dialog

For this particular dialog, I just wanted a Label telling the user which information we need from him/her, a TextBox for entering the answer, and then the usual Ok and Cancel buttons. I decided to add an icon to the dialog as well, for good looks. Here's the end result:



And here's the code for the dialog:

<Window x:Class="WpfTutorialSamples.Dialogs.InputDialogSample"

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

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

Title="Input" SizeToContent="WidthAndHeight" WindowStartupLocation="CenterScreen"

ContentRendered="Window\_ContentRendered">

<Grid Margin="15">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<Image Source="/WpfTutorialSamples;component/Images/question32.png" Width="32" Height="32" Grid.RowSpan="2" Margin="20,0" />

<Label Name="lblQuestion" Grid.Column="1">Question:</Label>

<TextBox Name="txtAnswer" Grid.Column="1" Grid.Row="1" MinWidth="250">Answer</TextBox>

<WrapPanel Grid.Row="2" Grid.ColumnSpan="2" HorizontalAlignment="Right" Margin="0,15,0,0">

<Button IsDefault="True" Name="btnDialogOk" Click="btnDialogOk\_Click" MinWidth="60" Margin="0,0,10,0">\_Ok</Button>

<Button IsCancel="True" MinWidth="60">\_Cancel</Button>

</WrapPanel>

</Grid>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.Dialogs

{

public partial class InputDialogSample : Window

{

public InputDialogSample(string question, string defaultAnswer = "")

{

InitializeComponent();

lblQuestion.Content = question;

txtAnswer.Text = defaultAnswer;

}

private void btnDialogOk\_Click(object sender, RoutedEventArgs e)

{

this.DialogResult = true;

}

private void Window\_ContentRendered(object sender, EventArgs e)

{

txtAnswer.SelectAll();

txtAnswer.Focus();

}

public string Answer

{

get { return txtAnswer.Text; }

}

}

}

The code is pretty simple, but here are the things that you should pay special attention to:

## XAML

In the **XAML** part, I've used a Grid for layout of the controls - nothing fancy here. I've removed the Width and Height properties of the Window and instead set it to automatically resize to match the content - this makes sense in a dialog, so you don't have to fine tune the size to make everything look alright. Instead, use margins and minimum sizes to ensure that things look the way you want them to, while still allowing the user to resize the dialog.

Another property which I've changed on the Window is the **WindowStartupLocation** property. For a dialog like this, and probably for most other non-main windows, you should change this value to CenterScreen or CenterOwner, to change the default behavior where your window will appear in a position decided by Windows, unless you manually specify **Top** and **Left** properties for it.

Also pay special attention to the two properties I've used on the dialog buttons: **IsCancel** and **IsDefault**. IsCancel tells WPF that if the user clicks this button, the **DialogResult** of the Window should be set to false which will also close the window. This also ensures that the user can press the **Esc** key on their keyboard to close the window, something that should always be possible in a Windows dialog.

The **IsDefault** property gives focus to the Ok button and also ensures that if the user presses the Enter key on their keyboard, this button is activated. An event handler is needed to set the DialogResult for this though, as described later.

## Code-behind

In **Code-behind**, I changed the constructor to take two parameters, with one being optional. This allows us to place the question and the default answer, if provided, into the designated UI controls.

The Ok button has an event handler which ensures that the special DialogResult property of the Window is set to true when clicked, to signal to the initiator of the dialog that the user accepted the entered value. We don't have one for the Cancel button, because WPF handles this for us when we set the **IsCancel** property to true, as described above.

To give focus to the TextBox upon showing the dialog, I've subscribed to the **ContentRendered** event, where I select all the text in the control and then give focus. If I just wanted to give focus, I could have use the FocusManager.FocusedElement attached property on the Window, but in this case, I also want to select the text, to allow the user to instantly overwrite the answer provided by default (if any).

A last detail is the **Answer** property which I've implemented. It simply gives access to the entered value of the TextBox control, but it's good practice to provide a property with the return value(s) of the dialog, instead of directly accessing controls from outside the window. This also allows you to influence the return value before returning it, if needed.

## Using the dialog

With all the above in place, we're now ready to actually use our dialog. It's a very simple task, so I've created a small application for testing it. Here's the code:

<Window x:Class="WpfTutorialSamples.Dialogs.InputDialogAppSample"

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

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

Title="InputDialogAppSample" Height="150" Width="300">

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">

<TextBlock>Hello, world. My name is:</TextBlock>

<TextBlock Name="lblName" Margin="0,10" TextAlignment="Center" FontWeight="Bold">[No name entered]</TextBlock>

<Button Name="btnEnterName" Click="btnEnterName\_Click">Enter name...</Button>

</StackPanel>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.Dialogs

{

public partial class InputDialogAppSample : Window

{

public InputDialogAppSample()

{

InitializeComponent();

}

private void btnEnterName\_Click(object sender, RoutedEventArgs e)

{

InputDialogSample inputDialog = new InputDialogSample("Please enter your name:", "John Doe");

if(inputDialog.ShowDialog() == true)

lblName.Text = inputDialog.Answer;

}

}

}



There's nothing special to it - just a couple of TextBlock controls and a Button for invoking the dialog. In the Click event handler, we instantiate the **InputDialogSample** window, providing a question and a default answer, and then we use the ShowDialog() method to show it - you should always use ShowDialog() method and not just Show() for a modal dialog like this.

If the result of the dialog is true, meaning that the user has activated the Ok button either by clicking it or pressing Enter, the result is assigned to the name Label. That's all there is to it!

**Common interface controls:**

# The WPF Menu control

One of the most common parts of a Windows application is the menu, sometimes referred to as the main menu because only one usually exists in the application. The menu is practical because it offers a lot of options, using only very little space, and even though Microsoft is pushing the Ribbon as a replacement for the good, old menu and toolbars, they definitely still have their place in every good developer's toolbox.

WPF comes with a fine control for creating menus called... Menu. Adding items to it is very simple - you simply add MenuItem elements to it, and each MenuItem can have a range of sub-items, allowing you to create hierarchical menus as you know them from a lot of Windows applications. Let's jump straight to an example where we use the Menu:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.MenuSample"

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

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

Title="MenuSample" Height="200" Width="200">

<DockPanel>

<Menu DockPanel.Dock="Top">

<MenuItem Header="\_File">

<MenuItem Header="\_New" />

<MenuItem Header="\_Open" />

<MenuItem Header="\_Save" />

<Separator />

<MenuItem Header="\_Exit" />

</MenuItem>

</Menu>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>



As in most Windows applications, my menu is placed in the top of the window, but in keeping with the enormous flexibility of WPF, you can actually place a Menu control wherever you like, and in any width or height that you may desire.

I have defined a single top-level item, with 4 child items and a separator. I use the **Header** property to define the label of the item, and you should notice the underscore before the first character of each label. It tells WPF to use that character as the accelerator key, which means that the user can press the Alt key followed by the given character, to activate the menu item. This works all the way from the top-level item and down the hierarchy, meaning that in this example I could press **Alt**, then **F** and then **N**, to activate the New item.

## Icons and checkboxes

Two common features of a menu item is the icon, used to more easily identify the menu item and what it does, and the ability to have checkable menu items, which can toggle a specific feature on and off. The WPF MenuItem supports both, and it's very easy to use:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.MenuIconCheckableSample"

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

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

Title="MenuIconCheckableSample" Height="150" Width="300">

<DockPanel>

<Menu DockPanel.Dock="Top">

<MenuItem Header="\_File">

<MenuItem Header="\_Exit" />

</MenuItem>

<MenuItem Header="\_Tools">

<MenuItem Header="\_Manage users">

<MenuItem.Icon>

<Image Source="/WpfTutorialSamples;component/Images/user.png" />

</MenuItem.Icon>

</MenuItem>

<MenuItem Header="\_Show groups" IsCheckable="True" IsChecked="True" />

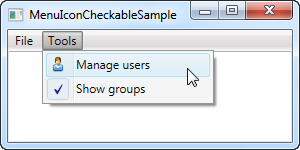
</MenuItem>

</Menu>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>



For this example I've created a secondary top-level item, where I've added two items: One with an icon defined, using the **Icon** property with a standard Image control inside of it, and one where we use the **IsCheckable** property to allow the user to check and uncheck the item. I even used the **IsChecked** property to have it checked by default. From Code-behind, this is the same property that you can read to know whether a given menu item is checked or not.

## Handling clicks

When the user clicks on a menu item, you will usually want something to happen. The easiest way is to simply add a click event handler to the MenuItem, like this:

<MenuItem Header="\_New" Click="mnuNew\_Click" />

In Code-behind you will then need to implement the mnuNew\_Click method, like this:

private void mnuNew\_Click(object sender, RoutedEventArgs e)

{

MessageBox.Show("New");

}

This will suffice for the more simple applications, or when prototyping something, but the WPF way is to use a Command for this.

## Keyboard shortcuts and Commands

You can easily handle the Click event of a menu item like we did above, but the more common approach is to use WPF commands. There's a lot of theory on using and creating commands, so they have their own category of articles here on the site, but for now, I can tell you that they have a couple of advantages when used in WPF, especially in combination with a Menu or a Toolbar.

First of all, they ensure that you can have the same action on a toolbar, a menu and even a context menu, without having to implement the same code in multiple places. They also make the handling of keyboard shortcuts a whole lot easier, because unlike with WinForms, WPF is not listening for keyboard shortcuts automatically if you assign them to e.g. a menu item - you will have to do that manually.

However, when using commands, WPF is all ears and will respond to keyboard shortcuts automatically. The text (Header) of the menu item is also set automatically (although you can overwrite it if needed), and so is the InputGestureText, which shows the user which keyboard shortcut can be used to invoke the specific menu item. Let's jump straight to an example of combining the Menu with WPF commands:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.MenuWithCommandsSample"

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

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

Title="MenuWithCommandsSample" Height="200" Width="300">

<Window.CommandBindings>

<CommandBinding Command="New" CanExecute="NewCommand\_CanExecute" Executed="NewCommand\_Executed" />

</Window.CommandBindings>

<DockPanel>

<Menu DockPanel.Dock="Top">

<MenuItem Header="\_File">

<MenuItem Command="New" />

<Separator />

<MenuItem Header="\_Exit" />

</MenuItem>

<MenuItem Header="\_Edit">

<MenuItem Command="Cut" />

<MenuItem Command="Copy" />

<MenuItem Command="Paste" />

</MenuItem>

</Menu>

<TextBox AcceptsReturn="True" Name="txtEditor" />

</DockPanel>

</Window>

using System;

using System.Windows;

using System.Windows.Input;

namespace WpfTutorialSamples.Common\_interface\_controls

{

public partial class MenuWithCommandsSample : Window

{

public MenuWithCommandsSample()

{

InitializeComponent();

}

private void NewCommand\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

{

e.CanExecute = true;

}

private void NewCommand\_Executed(object sender, ExecutedRoutedEventArgs e)

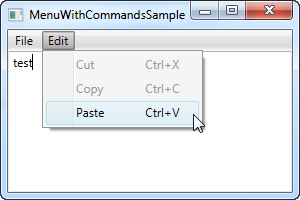
{

txtEditor.Text = "";

}

}

}



It might not be completely obvious, but by using commands, we just got a whole bunch of things for free: Keyboard shortcuts, text and **InputGestureText** on the items and WPF automatically enables/disables the items depending on the active control and its state. In this case, Cut and Copy are disabled because no text is selected, but Paste is enabled, because my clipboard is not empty!

And because WPF knows how to handle certain commands in combination with certain controls, in this case the Cut/Copy/Paste commands in combination with a text input control, we don't even have to handle their Execute events - they work right out of the box! We do have to handle it for the**New** command though, since WPF has no way of guessing what we want it to do when the user activates it. This is done with the **CommandBindings** of the Window, all explained in detail in the chapter on commands.

**Common interface controls:**

# The WPF ContextMenu

A context menu, often referred to as a popup or pop-up menu, is a menu which is shown upon certain user actions, usually a right-click with the mouse on a specific control or window. Contextual menus are often used to offer functionality that's relevant within a single control.

WPF comes with a ContextMenu control and because it's almost always tied to a specific control, that's also usually how you add it to the interface. This is done through the ContextProperty, which all controls exposes (it comes from the FrameworkElement which most WPF controls inherits from). Consider the next example to see how it's done:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.ContextMenuSample"

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

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

Title="ContextMenuSample" Height="250" Width="250">

<Grid>

<Button Content="Right-click me!" VerticalAlignment="Center" HorizontalAlignment="Center">

<Button.ContextMenu>

<ContextMenu>

<MenuItem Header="Menu item 1" />

<MenuItem Header="Menu item 2" />

<Separator />

<MenuItem Header="Menu item 3" />

</ContextMenu>

</Button.ContextMenu>

</Button>

</Grid>

</Window>



If you've already read the chapter on the regular menu, you will soon realize that the ContextMenu works exactly the same way, and no wonder, since they both inherit the MenuBase class. Just like we saw in the examples on using the regular Menu, you can of course add Click events to these items to handle when the user clicks on them, but a more WPF-suitable way is to use Commands.

## ContextMenu with Commands and icons

In this next example, I'm going to show you two key concepts when using the ContextMenu: The usage of WPF Commands, which will provide us with lots of functionality including a Click event handler, a text and a shortcut text, simply by assigning something to the Command property. I will also show you to use icons on your ContextMenu items. Have a look:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.ContextMenuWithCommandsSample"

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

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

Title="ContextMenuWithCommandsSample" Height="200" Width="250">

<StackPanel Margin="10">

<TextBox Text="Right-click here for context menu!">

<TextBox.ContextMenu>

<ContextMenu>

<MenuItem Command="Cut">

<MenuItem.Icon>

<Image Source="/WpfTutorialSamples;component/Images/cut.png" />

</MenuItem.Icon>

</MenuItem>

<MenuItem Command="Copy">

<MenuItem.Icon>

<Image Source="/WpfTutorialSamples;component/Images/copy.png" />

</MenuItem.Icon>

</MenuItem>

<MenuItem Command="Paste">

<MenuItem.Icon>

<Image Source="/WpfTutorialSamples;component/Images/paste.png" />

</MenuItem.Icon>

</MenuItem>

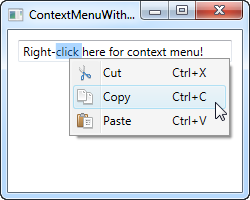
</ContextMenu>

</TextBox.ContextMenu>

</TextBox>

</StackPanel>

</Window>



Try running the example and see for yourself how much functionality we get for free by assigning commands to the items. Also notice how fairly simple it is to use icons on the menu items of the ContextMenu.

## Invoke ContextMenu from Code-behind

So far, the ContextMenu has been invoked when right-clicking on the control to which it belongs. WPF does this for us automatically, when we assign it to the **ContextMenu** property. However, in some situations, you might very well want to invoke it manually from code. This is pretty easy as well, so let's re-use the first example to demonstrate it with:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.ContextMenuManuallyInvokedSample"

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

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

Title="ContextMenuManuallyInvokedSample" Height="250" Width="250">

<Window.Resources>

<ContextMenu x:Key="cmButton">

<MenuItem Header="Menu item 1" />

<MenuItem Header="Menu item 2" />

<Separator />

<MenuItem Header="Menu item 3" />

</ContextMenu>

</Window.Resources>

<Grid>

<Button Content="Click me!" VerticalAlignment="Center" HorizontalAlignment="Center" Click="Button\_Click" />

</Grid>

</Window>

using System;

using System.Windows;

using System.Windows.Controls;

namespace WpfTutorialSamples.Common\_interface\_controls

{

public partial class ContextMenuManuallyInvokedSample : Window

{

public ContextMenuManuallyInvokedSample()

{

InitializeComponent();

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

ContextMenu cm = this.FindResource("cmButton") as ContextMenu;

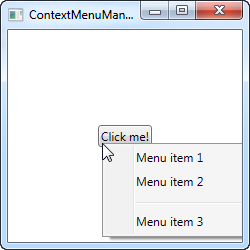
cm.PlacementTarget = sender as Button;

cm.IsOpen = true;

}

}

}



The first thing you should notice is that I've moved the ContextMenu away from the button. Instead, I've added it as a resource of the Window, to make it available from all everywhere within the Window. This also makes it a lot easier to find when we need to show it.

The Button now has a Click event handler, which I handle in Code-behind. From there, I simply find the ContextMenu instance within the window resources and then I do two things: I set it's PlacementTarget property, which tells WPF which element it should calculate the position based on, and then I set the IsOpen to true, to open the menu. That's all you need!

**Common interface controls:**

# The WPF ToolBar control

The toolbar is a row of commands, usually sitting right below the main menu of a standard Windows application. This could in fact be a simple panel with buttons on it, but by using the WPF ToolBar control, you get some extra goodies like automatic overflow handling and the possibility for the end-user to re-position your toolbars.

A WPF ToolBar is usually placed inside of a ToolBarTray control. The ToolBarTray will handle stuff like placement and sizing, and you can have multiple ToolBar controls inside of the ToolBarTray element. Let's try a pretty basic example, to see what it all looks like:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.ToolbarSample"

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

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

Title="ToolbarSample" Height="200" Width="300">

<Window.CommandBindings>

<CommandBinding Command="New" CanExecute="CommonCommandBinding\_CanExecute" />

<CommandBinding Command="Open" CanExecute="CommonCommandBinding\_CanExecute" />

<CommandBinding Command="Save" CanExecute="CommonCommandBinding\_CanExecute" />

</Window.CommandBindings>

<DockPanel>

<ToolBarTray DockPanel.Dock="Top">

<ToolBar>

<Button Command="New" Content="New" />

<Button Command="Open" Content="Open" />

<Button Command="Save" Content="Save" />

</ToolBar>

<ToolBar>

<Button Command="Cut" Content="Cut" />

<Button Command="Copy" Content="Copy" />

<Button Command="Paste" Content="Paste" />

</ToolBar>

</ToolBarTray>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>

using System;

using System.Windows;

using System.Windows.Input;

namespace WpfTutorialSamples.Common\_interface\_controls

{

public partial class ToolbarSample : Window

{

public ToolbarSample()

{

InitializeComponent();

}

private void CommonCommandBinding\_CanExecute(object sender, CanExecuteRoutedEventArgs e)

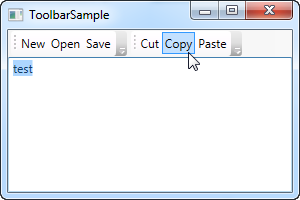
{

e.CanExecute = true;

}

}

}



Notice how I use commands for all the buttons. We discussed this in the previous chapter and using commands definitely gives us some advantages. Take a look at the Menu chapter, or the articles on commands, for more information.

In this example, I add a ToolBarTray to the top of the screen, and inside of it, two ToolBar controls. Each contains some buttons and we use commands to give them their behavior. In Code-behind, I make sure to handle the CanExecute event of the first three buttons, since that's not done automatically by WPF, contrary to the Cut, Copy and Paste commands, which WPF is capable of fully handling for us.

Try running the example and place the cursor over the left part of one of the toolbars (the dotted area). If you click and hold your left mouse button, you can now re-position the toolbar, e.g. below the other or even make them switch place.

## Images

While text on the toolbar buttons is perfectly okay, the normal approach is to have icons or at least a combination of an icon and a piece of text. Because WPF uses regular Button controls, adding icons to the toolbar items is very easy. Just have a look at this next example, where we do both:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.ToolbarIconSample"

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

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

Title="ToolbarIconSample" Height="200" Width="300">

<DockPanel>

<ToolBarTray DockPanel.Dock="Top">

<ToolBar>

<Button Command="Cut" ToolTip="Cut selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/cut.png" />

</Button>

<Button Command="Copy" ToolTip="Copy selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/copy.png" />

</Button>

<Button Command="Paste" ToolTip="Paste from Windows Clipboard.">

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/paste.png" />

<TextBlock Margin="3,0,0,0">Paste</TextBlock>

</StackPanel>

</Button>

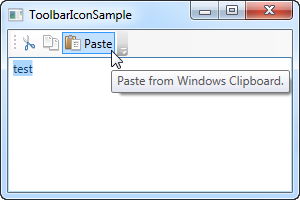
</ToolBar>

</ToolBarTray>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>

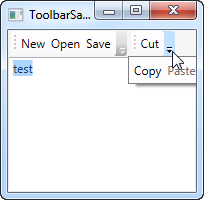


By specifying an **Image** control as the Content of the first two buttons, they will be icon based instead of text based. On the third button, I combine an **Image** control and a **TextBlock** control inside of a **StackPanel**, to achieve both icon and text on the button, a commonly used technique for buttons which are extra important or with a less obvious icon.

Notice how I've used the **ToolTip** property on each of the buttons, to add an explanatory text. This is especially important for those buttons with only an icon, because the purpose of the button might not be clear from only looking at the icon. With the ToolTip property, the user can hover the mouse over the button to get a description of what it does, as demonstrated on the screenshot.

## Overflow

As already mentioned, a very good reason for using the ToolBar control instead of just a panel of buttons, is the automatic overflow handling. It means that if there's no longer enough room to show all of the buttons on the toolbar, WPF will put them in a menu accessible by clicking on the arrow to the right of the toolbar. You can see how it works on this screenshot, which shows the first example, but with a smaller window, thereby leaving less space for the toolbars:



WPF even allows you to decide which items are suitable for overflow hiding and which should always be visible. Usually, when designing a toolbar, some items are less important than the others and some of them you might even want to have in the overflow menu all the time, no matter if there's space enough or not.

This is where the attached property **ToolBar.OverflowMode** comes into play. The default value is AsNeeded, which simply means that a toolbar item is put in the overflow menu if there's not enough room for it. You may use **Always** or **Never** instead, which does exactly what the names imply: Puts the item in the overflow menu all the time or prevents the item from ever being moved to the overflow menu. Here's an example on how to assign this property:

<ToolBar>

<Button Command="Cut" Content="Cut" ToolBar.OverflowMode="Always" />

<Button Command="Copy" Content="Copy" ToolBar.OverflowMode="AsNeeded" />

<Button Command="Paste" Content="Paste" ToolBar.OverflowMode="Never" />

</ToolBar>

## Position

While the most common position for the toolbar is indeed in the top of the screen, toolbars can also be found in the bottom of the application window or even on the sides. The WPF ToolBar of course supports all of this, and while the bottom placed toolbar is merely a matter of docking to the bottom of the panel instead of the top, a vertical toolbar requires the use of the **Orientation** property of the ToolBar tray. Allow me to demonstrate with an example:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.ToolbarPositionSample"

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

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

Title="ToolbarPositionSample" Height="200" Width="300">

<DockPanel>

<ToolBarTray DockPanel.Dock="Top">

<ToolBar>

<Button Command="Cut" ToolTip="Cut selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/cut.png" />

</Button>

<Button Command="Copy" ToolTip="Copy selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/copy.png" />

</Button>

<Button Command="Paste" ToolTip="Paste from Windows Clipboard.">

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/paste.png" />

<TextBlock Margin="3,0,0,0">Paste</TextBlock>

</StackPanel>

</Button>

</ToolBar>

</ToolBarTray>

<ToolBarTray DockPanel.Dock="Right" Orientation="Vertical">

<ToolBar>

<Button Command="Cut" ToolTip="Cut selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/cut.png" />

</Button>

<Button Command="Copy" ToolTip="Copy selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/copy.png" />

</Button>

<Button Command="Paste" ToolTip="Paste from Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/paste.png" />

</Button>

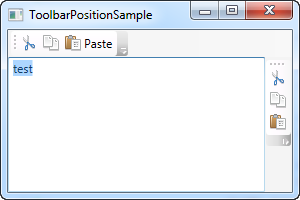
</ToolBar>

</ToolBarTray>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>



The trick here lies in the combination of the **DockPanel.Dock** property, that puts the ToolBarTray to the right of the application, and the **Orientation** property, that changes the orientation from horizontal to vertical. This makes it possible to place toolbars in pretty much any location that you might think of.

## Custom controls on the ToolBar

As you have seen on all of the previous examples, we use regular WPF Button controls on the toolbars. This also means that you can place pretty much any other WPF control on the toolbars, with no extra effort. Of course, some controls works better on a toolbar than others, but controls like the ComboBox and TextBox are commonly used on the toolbars in e.g. older versions of Microsoft Office, and you can do the same on your own WPF toolbars.

Another thing introduced in this example is the Separator element, which simply creates a separator between two sets of toolbar items. As you can see from the example, it's very easy to use!

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.ToolbarCustomControlsSample"

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

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

Title="ToolbarCustomControlsSample" Height="200" Width="300">

<DockPanel>

<ToolBarTray DockPanel.Dock="Top">

<ToolBar>

<Button Command="Cut" ToolTip="Cut selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/cut.png" />

</Button>

<Button Command="Copy" ToolTip="Copy selection to Windows Clipboard.">

<Image Source="/WpfTutorialSamples;component/Images/copy.png" />

</Button>

<Button Command="Paste" ToolTip="Paste from Windows Clipboard.">

<StackPanel Orientation="Horizontal">

<Image Source="/WpfTutorialSamples;component/Images/paste.png" />

<TextBlock Margin="3,0,0,0">Paste</TextBlock>

</StackPanel>

</Button>

<Separator />

<Label>Font size:</Label>

<ComboBox>

<ComboBoxItem>10</ComboBoxItem>

<ComboBoxItem IsSelected="True">12</ComboBoxItem>

<ComboBoxItem>14</ComboBoxItem>

<ComboBoxItem>16</ComboBoxItem>

</ComboBox>

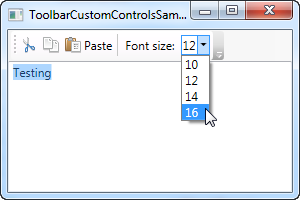
</ToolBar>

</ToolBarTray>

<TextBox AcceptsReturn="True" />

</DockPanel>

</Window>



**Common interface controls:**

# The WPF StatusBar control

With the top of the application window usually occupied by the main menu and/or toolbars, described in previous chapters, the bottom part of the window is usually the home of the status bar. The status bar is used to show various information about the current state of the application, like cursor position, word count, progress of tasks and so on. Fortunately for us, WPF comes with a nice StatusBar control, making it very easy to add status bar functionality to your applications.

Let's start off with a very basic example:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.StatusBarSample"

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

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

Title="StatusBarSample" Height="150" Width="300">

<DockPanel>

<StatusBar DockPanel.Dock="Bottom">

<StatusBarItem>

<TextBlock Name="lblCursorPosition" />

</StatusBarItem>

</StatusBar>

<TextBox AcceptsReturn="True" Name="txtEditor" SelectionChanged="txtEditor\_SelectionChanged" />

</DockPanel>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.Common\_interface\_controls

{

public partial class StatusBarSample : Window

{

public StatusBarSample()

{

InitializeComponent();

}

private void txtEditor\_SelectionChanged(object sender, RoutedEventArgs e)

{

int row = txtEditor.GetLineIndexFromCharacterIndex(txtEditor.CaretIndex);

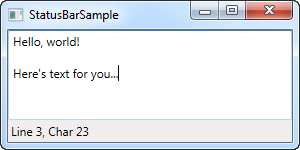
int col = txtEditor.CaretIndex - txtEditor.GetCharacterIndexFromLineIndex(row);

lblCursorPosition.Text = "Line " + (row + 1) + ", Char " + (col + 1);

}

}

}



It's all very simple - a TextBlock control that shows the current cursor position, just like in pretty much any other application that allows you to edit text. In this very basic form, the StatusBar could just as easily have been a panel with a set of controls on it, but the real advantage of the StatusBar comes when we need to divide it into several areas of information.

## Advanced StatusBar example

Let's try a more advanced example of using the StatusBar. The first thing we want to do is to make the StatusBar use another panel for the layout. By default, it uses the **DockPanel**, but when we want a more complex layout, with columns that adjusts its width in a certain way and aligned content, the **Grid** is a much better choice.

We'll divide the **Grid** into three areas, with the left and right one having a fixed width and the middle column automatically taking up the remaining space. We'll also add columns in between for **Separator** controls. Here's how it looks now:

<Window x:Class="WpfTutorialSamples.Common\_interface\_controls.StatusBarAdvancedSample"

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

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

Title="StatusBarAdvancedSample" Height="150" Width="400">

<DockPanel>

<StatusBar DockPanel.Dock="Bottom">

<StatusBar.ItemsPanel>

<ItemsPanelTemplate>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100" />

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="Auto" />

<ColumnDefinition Width="100" />

</Grid.ColumnDefinitions>

</Grid>

</ItemsPanelTemplate>

</StatusBar.ItemsPanel>

<StatusBarItem>

<TextBlock Name="lblCursorPosition" />

</StatusBarItem>

<Separator Grid.Column="1" />

<StatusBarItem Grid.Column="2">

<TextBlock Text="c:\path\of\current\file.txt" />

</StatusBarItem>

<Separator Grid.Column="3" />

<StatusBarItem Grid.Column="4">

<ProgressBar Value="50" Width="90" Height="16" />

</StatusBarItem>

</StatusBar>

<TextBox AcceptsReturn="True" Name="txtEditor" SelectionChanged="txtEditor\_SelectionChanged" />

</DockPanel>

</Window>

using System;

using System.Windows;

namespace WpfTutorialSamples.Common\_interface\_controls

{

public partial class StatusBarAdvancedSample : Window

{

public StatusBarAdvancedSample()

{

InitializeComponent();

}

private void txtEditor\_SelectionChanged(object sender, RoutedEventArgs e)

{

int row = txtEditor.GetLineIndexFromCharacterIndex(txtEditor.CaretIndex);

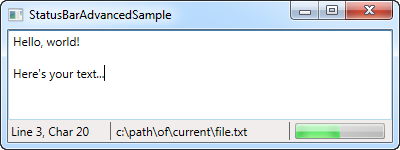
int col = txtEditor.CaretIndex - txtEditor.GetCharacterIndexFromLineIndex(row);

lblCursorPosition.Text = "Line " + (row + 1) + ", Char " + (col + 1);

}

}

}

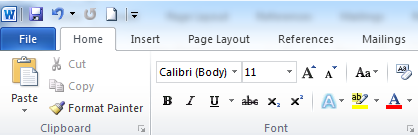


As you can see, I've added a bit of sample information, like the fake filename in the middle column and the progress bar to the right, showing a static value for now. You could easily make this work for real though, and it gives a pretty good idea on what you can do with the StatusBar control.

**Common interface controls:**

# The Ribbon control

The Ribbon interface was invented by Microsoft and first used in Office 2007. It combines the original menu and toolbar(s) into one control, with various functions grouped into tabs and groups. The most important purpose was to make it easier for the user to discover all the functionality, instead of hiding it in long menus. The Ribbon also allows for prioritization of functionality, with the ability to use different sizes of buttons.



WPF doesn't come with a built-in Ribbon control, but Microsoft has released one that you can and use for free, as long as you promise to follow their implementation guide when using it. You can read much more about it at [MSDN](http://msdn.microsoft.com/en-us/library/ff799534.aspx), where you'll also find a  [link](http://go.microsoft.com/fwlink/?LinkId=196621) for the Ribbon control.

**Rich Text controls:**

Introduction to WPF Rich Text controls

In other UI frameworks like WinForms, displaying large amounts of richly formatted text has been somewhat of a problem. Sure, you could load a file into a RichTextBox or you could create a WebBrowser object and load a local or remote web page, but specifying larger amounts of rich text in design-time wasn't really possible. It seems that Microsoft wanted to remedy that in WPF and even go beyond just simple viewing of the text.

The FlowDocument does indeed render rich text, and that even includes images, lists and tables, and elements can be floated, adjusted and so on, and using a FlowDocument, you can specify rich text in design-time as if it were HTML (thanks to XAML) and have it rendered directly in your WPF application.

The FlowDocument doesn't stand alone. Instead, it uses one of several built-in wrappers, which controls how the FlowDocument is laid out and whether the content can be edited by the user or not. WPF includes three controls for rendering a FlowDocument in read-only mode, which all has easy support for zooming and printing:

**FlowDocumentScrollViewer** - the simplest wrapper around a FlowDocument, which simply displays the document as one long document of text which you can scroll in.

**FlowDocumentPageViewer** - this wrapper will automatically split your document into pages, which the user can navigate back and forth between.

**FlowDocumentReader** - a combination of the *FlowDocumentScrollViewer* and the *FlowDocumentPageViewer*, which will let the user decide between the two rendering modes. It also offers the ability AND the interface to search in the document.

The FlowDocument is normally read-only, but put it inside of a **RichTextBox** control (described later in this tutorial) and you can now edit the text, much like in real word processors like Microsoft Word.

Read on through the next chapters, where we'll discuss all the wrappers that you can use with a FlowDocument, both read-only and editable. After that, we'll look into all of the possibilities you have when creating rich documents using the FlowDocument, including tables, lists, images and much more.

**Rich Text controls:**

# The FlowDocumentScrollViewer control

In the range of FlowDocument wrappers, discussed in the introduction, the FlowDocumentScrollViewer is the simplest one. It simply allows the users to scroll to long documents, using regular scrollbars. Since this is our first meeting with the FlowDocument used in any form, we'll start off with a basic "Hello World!" example, and besides the use of FlowDocumentScrollViewer, this article will also cover several concepts common between all of the wrappers. Here's the first example:

<Window x:Class="WpfTutorialSamples.Rich\_text\_controls.FlowDocumentScrollViewerSample"

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

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

Title="FlowDocumentScrollViewerSample" Height="200" Width="300">

<Grid>

<FlowDocumentScrollViewer>

<FlowDocument>

<Paragraph FontSize="36">Hello, world!</Paragraph>

<Paragraph FontStyle="Italic" TextAlignment="Left" FontSize="14" Foreground="Gray">The ultimate programming greeting!</Paragraph>

</FlowDocument>

</FlowDocumentScrollViewer>

</Grid>

</Window>



Notice how easy it was to specify the text, using simple markup tags, in this case the **Paragraph** tag. Now you might argue that this could have been achieved with a couple of TextBlock controls, and you would be absolutely right, but even with an extremely basic example like this, you get a bit of added functionality for free: You can select the text and copy it to the clipboard. It'll look like this:



## Zooming and scrollbar visibility

As previously mentioned, all of the FlowDocument wrappers support zooming out of the box. With the example above, you can simply hold down the Ctrl key while using the mouse wheel to zoom in and out. This might not be obvious to your end users though, so you can help them by displaying the built-in toolbar of the FlowDocumentScrollViewer, which has controls that will allow you to change the zoom level. Just set the **IsToolBarVisible** property to true on the FlowDocumentScrollViewer, and you're good to go, as you can see in the next example:

<Window x:Class="WpfTutorialSamples.Rich\_text\_controls.FlowDocumentScrollViewerZoomSample"

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

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

Title="FlowDocumentScrollViewerZoomSample" Height="180" Width="300">

<Grid>

<FlowDocumentScrollViewer IsToolBarVisible="True" Zoom="80" ScrollViewer.VerticalScrollBarVisibility="Auto">

<FlowDocument>

<Paragraph FontSize="36">Hello, world!</Paragraph>

<Paragraph FontStyle="Italic" TextAlignment="Left" FontSize="14" Foreground="Gray">The ultimate programming greeting!</Paragraph>

</FlowDocument>

</FlowDocumentScrollViewer>

</Grid>

</Window>



Now the user can control the zoom level using the slider and the buttons in the toolbar below the document. Notice also that we changed the default zoom level, using the **Zoom** property - it defines the zoom level in percentages, so in this case, the text is zoomed out to 80% by default.

The last thing I changed in this example, in comparison to the first one, is the use of the **ScrollViewer.VerticalScrollBarVisibility** property. By setting it to **Auto**, the scrollbars will be invisible until the content actually goes beyond the available space, which is usually what you want.

## Text alignment

You may have noticed that I specifically used the **TextAlignment** property in the above examples. That's because the text is rendered justified by default, in a WPF FlowDocument, meaning that each line of text is stretched to cover the entire available width, if needed. As you can see, this can be changed, either on a single paragraph or globally for the entire document by setting the same property on the FlowDocument element.

However, in many situations, justified text makes sense, but it can result in some very bad layout, with very excessive amounts of whitespace on lines where a linebreak is inserted right before a very long word.

The following example will illustrate that, as well as provide a solution that will help remedy the problem. By using the **IsOptimalParagraphEnabled** property in combination with the **IsHyphenationEnabled** property, you will give WPF a better chance of laying out the text in the best possible way.

**IsOptimalParagraphEnabled** allows WPF to look ahead in your text, to see if it would make more sense to break the text in a different position than right at the moment where it runs out of space. **IsHyphenationEnabled**allows WPF to split your words with a hyphen, if it would allow for a more natural layout of the text.

In the next example, I've rendered the same text twice - one without these properties, and one with. The difference is quite obvious:

<Window x:Class="WpfTutorialSamples.Rich\_text\_controls.FlowDocumentTextAlignmentSample"

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

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

Title="FlowDocumentTextAlignmentSample" Height="400" Width="330">

<StackPanel>

<FlowDocumentScrollViewer ScrollViewer.VerticalScrollBarVisibility="Auto">

<FlowDocument>

<Paragraph FontStyle="Italic" FontSize="14" Foreground="Gray">

By setting the

<Bold>IsOptimalParagraphEnabled</Bold> property to true,

you will allow WPF to look ahead on the lines to come, before deciding

where to break. This will usually result in a more pleasant reading

experience. It works especially well in combination with the

<Bold>IsHyphenationEnabled</Bold> property.

</Paragraph>

</FlowDocument>

</FlowDocumentScrollViewer>

<FlowDocumentScrollViewer ScrollViewer.VerticalScrollBarVisibility="Auto">

<FlowDocument IsOptimalParagraphEnabled="True" IsHyphenationEnabled="True">

<Paragraph FontStyle="Italic" FontSize="14" Foreground="Gray">

By setting the <Bold>IsOptimalParagraphEnabled</Bold> property to true,

you will allow WPF to look ahead on the lines to come, before deciding

where to break. This will usually result in a more pleasant reading

experience. It works especially well in combination with the

<Bold>IsHyphenationEnabled</Bold> property.

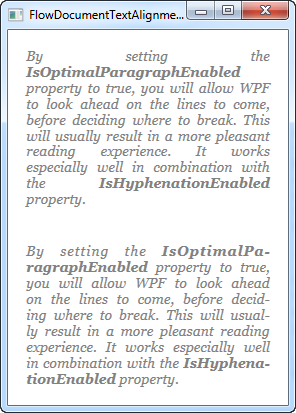
</Paragraph>

</FlowDocument>

</FlowDocumentScrollViewer>

</StackPanel>

</Window>



IsOptimalParagraphEnabled is not enabled by default because it does require a bit more CPU power when rendering the text, especially if the window is frequently resized. For most situations this shouldn't be a problem though.

If you have a lot of FlowDocument instances in your application and you prefer this optimal rendering method, you can enable it on all of your FlowDocument instances by specifying a global style that enables it, in your App.xaml. Here's an example:

<Application x:Class="WpfTutorialSamples.App"

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

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

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

StartupUri="Rich text controls/FlowDocumentTextAlignmentSample.xaml">

<Application.Resources>

<Style TargetType="FlowDocument">

<Setter Property="IsOptimalParagraphEnabled" Value="True" />

<Setter Property="IsHyphenationEnabled" Value="True" />

</Style>

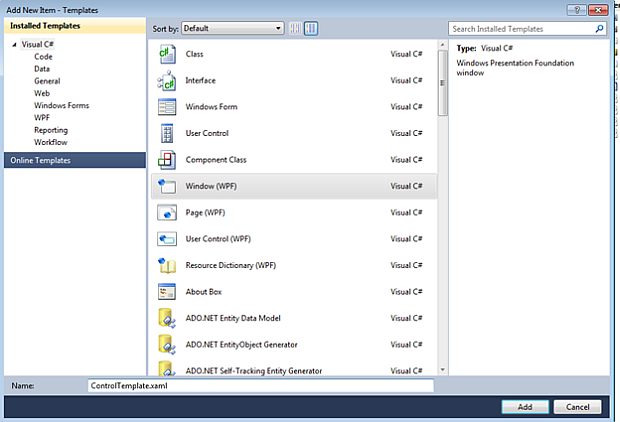
</Application.Resources>

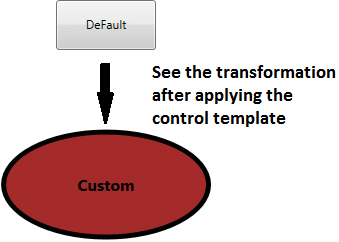
</Application>

**Introduction**  
Each control has its own default template associated with it. Using styles, you can only modify the default associated template. WPF enables you to change the look and feel of the controls and this can be achieved by using the templates.  
  
There are four types of templates, which are shown below.

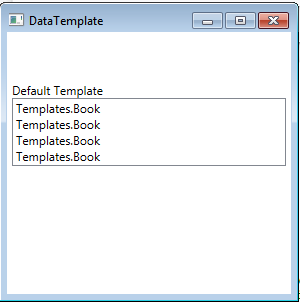
* Control Template
* Data Template
* ItemsPanel Template
* HierarchalData Template

**Control Template**  
  
Control Template enables you to customize the default appearance and behavior of the control. This can be achieved by setting the dependency property “Template” to an instance of Control Template.  
  
**Example**  
Let’s create a Control Template for a button.

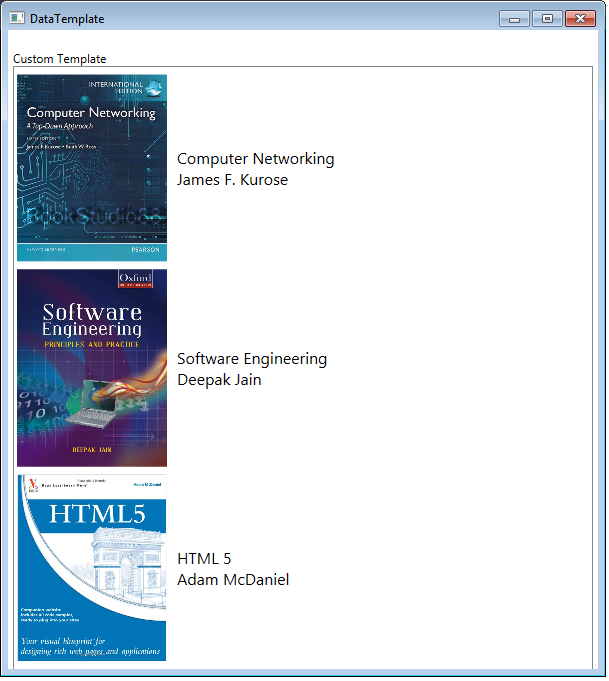
1. Create a new Project “Templates” and add XAML file ControlTemplate.xaml  
     
   
2. Place a button in it.
   1. <Grid>
   2. <Button Margin="50” Foreground=" Black " Content="Custom " ></Button>
   3. </Grid
3. Add Button.Template tag and create ControlTemplate within it, as shown below.
   1. <Button Margin="50” Foreground=" Black " Content="Custom " >
   2. <Button.Template>
   3. <ControlTemplate TargetType="{x:Type Button} ">
   4. <Grid>
   5. <Ellipse Width="210 " Height="110 " Fill="Black "/>
   6. <Ellipse Width="200 " Height="100 " Name="Button " Fill="Brown” />
   7. <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" /> </Grid>
   8. </ControlTemplate>
   9. </Button.Template>
   10. </Button>

As soon as you create a ControlTemplate, the button will replace its default template with the template created by you.  
  
  
  
**Data Template**  
  
Data Template enables you to customize the appearance of the data objects. It is beneficial when a collection of objects binds with ItemControls like ListView, ListBox,ComboBox.  
  
To understand the importance of Data Templates, let’s see what happens when a ListBox is created without Data Template.  
  
**Steps - ListBox without DataTemplate**

1. Add the new XAML file DataTemplate.xaml and place a Listbox control in it.
   1. <Grid>
   2. <ListBox Name="dataTemplate"></ListBox>
   3. </Grid>
2. Create a class “Book”, as shown below.
   1. **public** **class** Book {
   2. **public** string CoverImage {
   3. get;
   4. set;
   5. }
   6. **public** string Name {
   7. get;
   8. set;
   9. }
   10. **public** string Author {
   11. get;
   12. set;
   13. }
   14. }
3. Create a collection of instance of class Book and bind the collection with the ListBox, as shown below.
   1. **public** partial **class** DataTemplate: Window {
   2. **public** DataTemplate() {
   3. InitializeComponent();
   4. // Create the Collection
   5. List < Book > bookList = **new** List < Book > ();
   6. bookList.Add(**new** Book() {
   7. CoverImage = @ "images\ComputerNetworking6E.jpg",
   8. Name = "Computer Networking",
   9. Author = "James F. Kurose"
   10. });
   11. bookList.Add(**new** Book() {
   12. CoverImage = @ "images\software-engineering-oup.jpg",
   13. Name = "Software Engineering",
   14. Author = "Deepak Jain"
   15. });
   16. bookList.Add(**new** Book() {
   17. CoverImage = @ "images\MyCoverImage.jpg",
   18. Name = "HTML 5",
   19. Author = "Adam McDaniel"
   20. });
   21. bookList.Add(**new** Book() {
   22. CoverImage = @ "images\9780134133164.jpg",
   23. Name = "Visual Studio 2015",
   24. Author = "Lars Powers"
   25. });
   26. //Bind it with the ListBox
   27. **this**.dataTemplate.ItemsSource = bookList;
   28. }
   29. }

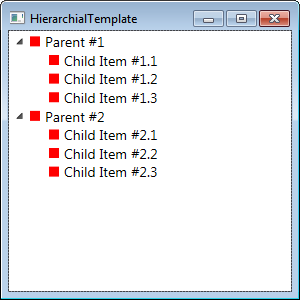
**Output**  
  
  
The default template of the data object is a Textblock. Thus, if we bind the objects to it without the data template, ToString() method is called on it and the data is shown as the string.  
  
Now, let’s see what happens with the data template.  
  
**Additional Step - ListBox with DataTemplate**  
  
Add ListBox.ItemTemplate tag and create the Data Template within it, as shown below.

1. <ListBox Name="dataTemplate">
2. <ListBox.ItemTemplate>
3. <DataTemplate>
4. <StackPanel Orientation="Horizontal" VerticalAlignment="Center">
5. <Image Source="{Binding CoverImage}" Height="200" Width="150"></Image>
6. <StackPanel Orientation="Vertical" VerticalAlignment="Center">
7. <TextBlock Text="{Binding Name}" FontSize="16"></TextBlock>
8. <TextBlock Text="{Binding Author}" FontSize="16"></TextBlock>
9. </StackPanel>
10. </StackPanel>
11. </DataTemplate>
12. </ListBox.ItemTemplate>
13. </ListBox>

**Output**  
  
  
For proper pictorial presentation of the data objects, we should create ItemsControl with DataTemplate but what if we need to customize the default layout of items? In this case ItemsPanelTemplate comes into the picture.  
  
**ItemsPanelTemplate**  
  
ItemsPanelTemplate enables you to customize the panel, which defines the layout of items in ItemControls like ListBox and ListView. Every ItemControl has its default panel.  
  
For example, Default panel for ListBox is VirtualizingStackPanel.  
  
To understand it in more detail, let's customize the layout of ListBox in the example stated above. Listbox renders all the items vertically aligned one after the other, each item occupying the whole row. This layout can be customized, as shown below.  
  
Add ListBox.ItemsPanel tag and create ItemsPanelTemplate within it.

1. <ListBox.ItemsPanel>
2. <ItemsPanelTemplate>
3. <UniformGrid Columns="3" /> </ItemsPanelTemplate>
4. </ListBox.ItemsPanel>

  
  
**Output**  
  
  
  
It’s clear now that the template of ItemControls like ListBox & ListView can be customized, using DataTemplate and ItemsPanelTemplate. WPF also provides an ItemControl called TreeView, which is hierarchical in nature. DataTemplate and ItemPanelTemplate are of no use in this case.  
  
**HierarchialDataTemplate**  
  
HierarchialDataTemplate enables you to customize the template of Parent TreeViewItems as well as their Child TreeViewItems.  
  
Let’s take an example to understand it in more detail

1. Create a Child class and declare a string type property “Title”, as shown below.
   1. **public** **class** Child {
   2. **public** Child(string title) {
   3. Title = title;
   4. }
   5. **public** string Title {
   6. get;
   7. set;
   8. }
   9. }
2. Create a Parent class and declare a string type property “Title” and a List of type Child class “ChildItems”, as shown below.
   1. **public** **class** Parent {
   2. **public** Parent(string title) {
   3. Title = title;
   4. ChildItems = **new** List < Child > ();
   5. }
   6. **public** string Title {
   7. get;
   8. set;
   9. }
   10. **public** List < Child > ChildItems {
   11. get;
   12. set;
   13. }
   14. }
3. Now, create a dummy Hierarchical collection.
   1. **var** parent1 = **new** Parent("Parent #1") {
   2. ChildItems = {
   3. **new** Child("Child Item #1.1"),
   4. **new** Child("Child Item #1.2"),
   5. **new** Child("Child Item #1.3")
   6. }
   7. };
   8. **var** parent2 = **new** Parent("Parent #2") {
   9. ChildItems = {
   10. **new** Child("Child Item #2.1"),
   11. **new** Child("Child Item #2.2"),
   12. **new** Child("Child Item #2.3")
   13. }
   14. };
   15. **this**.treeView.Items.Clear();
   16. List < Parent > parent = **new** List < Parent > ();
   17. parent.Add(parent1);
   18. parent.Add(parent2);
4. Add a TreeView and create a HierarchialDataTemplate for it
   1. <Grid>
   2. <TreeView Name="treeView">
   3. <TreeView.ItemTemplate>
   4. <HierarchicalDataTemplate ItemsSource="{Binding ChildItems}">
   5. <StackPanel Orientation="Horizontal">
   6. <Rectangle Height="10" Width="10" Fill="Red"></Rectangle>
   7. <TextBlock Text="{Binding Title}"></TextBlock>
   8. </StackPanel>
   9. </HierarchicalDataTemplate>
   10. </TreeView.ItemTemplate>
   11. </TreeView>
   12. </Grid>
5. Bind the hierarchical collection parent to the TreeView this.treeView.ItemsSource = parent;  
     
   **Output**  
     
     
     
   The template has changed the look and feel of the parent as well as their child items.

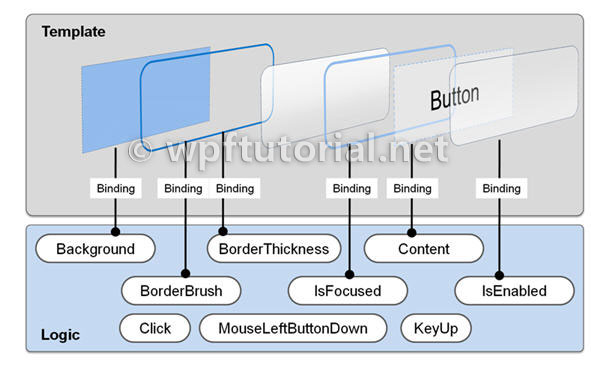
# Control Templates

## Introduction

Controls in WPF are separated into **logic**, that defines the states, events and properties and **template**, that defines the visual appearance of the control. The wireup between the logic and the template is done by DataBinding.

Each control has a default template. This gives the control a basic appearance. The default template is typically shipped together with the control and available for all common windows themes. It is by convention wrapped into a style, that is identified by value of the DefaultStyleKey property that every control has.

The template is defined by a dependency property called Template. By setting this property to another instance of a control template, you can completely replace the appearance (visual tree) of a control.



The control template is often included in a style that contains other property settings. The following code sample shows a simple control template for a button with an ellipse shape.

<Style x:Key="DialogButtonStyle" TargetType="Button">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type Button}">

<Grid>

<Ellipse Fill="{TemplateBinding Background}"

Stroke="{TemplateBinding BorderBrush}"/>

<ContentPresenter HorizontalAlignment="Center"

VerticalAlignment="Center"/>

</Grid>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

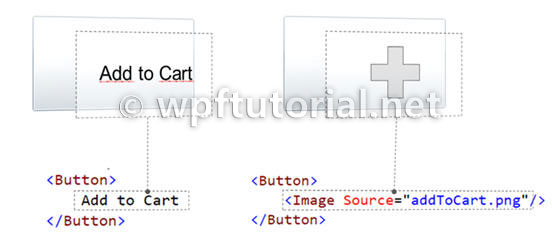
<Button Style="{StaticResource DialogButtonStyle}" />



*A Button without and with a custom control template*

## ContentPresenter

When you create a custom control template and you want to define a placeholder that renders the content, you can use the ContentPresenter. By default it adds the content of the Content property to the visual tree of the template. To display the content of another property you can set the ContentSource to the name of the property you like.



## Triggers

### {RelativeSource TemplatedParent} not working in DataTriggers of a ControlTemplate

If you want to bind to a property of a property on your control like Data.IsLoaded you cannot use a normal Trigger, since it does not support this notation, you have to use a DataTrigger.

But when you are using a DataTrigger, with {RelativeSource TemplatedParent} it will not work. The reason is, that **TemplatedParent can only be used within the ControlTemplate**. It is not working in the Trigger section. You have to use the {RelativeSource Self} instead.

## What if a Binding working or a Setter is not applied when using a control template

There is something you need to know when setting a value of an element within a control template: The value does have a lower precendence as the local value! So if you are setting the local value in the constructor of the contained element, you cannot override it within the controltemplate. But if you use the element directly in your view, it will work. So be aware of this behavior!.  
  
Here you can find more information about DependencyProperty value precendence: [Dependency Property Value Precedence](http://msdn.microsoft.com/en-us/library/ms743230.aspx)

# Data Templates

## Introduction

Data Template are a similar concept as [Control Templates](https://www.wpftutorial.net/templates.html). They give you a very flexible and powerful solution to **replace the visual appearance of a data item** in a control like ListBox, ComboBox or ListView. In my opinion this is one of the key success factory of WPF.

If you don't specify a data template, WPF takes the default template that is just a TextBlock. If you bind complex objects to the control, it just calls ToString() on it. Within a DataTemplate, the DataContext is set the data object. So you can easily bind against the data context to display various members of your data object

## DataTemplates in Action: Building a simple PropertyGrid

Whereas it was really hard to display complex data in a ListBox with WinForms, its super easy with WPF. The following example shows a ListBox with a list of DependencyPropertyInfo instances bound to it. Without a DataTemplate you just see the result of calling ToString() on the object. With the data template we see the name of the property and a TextBox that even allows us to edit the value.



*<!-- Without DataTemplate -->*

<ListBox ItemsSource="{Binding}" />

*<!-- With DataTemplate -->*

<ListBox ItemsSource="{Binding}" BorderBrush="Transparent"

Grid.IsSharedSizeScope="True"

HorizontalContentAlignment="Stretch">

<ListBox.ItemTemplate>

<DataTemplate>

<Grid Margin="4">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" SharedSizeGroup="Key" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<TextBlock Text="{Binding Name}" FontWeight="Bold" />

<TextBox Grid.Column="1" Text="{Binding Value }" />

</Grid>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

## How to use a DataTemplateSelector to switch the Template depending on the data

Our property grid looks nice so far, but it would be much more usable if we could switch the editor depending on the type of the property.

The simplest way to do this is to use a DataTemplateSelector. The DataTemplateSelector has a single method to override: SelectTemplate(object item, DependencyObject container). In this method we decide on the provided item which DataTemplate to choose.

The following exmple shows an DataTemplateSelector that decides between tree data templates:

**public** **class** PropertyDataTemplateSelector : DataTemplateSelector

{

**public** DataTemplate DefaultnDataTemplate { get; set; }

**public** DataTemplate BooleanDataTemplate { get; set; }

**public** DataTemplate EnumDataTemplate { get; set; }

**public** **override** DataTemplate SelectTemplate(**object** item,

DependencyObject container)

{

DependencyPropertyInfo dpi = item **as** DependencyPropertyInfo;

**if** (dpi.PropertyType == **[typeof](http://www.google.com/search?q=typeof+msdn.microsoft.com)**(**bool**))

{

**return** BooleanDataTemplate;

}

**if** (dpi.PropertyType.IsEnum)

{

**return** EnumDataTemplate;

}

**return** DefaultnDataTemplate;

}

}

<Window x:Class="DataTemplates.Window1"

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

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

xmlns:l="clr-namespace:DataTemplates"

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

<Window.Resources>

*<!-- Default DataTemplate -->*

<DataTemplate x:Key="DefaultDataTemplate">

...

</DataTemplate>

*<!-- DataTemplate for Booleans -->*

<DataTemplate x:Key="BooleanDataTemplate">

...

</DataTemplate>

*<!-- DataTemplate for Enums -->*

<DataTemplate x:Key="EnumDataTemplate">

...

</DataTemplate>

*<!-- DataTemplate Selector -->*

<l:PropertyDataTemplateSelector x:Key="templateSelector"

DefaultnDataTemplate="{StaticResource DefaultDataTemplate}"

BooleanDataTemplate="{StaticResource BooleanDataTemplate}"

EnumDataTemplate="{StaticResource EnumDataTemplate}"/>

</Window.Resources>

<Grid>

<ListBox ItemsSource="{Binding}" Grid.IsSharedSizeScope="True"

HorizontalContentAlignment="Stretch"

ItemTemplateSelector="{StaticResource templateSelector}"/>

</Grid>

</Window>

## How to react to IsSelected in the DataTemplate

If you want to change the appearance of a ListBoxItem when it is selected, you have to bind the IsSelected property of the ListBoxItem. But this is a bit tricky, you have to use a relative source with FindAcestor to navigate up the visual tree until you reach the ListBoxItem.

<DataTemplate x:Key="DefaultDataTemplate">

<Border x:Name="border" Height="50">

...

</Border>

<DataTemplate.Triggers>

<DataTrigger Binding="{Binding RelativeSource=

{RelativeSource Mode=FindAncestor, AncestorType=

{x:Type ListBoxItem}},Path=IsSelected}" Value="True">

<Setter TargetName="border" Property="Height" Value="100"/>

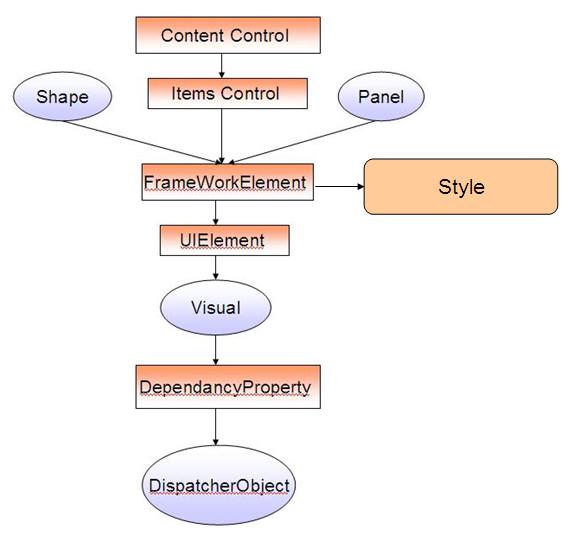
</DataTrigger>

</DataTemplate.Triggers>

</DataTemplate>

## Style

WPF exposes a property Style for every Control. If you look into the object Hierarchy, the Style is basically a property which exposes an object of Style in FrameworkElement. So each object can associate it and define custom setters to manipulate the basic look and feel of a control.



Clearly, the above diagram shows the association of Style in FrameworkElement and from the object hierarchy every control somehow inherits from FrameworkElement and hence style will be available to it. Style is also a WPF object which is inherited form DispatcherObject which helps in setting different properties of your UI Element.

### How Style differs from Theme ?

Before we move further into Styles lets talk about Themes. Theme is totally different from Styles. Themes are defined at OS level, or more precisely a Theme can take part of delivering styles all over the Desktop while Styles are restricted to the contextual area of a WPF window. WPF are capable of retrieving the color scheme which is defined in OS level. Say for instance, if you do not define style for your application, the elements in the screen will automatically get styles from external environment. Say for instance, in XP if you change the theme to something else you would see that the buttons, TextBox on your WPF window will change its color instantly. You can even set the Theme which the application would use [programmatically](http://arbel.net/blog/archive/2006/11/03/Forcing-WPF-to-use-a-specific-Windows-theme.aspx) [[^](http://arbel.net/blog/archive/2006/11/03/Forcing-WPF-to-use-a-specific-Windows-theme.aspx)] from your code.

### What about Templates ?

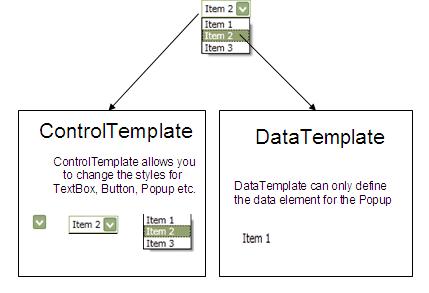
Every control defines a ControlTemplate. A ControlTemplate defines the overall structure of the control. As I have already told you, say for instance you have a Button. Button is a control that is made up of more than one control. It would have a ContentPresenter which writes the Text over the control, it would have a Rectangle which keeps the boundary of the Button etc. So Template is a special property associated with a Control which specifies how the control will look like structurally. We can easily define our Template and change the overall structure of a control.

Templates are basically of 2 types :

* 1. **ControlTemplate**
  2. **DataTemplate**

ControlTemplate defines the structure of the Control. It means say for instance, you define the ControlTemplate for a ComboBox. So from ControlTemplate you can easily change the ToggleButton associated with the ComboBox which opens the DropDown, you can change the structure of the TextBox, the Popup etc. So ControlTemplate allows you to change the overall structure of the Control.

Each control is made up of Data. Say for instance a ItemsControl contains a number of Data Element which builds the items inside the Popup. The DataTemplate could be associated with ItemsTemplate and will build up the Data Block for the ComboBox.



So, you should always remember, ControlTemplate defines the whole Control while the DataTemplate defines each individual Data Element.

### How to define Style?

Normally a style is an unique object which is used to style WPF controls. Each WPF element contains a number of Dependency Properties. A dependency property defines the basic behavior and look of the control in UI. Styles maintains a collection of Setters which enumerates a DependencyProperty with its value.

Thus you can say a style is a collection of DependencyProperty settings which when applied on a Target will change the behavior of it.

Let us suppose you are going to style a TextBox.



Hide   Copy Code

<TextBox Text="This is a TextBox without Styles"

HorizontalAlignment="Center"

VerticalAlignment="Center"

CharacterCasing="Lower"

FlowDirection="RightToLeft"

FontSize="20"

FontWeight="UltraBlack"

Width="400"

Height="40">

<TextBox.Background>

<LinearGradientBrush>

<GradientStop Color="Cyan" Offset="0.0"/>

<GradientStop Color="Yellow" Offset="0.5"/>

<GradientStop Color="Red" Offset="1.0"/>

</LinearGradientBrush>

</TextBox.Background>

<TextBox.Foreground>

<SolidColorBrush Color="Black"/>

</TextBox.Foreground>

<TextBox.Effect>

<DropShadowEffect BlurRadius="40" Color="Maroon"

Direction="50" Opacity="0.5"/>

</TextBox.Effect>

</TextBox>

So I have just designed a TextBox in the above code. The XAML looks straight forward, where I have configured different properties of the TextBox control to create my stylish TextBox. But looking at the code, you might wonder how difficult it would be if you need to redo the same thing again and again for every TextBox you define in your application. This is what the problem is. So WPF comes with an alternative with style. A style is an object that holds this behaviors into a collection of Setters. So lets redefine the same with Styles.



Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

<TextBox>

<TextBox.Style>

<Style TargetType="{x:Type TextBox}">

<Setter Property="Text" Value="This is a TextBox with Styles"/>

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="VerticalAlignment" Value="Center"/>

<Setter Property="CharacterCasing" Value="Lower"/>

<Setter Property="FlowDirection" Value="RightToLeft"/>

<Setter Property="FontSize" Value="20"/>

<Setter Property="FontWeight" Value="UltraBlack"/>

<Setter Property="Width" Value="400"/>

<Setter Property="Height" Value="40"/>

<Setter Property="Background">

<Setter.Value>

<LinearGradientBrush>

<GradientStop Color="Cyan" Offset="0.0"/>

<GradientStop Color="Yellow" Offset="0.5"/>

<GradientStop Color="Red" Offset="1.0"/>

</LinearGradientBrush>

</Setter.Value>

</Setter>

<Setter Property="Foreground">

<Setter.Value>

<SolidColorBrush Color="Black"/>

</Setter.Value>

</Setter>

<Setter Property="Effect" >

<Setter.Value>

<DropShadowEffect BlurRadius="40"

Color="Maroon" Direction="50" Opacity="0.5"/>

</Setter.Value>

</Setter>

</Style>

</TextBox.Style>

</TextBox>

So you can see, I have defined the Style inside the TextBox and the textbox looks almost the same. The Setters allows you to enumerate all the properties for the TextBox and produced a style inside it whose TargetType is set to {x:Type Button}

Now how this style can be made reusable for many controls ? Yes, this might be your first question that arose in your mind. Yes, if you have read my previous articles, you should already know the use of ResourceDictionaries. So in our case I will shift the style to Resource section for the Window and reuse the code just by calling the Resource key from the Textbox.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

<Grid>

<Grid.Resources>

<ResourceDictionary>

<Style TargetType="{x:Type TextBox}" x:Key="MyTextBoxStyle">

<Setter Property="Text" Value="This is a TextBox with Styles"/>

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="VerticalAlignment" Value="Center"/>

<Setter Property="CharacterCasing" Value="Lower"/>

<Setter Property="FlowDirection" Value="RightToLeft"/>

<Setter Property="FontSize" Value="20"/>

<Setter Property="FontWeight" Value="UltraBlack"/>

<Setter Property="Width" Value="400"/>

<Setter Property="Height" Value="40"/>

<Setter Property="Margin" Value="0,20,0,10" />

<Setter Property="Background">

<Setter.Value>

<LinearGradientBrush>

<GradientStop Color="Cyan" Offset="0.0"/>

<GradientStop Color="Yellow" Offset="0.5"/>

<GradientStop Color="Red" Offset="1.0"/>

</LinearGradientBrush>

</Setter.Value>

</Setter>

<Setter Property="Foreground">

<Setter.Value>

<SolidColorBrush Color="Black"/>

</Setter.Value>

</Setter>

<Setter Property="Effect" >

<Setter.Value>

<DropShadowEffect BlurRadius="40"

Color="Maroon" Direction="50" Opacity="0.5"/>

</Setter.Value>

</Setter>

</Style>

</ResourceDictionary>

</Grid.Resources>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<TextBox Style="{StaticResource MyTextBoxStyle}" Grid.Row="0" />

<TextBox Style="{StaticResource MyTextBoxStyle}" Grid.Row="1"

Text="The Style is modified here"

FlowDirection="LeftToRight"/>

</Grid>



So here I have shifted the Style into Resource section and used MyTextBoxStyle key to refer for each TextBox i defined. Notably, the style of both the textboxes remains same, while you can see I have also overridden certain settings in the control itself and it works the same. I have modified the Text of the 2nd TextBox to **"The Style is modified here"** and also made the FlowDirection to LeftToRight.

Another important thing, that you should always keep into mind, that if you do not define the Key element for the Style in Resource section, it will automatically be applied to all the TextBox you define.

Hide   Copy Code

<Style TargetType="{x:Type TextBox}">

</Style>

Say the style you define does not contain any Key. So all the TextBoxes will automatically apply the style when appeared. You can eventually use

Hide   Copy Code

<TextBox Style="{x:Null}"/>

to revert the style.

## Members of Style



The styling of WPF controls is made up with the help of a class called Style. The style object exposes few properties which help you to define various behavior. Lets look into the properties:

* + **Resources** : It holds the reference for the ResourceDictionary where the Style is defined.
  + **Setters** : It is a collection which holds all the DependencyProperty configuration for the whole control.
  + **TargetType** : TargetType defines the type of the control for which the Style can be applied. So based on the TargetType the Style setters are defined to. So if you define a style for TextBox you cannot use Content as property Setter.
  + **BasedOn** : This is used to allow Style inheritance. You can use an existing style key to inherit all the properties to a new Style.
  + **Triggers** : A collection of Setters which would be applied based on certain conditions.

Using those properties you can define your own styles.

## What about Explicit and Implicit Styles ?

WPF controls can have two type of styles associated with it. A control can have a style defined in the application and applied to its Style property. If your control is using a Style to define its look and feel or basically your control has set an object of Style into its Style property, then it is using an Explicit Style.

On the other hand, if your control takes the style from external environment (Theme) and the Style property is set to Null, then your control is using Implicit Style. Basically any WPF control automatically defines a DefaultStyle for it, so that you can set only the portion of the control which you need to change.

Say for instance, you have a Button. If you want to have its Text to be colored Red, you just need to change the Foreground of the Button. You need not to define the whole style. If there is no Default Style defined for Buttons, you need to define all the properties individually to make it appear. Thus the default color of the Text is Black if not defined otherwise.

## Triggers

Triggers are a set of styles that work on a particular condition. You can think Trigger as a part of Style which will be set only when the Condition defined for the Trigger is met.

There are few types of Triggers :

* + **Property Trigger** : Will be set only when the DependencyProperty of a certain object has been set to a Value.
  + **Data Trigger** : Will work for any normal Properties using on Binding.
  + **Event Trigger** : Will work only when some event is triggered from the control.

Now to demonstrate let us look into the code below :

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

<Style TargetType="{x:Type TextBox}" x:Key="MyTextBoxStyle">

<Setter Property="Text" Value="This is a TextBox with Styles"/>

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="VerticalAlignment" Value="Center"/>

<Setter Property="CharacterCasing" Value="Lower"/>

<Setter Property="FlowDirection" Value="RightToLeft"/>

<Setter Property="FontSize" Value="20"/>

<Setter Property="FontWeight" Value="UltraBlack"/>

<Setter Property="Width" Value="400"/>

<Setter Property="Height" Value="40"/>

<Setter Property="Margin" Value="0,20,0,10" />

<Setter Property="Background">

<Setter.Value>

<LinearGradientBrush>

<GradientStop Color="Cyan" Offset="0.0"/>

<GradientStop Color="Yellow" Offset="0.5"/>

<GradientStop Color="Red" Offset="1.0"/>

</LinearGradientBrush>

</Setter.Value>

</Setter>

<Setter Property="Foreground">

<Setter.Value>

<SolidColorBrush Color="Black"/>

</Setter.Value>

</Setter>

<Setter Property="Effect" >

<Setter.Value>

<DropShadowEffect BlurRadius="40" Color="Maroon" Direction="50"

Opacity="0.5"/>

</Setter.Value>

</Setter>

<Style.Triggers>

<Trigger Property="IsFocused" Value="True">

<Setter Property="Effect">

<Setter.Value>

<DropShadowEffect BlurRadius="40" Color="Red"

Direction="50" Opacity="0.9"/>

</Setter.Value>

</Setter>

</Trigger>

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="IsFocused" Value="True"/>

<Condition Property="IsMouseOver" Value="True"/>

</MultiTrigger.Conditions>

<Setter Property="Effect">

<Setter.Value>

<DropShadowEffect BlurRadius="40" Color="Violet"

Direction="50" Opacity="0.9"/>

</Setter.Value>

</Setter>

<Setter Property="Foreground" Value="White" />

<Setter Property="Background" Value="Maroon" />

</MultiTrigger>

</Style.Triggers>

</Style>

Here you can see I have used Property Trigger to change the DropShadowEffect of TextBox when it is focussed. Every WPF control exposes few properties to work with Property Triggers, which will be set to true based on the control appearance changes. You can use these properties like IsFocused, IsMouseDown etc to work around with Property Triggers.

On the second occasion, I have defined a MultiTrigger. MultiTrigger allows you to mention Condition, so that when all the conditions of the MultiTrigger is met, the Property Setters for the object is applied.



So you can see when you hover your mouse over the TextBox and your textbox has its focus in it, only then you see the TextBox to appear in Maroon background and Violet DropShadow effect.

## Animation Basics

Another interesting thing that you might think very interesting is the support of Animation for WPF. Basically, by the word Animation, we generally think of large Texture graphics in 3D space, which would probably be created in 3DS MAX studio or MAC etc. But believe me there is nothing to worry about this in case of WPF. WPF simplifies the concept Animation to be the change of a property over time.

Say for instance, say you want your textbox to change its color over time, you would write a simple color animation to do this or say you want to change the Opacity of a Border element during time, you need DoubleAnimation to do this. Animation is cool if you are clear about how it works.

### Type of Animation

I must say, don't make yourself more confused by seeing the types of Animation. Animation is actually categorized in the same way as you categorize variables. Say for instance :

* 1. **DoubleAnimation** : This will animate a Double Value from one value to another. So if you want to change the Width of a TextBox over time you need DoubleAnimation.
  2. **ColorAnimation** : Same as the above if the type of Changing element is Color, you need ColorAnimation.
  3. SingleAnimation, RectAnimation, PointAnimation, Int32Animaition, ThicknessAnimation etc each of them bears the same meaning.

So basically the basis of Animation types is based on the type of the property for which you want your animation to work on.

Animation can also be categorized into two basic ways :

* 1. **Animation Without KeyFrames** : These are animation that only needs two values, From and To. It gives you a smooth animation based on the Timeline.DesiredFramerate property for the animation.
  2. **Animation With KeyFrames** : Allows you to specify a KeyFrame collection which lets you define the KeyFrame value on a specified time. So that you can adjust your own animation based on specific time intervals.

Let us take a look at a few examples to make you understand animation feature of WPF:

Hide   Copy Code

<Window.Triggers>

<EventTrigger RoutedEvent="Loaded">

<BeginStoryboard>

<Storyboard RepeatBehavior="Forever">

<DoubleAnimation Storyboard.TargetProperty="Width"

From="300" To="200" AutoReverse="True" Duration="0:0:5" ></DoubleAnimation>

<DoubleAnimation Storyboard.TargetProperty="Height"

From="300" To="200" AutoReverse="True" Duration="0:0:5"></DoubleAnimation>

</Storyboard>

</BeginStoryboard>

</EventTrigger>

</Window.Triggers>

In the above code, I have defined an EventTrigger which lets you have a DoubleAnimation(as Width is double value) on Width of the Window. We use Loaded Event to start a StoryBoard.

### What is a StoryBoard ?

A StoryBoard can be defined as a Container for TimeLines or a collection of animation timelines for which the object specified in Target will animate. We use StoryBoard to specify Animation within it.

Few important properties of StoryBoard :

* 1. **RepeatBehaviour** : Specifies the number of times for which the StoryBoard repeat the animation.
  2. **Target** : Specifies the Target for which the storyboard will be applied to.
  3. **TargetName** : Defines the target and reference it by its name attribute.
  4. **TargetProperty** : Specifies the property for which the animation will be applied for.
  5. **AccelerationRatio / DecelerationRatio** : Defines the acceleration or deceleration for the animation.
  6. **AutoReverse** : Defines whether the StoryBoard will be reversed automatically. This is really cool concept, which allows you to get the reverse of the storyboard timeline automatically generated by the WPF.

Animation can also be applied from code.

Hide   Copy Code

DoubleAnimation myDoubleAnimation = new DoubleAnimation();

myDoubleAnimation.From = 1.0;

myDoubleAnimation.To = 0.0;

myDoubleAnimation.Duration = new Duration(TimeSpan.FromSeconds(5));

In the above code I have declared a DoubleAnimation which starts From 1.0 and moves to 0.0 in 5 seconds.

### Animation with KeyFrames

Animation can be defined either using KeyFrames or without KeyFrames. KeyFrame allows you to define an intermediate frame so that the animation occurs for each individual frame intervals. There are three types of interpolation for an AnimationwithKeyFrames.

* 1. Linear
  2. Discrete
  3. Spline

#### Linear

Lets create an animation using KeyFrames :

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

<Border Background="Violet"

HorizontalAlignment="Stretch"

VerticalAlignment="Stretch" >

<Border.Triggers>

<EventTrigger RoutedEvent="Border.MouseLeftButtonDown">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimationUsingKeyFrames

Storyboard.TargetName="transformObj"

Storyboard.TargetProperty="X"

Duration="0:0:15">

<LinearDoubleKeyFrame Value="500"

KeyTime="0:0:3" />

<LinearDoubleKeyFrame Value="50"

KeyTime="0:0:7" />

<LinearDoubleKeyFrame Value="300"

KeyTime="0:0:13" />

</DoubleAnimationUsingKeyFrames>

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

</Border.Triggers>

<Border.RenderTransform>

<TranslateTransform x:Name="transformObj" X="0" Y="0" />

</Border.RenderTransform>

</Border>

Here the animation is applied as LinearDoubleKeyFrame, which means the animation would be smooth while we define each KeyFrame value based on KeyTime. Here we change the Translation of the Border based on different KeyTime specified such that on 3rd second, the Rectangle will move to 500, at 7th second it will be at 50 and at 13th second it will be at 300. The animation is LinearDouble so the animation is smooth and steady.

#### Discrete

If I change the animation to DiscreteAnimation it will place the object only at the KeyTime specified

Hide   Copy Code

<DoubleAnimationUsingKeyFrames

Storyboard.TargetName="transformObj"

Storyboard.TargetProperty="X"

Duration="0:0:15">

<DiscreteDoubleKeyFrame Value="500" KeyTime="0:0:3" />

<DiscreteDoubleKeyFrame Value="50" KeyTime="0:0:7" />

<DiscreteDoubleKeyFrame Value="300" KeyTime="0:0:13" />

</DoubleAnimationUsingKeyFrames>

Thus changing the LinearDouble with DiscreteDouble makes it change its position all of a sudden based on the KeyTime specified for the animation.

#### Spline

SplineAnimation is used to define more realistic animation behavior for your control. It lets you control acceleration and deceleration of the animation. With KeySpline you can define the the cubic bazier curve using Spline Key frame. Lets look at the example

Hide   Copy Code

<DoubleAnimationUsingKeyFrames

Storyboard.TargetName="transformObj"

Storyboard.TargetProperty="X"

Duration="0:0:15">

<SplineDoubleKeyFrame Value="500" KeyTime="0:0:3" KeySpline="0.0,0.1 0.1,0.1" />

<SplineDoubleKeyFrame Value="50" KeyTime="0:0:7" KeySpline="0.0,0.1 0.1,0.1"/>

<SplineDoubleKeyFrame Value="300" KeyTime="0:0:13" KeySpline="0.0,0.1 0.1,0.1"/>

</DoubleAnimationUsingKeyFrames>