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>

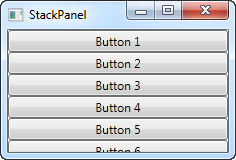
<Button>Button 4</Button>

<Button>Button 5</Button>

<Button>Button 6</Button>

</StackPanel>

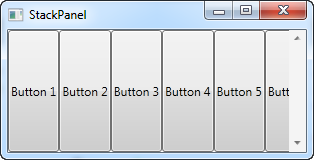
</Window>



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>

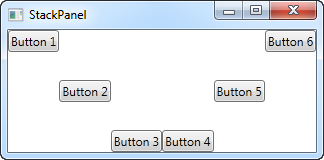
<Button VerticalAlignment="Bottom">Button 4</Button>

<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>

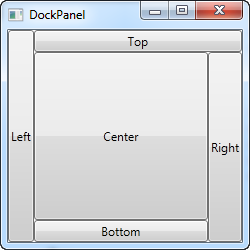
<Button DockPanel.Dock="Right">Right</Button>

<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>

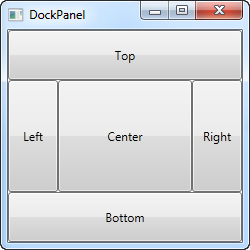
<Button DockPanel.Dock="Left" Width="50">Left</Button>

<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>

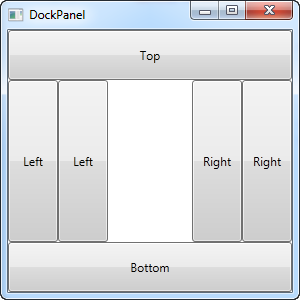
<Button DockPanel.Dock="Left" Width="50">Left</Button>

<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">

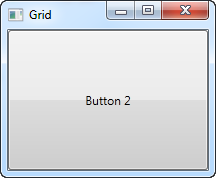
<Grid>

<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="\*" />

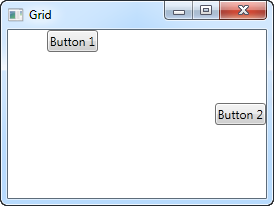
</Grid.ColumnDefinitions>

<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>

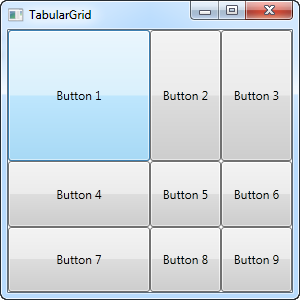
<Button Grid.Row="2">Button 7</Button>

<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>

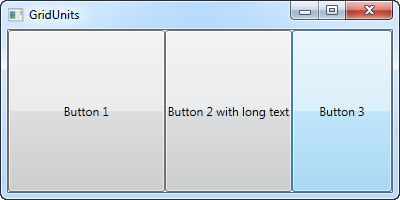
<Button>Button 1</Button>

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

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

</Grid>

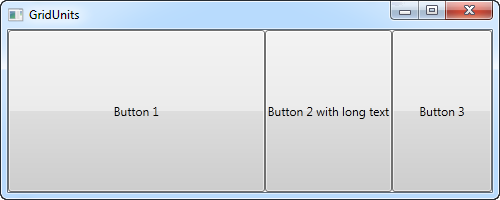
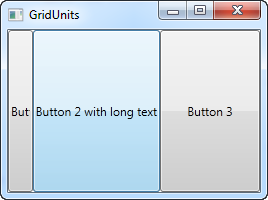
</Window>



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>

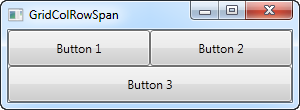
<Button>Button 1</Button>

<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>

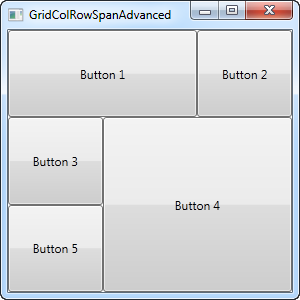
<Button Grid.Row="1">Button 3</Button>

<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>

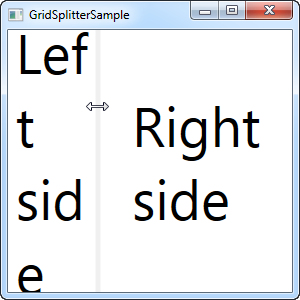
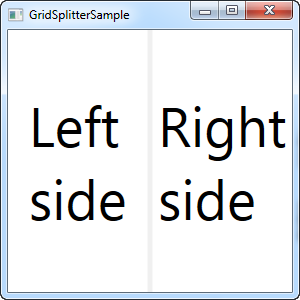
<TextBlock FontSize="55" HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap">Left side</TextBlock>

<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>

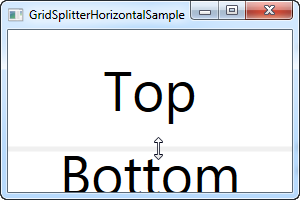
<TextBlock FontSize="55" HorizontalAlignment="Center" VerticalAlignment="Center" TextWrapping="Wrap">Top</TextBlock>

<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>

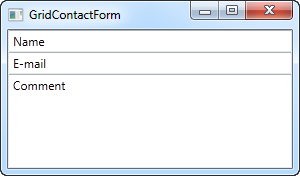
<TextBox>Name</TextBox>

<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>

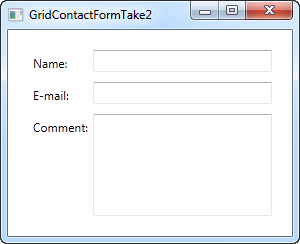
<TextBox Grid.Row="1" Grid.Column="1" Margin="0,0,0,10" />

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

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

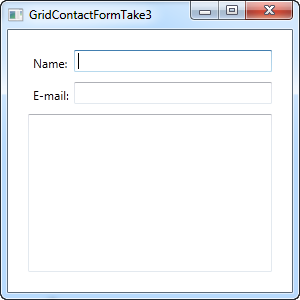
</Grid>

</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>

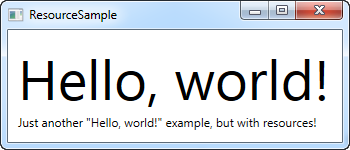
<StackPanel Margin="10">

<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>

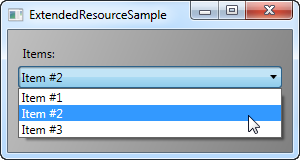
<StackPanel Margin="10">

<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());

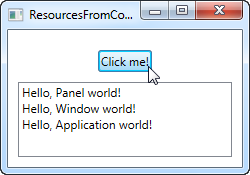
lbResult.Items.Add(this.FindResource("strWindow").ToString());

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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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"

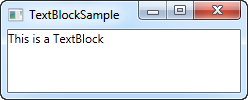
Title="TextBlockSample" Height="100" Width="200">

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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"

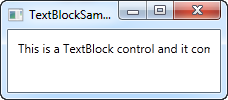
Title="TextBlockSample" Height="100" Width="200">

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

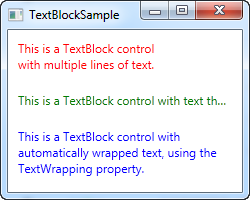
<TextBlock Margin="10" TextWrapping="Wrap" Foreground="Blue">

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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

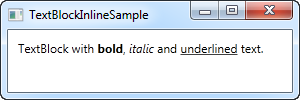
<TextBlock Margin="10" TextWrapping="Wrap">

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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

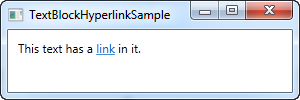
<TextBlock Margin="10" TextWrapping="Wrap">

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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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">

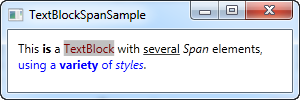
using a <Bold>variety</Bold> of <Italic>styles</Italic>

</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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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"

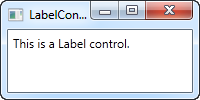
Title="LabelControlSample" Height="100" Width="200">

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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}" />

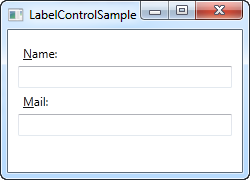
<TextBox Name="txtName" />

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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:" />

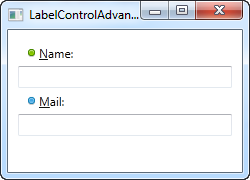
</StackPanel>

</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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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"

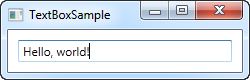
Title="TextBoxSample" Height="80" Width="250">

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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"

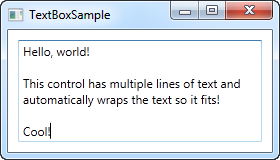
Title="TextBoxSample" Height="160" Width="280">

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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"

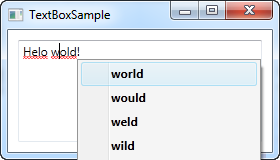
Title="TextBoxSample" Height="160" Width="280">

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

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;

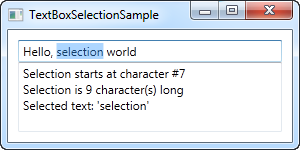
txtStatus.Text += "Selection is " + textBox.SelectionLength + " character(s) long" + Environment.NewLine;

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:**

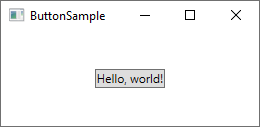
# The Button control

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:

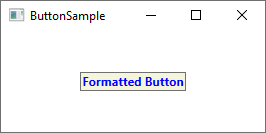
private void HelloWorldButton\_Click(object sender, RoutedEventArgs e)  
{  
    MessageBox.Show("Hello, world!");  
}

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>



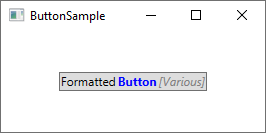
By setting these properties directly on the Button, you are of course limited to applying the same formatting for all of the content, but if that's not good enough, just read on for even more advanced content formatting.

## 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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>



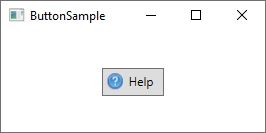
But of course, you are not limited to just text - you can put whatever you want inside your buttons, which leads us to a subject that I know many people will ask for. Buttons with images!

### 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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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

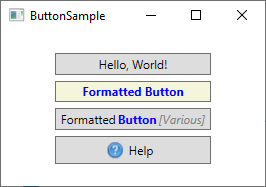
You may have noticed that buttons in the WPF framework doesn't come with any padding by default. This means that the text is very close to the borders, which might look a little bit strange, because most buttons found elsewhere (web, other applications etc.) do have at least some padding in the sides. 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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

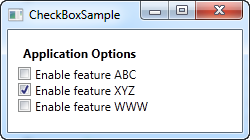
<CheckBox>Enable feature ABC</CheckBox>

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

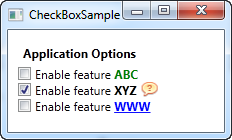
Enable feature <Run Foreground="Blue" TextDecorations="Underline" FontWeight="Bold">WWW</Run>

</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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

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;

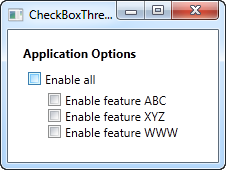
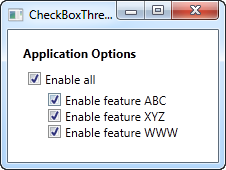
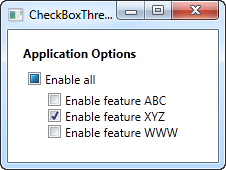
if((cbFeatureAbc.IsChecked == false) && (cbFeatureXyz.IsChecked == false) && (cbFeatureWww.IsChecked == false))

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.

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

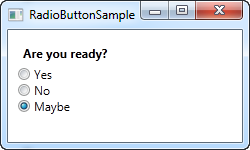
<RadioButton>Yes</RadioButton>

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

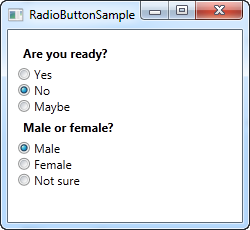
<RadioButton GroupName="sex">Male</RadioButton>

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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" />

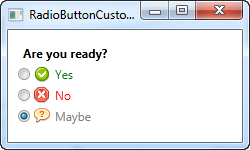
<TextBlock Text="Maybe" Foreground="Gray" />

</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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

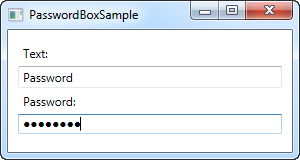
<TextBox />

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

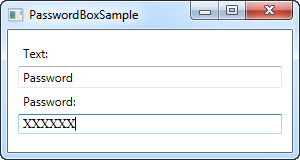
<TextBox />

<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.

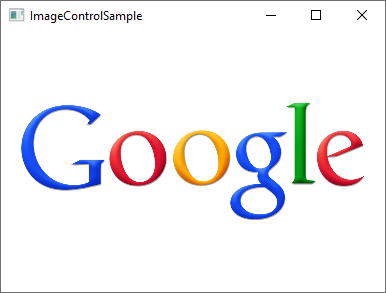
This may or may not be important to you - as already stated, you can still read the password from Code-behind, but for MVVM implementations or if you just love data bindings, a workaround has been developed. You can read much more about it here: <http://blog.functionalfun.net/2008/06/wpf-passwordbox-and-data-binding.html>

# 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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

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:

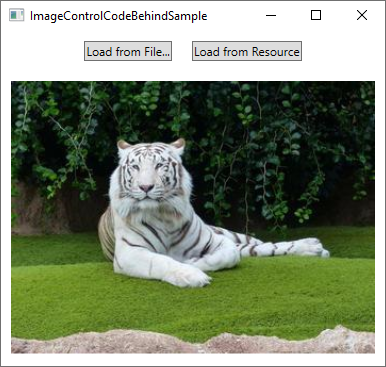
[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

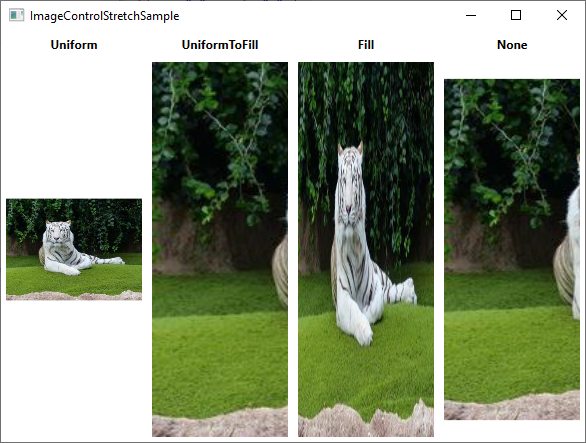
<Window x:Class="WpfTutorialSamples.Basic\_controls.ImageControlCodeBehindSample"  
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
 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.

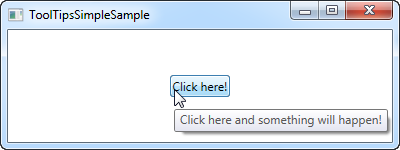
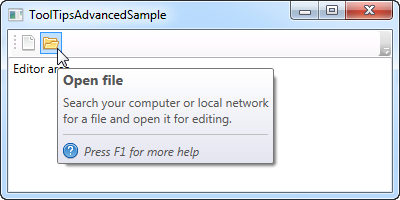
As you can see from this next example, the Stretch property can make quite a bit of difference in how an image is displayed:



[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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:
* [Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)
* <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):

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

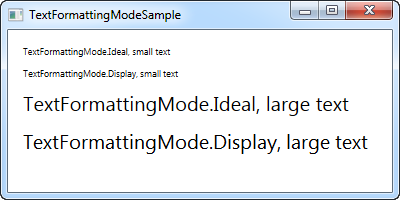
<Label TextOptions.TextFormattingMode="Display" FontSize="9">TextFormattingMode.Display, small text</Label>

<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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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>

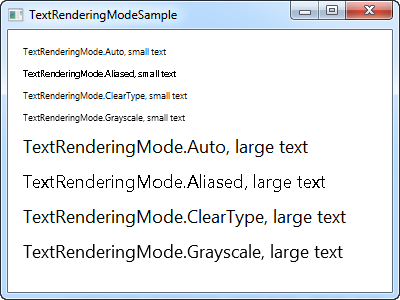
<Label TextOptions.TextRenderingMode="Aliased" FontSize="18">TextRenderingMode.Aliased, large text</Label>

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

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

</StackPanel>

</Window>

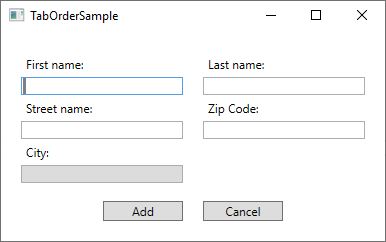


As you can see, the resulting text differs quite a bit in how it looks and once again, you should mainly change this in special circumstances.

# 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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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

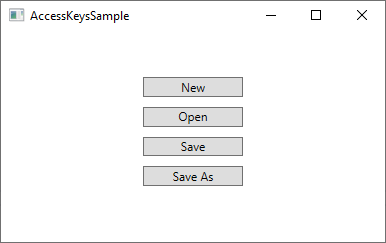
The concept of **Access Keys**, sometimes referred to as Accelerator Keys or Keyboard Accelerators, allows you to reach a specific control inside a window by holding down the Alt key and then pressing another key on the keyboard. This enhances the usability of your windows, because it allows the user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

## Defining Access Keys

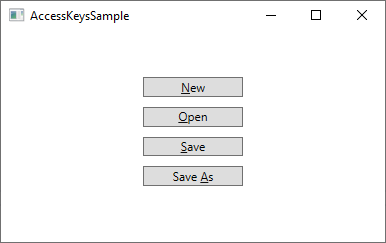
Defining access keys for your WPF control is very easy, but the method might surprise you a bit. Normally, there would be a property for this, but not for Access Keys. Instead, you define the Access Key by prefixing the letter with an underscore in the Text/Content property of the control. For instance,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

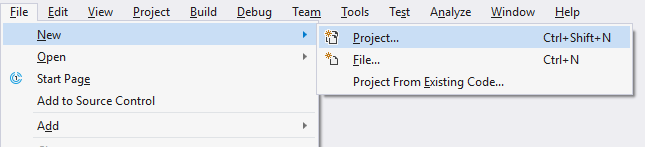


However, as soon as you press the **Alt** key on your Keyboard, the available Access Keys are highlighted by underlining them:



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:

[Download & run this example](https://www.wpf-tutorial.com/download-wpf-tutorial-pdf-with-sample-code/?utm_source=website&utm_medium=link&utm_campaign=codebox)

<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:

