

# WPF Tutorial

# 1.1. About WPF

---

## 1.1.1. What is WPF?

WPF, which stands for Windows Presentation Foundation, is Microsoft's latest approach to a GUI framework, used with the .NET framework.

But what IS a GUI framework? GUI stands for Graphical User Interface, and you're probably looking at one right now. Windows has a GUI for working with your computer, and the browser that you're likely reading this document in has a GUI that allows you to surf the web.

A GUI framework allows you to create an application with a wide range of GUI elements, like labels, textboxes and other well known elements. Without a GUI framework you would have to draw these elements manually and handle all of the user interaction scenarios like text and mouse input. This is a LOT of work, so instead, most developers will use a GUI framework which will do all the basic work and allow the developers to focus on making great applications.

There are a lot of GUI frameworks out there, but for .NET developers, the most interesting ones are currently WinForms and WPF. WPF is the newest, but Microsoft is still maintaining and supporting WinForms. As you will see in the next chapter, there are quite a few differences between the two frameworks, but their purpose is the same: To make it easy to create applications with a great GUI.

In the next chapter, we will look at the differences between WinForms and WPF.

## 1.1.2. WPF vs. WinForms

In the previous chapter, we talked about what WPF is and a little bit about WinForms. In this chapter, I will try to compare the two, because while they do serve the same purpose, there is a LOT of differences between them. If you have never worked with WinForms before, and especially if WPF is your very first GUI framework, you may skip this chapter, but if you're interested in the differences then read on.

The single most important difference between WinForms and WPF is the fact that while WinForms is simply a layer on top of the standard Windows controls (e.g. a TextBox), WPF is built from scratch and doesn't rely on standard Windows controls in almost all situations. This might seem like a subtle difference, but it really isn't, which you will definitely notice if you have ever worked with a framework that depends on Win32/WinAPI.

A great example of this is a button with an image and text on it. This is not a standard Windows control, so WinForms doesn't offer you this possibility out of the box. Instead you will have to draw the image yourself, implement your own button that supports images or use a 3rd party control. With WPF, a button can contain anything because it's essentially a border with content and various states (e.g. untouched, hovered, pressed). The WPF button is "look-less", as are most other WPF controls, which means that it can contain a range of other controls inside of it. You want a button with an image and some text? Just put an Image and a TextBlock control inside of the button and you're done! You simply don't get this kind of flexibility out of the standard WinForms controls, which is why there's a big market for rather simple implementations of controls like buttons with images and so on.

The drawback to this flexibility is that sometimes you will have to work harder to achieve something that was very easy with WinForms, because it was created for just the scenario you need it for. At least that's how it feels in the beginning, where you find yourself creating templates to make a ListView with an image and some nicely aligned text, something that the WinForms ListViewItem does in a single line of code.

This was just one difference, but as you work with WPF, you will realize that it is in fact the underlying reason for many of the other differences - WPF is simply just doing things in its own way, for better and for worse. You're no longer constrained to doing things the Windows way, but to get this kind of flexibility, you pay with a little more work when you're really just looking to do things the Windows way.

The following is a completely subjective list of the key advantages for WPF and WinForms. It should give you a better idea of what you're going into.

### 1.1.2.1. WPF advantages

- It's newer and thereby more in tune with current standards
- Microsoft is using it for a lot of new applications, e.g. Visual Studio
- It's more flexible, so you can do more things without having to write or buy new controls
- When you do need to use 3rd party controls, the developers of these controls will likely be more focused on WPF because it's newer
- XAML makes it easy to create and edit your GUI, and allows the work to be split between a designer (XAML) and a programmer (C#, VB.NET etc.)

- Databinding, which allows you to get a more clean separation of data and layout
- Uses hardware acceleration for drawing the GUI, for better performance
- It allows you to make user interfaces for both Windows applications and web applications (Silverlight/XBAP)

#### 1.1.2.2. WinForms advantages

- It's older and thereby more tried and tested
- There are already a lot of 3rd party controls that you can buy or get for free
- The designer in Visual Studio is still, as of writing, better for WinForms than for WPF, where you will have to do more of the work yourself with WPF

## 1.2. Getting started

---

### 1.2.1. Visual Studio Community

WPF is, as already described, a combination of XAML (markup) and C#/VB.NET/any other .NET language. All of it can be edited in any text editor, even Notepad included in Windows, and then compiled from the command line. However, most developers prefer to use an IDE (Integrated Development Environment), since it makes everything, from writing code to designing the interface and compiling it all so much easier.

The preferred choice for a .NET/WPF IDE is Visual Studio, which costs quite a bit of money though. Luckily, Microsoft has decided to make it easy and absolutely free for everyone to get started with .NET and WPF, so they have created a free version of Visual Studio, called Visual Studio Community. This version contains slightly less functionality than the real Visual Studio, but it has everything that you need to get started learning WPF and make real applications.

So go and download Visual Studio Community from Microsoft - it's free and easy to install and use:

<https://www.visualstudio.com/vs/community/>

Once downloaded and installed, click on to the next article to get started with the WPF learning process!

## 1.2.2. Hello, WPF!

The first and very classic example in pretty much any programming tutorial is the "Hello, world!" example, but in this tutorial we'll go nuts and change that into "Hello, WPF!" instead. The goal is simply to get this piece of text onto the screen, to show you how easy it is to get started.

The rest of this tutorial assumes that you have an IDE installed, preferably Visual Studio or Visual Studio Community (see the previous article for instructions on how to get it). If you're using another product, you will have to adapt the instructions to your product.

In Visual Studio, start by selecting **New project** from the **File** menu. On the left, you should have a tree of categories. This tutorial will focus on C# whenever code is involved, so you should select that from the list of templates, and since we'll be creating Windows applications, you should select **Windows** from the tree. This will give you a list of possible Windows application types to the right, where you should select a **WPF Application**. I named my project "HelloWPF" in the **Name** text field. Make sure that the rest of the settings in the bottom part of the dialog are okay and then press the **Ok** button.

Your new project will have a couple of files, but we will focus on just one of them now: *MainWindow.xaml*. This is the applications primary window, the one shown first when launching the application, unless you specifically change this. The XAML code found in it (XAML is discussed in details in another chapter of this tutorial) should look something like this:

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

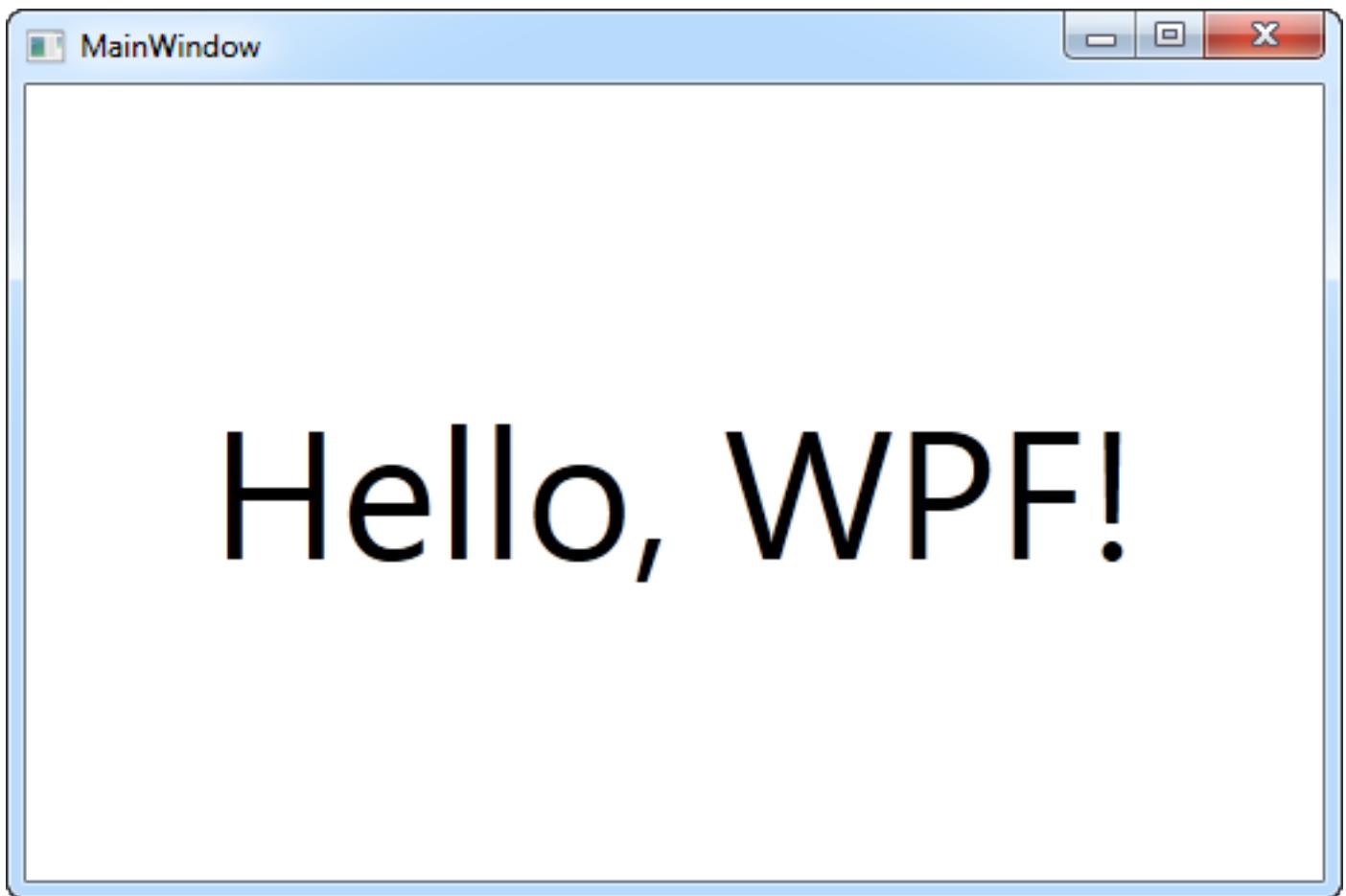
This is the base XAML that Visual Studio creates for our window, all parts of it explained in the chapters on XAML and "The Window". You can actually run the application now (select Debug -> Start debugging or press **F5**) to see the empty window that our application currently consists of, but now it's time to get our message on the screen.

We'll do it by adding a **TextBlock** control to the Grid panel, with our aforementioned message as the content:

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

```
Title="MainWindow" Height="350" Width="525">
<Grid>
    <TextBlock HorizontalAlignment="Center"
    VerticalAlignment="Center" FontSize="72">
        Hello, WPF!
    </TextBlock>
</Grid>
</Window>
```

Try running the application now (select Debug -> Start debugging or press **F5**) and see the beautiful result of your hard work - your first WPF application:



You will notice that we used three different attributes on the `TextBlock` to get a custom alignment (in the middle of the window), as well the `FontSize` property to get bigger text. All of these concepts will be treated in later articles.

Congratulations on making it this far. Now go read the rest of the tutorial and soon you will master WPF!

## 1.3. XAML

---

### 1.3.1. What is XAML?

XAML, which stands for eXtensible Application Markup Language, is Microsoft's variant of XML for describing a GUI. In previous GUI frameworks, like WinForms, a GUI was created in the same language that you would use for interacting with the GUI, e.g. C# or VB.NET and usually maintained by the designer (e.g. Visual Studio), but with XAML, Microsoft is going another way. Much like with HTML, you are able to easily write and edit your GUI.

This is not really a XAML tutorial, but I will briefly tell you about how you use it, because it's such an essential part of WPF. Whether you're creating a Window or a Page, it will consist of a XAML document and a CodeBehind file, which together creates the Window/Page. The XAML file describes the interface with all its elements, while the CodeBehind handles all the events and has access to manipulate with the XAML controls.

In the next chapters, we will have a look at how XAML works and how you use it to create your interface.

### 1.3.2. Basic XAML

In the previous chapter, we talked about what XAML is and what you use it for, but how do you create a control in XAML? As you will see from the next example, creating a control in XAML is as easy as writing its name, surrounded by angle brackets. For instance, a Button looks like this:

```
<Button>
```

XAML tags has to be ended, either by writing the end tag or by putting a forward slash at the end of the start tag:

```
<Button></Button>
```

Or

```
<Button />
```

A lot of controls allow you to put content between the start and end tags, which is then the content of the control. For instance, the Button control allows you to specify the text shown on it between the start and end tags:

```
<Button>A button</Button>
```

HTML is not case-sensitive, but XAML is, because the control name has to correspond to a type in the .NET framework. The same goes for attribute names, which corresponds to the properties of the control. Here's a button where we define a couple of properties by adding attributes to the tag:

```
<Button FontWeight="Bold" Content="A button" />
```

We set the FontWeight property, giving us bold text, and then we set the Content property, which is the same as writing the text between the start and end tag. However, all attributes of a control may also be defined like this, where they appear as child tags of the main control, using the Control-Dot-Property notation:

```
<Button>
    <Button.FontWeight>Bold</Button.FontWeight>
    <Button.Content>A button</Button.Content>
</Button>
```

The result is exactly the same as above, so in this case, it's all about syntax and nothing else. However, a lot of controls allow content other than text, for instance other controls. Here's an example where we have text in different colors on the same button by using several TextBlock controls inside of the Button:

```
<Button>
```

```

<Button.FontWeight>Bold</Button.FontWeight>
<Button.Content>
    <WrapPanel>
        <TextBlock Foreground="Blue">Multi</TextBlock>
        <TextBlock Foreground="Red">Color</TextBlock>
        <TextBlock>Button</TextBlock>
    </WrapPanel>
</Button.Content>
</Button>

```

The Content property only allows for a single child element, so we use a WrapPanel to contain the differently colored blocks of text. Panels, like the WrapPanel, plays an important role in WPF and we will discuss them in much more details later on - for now, just consider them as containers for other controls.

The exact same result can be accomplished with the following markup, which is simply another way of writing the same:

```

<Button FontWeight="Bold">
    <WrapPanel>
        <TextBlock Foreground="Blue">Multi</TextBlock>
        <TextBlock Foreground="Red">Color</TextBlock>
        <TextBlock>Button</TextBlock>
    </WrapPanel>
</Button>

```

### 1.3.2.1. Code vs. XAML

Hopefully the above examples show you that XAML is pretty easy to write, but with a lot of different ways of doing it, and if you think that the above example is a lot of markup to get a button with text in different colors, then try comparing it to doing the exact same thing in C#:

```

Button btn = new Button();
btn.FontWeight = FontWeights.Bold;

WrapPanel pnl = new WrapPanel();

TextBlock txt = new TextBlock();
txt.Text = "Multi";
txt.Foreground = Brushes.Blue;
pnl.Children.Add(txt);

txt = new TextBlock();

```

```
txt.Text = "Color";
txt.Foreground = Brushes.Red;
pnl.Children.Add(txt);

txt = new TextBlock();
txt.Text = "Button";
pnl.Children.Add(txt);

btn.Content = pnl;
pnlMain.Children.Add(btn);
```

Of course the above example could be written less explicitly and using more syntactical sugar, but I think the point still stands: XAML is pretty short and concise for describing interfaces.

### 1.3.3. Events in XAML

Most modern UI frameworks are event driven and so is WPF. All of the controls, including the Window (which also inherits the Control class) exposes a range of events that you may subscribe to. You can subscribe to these events, which means that your application will be notified when they occur and you may react to that.

There are many types of events, but some of the most commonly used are there to respond to the user's interaction with your application using the mouse or the keyboard. On most controls you will find events like KeyDown, KeyUp, MouseDown, MouseEnter, MouseLeave, MouseUp and several others.

We will look more closely at how events work in WPF, since this is a complex topic, but for now, you need to know how to link a control event in XAML to a piece of code in your Code-behind file. Have a look at this example:

```
<Window x:Class="WpfTutorialSamples.XAML.EventsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="EventsSample" Height="300" Width="300">
    <Grid Name="pnlMainGrid" MouseUp="pnlMainGrid_MouseUp" Background
="LightBlue">

    </Grid>
</Window>
```

Notice how we have subscribed to the MouseUp event of the Grid by writing a method name. This method needs to be defined in code-behind, using the correct event signature. In this case it should look like this:

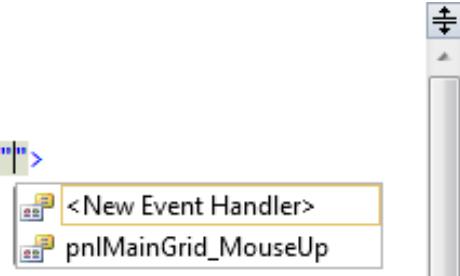
```
private void pnlMainGrid_MouseUp(object sender, MouseButtonEventArgs e)
{
    MessageBox.Show("You clicked me at " + e.GetPosition(this
).ToString());
}
```

The MouseUp event uses a delegate called **MouseButtonEventHandler**, which you subscribe to. It has two parameters, a sender (the control which raised the event) and a MouseButtonEventArgs object that will contain useful information. We use it in the example to get the position of the mouse cursor and tell the user about it.

Several events may use the same delegate type - for instance, both MouseUp and MouseDown uses the **MouseButtonEventHandler** delegate, while theMouseMove event uses the **MouseEventHandler** delegate. When defining the event handler method, you need to know which delegate it uses and if you don't know that, you can look it up in the documentation.

Fortunately, Visual Studio can help us to generate a correct event handler for an event. The easiest way to do this is to simply write the name of the event in XAML and then let the IntelliSense of VS do the rest for you:

```
    >useUp="pnlMainGrid_MouseUp" Background="LightBlue" MouseDown="">
```



When you select &lt;New Event Handler&gt;, Visual Studio will generate an appropriate event handler in your Code-behind file. It will be named <control name>\_<event name>, in our case **pnlMainGrid\_MouseDown**. Right-click in the event name and select **Navigate to Event Handler** and VS will take you right to it.

#### 1.3.3.1. Subscribing to an event from Code-behind

The most common way to subscribe to events is explained above, but there may be times where you want to subscribe to the event directly from Code-behind instead. This is done using the += C# syntax, where you add an event handler to event directly on the object. The full explanation of this belongs in a dedicated C# example, but for comparison, here's an example:

```
using System;
using System.Windows;
using System.Windows.Input;

namespace WpfTutorialSamples.XAML
{
    public partial class EventsSample : Window
    {
        public EventsSample()
        {
            InitializeComponent();
            pnlMainGrid.MouseUp += new
MouseButtonEventHandler(pnlMainGrid_MouseUp);
        }

        private void pnlMainGrid_MouseUp(object sender,
MouseButtonEventArgs e)
        {
```

```

        MessageBox.Show( "You clicked me at " + e.GetPosition(this)
).ToString() );
}

}

```

Once again, you need to know which delegate to use, and once again, Visual Studio can help you with this. As soon as you write:

```
pnlMainGrid.MouseDown +=
```

Visual Studio will offer its assistance:

```

public EventsSample()
{
    InitializeComponent();
    pnlMainGrid.MouseUp += new MouseButtonEventHandler(pnlMainGrid_MouseUp);
    pnlMainGrid.MouseDown += |
}~
```

**new MouseButtonEventHandler(pnlMainGrid\_MouseDown); (Press TAB to insert)**

Simply press the [Tab] key twice to have Visual Studio generate the correct event handler for you, right below the current method, ready for implementation. When you subscribe to the events like this, you don't need to do it in XAML.

## 1.4. A WPF application

---

### 1.4.1. A WPF Application - Introduction

In this tutorial, our primary focus will be on using WPF to create applications. As you may know, .NET can be executed on all platforms which have a .NET implementation, but the most common platform is still Microsoft Windows. When we talk about Windows applications in this tutorial, it really just means an application that runs on Windows (or another .NET compatible platform) and not in a browser or remotely over the Internet.

A WPF application requires the .NET framework to run, just like any other .NET application type. Fortunately, Microsoft has been including the .NET framework on all versions of Windows since Vista, and they have been pushing out the framework on older versions through Windows Update. In other words, you can be pretty sure that most Windows users out there will be able to run your WPF application.

In the following chapters we will have a look at the structure and various aspects of a WPF application.

## 1.4.2. The Window

When creating a WPF application, the first thing you will meet is the `Window` class. It serves as the root of a window and provides you with the standard border, title bar and maximize, minimize and close buttons. A WPF window is a combination of a XAML (.xaml) file, where the `<Window>` element is the root, and a CodeBehind (.cs) file. If you're using Visual Studio (Express) and you create a new WPF application, it will create a default window for you, which will look something like this:

```
<Window x:Class="WpfApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>

    </Grid>
</Window>
```

The `x:Class` attribute tells the XAML file which class to use, in this case `Window1`, which Visual Studio has created for us as well. You will find it in the project tree in VS, as a child node of the XAML file. By default, it looks something like this:

```
using System;
using System.Windows;
using System.Windows.Controls;
//...more using statements

namespace WpfApplication1
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

As you can see, the `Window1` class is defined as partial, because it's being combined with your XAML file in runtime to give you the full window. This is actually what the call to `InitializeComponent()` does, which is why it's required to get a full functioning window up and running.

If we go back to the XAML file, you will notice a couple of other interesting attributes on the Window element, like Title, which defines the title of the window (shown in the title bar) as well as the starting width and height. There are also a couple of namespace definitions, which we will talk about in the XAML chapters.

You will also notice that Visual Studio has created a Grid control for us inside the Window. The Grid is one of the WPF panels, and while it could be any panel or control, the Window can only have ONE child control, so a Panel, which in turn can contain multiple child controls, is usually a good choice. Later in this tutorial, we will have a much closer look into the different types of panels that you can use, as they are very important in WPF.

#### 1.4.2.1. Important Window properties

The WPF Window class has a bunch of interesting attributes that you may set to control the look and behavior of your application window. Here's a short list of the most interesting ones:

**Icon** - Allows you to define the icon of the window, which is usually shown in the upper left corner, to the left of the window title.

**ResizeMode** - This controls whether and how the end-user can resize your window. The default is CanResize, which allows the user to resize the window like any other window, either by using the maximize/minimize buttons or by dragging one of the edges. CanMinimize will allow the user to minimize the window, but not to maximize it or drag it bigger or smaller. NoResize is the strictest one, where the maximize and minimize buttons are removed and the window can't be dragged bigger or smaller.

**ShowInTaskbar** - The default is true, but if you set it to false, your window won't be represented in the Windows taskbar. Useful for non-primary windows or for applications that should minimize to the tray.

**SizeToContent** - Decide if the Window should resize itself to automatically fit its content. The default is Manual, which means that the window doesn't automatically resize. Other options are Width, Height and WidthAndHeight, and each of them will automatically adjust the window size horizontally, vertically or both.

**Topmost** - The default is false, but if set to true, your Window will stay on top of other windows unless minimized. Only useful for special situations.

**WindowStartupLocation** - Controls the initial position of your window. The default is Manual, which means that the window will be initially positioned according to the Top and Left properties of your window. Other options are CenterOwner, which will position the window in the center of its owner window, and CenterScreen, which will position the window in the center of the screen.

**WindowState** - Controls the initial window state. It can be either Normal, Maximized or Minimized. The default is Normal, which is what you should use unless you want your window to start either maximized or minimized.

There are lots of other attributes though, so have a look for yourself and then move on to the next chapter.

## 1.4.3. Working with App.xaml

App.xaml is the declarative starting point of your application. Visual Studio will automatically create it for you when you start a new WPF application, including a Code-behind file called App.xaml.cs. They work much like for a Window, where the two files are partial classes, working together to allow you to work in both markup (XAML) and Code-behind.

App.xaml.cs extends the Application class, which is a central class in a WPF Windows application. .NET will go to this class for starting instructions and then start the desired Window or Page from there. This is also the place to subscribe to important application events, like application start, unhandled exceptions and so on. More about that later.

One of the most commonly used features of the App.xaml file is to define global resources that may be used and accessed from all over an application, for instance global styles. This will be discussed in detail later on.

### 1.4.3.1. App.xaml structure

When creating a new application, the automatically generated App.xaml will look something like this:

```
<Application x:Class="WpfTutorialSamples.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">

    <Application.Resources>

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

The main thing to notice here is the StartupUri property. This is actually the part that instructs which Window or Page to start up when the application is launched. In this case, MainWindow.xaml will be started, but if you would like to use another window as the starting point, you can simply change this.

In some situations, you want more control over how and when the first window is displayed. In that case, you can remove the StartupUri property and value and then do it all from Code-Behind instead. This will be demonstrated below.

### 1.4.3.2. App.xaml.cs structure

The matching App.xaml.cs will usually look like this for a new project:

```
using System;
using System.Collections.Generic;
using System.Windows;
```

```

namespace WpfTutorialSamples
{
    public partial class App : Application
    {

    }
}

```

You will see how this class extends the Application class, allowing us to do stuff on the application level. For instance, you can subscribe to the Startup event, where you can manually create your starting window.

Here's an example:

```

<Application x:Class="WpfTutorialSamples.App"
             xmlns
             ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Startup="Application_Startup">
    <Application.Resources></Application.Resources>
</Application>

```

Notice how the StartupUri has been replaced with a subscription to the Startup event (subscribing to events through XAML is explained in another chapter). In Code-Behind, you can use the event like this:

```

using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples
{
    public partial class App : Application
    {

        private void Application_Startup(object sender,
StartupEventArgs e)
        {
            // Create the startup window
            MainWindow wnd = new MainWindow();
            // Do stuff here, e.g. to the window
            wnd.Title = "Something else";
        }
}

```

```
// Show the window
wnd.Show();
}

}
```

The cool thing in this example, compared to just using the StartupUri property, is that we get to manipulate the startup window before showing it. In this, we change the title of it, which is not terribly useful, but you could also subscribe to events or perhaps show a splash screen. When you have all the control, there are many possibilities. We will look deeper into several of them in the next articles of this tutorial.

#### 1.4.4. Command-line parameters in WPF

Command-line parameters are a technique where you can pass a set of parameters to an application that you wish to start, to somehow influence it. The most common example is to make the application open with a specific file, e.g. in an editor. You can try this yourself with the built-in Notepad application of Windows, by running (select Run from the Start menu or press [WindowsKey-R]):

```
notepad.exe c:\Windows\win.ini
```

This will open Notepad with the win.ini file opened (you may have to adjust the path to match your system). Notepad simply looks for one or several parameters and then uses them and your application can do the same!

Command-line parameters are passed to your WPF application through the Startup event, which we subscribed to in the App.xaml article. We will do the same in this example, and then use the value passed on to through the method arguments. First, the App.xaml file:

```
<Application x:Class="WpfTutorialSamples.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Startup="Application_Startup">
    <Application.Resources></Application.Resources>
</Application>
```

All we do here is to subscribe to the **Startup** event, replacing the **StartupUri** property. The event is then implemented in App.xaml.cs:

```
using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples
{
    public partial class App : Application
    {

        private void Application_Startup(object sender,
StartupEventArgs e)
        {
            MainWindow wnd = new MainWindow();
            if(e.Args.Length == 1)
                MessageBox.Show("Now opening file: \n\n" +
e.Args[0]);
        }
}
```

```

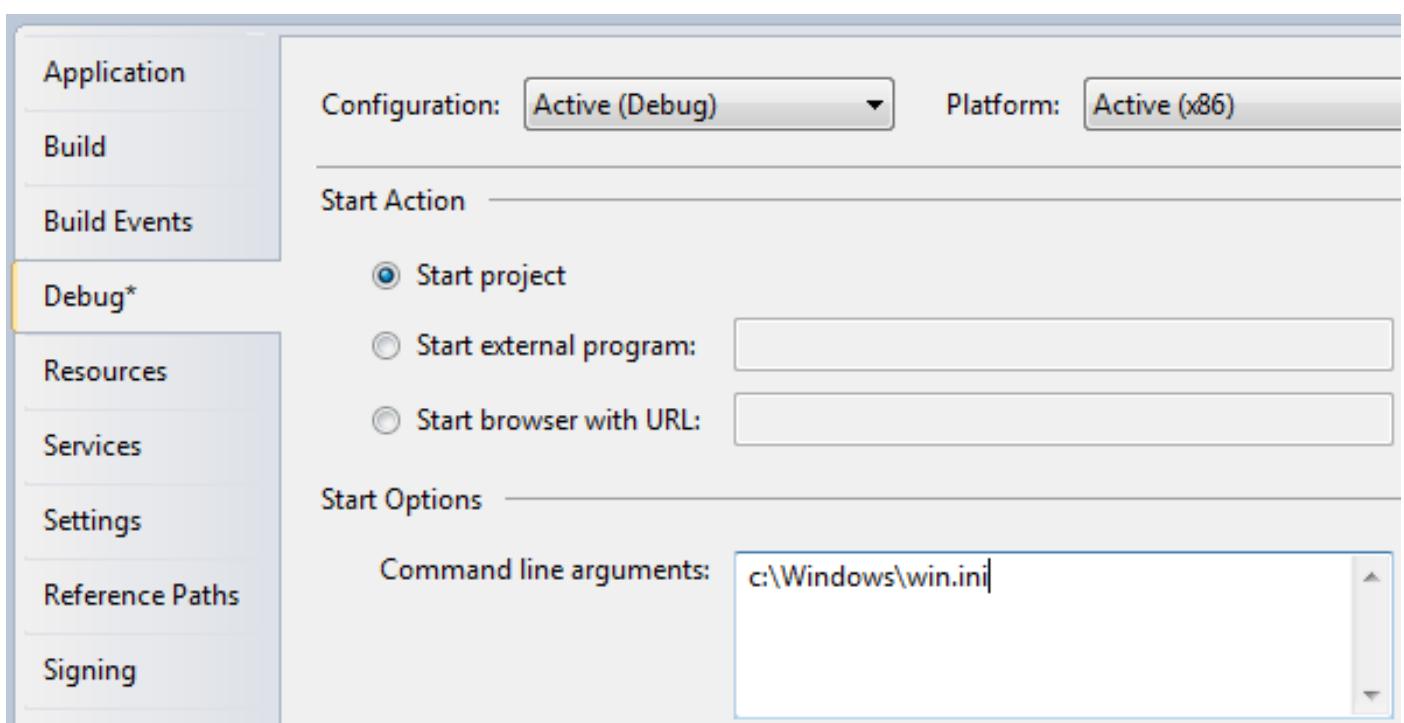
e.Args[0]);
    wnd.Show();
}
}
}

```

The **StartupEventArgs** is what we use here. It's passed into the Application Startup event, with the name `e`. It has the property **Args**, which is an array of strings. Command-line parameters are separated by spaces, unless the space is inside a quoted string.

#### 1.4.4.1. Testing the command-line parameter

If you run the above example, nothing will happen, because no command-line parameters have been specified. Fortunately, Visual Studio makes it easy to test this in your application. From the **Project** menu select "**[Project name] properties**" and then go to the **Debug** tab, where you can define a command-line parameter. It should look something like this:



Try running the application and you will see it respond to your parameter.

Of course, the message isn't terribly useful. Instead you might want to either pass it to the constructor of your main window or call a public open method on it, like this:

```

using System;
using System.Collections.Generic;
using System.Windows;

```

```

namespace WpfTutorialSamples
{
    public partial class App : Application
    {

        private void Application_Startup(object sender,
StartupEventArgs e)
        {
            MainWindow wnd = new MainWindow();
            // The OpenFile() method is just an example of what you
could do with the
            // parameter. The method should be declared on your
MainWindow class, where
            // you could use a range of methods to process the passed
file path
            if(e.Args.Length == 1)
                wnd.OpenFile(e.Args[0]);
            wnd.Show();
        }
    }
}

```

#### 1.4.4.2. Command-line possibilities

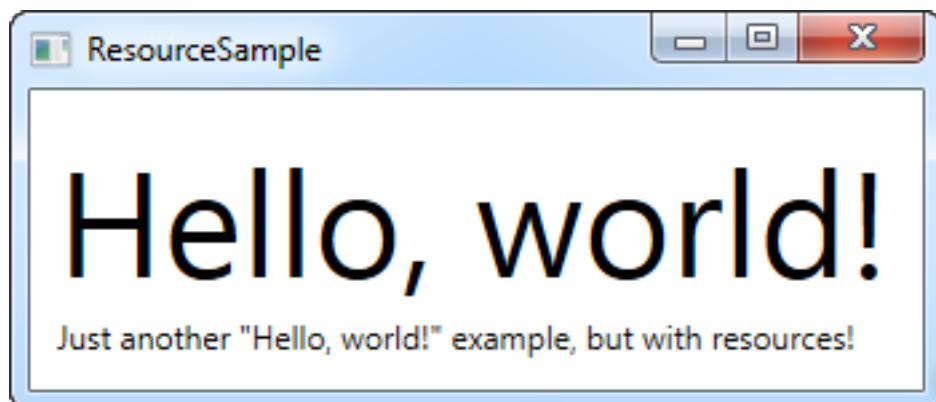
In this example, we test if there is exactly one argument and if so, we use it as a filename. In a real world example, you might collect several arguments and even use them for options, e.g. toggling a certain feature on or off. You would do that by looping through the entire list of arguments passed while collecting the information you need to proceed, but that's beyond the scope of this article.

## 1.4.5. 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 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.

#### 1.4.5.1. 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.

#### 1.4.5.2. 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>

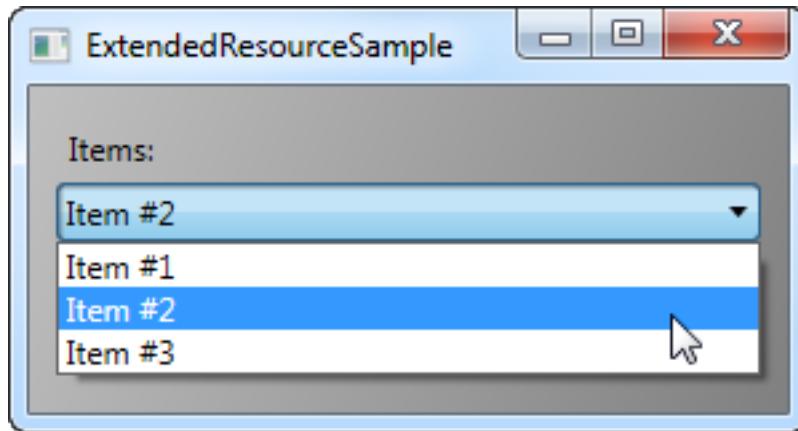
        <x:Array x:Key="ComboBoxItems" Type="sys:String">
            <sys:String>Item #1</sys:String>
            <sys:String>Item #2</sys:String>
            <sys:String>Item #3</sys:String>
        </x:Array>

        <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.

#### 1.4.5.3. 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}" />
```

#### 1.4.5.4. 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;

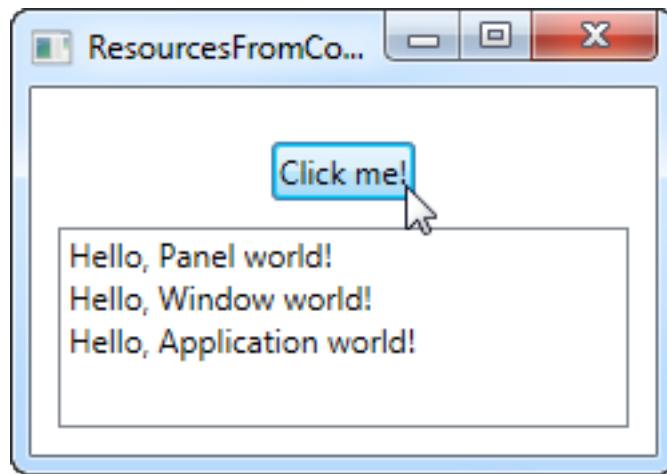
namespace WpfTutorialSamples.WPF_Application
{
    public partial class ResourcesFromCodeBehindSample : Window
    {
        public ResourcesFromCodeBehindSample()
        {
```

```

        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 principle, 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.

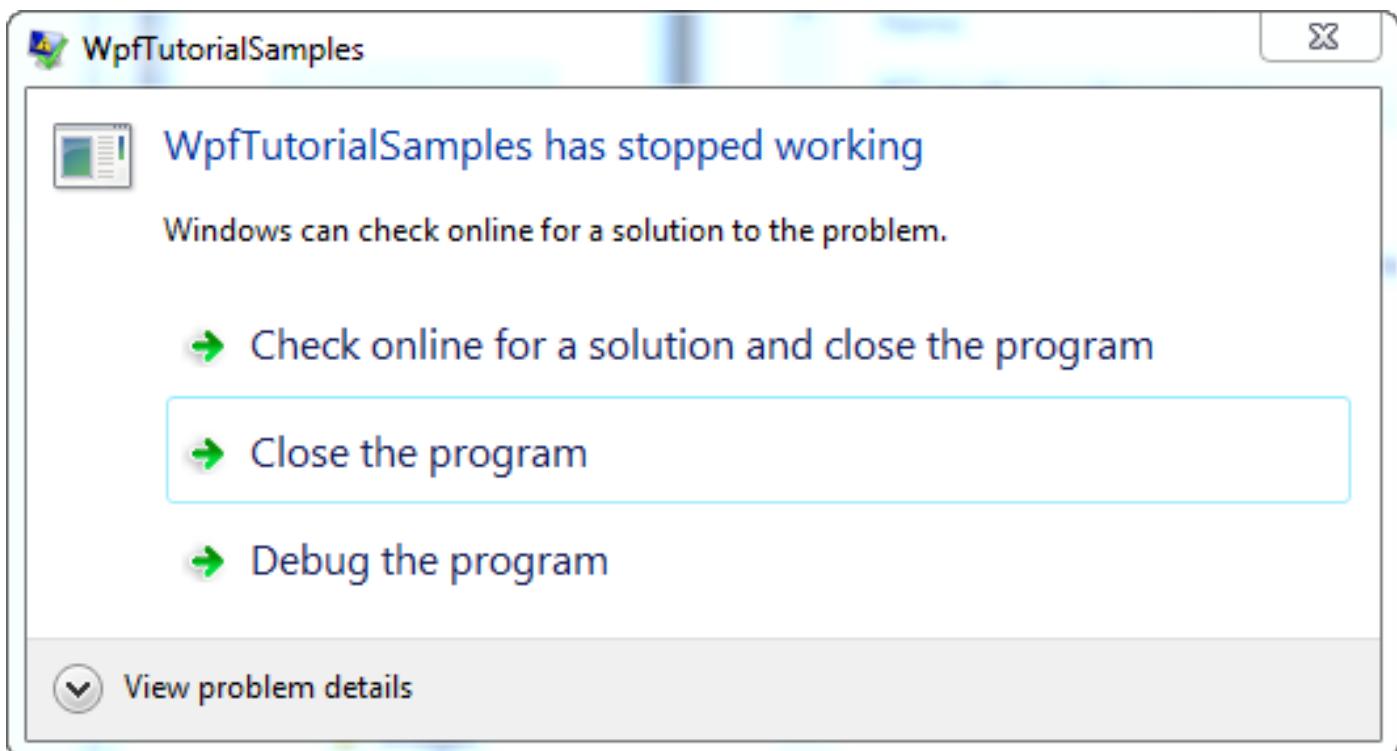
The same is not true the other way around - the search doesn't navigate down the tree, so you can't start looking for a resource on the application level, if it has been defined locally for the control or for the window.

#### 1.4.6. Handling exceptions in WPF

If you're familiar with C# or any of the other .NET languages that you may use with WPF, then exception handling should not be new to you: Whenever you have a piece of code that are likely to throw an exception, then you should wrap it in a try-catch block to handle the exception gracefully. For instance, consider this example:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string s = null;
    s.Trim();
}
```

Obviously it will go wrong, since I try to perform the Trim() method on a variable that's currently null. If you don't handle the exception, your application will crash and Windows will have to deal with the problem. As you can see, that isn't very user friendly:



In this case, the user would be forced to close your application, due to such a simple and easily avoided error. So, if you know that things might go wrong, then you should use a try-catch block, like this:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string s = null;
    try
    {
```

```

        s.Trim();
    }
    catch(Exception ex)
    {
        MessageBox.Show("A handled exception just occurred: " +
ex.Message, "Exception Sample", MessageBoxButton.OK,
MessageBoxImage.Warning);
    }
}

```

However, sometimes even the simplest code can throw an exception, and instead of wrapping every single line of code with a try- catch block, WPF lets you handle all unhandled exceptions globally. This is done through the **DispatcherUnhandledException** event on the Application class. If subscribed to, WPF will call the subscribing method once an exception is thrown which is not handled in your own code. Here's a complete example, based on the stuff we just went through:

```

<Window x:Class
        ="WpfTutorialSamples.WPF_Application.ExceptionHandlingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ExceptionHandlingSample" Height="200" Width="200">
    <Grid>
        <Button HorizontalAlignment="Center" VerticalAlignment
        ="Center" Click="Button_Click">
            Do something bad!
        </Button>
    </Grid>
</Window>

using System;
using System.Windows;

namespace WpfTutorialSamples.WPF_Application
{
    public partial class ExceptionHandlingSample : Window
    {
        public ExceptionHandlingSample()
        {
            InitializeComponent();
        }
}

```

```

        private void Button_Click(object sender, RoutedEventArgs e)
    {
        string s = null;
        try
        {
            s.Trim();
        }
        catch(Exception ex)
        {
            MessageBox.Show("A handled exception just occurred:
" + ex.Message, "Exception Sample", MessageBoxButton.OK,
MessageBoxImage.Warning);
        }
        s.Trim();
    }
}

```

Notice that I call the Trim() method an extra time, outside of the try-catch block, so that the first call is handled, while the second is not. For the second one, we need the App.xaml magic:

```

<Application x:Class="WpfTutorialSamples.App"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    DispatcherUnhandledException
    ="Application_DispatcherUnhandledException"
    StartupUri="WPF
Application/ExceptionHandlingSample.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>

using System;
using System.Windows;

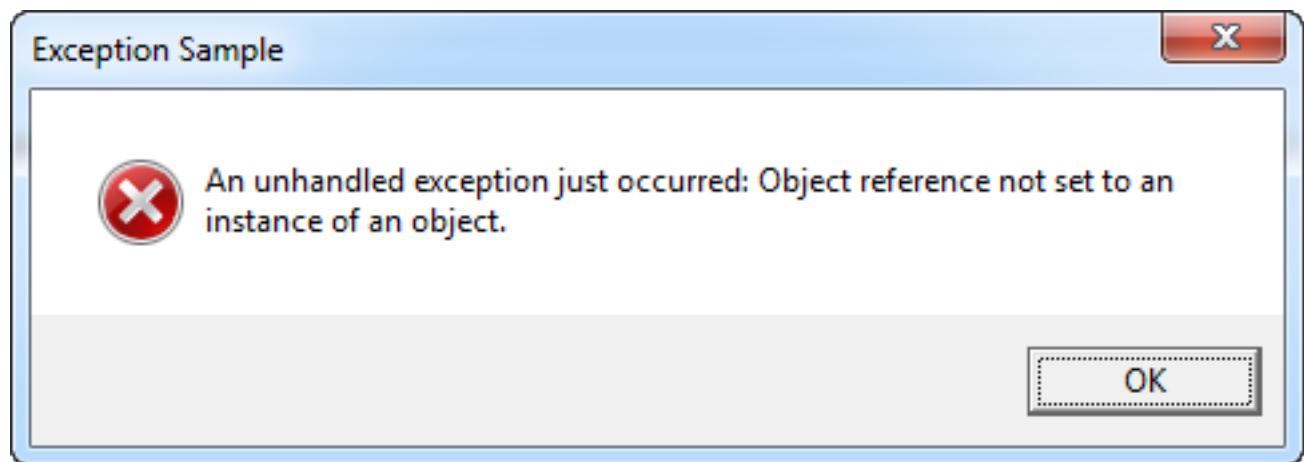
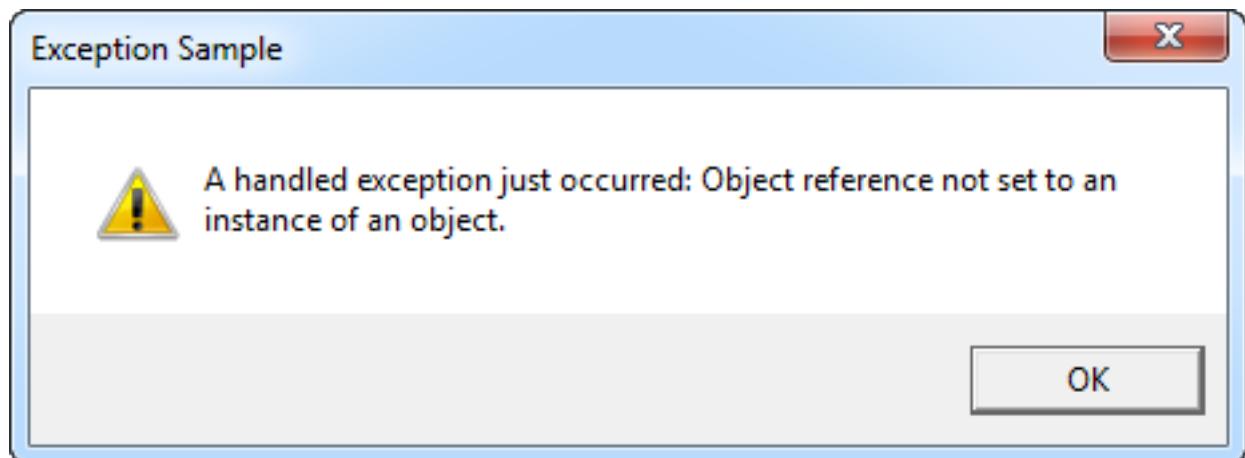
namespace WpfTutorialSamples
{
    public partial class App : Application
    {

```

```

    private void Application_DispatcherUnhandledException(object
sender, System.Windows.Threading.DispatcherUnhandledExceptionEventArgs
e)
{
    MessageBox.Show("An unhandled exception just occurred: "
+ e.Exception.Message, "Exception Sample", MessageBoxButton.OK,
MessageBoxImage.Warning);
    e.Handled = true;
}
}

```



We handle the exception much like the local one, but with a slightly different text and image in the message box. Also, notice that I set the `e.Handled` property to true. This tells WPF that we're done dealing with this exception and nothing else should be done about it.

#### 1.4.6.1. Summary

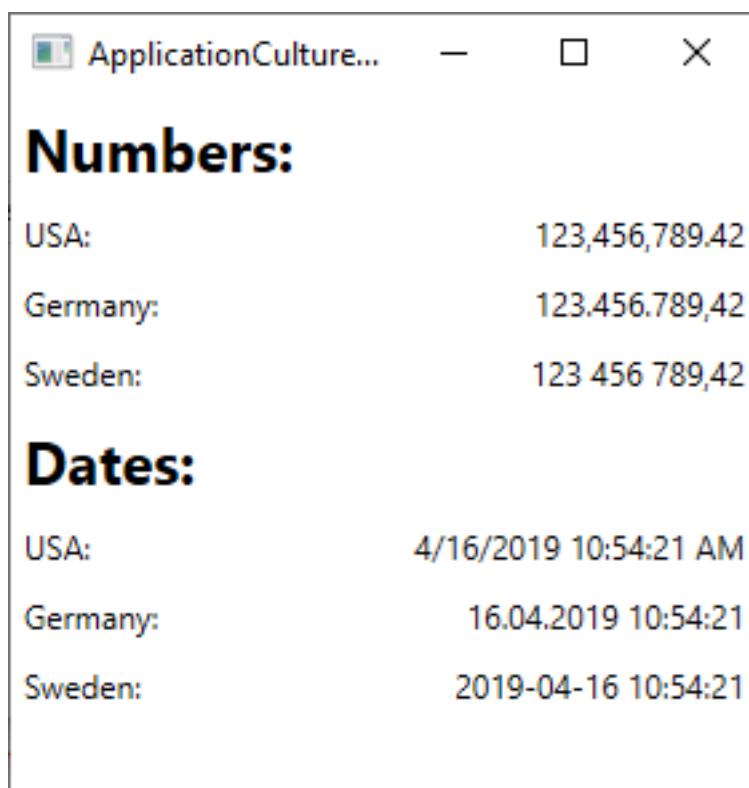
Exception handling is a very important part of any application and fortunately, WPF and .NET makes it very easy to handle exceptions both locally and globally. You should handle exceptions locally when it makes

sense and only use the global handling as a fallback mechanism, since local handling allows you to be more specific and deal with the problem in a more specialized way.

## 1.4.7. Application Culture / UICulture

If you have worked a little bit with numbers or dates in your WPF application, e.g. because of one of the articles found in this tutorial, you may have noticed something cool: Numbers and dates are automatically displayed in a format that matches the format used on your computer. If you live in an English-speaking country, this is probably doesn't seem like a big thing, but if you live in one of the MANY countries where dates and/or numbers are displayed differently, this is really cool.

And if you're thinking "surely there can't be that many differences when formatting simple stuff like numbers and dates?", I suggest that you have a look at this sample app, where I have formatted the same number and the same date in accordance with how they prefer it in the US, Germany and Sweden:



So as you can see, there are many subtle differences in how numbers and dates are displayed. The good news is that the .NET framework can help you out a lot - in fact, it already does: By default, dates and numbers are formatted in accordance with the system settings of the computer where your application are executed. The bad news is that this behavior might not always be what you want. But don't worry - you can easily change this. It all relates to the use of the **CultureInfo** class, which you can read much more about in the [C# Tutorial article on CultureInfo](#). For now, let's discuss how to apply these techniques to your WPF application.

### 1.4.7.1. Ad-hoc formatting

If you only need to apply formatting for a specific piece of information, e.g. the contents of a single Label control, you can easily do this, on-the-fly, using a combination of the **ToString()** method and the **CultureInfo** class. For instance, in the example above, I applied different, culture-based formatting like this:

```

double largeNumber = 123456789.42;

CultureInfo usCulture = new CultureInfo("en-US");
CultureInfo deCulture = new CultureInfo("de-DE");
CultureInfo seCulture = new CultureInfo("sv-SE");

lblNumberUs.Content = largeNumber.ToString("N2", usCulture);
lblNumberDe.Content = largeNumber.ToString("N2", deCulture);
lblNumberSe.Content = largeNumber.ToString("N2", seCulture);

```

This might suffice for some cases, where you just need special formatting in a couple of places, but in general, you should decide if your application should use system settings (the default) or if you should override this behavior with a specific culture-setting for the entire application.

#### 1.4.7.2. CurrentCulture & CurrentUICulture

Applying another culture to your WPF application is quite easy. You will, potentially, be dealing with two attributes, found on the **CurrentThread** property of the **Thread** class: **CurrentCulture** and **CurrentUICulture**. But what's the difference?

The **CurrentCulture** property is the one that controls how numbers and dates etc. are formatted. The default value comes from the operating system of the computer executing the application and can be changed independently of the language used by their operating system. It is, for instance, very common for a person living in Germany to install Windows with English as their interface language, while still preferring German-notation for numbers and dates. For a situation like this, the CurrentCulture property would default to German.

The **CurrentUICulture** property specifies the language that the interface should use. This is only relevant if your application supports multiple languages, e.g. through the use of language-resource files. Once again, this allows you to use one culture for the language (e.g. English), while using another (e.g. German) when dealing with input/output of numbers, dates etc.

#### 1.4.7.3. Changing the application Culture

With that in mind, you now have to decide whether to change the CurrentCulture and/or the CurrentUICulture. It can be done pretty much whenever you want, but it makes the most sense to do it when starting your application - otherwise, some output might already be generated with the default culture, before the switch. Here's an example where we change the Culture, as well as the UICulture, in the Application\_Startup() event which can be used in the App.xaml.cs file of your WPF application:

```

private void Application_Startup(object sender, StartupEventArgs e)
{
    Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
    Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-US");
}

```

```
}
```

Since we use the `Thread` class as well as the `CultureInfo`, don't forget to add the required namespaces to your file, if they are not already present:

```
using System.Threading;
using System.Globalization;
```

With this in place, numbers and dates will now be formatted according to how they prefer it in German (**de-DE**). As mentioned, you can leave out the line defining the culture for the `UICulture` (the last line) if your application doesn't support multiple languages.

Changing the culture during the **Application\_Startup** event, or at the latest in the constructor of your main window, makes most sense, because values that are already generated aren't updated automatically when you change the **CurrentCulture** property. That doesn't mean that you can't do it though, as illustrated by this next example, which also serves as a fine demonstration of how the output is affected by the **CurrentCulture** property:

```
<Window x:Class
        ="WpfTutorialSamples.WPF_Application.ApplicationCultureSwitchSample"
        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.WPF_Application"
        mc:Ignorable="d"
        Title="ApplicationCultureSwitchSample" Height="200" Width
        ="320">
    <StackPanel Margin="20">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Label>Number:</Label>
            <Label Name="lblNumber" Grid.Column="1" />
```

```

        <Label Grid.Row="1">Date:</Label>
        <Label Name="lblDate" Grid.Row="1" Grid.Column="1" />
    </Grid>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center" Margin="0,20">
        <Button Tag="en-US" Click="CultureInfoSwitchButton_Click" HorizontalContentAlignment="Stretch">English (US)</Button>
        <Button Tag="de-DE" Click="CultureInfoSwitchButton_Click" HorizontalContentAlignment="Stretch" Margin="10,0">German (DE)</Button>
        <Button Tag="sv-SE" Click="CultureInfoSwitchButton_Click" HorizontalContentAlignment="Stretch">Swedish (SE)</Button>
    </StackPanel>
</StackPanel>
</Window>

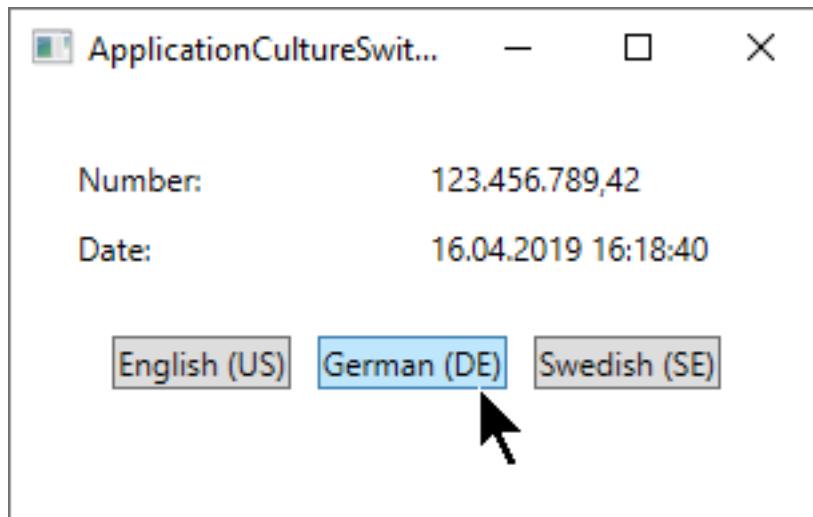
using System;
using System.Globalization;
using System.Threading;
using System.Windows;
using System.Windows.Controls;

namespace WpfTutorialSamples.WPF_Application
{
    public partial class ApplicationCultureSwitchSample : Window
    {
        public ApplicationCultureSwitchSample()
        {
            InitializeComponent();
        }

        private void CultureInfoSwitchButton_Click(object sender,
RoutedEventArgs e)
        {
            Thread.CurrentThread.CurrentCulture = new CultureInfo((sender as Button).Tag.ToString());
            lblNumber.Content = (123456789.42d).ToString("N2");
            lblDate.Content = DateTime.Now.ToString();
        }
    }
}

```

The interesting part is found in the CultureInfoSwitchButton\_Click event, where we set CurrentCulture based on which of the buttons were clicked, and then update the two labels containing a number and a date:



#### 1.4.7.4. Culture & Threads: The DefaultThreadCurrentCulture property

If your application uses more than one thread, you should consider using the **DefaultThreadCurrentCulture** property. It can be found on the CultureInfo class (introduced in .NET framework version 4.5) and will ensure that not only the current thread, but also future threads will use the same culture. You can use it like this, e.g. in the **Application\_Startup** event:

```
CultureInfo.DefaultThreadCurrentCulture = new CultureInfo("de-DE");
```

So, will you have to set both the **CurrentCulture** AND the **DefaultThreadCurrentCulture** properties? Actually, no - if you have not already changed the CurrentCulture property, setting the DefaultThreadCurrentCulture property will also be applied to the CurrentCulture property. In other words, it makes sense to use the DefaultThreadCurrentCulture instead of CurrentCulture if you plan on using multiple threads in your application - it will take care of all scenarios.

#### 1.4.7.5. Summary

Dealing with the culture of your WPF application is very important, but fortunately for you, WPF will do a lot of it for you completely out-of-the-box. If you need to change the default behavior, it's quite easy as well, using the **CurrentCulture** and **CurrentUICulture** properties, as illustrated in the numerous examples of this article.

## 1.5. Basic controls

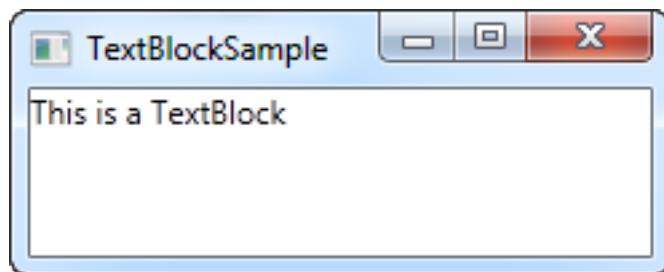
### 1.5.1. The TextBlock control

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

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

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBlockSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBlockSample" Height="100" Width="200">
    <Grid>
        <TextBlock>This is a TextBlock</TextBlock>
    </Grid>
</Window>
```



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

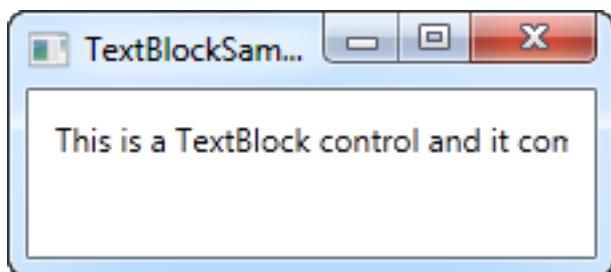
For the next example, let's try a longer text to show how the TextBlock deals with that. I've also added a bit of margin, to make it look just a bit better:

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBlockSample"
```

```

    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBlockSample" Height="100" Width="200">
    <Grid>
        <TextBlock Margin="10">This is a TextBlock control and it
comes with a very long text</TextBlock>
    </Grid>
</Window>

```



### 1.5.1.1. Dealing with long strings

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

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

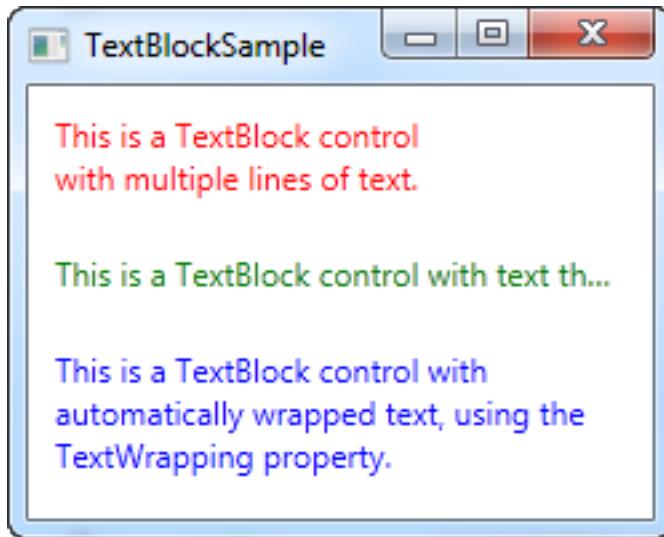
```

<Window x:Class="WpfTutorialSamples.Basic_controls.TextBlockSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBlockSample" Height="200" Width="250">
    <StackPanel>
        <TextBlock Margin="10" Foreground="Red">
            This is a TextBlock control<LineBreak />
            with multiple lines of text.
        </TextBlock>
        <TextBlock Margin="10" TextTrimming="CharacterEllipsis"
Foreground="Green">
            This is a TextBlock control with text that may not be
rendered completely, which will be indicated with an ellipsis.
        </TextBlock>
        <TextBlock Margin="10" TextWrapping="Wrap" Foreground="Blue">

```

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

```
</TextBlock>  
</StackPanel>  
</Window>
```



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

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

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

The blue TextBlock uses the **TextWrapping** property with the value **Wrap**, to make the TextBlock wrap to the next line whenever it can't fit anymore text into the previous line. Contrary to the first TextBlock, where we manually define where to wrap the text, this happens completely automatic and even better: It's also automatically adjusted as soon as the TextBlock gets 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.

## 1.5.2. The TextBlock control - Inline formatting

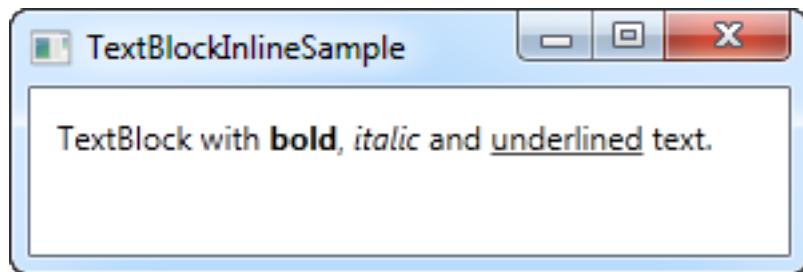
In the last article we looked at the core functionality of the TextBlock control: Displaying a simple string and wrapping it if necessary. We even used another color than the default for rendering the text, but what if you wanted to do more than just define a static color for all the text in the TextBlock?

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.

### 1.5.2.1. Bold, Italic and Underline

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBlockInlineSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBlockInlineSample" Height="100" Width="300">
    <Grid>
        <TextBlock Margin="10" TextWrapping="Wrap">
            TextBlock with <Bold>bold</Bold>, <Italic>italic</Italic>
            and <Underline>underlined</Underline> text.
        </TextBlock>
    </Grid>
</Window>
```



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

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

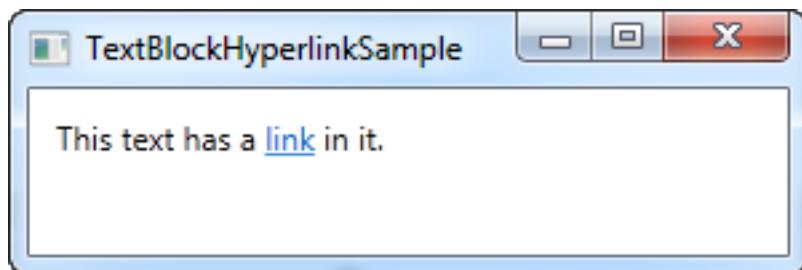
### 1.5.2.2. LineBreak

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

### 1.5.2.3. Hyperlink

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBlockHyperlinkSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBlockHyperlinkSample" Height="100" Width="300">
    <Grid>
        <TextBlock Margin="10" TextWrapping="Wrap">
            This text has a <Hyperlink RequestNavigate="Hyperlink_RequestNavigate" NavigateUri="https://www.google.com">link</Hyperlink> in it.
        </TextBlock>
    </Grid>
</Window>
```



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

```
private void Hyperlink_RequestNavigate(object sender,
System.Windows.Navigation.RequestEventArgs e)
{
```

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

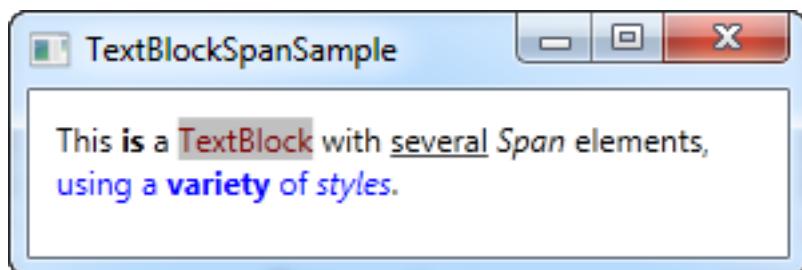
#### 1.5.2.4. Run

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.

#### 1.5.2.5. Span

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBlockSpanSample"  
        xmlns  
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        Title="TextBlockSpanSample" Height="100" Width="300">  
    <Grid>  
        <TextBlock Margin="10" TextWrapping="Wrap">  
            This <Span FontWeight="Bold">is</Span> a  
            <Span Background="Silver" Foreground="Maroon">TextBlock</Span>  
            with <Span TextDecorations="Underline">several</Span>  
            <Span FontStyle="Italic">Span</Span> elements,  
            <Span Foreground="Blue">  
                using a <Bold>variety</Bold> of <Italic>styles</Italic>  
            </Span>.  
        </TextBlock>  
    </Grid>  
</Window>
```



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

#### 1.5.2.6. Formatting text from C#/Code-Behind

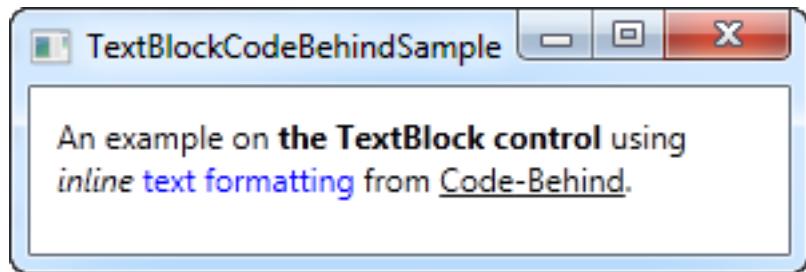
As you can see, formatting text through XAML is very easy, but in some cases, you might prefer or even need to do it from your C#/Code-Behind file. This is a bit more cumbersome, but here's an example on how you may do it:

```
<Window x:Class
        ="WpfTutorialSamples.Basic_controls.TextBlockCodeBehindSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBlockCodeBehindSample" Height="100" Width="300">
    <Grid></Grid>
</Window>

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Media;

namespace WpfTutorialSamples.Basic_controls
{
    public partial class TextBlockCodeBehindSample : Window
    {
        public TextBlockCodeBehindSample()
        {
            InitializeComponent();
            TextBlock tb = new TextBlock();
            tb.TextWrapping = TextWrapping.Wrap;
            tb.Margin = new Thickness(10);
            tb.Inlines.Add("An example on ");
            tb.Inlines.Add(new Run("the TextBlock control ") {
                FontWeight = FontWeights.Bold });
            tb.Inlines.Add("using ");
            tb.Inlines.Add(new Run("inline ") { FontStyle =
FontStyles.Italic });
            tb.Inlines.Add(new Run("text formatting ") { Foreground =
Brushes.Blue });
        }
    }
}
```

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



It's great to have the possibility, and it can be necessary to do it like this in some cases, but this example will probably make you appreciate XAML even more.

### 1.5.3. The Label control

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.LabelControlSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="LabelControlSample" Height="100" Width="200">
    <Grid>
        <Label Content="This is a Label control." />
    </Grid>
</Window>
```



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

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

#### 1.5.3.1. 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:

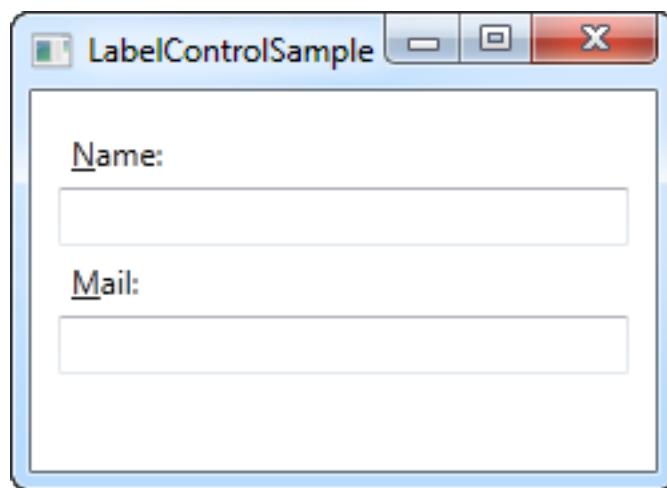
- Specify a border
- Render other controls, e.g. an image
- Use templated content through the ContentTemplate property
- **<b>Use access keys to give focus to related controls</b>**

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.

### 1.5.3.2. Label and Access keys (mnemonics)

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.LabelControlSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="LabelControlSample" Height="180" Width="250">
    <StackPanel Margin="10">
        <Label Content="_Name:" Target="{Binding ElementName=txtName}" />
        <TextBox Name="txtName" />
        <Label Content="_Mail:" Target="{Binding ElementName=txtMail}" />
        <TextBox Name="txtMail" />
    </StackPanel>
</Window>
```



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

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

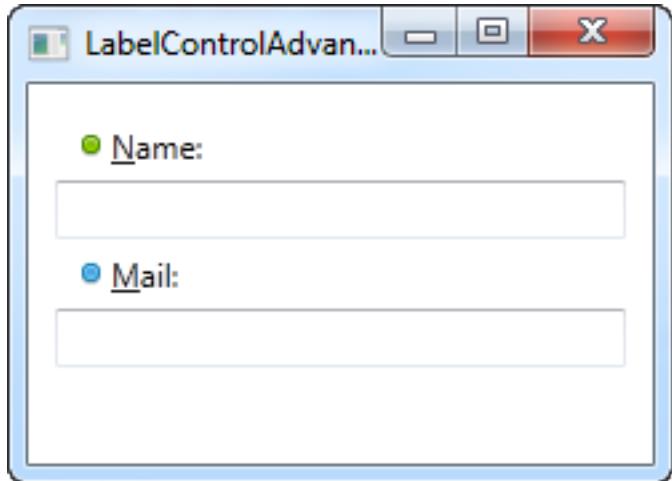
We use the **Target** property to connect the Label and the designated control. We use a standard WPF binding for this, using the **ElementName** property, all of which we will describe later on in this tutorial. The binding is based on the name of the control, so if you change this name, you will also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

#### 1.5.3.3. Using controls as Label content

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

```
<Window x:Class
    ="WpfTutorialSamples.Basic_controls.LabelControlAdvancedSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="LabelControlAdvancedSample" Height="180" Width="250">
    <StackPanel Margin="10">
        <Label Target="{Binding ElementName=txtName}">
            <StackPanel Orientation="Horizontal">
                <Image Source
    ="http://cdn1.iconfinder.com/data/icons/fatcow/16/bullet_green.png" />
                <AccessText Text="_Name:" />
            </StackPanel>
        </Label>
        <TextBox Name="txtName" />
        <Label Target="{Binding ElementName=txtMail}">
            <StackPanel Orientation="Horizontal">
                <Image Source
    ="http://cdn1.iconfinder.com/data/icons/fatcow/16/bullet_blue.png" />
                <AccessText Text="_Mail:" />
            </StackPanel>
        </Label>
        <TextBox Name="txtMail" />
    </StackPanel>
</Window>
```

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

#### 1.5.3.4. Summary

In most situations, the Label control does exactly what the name implies: It acts as a text label for another control. This is the primary purpose of it. For most other cases, you should probably use a TextBlock control or one of the other text containers that WPF offers.

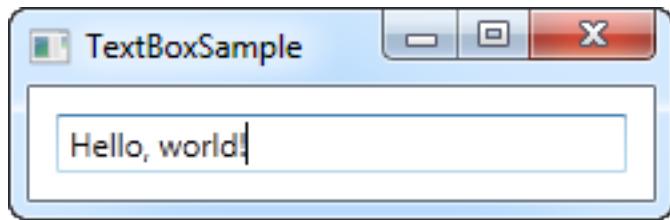
## 1.5.4. The TextBox control

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.

### 1.5.4.1. Single-line TextBox

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBoxSample" Height="80" Width="250">
    <StackPanel Margin="10">
        <TextBox />
    </StackPanel>
</Window>
```



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

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

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

### 1.5.4.2. Multi-line TextBox

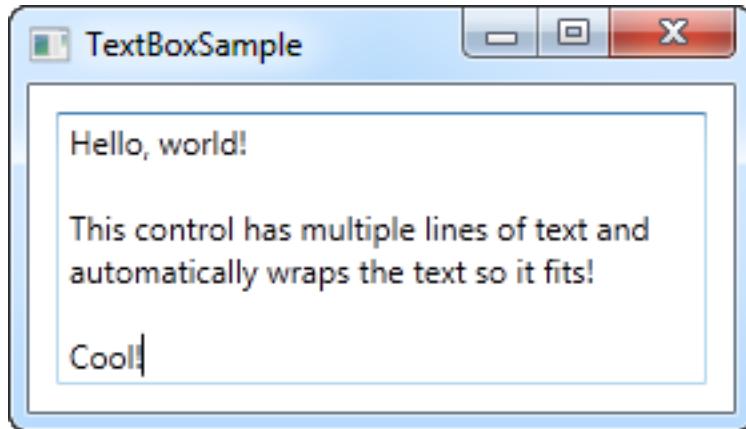
If you run the above example, you will notice that the TextBox control by default is a single-line control. Nothing happens when you press Enter and if you add more text than what can fit on a single line, the control just scrolls. However, making the TextBox control into a multi-line editor is very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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TextBoxSample" Height="160" Width="280">
<Grid Margin="10">
    <TextBox AcceptsReturn="True" TextWrapping="Wrap" />
</Grid>
</Window>

```



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

#### 1.5.4.3. Spellcheck with TextBox

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

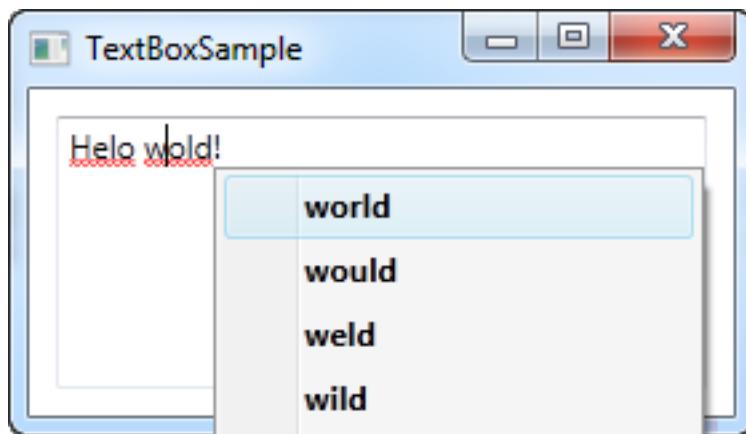
It works much like in Microsoft Word, where spelling errors are underlined and you can right-click it for suggested alternatives. Enabling spell checking is very easy:

```

<Window x:Class="WpfTutorialSamples.Basic_controls.TextBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBoxSample" Height="160" Width="280">
<Grid Margin="10">
    <TextBox AcceptsReturn="True" TextWrapping="Wrap"
    SpellCheck.IsEnabled="True" Language="en-US" />
</Grid>
</Window>

```

We have used the previous, multi-line textbox example as the basis and then I have added two new



properties: The attached property from the SpellCheck class called IsEnabled, which simply enables spell checking on the parent control, and the Language property, which instructs the spell checker which language to use.

#### 1.5.4.4. Working with TextBox selections

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.TextBoxSelectionSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextBoxSelectionSample" Height="150" Width="300">
    <DockPanel Margin="10">
        <TextBox SelectionChanged="TextBox_SelectionChanged" DockPanel.Dock="Top" />
        <TextBox Name="txtStatus" AcceptsReturn="True" TextWrapping="Wrap" IsReadOnly="True" />
    </DockPanel>
</Window>
```

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

```
using System;
using System.Text;
using System.Windows;
```

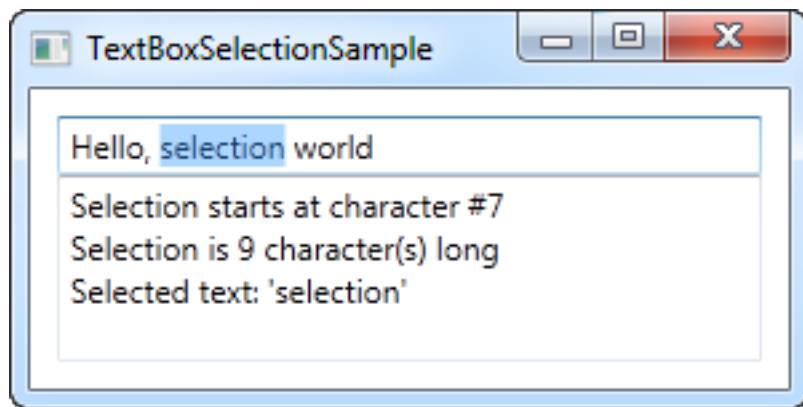
```

using System.Windows.Controls;

namespace WpfTutorialSamples.Basic_controls
{
    public partial class TextBoxSelectionSample : Window
    {
        public TextBoxSelectionSample()
        {
            InitializeComponent();
        }

        private void TextBox_SelectionChanged(object sender,
RoutedEventArgs e)
        {
            TextBox textBox = sender as TextBox;
            txtStatus.Text = "Selection starts at character #" +
textBox.SelectionStart + Environment.NewLine;
            txtStatus.Text += "Selection is " +
textBox.SelectionLength + " character(s) long" + Environment.NewLine;
            txtStatus.Text += "Selected text: '" +
textBox.SelectedText + "'";
        }
    }
}

```



We use three interesting properties to accomplish this:

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

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

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

#### 1.5.4.5. 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.

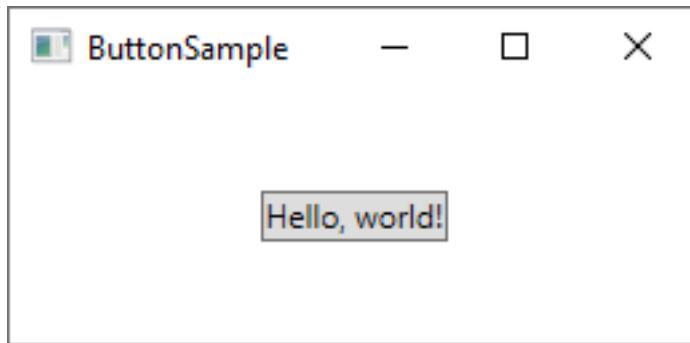
## 1.5.5. The Button control

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.

### 1.5.5.1. 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:

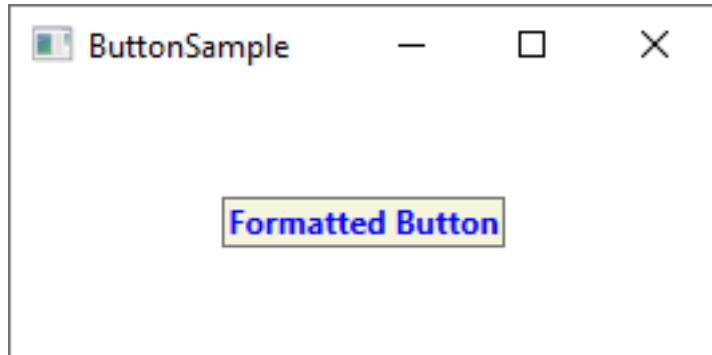
```
private void HelloWorldButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hello, world!");
}
```

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

### 1.5.5.2. 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>
```

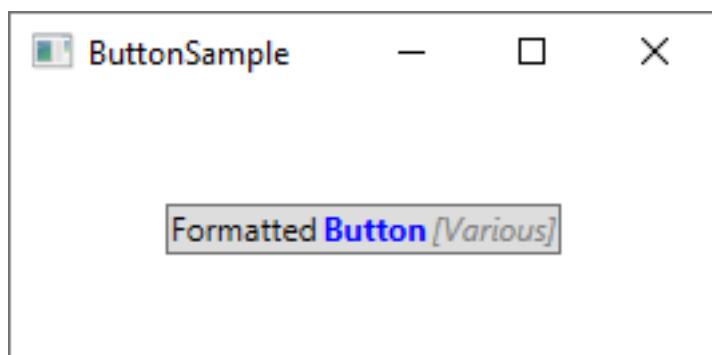


By setting these properties directly on the Button, you are of course limited to applying the same formatting for all of the content, but if that's not good enough, just read on for even more advanced content formatting.

#### 1.5.5.3. Buttons with advanced content

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

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

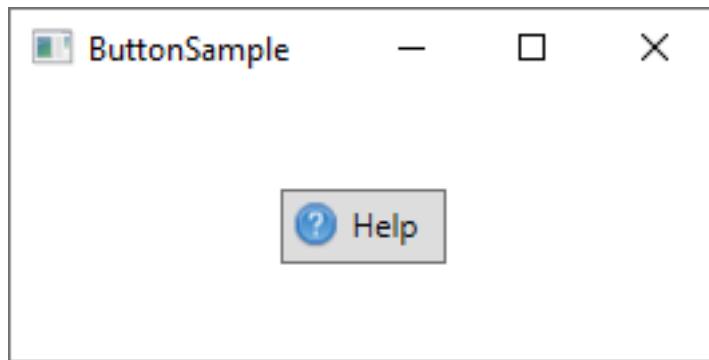


But of course, you are not limited to just text - you can put whatever you want inside your buttons, which leads us to a subject that I know many people will ask for. Buttons with images!

#### 1.5.5.4. Buttons with Images (ImageButton)

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

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



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

#### 1.5.5.5. Button Padding

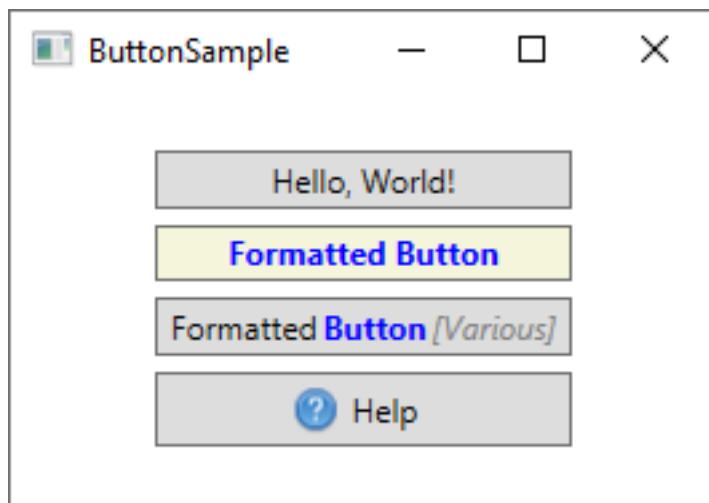
You may have noticed that buttons in the WPF framework doesn't come with any padding by default. This means that the text is very close to the borders, which might look a little bit strange, because most buttons found elsewhere (web, other applications etc.) do have at least some padding in the sides.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



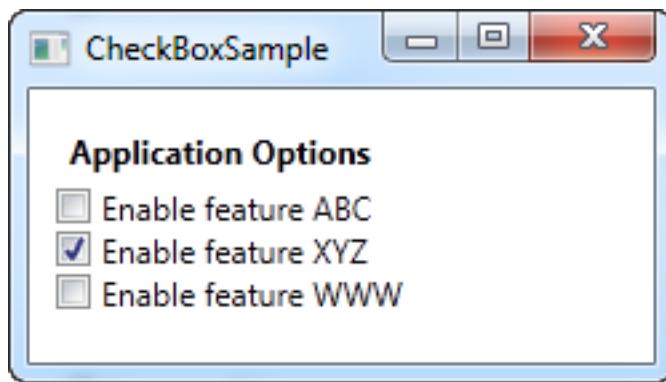
#### 1.5.5.6. Summary

As you can see from this article, using buttons in the WPF framework is very easy and you can customize this important control almost endlessly.

## 1.5.6. The CheckBox control

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.CheckBoxSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CheckBoxSample" Height="140" Width="250">
    <StackPanel Margin="10">
        <Label FontWeight="Bold">Application Options</Label>
        <CheckBox>Enable feature ABC</CheckBox>
        <CheckBox IsChecked="True">Enable feature XYZ</CheckBox>
        <CheckBox>Enable feature WWW</CheckBox>
    </StackPanel>
</Window>
```



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

### 1.5.6.1. Custom content

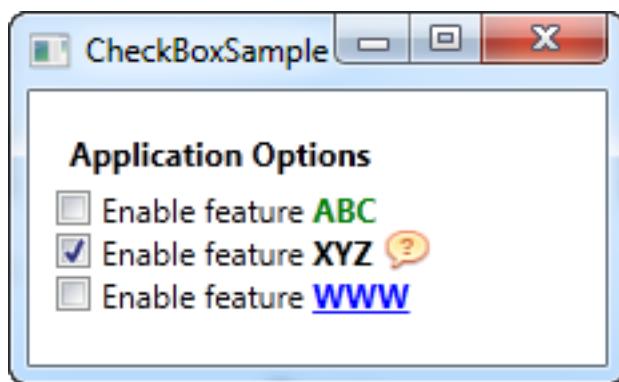
The CheckBox control inherits from the ContentControl class, which means that it can take custom content and display next to it. If you just specify a piece of text, like I did in the example above, WPF will put it inside a TextBlock control and display it, but this is just a shortcut to make things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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.CheckBoxSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```

    Title="CheckBoxSample" Height="140" Width="250">
<StackPanel Margin="10">
    <Label FontWeight="Bold">Application Options</Label>
    <CheckBox>
        <TextBlock>
            Enable feature <Run Foreground="Green" FontWeight="Bold">ABC</Run>
        </TextBlock>
    </CheckBox>
    <CheckBox IsChecked="True">
        <WrapPanel>
            <TextBlock>
                Enable feature <Run FontWeight="Bold">XYZ</Run>
            </TextBlock>
            <Image Source
=" /WpfTutorialSamples;component/Images/question.png" Width="16" Height
="16" Margin="5,0" />
        </WrapPanel>
    </CheckBox>
    <CheckBox>
        <TextBlock>
            Enable feature <Run Foreground="Blue"
TextDecorations="Underline" FontWeight="Bold">WWW</Run>
        </TextBlock>
    </CheckBox>
</StackPanel>
</Window>

```



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

CheckBox and toggle it on or off.

### 1.5.6.2. The IsThreeState property

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

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

```
<Window x:Class
    ="WpfTutorialSamples.Basic_controls.CheckBoxThreeStateSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CheckBoxThreeStateSample" Height="170" Width="300">
    <StackPanel Margin="10">
        <Label FontWeight="Bold">Application Options</Label>
        <StackPanel Margin="10,5">
            <CheckBox IsThreeState="True" Name="cbAllFeatures"
Checked="cbAllFeatures_CheckedChanged" Unchecked
="cbAllFeatures_CheckedChanged">Enable all</CheckBox>
            <StackPanel Margin="20,5">
                <CheckBox Name="cbFeatureAbc" Checked
="cbFeature_CheckedChanged" Unchecked="cbFeature_CheckedChanged">Enable
feature ABC</CheckBox>
                <CheckBox Name="cbFeatureXyz" IsChecked="True"
Checked="cbFeature_CheckedChanged" Unchecked="cbFeature_CheckedChanged">Enable
feature XYZ</CheckBox>
                <CheckBox Name="cbFeatureWww" Checked
="cbFeature_CheckedChanged" Unchecked="cbFeature_CheckedChanged">Enable
feature WWW</CheckBox>
            </StackPanel>
        </StackPanel>
    </StackPanel>
</Window>

using System;
using System.Windows;
```

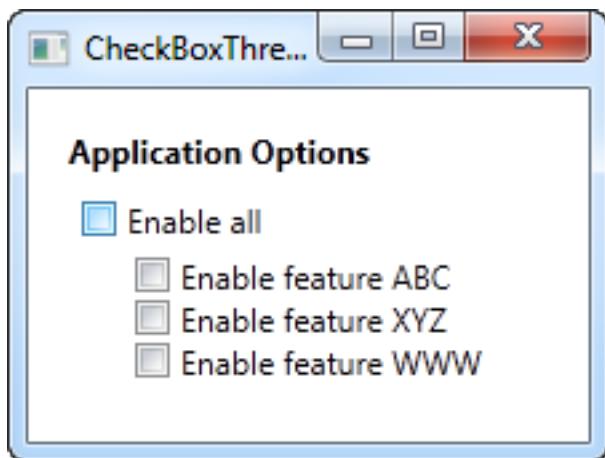
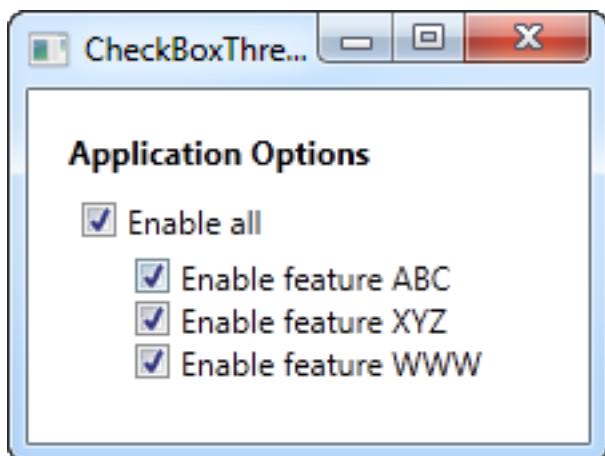
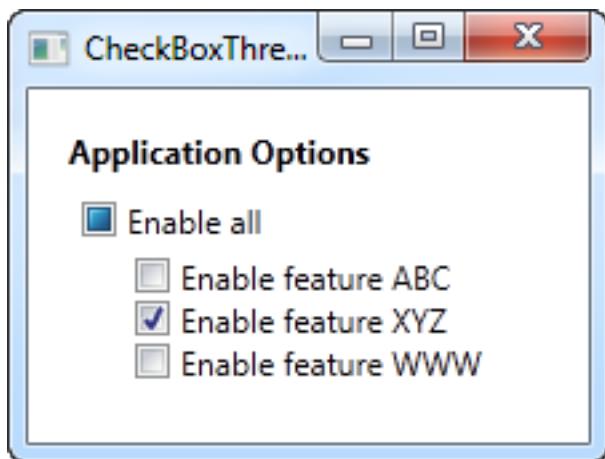
```

namespace WpfTutorialSamples.Basic_controls
{
    public partial class CheckBoxThreeStateSample : Window
    {
        public CheckBoxThreeStateSample()
        {
            InitializeComponent();
        }

        private void cbAllFeatures_CheckedChanged(object sender,
RoutedEventArgs e)
        {
            bool newVal = (cbAllFeatures.IsChecked == true);
            cbFeatureAbc.IsChecked = newVal;
            cbFeatureXyz.IsChecked = newVal;
            cbFeatureWww.IsChecked = newVal;
        }

        private void cbFeature_CheckedChanged(object sender,
RoutedEventArgs e)
        {
            cbAllFeatures.IsChecked = null;
            if((cbFeatureAbc.IsChecked == true) &&
(cbFeatureXyz.IsChecked == true) && (cbFeatureWwww.IsChecked == true))
                cbAllFeatures.IsChecked = true;
            if((cbFeatureAbc.IsChecked == false) &&
(cbFeatureXyz.IsChecked == false) && (cbFeatureWwww.IsChecked == false))
                cbAllFeatures.IsChecked = false;
        }
    }
}

```



This example works from two different angles: If you check or uncheck the "Enable all" CheckBox, then all of the child check boxes, each representing an application feature in our example, is either checked or unchecked. It also works the other way around though, where checking or unchecking a child CheckBox affects the "Enable all" CheckBox state: If they are all checked or unchecked, then the "Enable all" CheckBox gets the same state - otherwise the value will be left with a null, which forces the CheckBox into the indeterminate state.

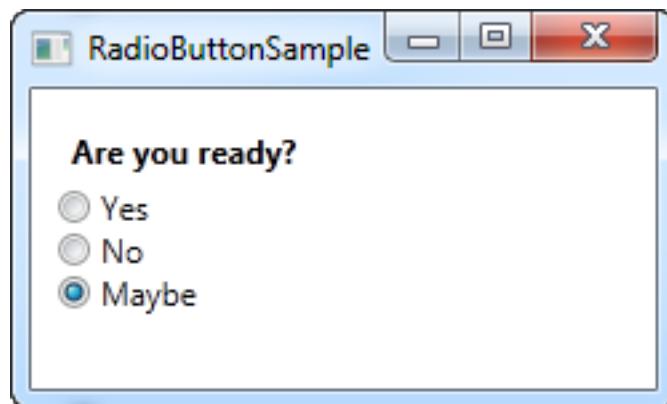
All of this behavior can be seen on the screenshots above, and is achieved by subscribing to the Checked and Unchecked events of the CheckBox controls. In a real world example, you would likely bind the values instead, but this example shows the basics of using the IsThreeState property to create a "Toggle all"

effect.

## 1.5.7. The RadioButton control

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.RadioButtonSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="RadioButtonSample" Height="150" Width="250">
    <StackPanel Margin="10">
        <Label FontWeight="Bold">Are you ready?</Label>
        <RadioButton>Yes</RadioButton>
        <RadioButton>No</RadioButton>
        <RadioButton IsChecked="True">Maybe</RadioButton>
    </StackPanel>
</Window>
```



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

### 1.5.7.1. RadioButton groups

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

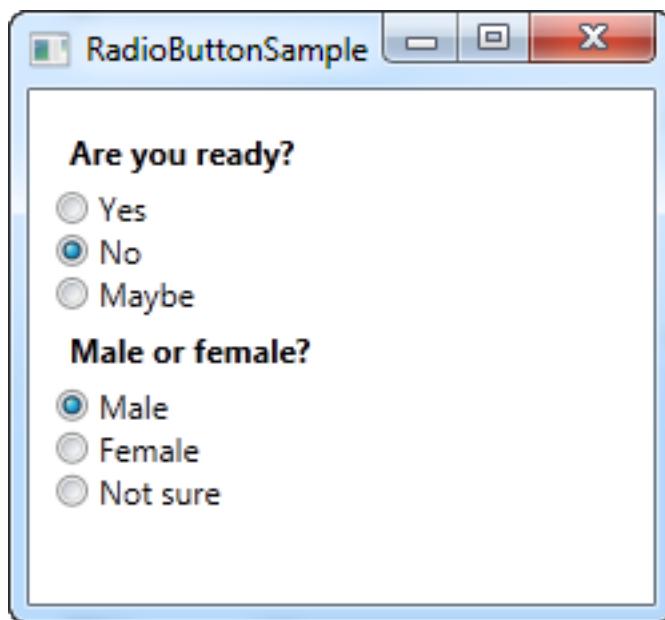
```
<Window x:Class="WpfTutorialSamples.Basic_controls.RadioButtonSample"
        xmlns="
```

```

="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="RadioButtonSample" Height="230" Width="250">
<StackPanel Margin="10">
    <Label FontWeight="Bold">Are you ready?</Label>
    <RadioButton GroupName="ready">Yes</RadioButton>
    <RadioButton GroupName="ready">No</RadioButton>
    <RadioButton GroupName="ready" IsChecked="True">Maybe</
RadioButton>

    <Label FontWeight="Bold">Male or female?</Label>
    <RadioButton GroupName="sex">Male</RadioButton>
    <RadioButton GroupName="sex">Female</RadioButton>
    <RadioButton GroupName="sex" IsChecked="True">Not sure</
RadioButton>
</StackPanel>
</Window>

```



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

#### 1.5.7.2. Custom content

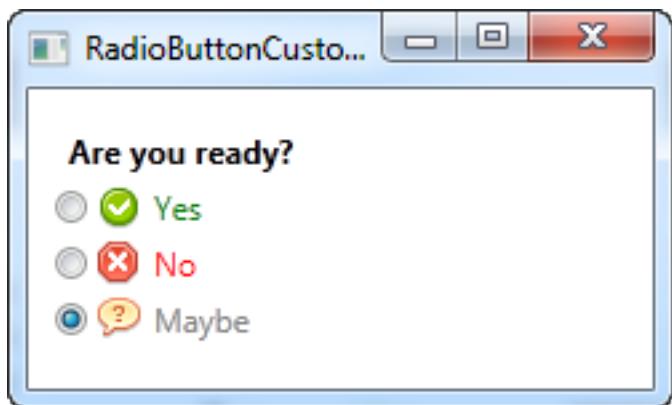
The `RadioButton` inherits from the `ContentControl` class, which means that it can take custom content and display next to it. If you just specify a piece of text, like I did in the example above, WPF will put it inside a `TextBlock` control and display it, but this is just a shortcut to make things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

```

<Window x:Class
        ="WpfTutorialSamples.Basic_controls.RadioButtonCustomContentSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="RadioButtonCustomContentSample" Height="150" Width
        ="250">
    <StackPanel Margin="10">
        <Label FontWeight="Bold">Are you ready?</Label>
        <RadioButton>
            <WrapPanel>
                <Image Source
                    ="/WpfTutorialSamples;component/Images/accept.png" Width="16" Height
                    ="16" Margin="0,0,5,0" />
                <TextBlock Text="Yes" Foreground="Green" />
            </WrapPanel>
        </RadioButton>
        <RadioButton Margin="0,0,5,0">
            <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



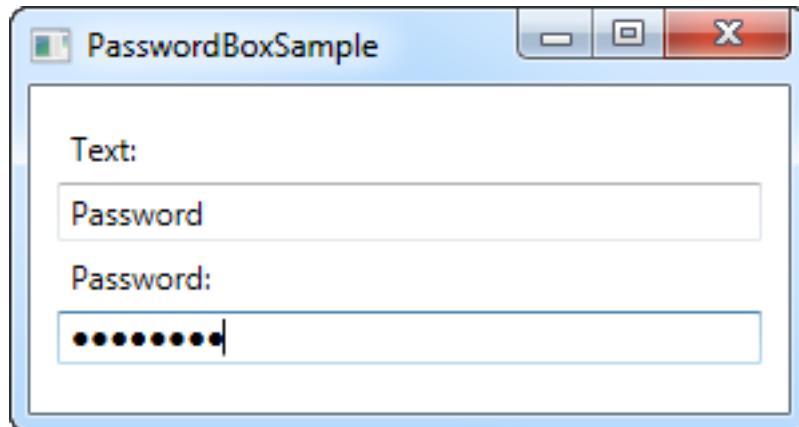
display an image for each choice.

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.

## 1.5.8. The PasswordBox control

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

```
<Window x:Class="WpfTutorialSamples.Basic_controls.PasswordBoxSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PasswordBoxSample" Height="160" Width="300">
    <StackPanel Margin="10">
        <Label>Text:</Label>
        <TextBox />
        <Label>Password:</Label>
        <PasswordBox />
    </StackPanel>
</Window>
```



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

```
<PasswordBox PasswordChar="X" />
```

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

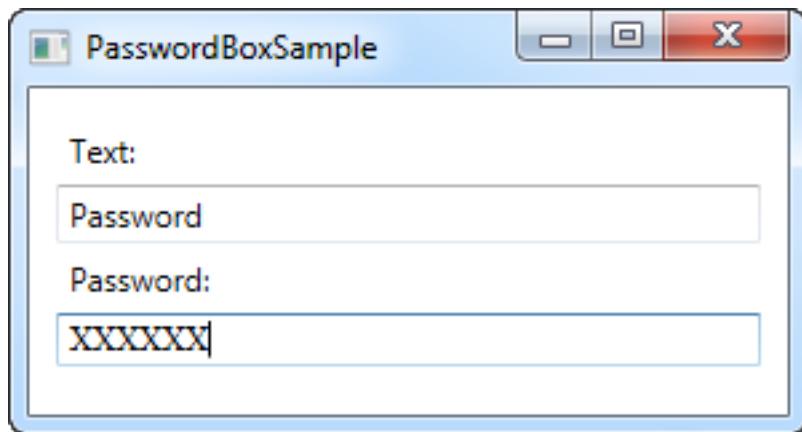
```
<PasswordBox MaxLength="6" />
```

I have used both properties in this updated example:

```

<Window x:Class="WpfTutorialSamples.Basic_controls.PasswordBoxSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PasswordBoxSample" Height="160" Width="300">
    <StackPanel Margin="10">
        <Label>Text:</Label>
        <TextBox />
        <Label>Password:</Label>
        <PasswordBox MaxLength="6" PasswordChar="X" />
    </StackPanel>
</Window>

```



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

### 1.5.8.1. 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>

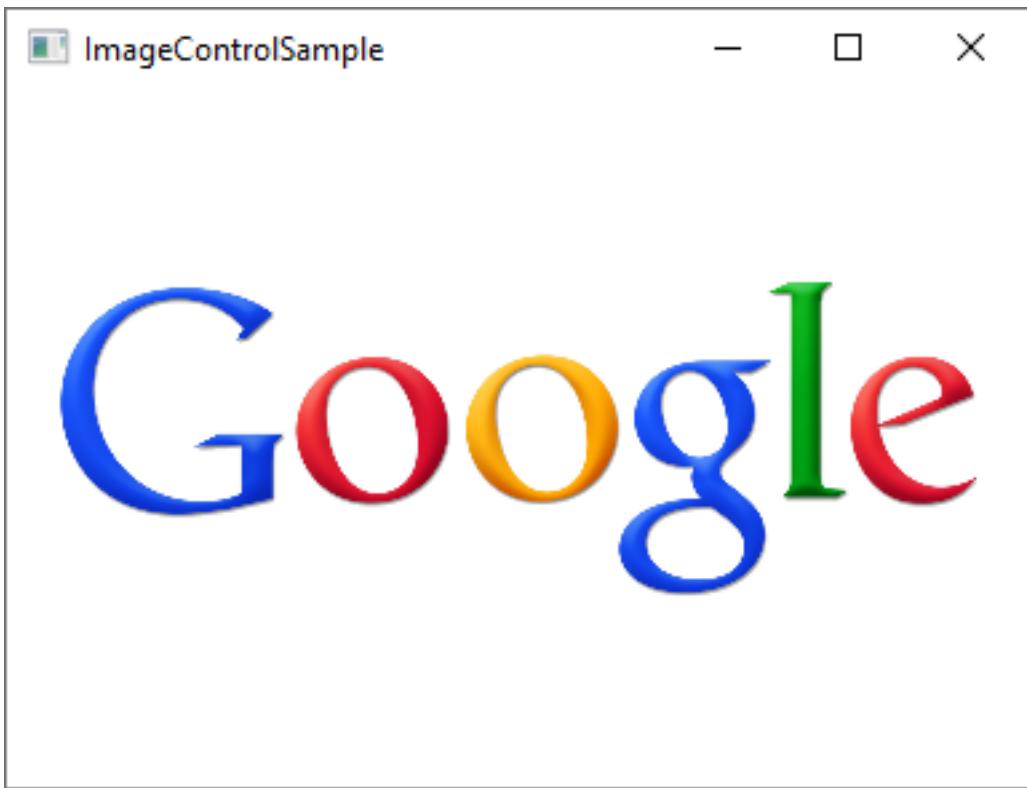
---

## 1.5.9. 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.

### 1.5.9.1. 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 (<em>/WpfTutorialSamples;component</em>), where the assembly name (<strong>WpfTutorialSamples</strong> in my application) is combined with the word "component"
- The second part, where the relative path of the resource is specified:  
<em>/Images/google.png</em>

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" />
```

#### 1.5.9.2. Loading images dynamically (Code-behind)

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

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

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

```
private void BtnLoadFromResource_Click(object sender, RoutedEventArgs e)
{
```

```

        Uri resourceUri = new Uri( "/Images/white_bengal_tiger.jpg" ,
UriKind.Relative);
        imgDynamic.Source = new BitmapImage(resourceUri);
}

```

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

```

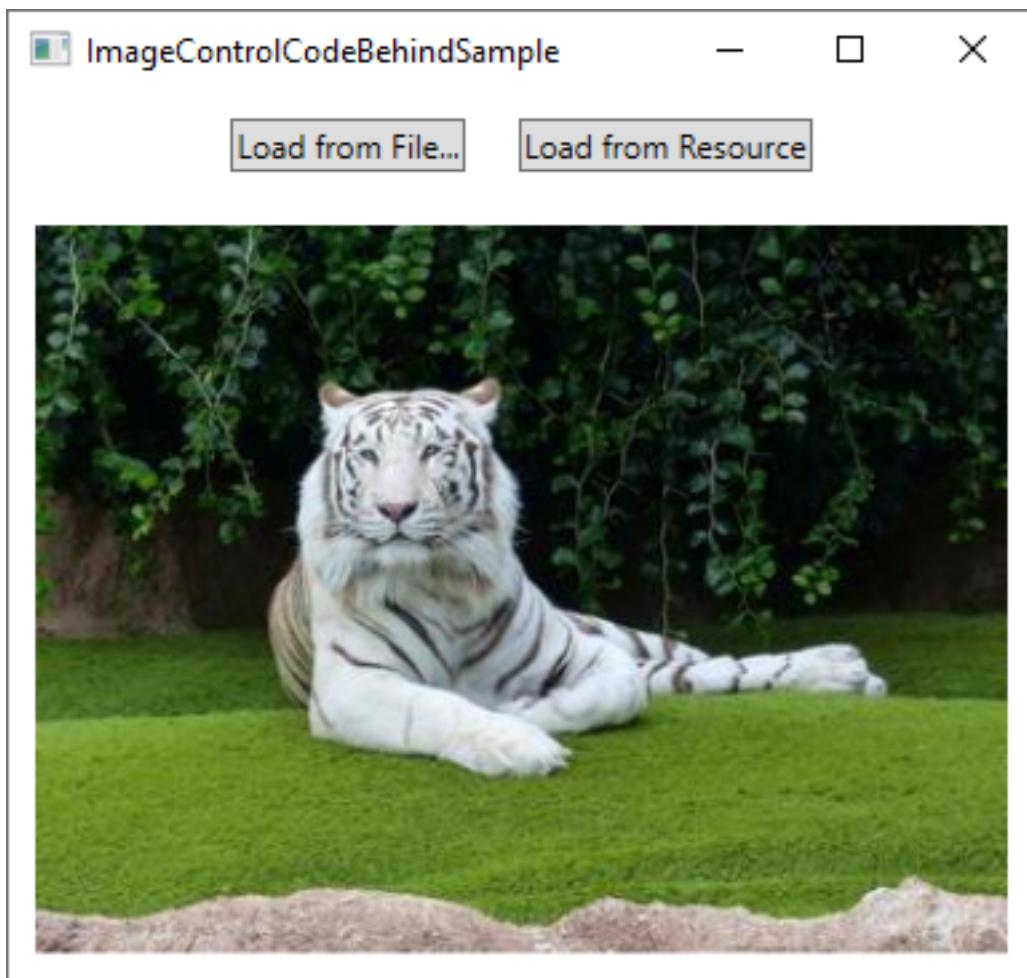
<Window x:Class
        ="WpfTutorialSamples.Basic_controls.ImageControlCodeBehindSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
        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>

```

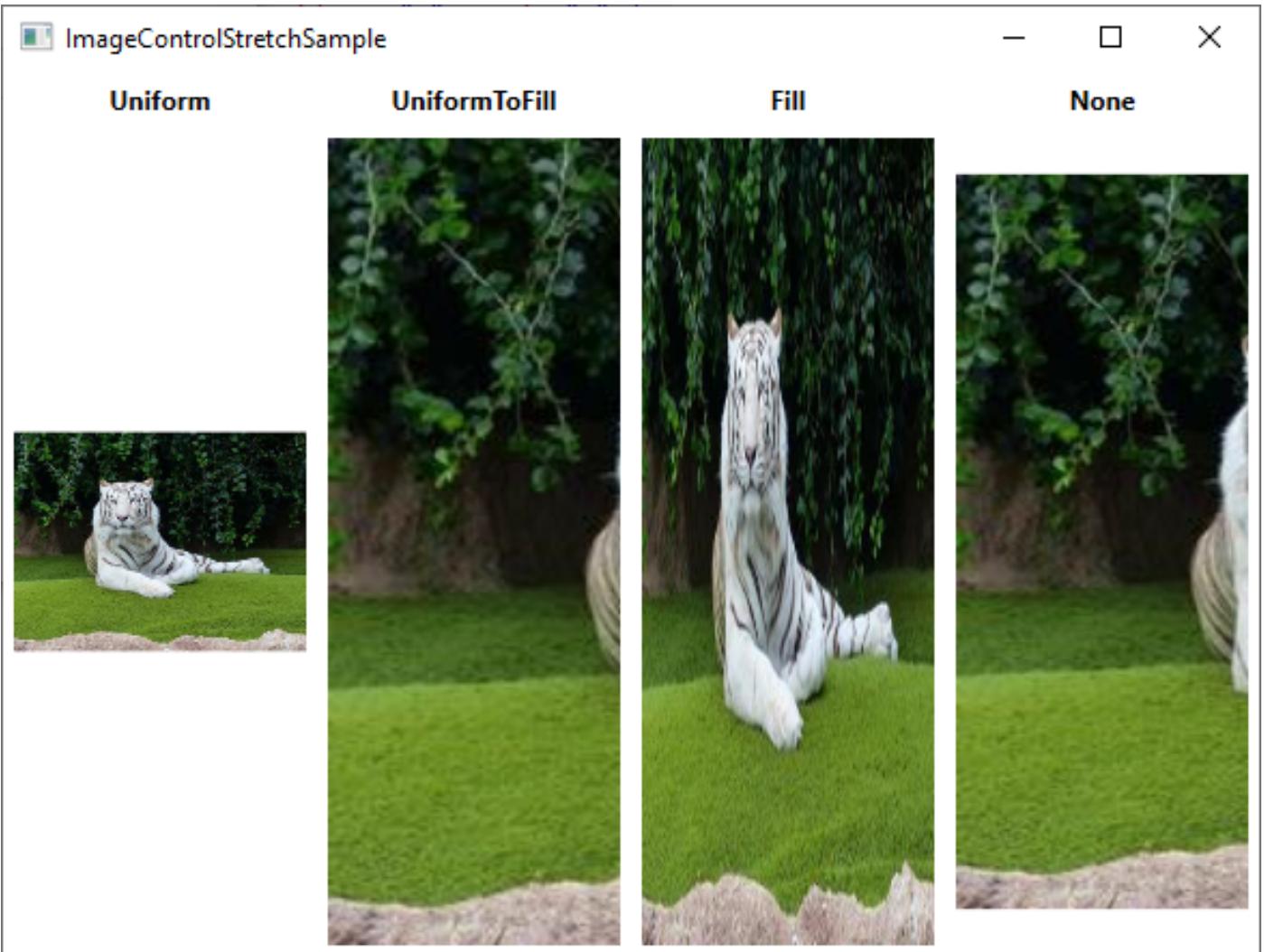
### 1.5.9.3. The Stretch property

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

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



```
<Window x:Class
        ="WpfTutorialSamples.Basic_controls.ImageControlStretchSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:WpfTutorialSamples.Basic_controls"
        mc:Ignorable="d"
        Title="ImageControlStretchSample" Height="450" Width="600">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
```



```
<RowDefinition Height="Auto" />
<RowDefinition Height="*" />
</Grid.RowDefinitions>
<Label Grid.Column="0" HorizontalAlignment="Center" FontWeight="Bold">Uniform</Label>
<Label Grid.Column="1" HorizontalAlignment="Center" FontWeight="Bold">UniformToFill</Label>
<Label Grid.Column="2" HorizontalAlignment="Center" FontWeight="Bold">Fill</Label>
<Label Grid.Column="3" HorizontalAlignment="Center" FontWeight="Bold">None</Label>
<Image Source="/Images/white_bengal_tiger.jpg" Stretch="Uniform" Grid.Column="0" Grid.Row="1" Margin="5" />
<Image Source="/Images/white_bengal_tiger.jpg" Stretch="UniformToFill" Grid.Column="1" Grid.Row="1" Margin="5" />
<Image Source="/Images/white_bengal_tiger.jpg" Stretch="Fill" Grid.Column="2" Grid.Row="1" Margin="5" />
<Image Source="/Images/white_bengal_tiger.jpg" Stretch="None" Grid.Column="3" Grid.Row="1" Margin="5" />
```

```
Grid.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.

#### 1.5.9.4. Summary

The WPF **Image** control makes it easy for you to display an image in your application, whether from a remote source, an embedded resource or from the local computer, as demonstrated in this article.

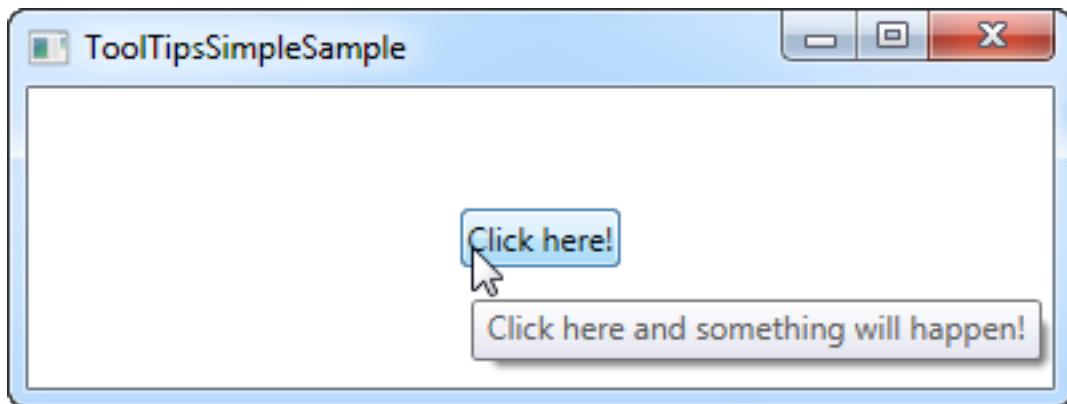
# 1.6. Control concepts

## 1.6.1. Control ToolTips

Tooltips, infotips or hints - various names, but the concept remains the same: The ability to get extra information about a specific control or link by hovering the mouse over it. WPF obviously supports this concept as well, and by using the **ToolTip** property found on the **FrameworkElement** class, which almost any WPF control inherits from.

Specifying a tooltip for a control is very easy, as you will see in this first and very basic example:

```
<Window x:Class  
= "WpfTutorialSamples.Control_concepts.ToolTipssimpleSample"  
    xmlns  
= "http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Title="ToolTipsSimpleSample" Height="150" Width="400">  
    <Grid VerticalAlignment="Center" HorizontalAlignment="Center">  
  
        <Button ToolTip="Click here and something will happen!">Click  
here!</Button>  
  
    </Grid>  
</Window>
```



As you can see on the screenshots, this results in a floating box with the specified string, once the mouse hovers over the button. This is what most UI frameworks offers - the display of a text string and nothing more.

However, in WPF, the **ToolTip** property is actually not a string type, but instead an object type, meaning that we can put whatever we want in there. This opens up for some pretty cool possibilities, where we can provide the user with much richer and more helpful tooltips. For instance, consider this example and

compare it to the first one:

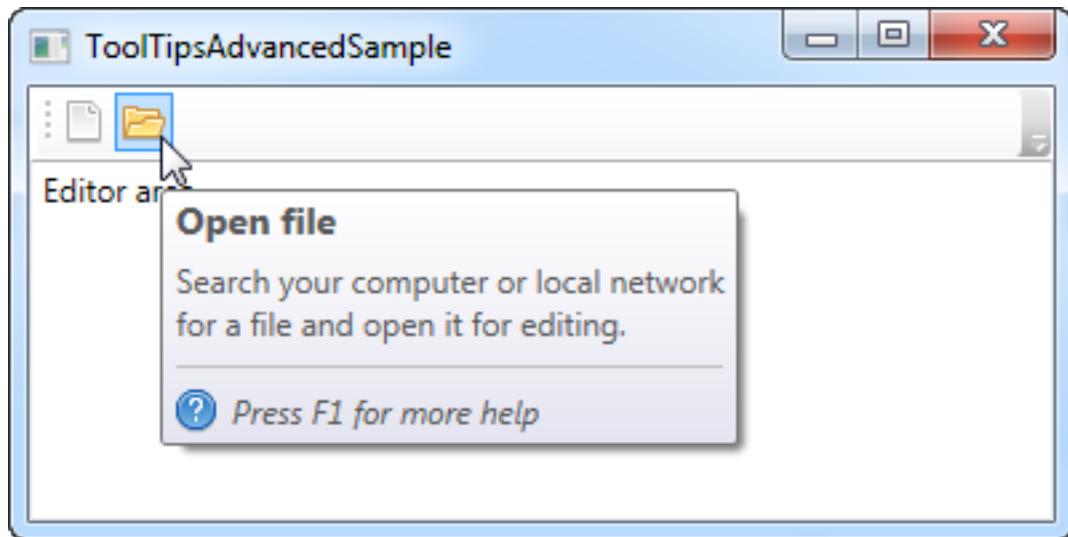
```
<Window x:Class
        ="WpfTutorialSamples.Control_concepts.ToolTipsAdvancedSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ToolTipsAdvancedSample" Height="200" Width="400"
        UseLayoutRounding="True">
    <DockPanel>
        <ToolBar DockPanel.Dock="Top">
            <Button ToolTip="Create a new file">
                <Button.Content>
                    <Image Source
                           ="/WpfTutorialSamples;component/Images/page_white.png" Width="16" Height
                           ="16" />
                </Button.Content>
            </Button>
            <Button>
                <Button.Content>
                    <Image Source
                           ="/WpfTutorialSamples;component/Images/folder.png" Width="16" Height
                           ="16" />
                </Button.Content>
                <Button.ToolTip>
                    <StackPanel>
                        <TextBlock FontWeight="Bold" FontSize="14"
Margin="0,0,0,5">Open file</TextBlock>
                        <TextBlock>
                            Search your computer or local network
                            <LineBreak />
                            for a file and open it for editing.
                        </TextBlock>
                    <Border BorderBrush="Silver"
BorderThickness="0,1,0,0" Margin="0,8" />
                    <WrapPanel>
                        <Image Source
                               ="/WpfTutorialSamples;component/Images/help.png" Margin="0,0,5,0" />
                        <TextBlock FontStyle="Italic">Press
F1 for more help</TextBlock>
                
```

```

        </WrapPanel>
    </StackPanel>
</Button.ToolTip>
</Button>
</ToolBar>

<TextBox>
    Editor area...
</TextBox>
</DockPanel>
</Window>

```



Notice how this example uses a simple string tooltip for the first button and then a much more advanced one for the second button. In the advanced case, we use a panel as the root control and then we're free to add controls to that as we please. The result is pretty cool, with a header, a description text and a hint that you can press F1 for more help, including a help icon.

#### 1.6.1.1. 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 documentation:

### 1.6.1.2. Summary

Tooltips can be a great help for the user, and in WPF, they are both easy to use and extremely flexible. Combine the fact that you can completely control the design and content of the tooltip, with properties from the **ToolTipService** class, to create more user friendly inline help in your applications.

## 1.6.2. 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.

### 1.6.2.1. 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.

### 1.6.2.2. TextFormattingMode

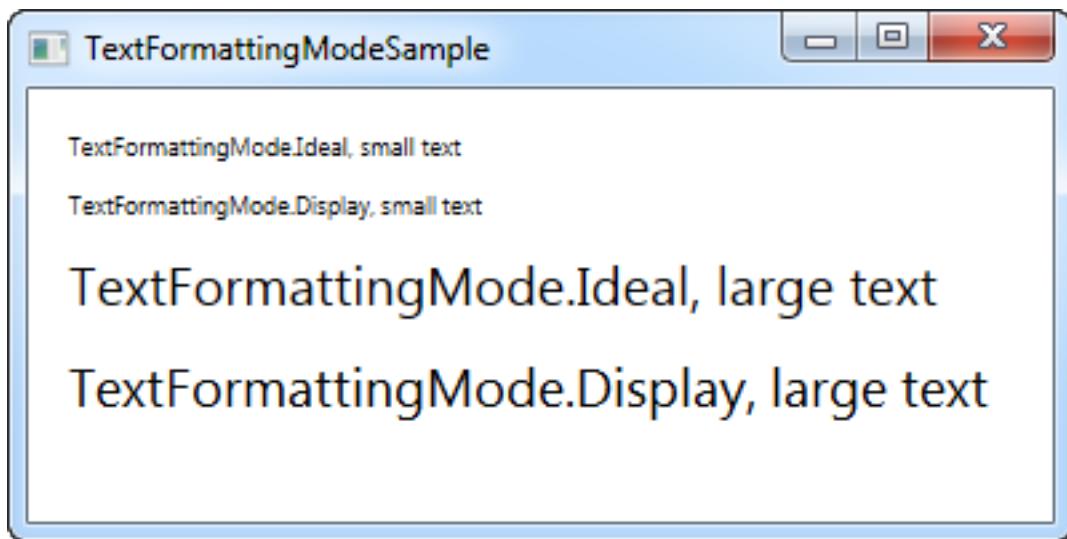
Using the **TextFormattingMode** property, you get to decide which algorithm should be used when formatting the text. You can choose between **Ideal** (the default value) and **Display**. You would normally want to leave this property untouched, since the Ideal setting will be best for most situations,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

```
<Window x:Class
        ="WpfTutorialSamples.Control_concepts.TextFormattingModeSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TextFormattingModeSample" Height="200" Width="400">
    <StackPanel Margin="10">
        <Label TextOptions.TextFormattingMode="Ideal" FontSize="9">
TextFormattingMode.Ideal, small text</Label>
        <Label TextOptions.TextFormattingMode="Display" FontSize="9">
TextFormattingMode.Display, small text</Label>
```

```

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

```



#### 1.6.2.3. TextRenderingMode

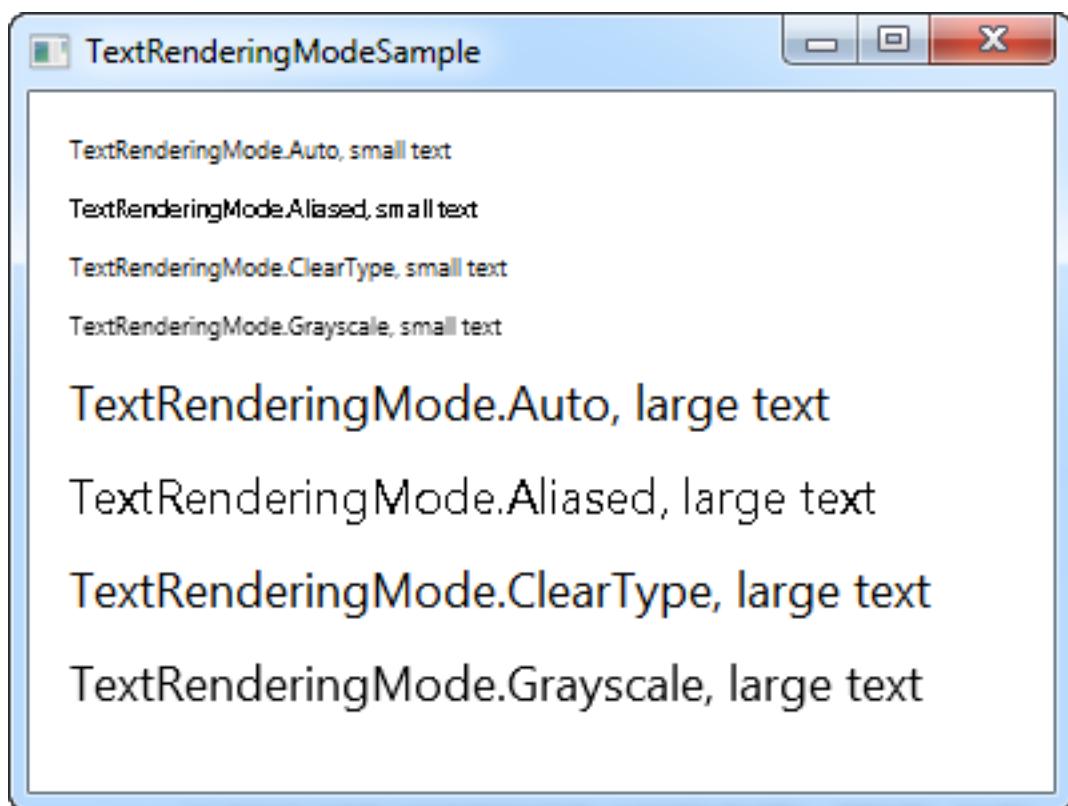
The **TextRenderingMode** property gives you control of which antialiasing algorithm is used when rendering text. It has the biggest effect in combination with the **Display** setting for the **TextFormattingMode** property, which we'll use in this example to illustrate the differences:

```

<Window x:Class
="WpfTutorialSamples.Control_concepts.TextRenderingModeSample"
    xmlns
="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TextRenderingModeSample" Height="300" Width="400">
    <StackPanel Margin="10" TextOptions.TextFormattingMode="Display">
        <Label TextOptions.TextRenderingMode="Auto" FontSize="9">
TextRenderingMode.Auto, small text</Label>
        <Label TextOptions.TextRenderingMode="Aliased" FontSize="9">
TextRenderingMode.Aliased, small text</Label>
        <Label TextOptions.TextRenderingMode="ClearType" FontSize="9">
TextRenderingMode.ClearType, small text</Label>
        <Label TextOptions.TextRenderingMode="Grayscale" FontSize="9">
TextRenderingMode.Grayscale, small text</Label>
        <Label TextOptions.TextRenderingMode="Auto" FontSize="18">

```

```
TextRenderingMode.Auto, large text</Label>
    <Label TextOptions.TextRenderingMode="Aliased" FontSize="18">
TextRenderingMode.Aliased, large text</Label>
    <Label TextOptions.TextRenderingMode="ClearType" FontSize="18" 
>TextRenderingMode.ClearType, large text</Label>
    <Label TextOptions.TextRenderingMode="Grayscale" FontSize="18" 
>TextRenderingMode.Grayscale, large text</Label>
</StackPanel>
</Window>
```

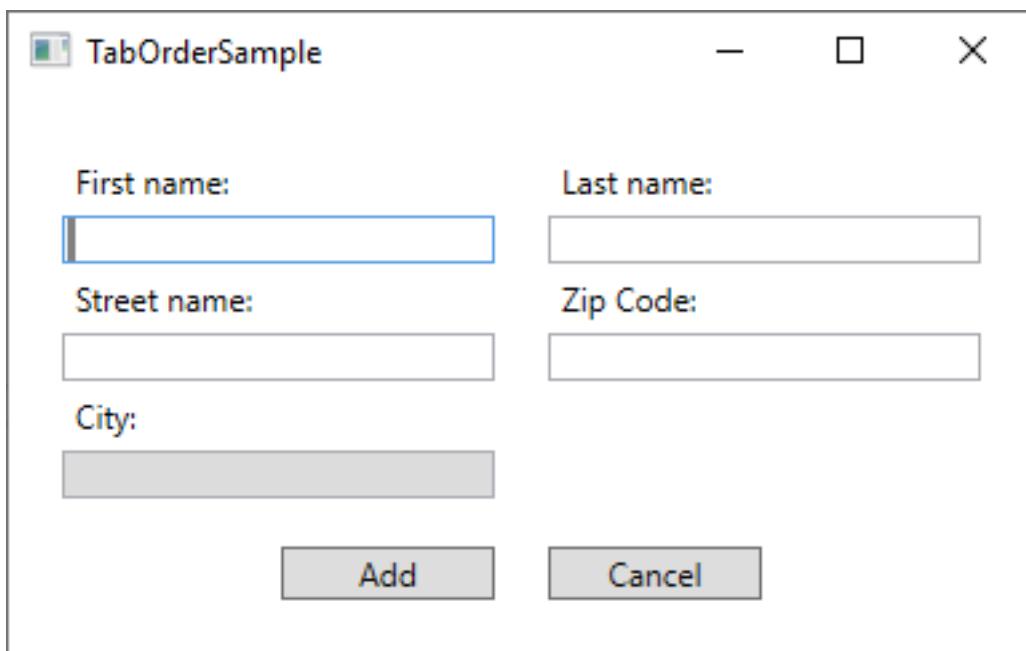


As you can see, the resulting text differs quite a bit in how it looks and once again, you should mainly change this in special circumstances.

### 1.6.3. Tab Order

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

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



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

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

```
<Window x:Class="WpfTutorialSamples.Control_concepts.TabOrderSample"
```

```

    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-
namespace:WpfTutorialSamples.Control_concepts"
        mc:Ignorable="d"
        Title="TabOrderSample" Height="250" Width="400">
<Grid Margin="20">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="20" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <StackPanel>
        <Label>First name:</Label>
        <TextBoxTabIndex="0" />
        <Label>Street name:</Label>
        <TextBoxTabIndex="2" />
        <Label>City:</Label>
        <TextBoxTabIndex="5" IsReadOnly="True" IsTabStop="False"
Background="Gainsboro" />
    </StackPanel>
    <StackPanel Grid.Column="2">
        <Label>Last name:</Label>
        <TextBoxTabIndex="1" />
        <Label>Zip Code:</Label>
        <TextBoxTabIndex="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!

#### 1.6.3.1. Summary

Controlling the tab order of a dialog is very important, but fortunately for us, WPF does a very good job of automatically defining a proper tab order for you. However, in some cases, it will make sense to go in and take control, using the **TabIndex** and **IsTabStop** properties, as illustrated in the example above.

## 1.6.4. Access Keys

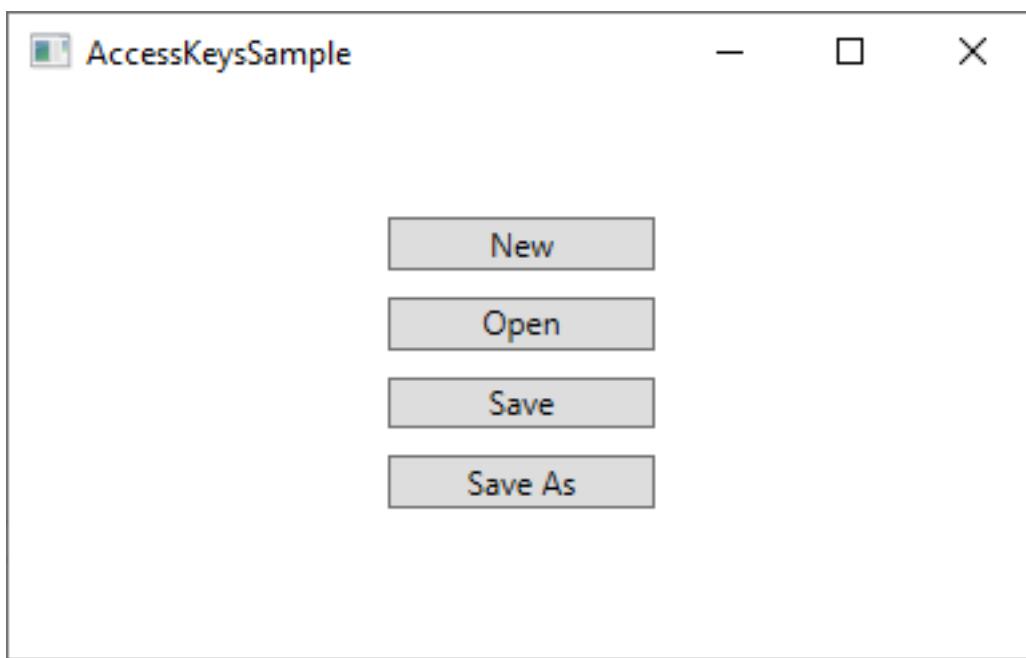
The concept of **Access Keys**, sometimes referred to as *Accelerator Keys* or *Keyboard Accelerators*, allows you to reach a specific control inside a window by holding down the Alt key and then pressing another key on the keyboard. This enhances the usability of your windows, because it allows the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

### 1.6.4.1. 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>
```

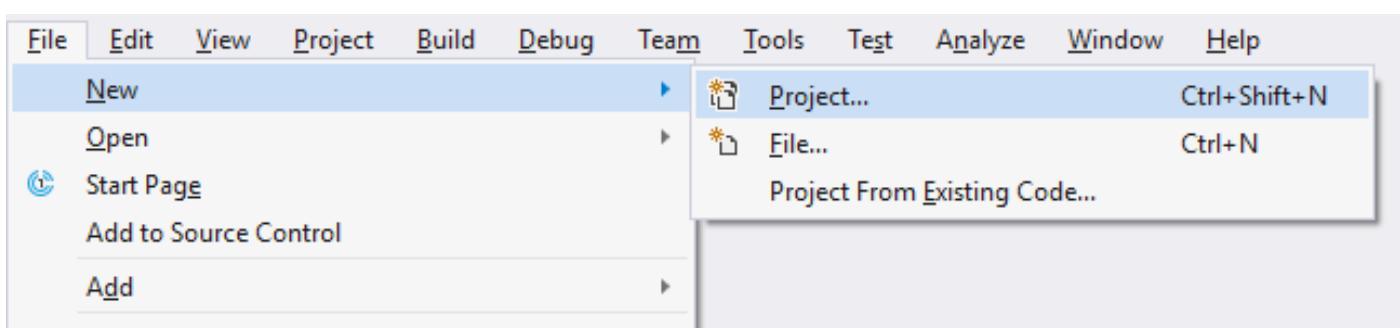
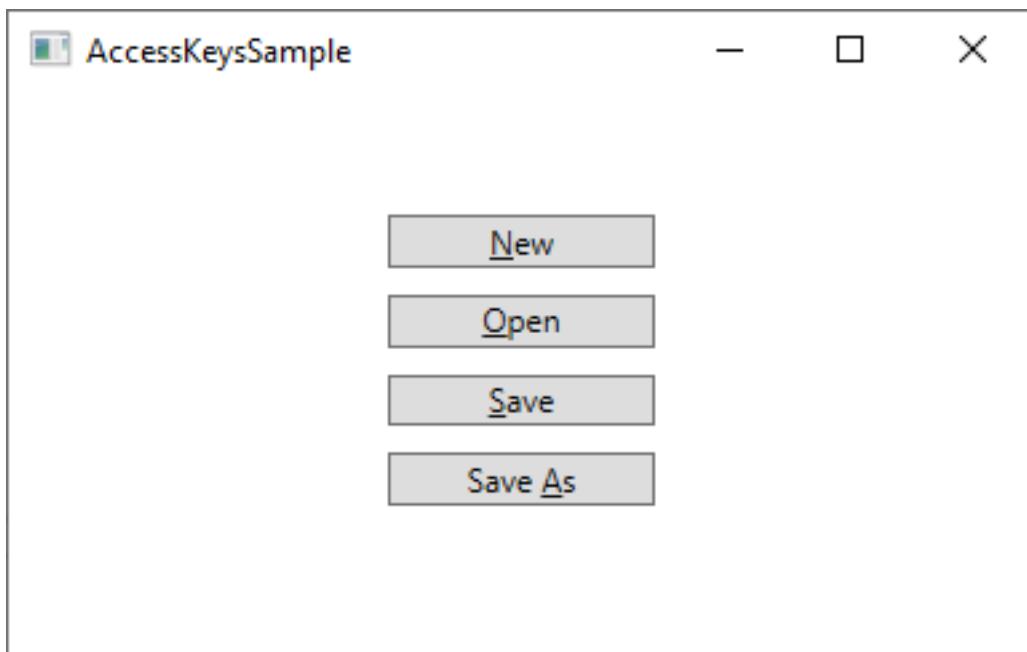
Notice the underscore (\_) just before the N character - this will turn the N key into the designated Access Key for this Button control. By default, the look of your control(s) doesn't change, as you can see from this example where I have defined Access Keys for all the buttons:



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.

#### 1.6.4.2. 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 **I** in *Save All*)

- In general, you may want to avoid narrow characters like <em>i</em> and <em>l</em>, and go for the wider characters like <em>m</em>, <em>s</em>, <em>w</em> etc.

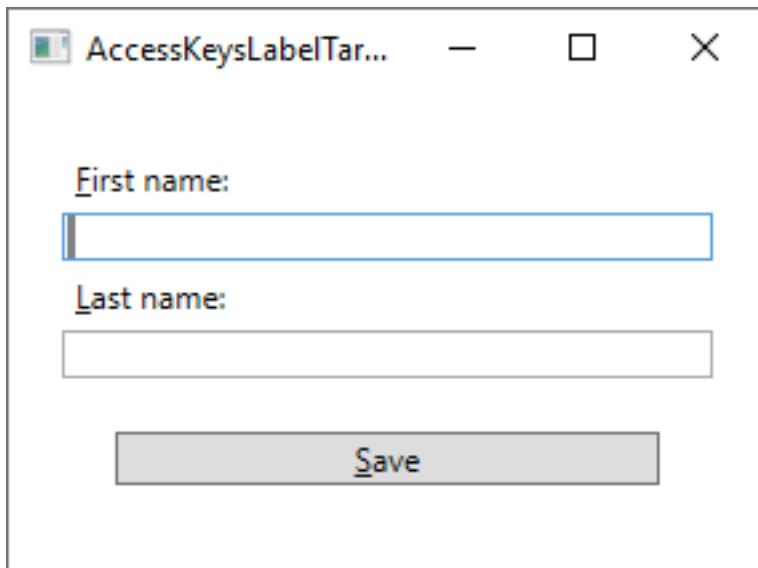
#### 1.6.4.3. Tying together two controls

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

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

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

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



#### 1.6.4.4. Summary

By using Access Keys in your windows/dialogs, you are making it much easier for people to navigate using only their keyboards. This is especially popular among power-users, who will use the keyboard in favor of the mouse as much as possible. You should always use Access Keys, especially for your menus.

# 1.7. Panels

---

## 1.7.1. Introduction to WPF panels

Panels are one of the most important control types of WPF. They act as containers for other controls and control the layout of your windows/pages. Since a window can only contain ONE child control, a panel is often used to divide up the space into areas, where each area can contain a control or another panel (which is also a control, of course).

Panels come in several different flavors, with each of them having its own way of dealing with layout and child controls. Picking the right panel is therefore essential to getting the behavior and layout you want, and especially in the start of your WPF career, this can be a difficult job. The next section will describe each of the panels shortly and give you an idea of when to use it. After that, move on to the next chapters, where each of the panels will be described in detail.

### 1.7.1.1. Canvas

A simple panel, which mimics the WinForms way of doing things. It allows you to assign specific coordinates to each of the child controls, giving you total control of the layout. This is not very flexible though, because you have to manually move the child controls around and make sure that they align the way you want them to. Use it (only) when you want complete control of the child control positions.

### 1.7.1.2. WrapPanel

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.

### 1.7.1.3. StackPanel

The StackPanel acts much like the WrapPanel, but instead of wrapping if the child controls take up too much room, it simply expands itself, if possible. Just like with the WrapPanel, the orientation can be either horizontal or vertical, but instead of adjusting the width or height of the child controls based on the largest item, each item is stretched to take up the full width or height. Use the StackPanel when you want a list of controls that takes up all the available room, without wrapping.

### 1.7.1.4. DockPanel

The DockPanel allows you to dock the child controls to the top, bottom, left or right. By default, the last control, if not given a specific dock position, will fill the remaining space. You can achieve the same with the Grid panel, but for the simpler situations, the DockPanel will be easier to use. Use the DockPanel whenever you need to dock one or several controls to one of the sides, like for dividing up the window into specific areas.

### 1.7.1.5. Grid

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.

#### 1.7.1.6. UniformGrid

The UniformGrid is just like the Grid, with the possibility of multiple rows and columns, but with one important difference: All rows and columns will have the same size! Use this when you need the Grid behavior without the need to specify different sizes for the rows and columns.

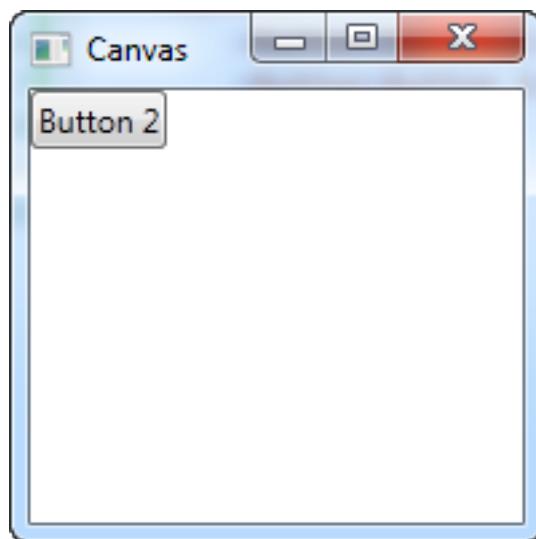
## 1.7.2. The Canvas control

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.

More about that later, let's get into a simple example. This one is mostly about showing you just how little the Canvas does by default:

```
<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>
```



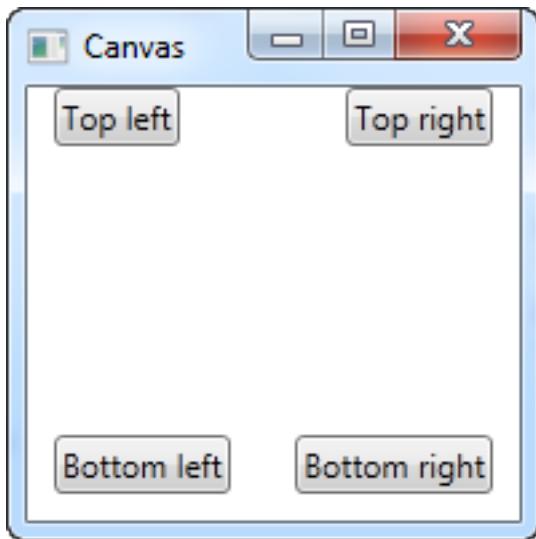
As you can see, even though we have two buttons, they are both placed in the exact same place, so only the last one is visible. The Canvas does absolutely nothing until you start giving coordinates to the child controls. This is done using the Left, Right, Top and Bottom attached properties from the Canvas control.

These properties allow you to specify the position relative to the four edges of the Canvas. By default, they are all set to NaN (Not a Number), which will make the Canvas place them in the upper left corner, but as mentioned, you can easily change this:

```

<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>

```



Notice how I only set the property or properties that I need. For the first two buttons, I only wish to specify a value for the X axis, so I use the Left and Right properties to push the buttons towards the center, from each direction.

For the bottom buttons, I use both Left/Right and Bottom to push them towards the center in both directions. You will usually specify either a Top or a Bottom value and/or a Left or a Right value.

As mentioned, since the Canvas gives you complete control of positions, it won't really care whether or not there's enough room for all your controls or if one is on top of another. This makes it a bad choice for pretty much any kind of dialog design, but the Canvas is, as the name implies, great for at least one thing: Painting. WPF has a bunch of controls that you can place inside a Canvas, to make nice illustrations.

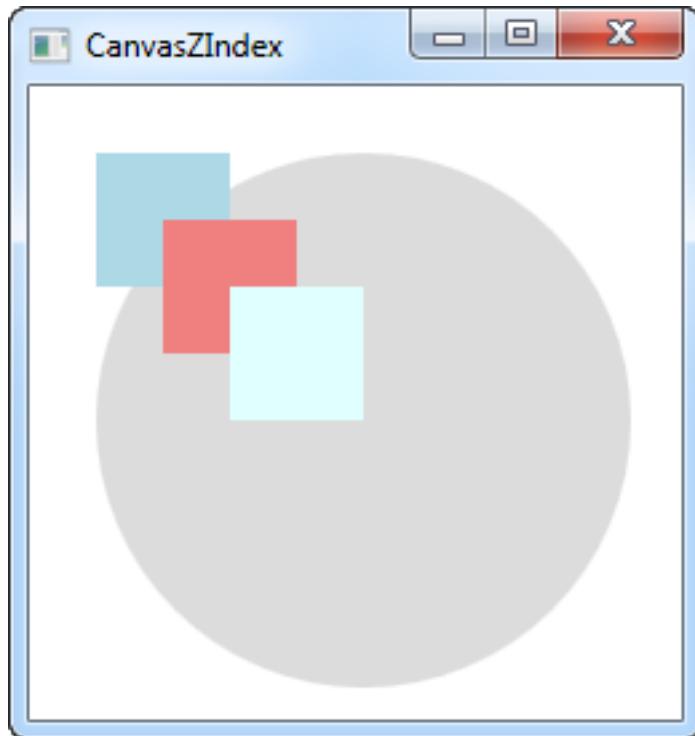
### 1.7.2.1. Z-Index

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>
```

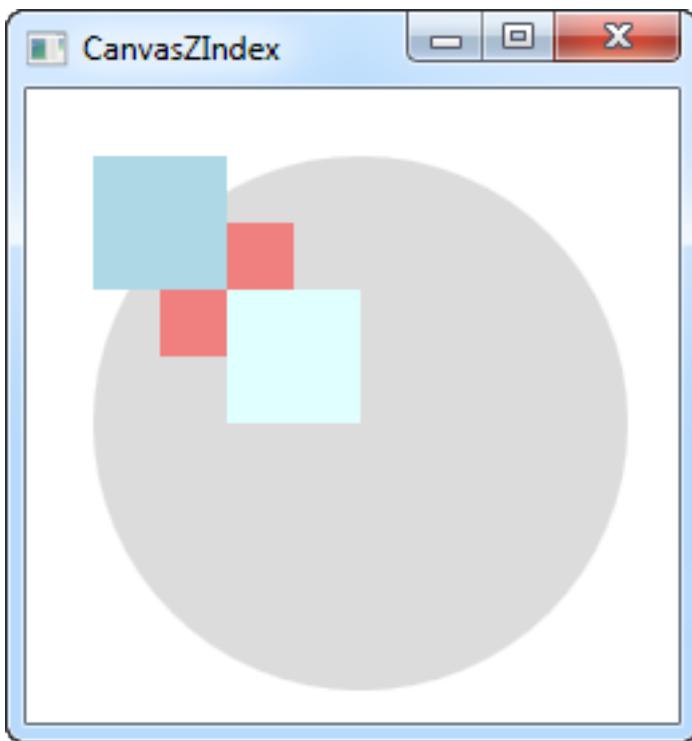


Notice that because each of the rectangles are defined after the circle, they all overlap the circle, and each of them will overlap the previously defined one. Let's try changing that:

```

<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 Panel.ZIndex="2" Fill="Gainsboro" Canvas.Left="25"
        Canvas.Top="25" Width="200" Height="200" />
        <Rectangle Panel.ZIndex="3" Fill="LightBlue" Canvas.Left="25"
        Canvas.Top="25" Width="50" Height="50" />
        <Rectangle Panel.ZIndex="2" Fill="LightCoral" Canvas.Left="50"
        Canvas.Top="50" Width="50" Height="50" />
        <Rectangle Panel.ZIndex="4" Fill="LightCyan" Canvas.Left="75"
        Canvas.Top="75" Width="50" Height="50" />
    </Canvas>
</Window>

```



The default ZIndex value is 0, but we assign a new one to each of the shapes. The rule is that the element with the higher z-index overlaps the ones with the lower values. If two values are identical, the last defined element "wins". As you can see from the screenshot, changing the ZIndex property gives quite another look.

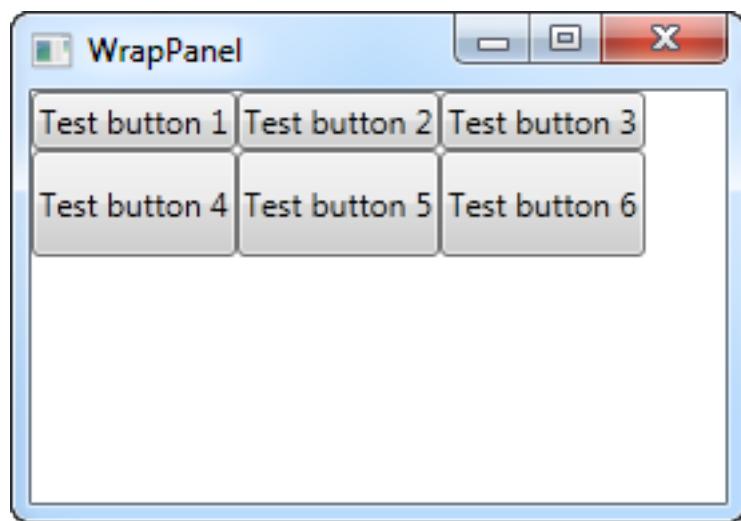
### 1.7.3. The WrapPanel control

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.

In the first example, we'll check out a WrapPanel with the default (Horizontal) orientation:

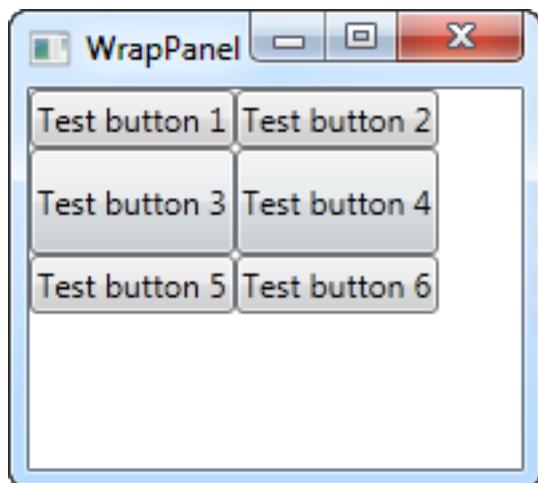
```
<Window x:Class="WpfTutorialSamples.Panels.WrapPanel"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WrapPanel" Height="300" Width="300">
    <WrapPanel>
        <Button>Test button 1</Button>
        <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>
```



Notice how I set a specific height on one of the buttons in the second row. In the resulting screenshot, you will see that this causes the entire row of buttons to have the same height instead of the height required, as seen on the first row. You will also notice that the panel does exactly what the name implies: It wraps the content when it can't fit any more of it in. In this case, the fourth button couldn't fit in on the first line, so it

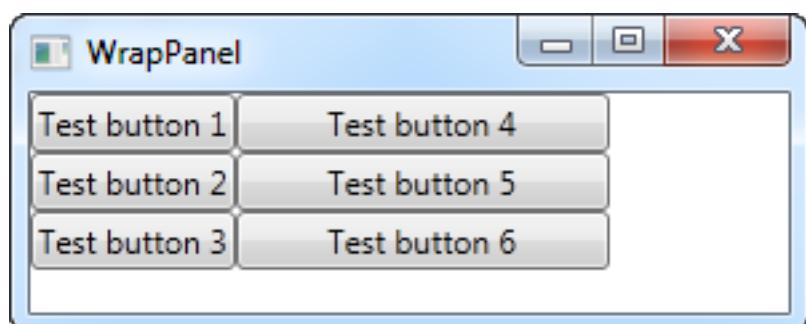
automatically wraps to the next line.

Should you make the window, and thereby the available space, smaller, you will see how the panel immediately adjusts to it:



All of this behavior is also true when you set the Orientation to Vertical. Here's the exact same example as before, but with a Vertical WrapPanel:

```
<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>
```

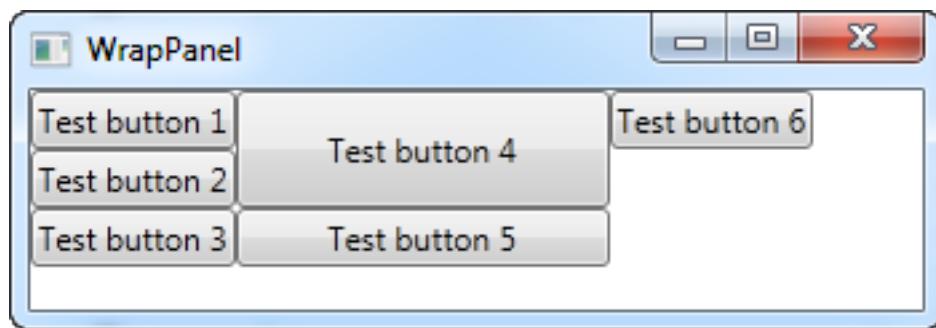


You can see how the buttons go vertical instead of horizontal, before they wrap because they reach the bottom of the window. In this case, I gave a wider width to the fourth button, and you will see that the buttons in the same column also gets the same width, just like we saw with the button height in the Horizontal example.

Please be aware that while the Horizontal WrapPanel will match the height in the same row and the Vertical WrapPanel will match the width in the same column, height is not matched in a Vertical WrapPanel and width is not matched in a Horizontal WrapPanel. Take a look in this example, which is the Vertical WrapPanel but where the fourth button gets a custom width AND height:

```
<Button Width="140" Height="44">Test button 4</Button>
```

It will look like this:

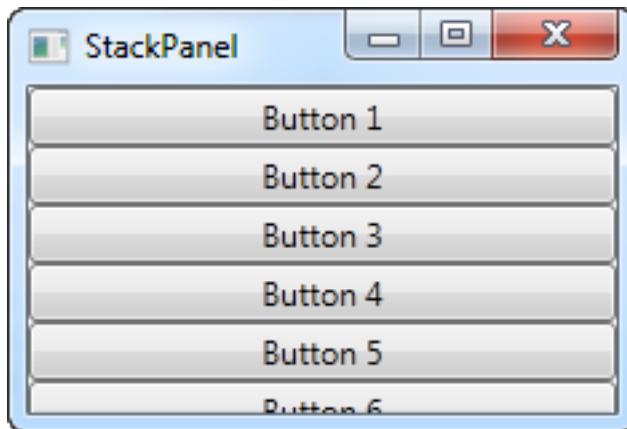


Notice how button 5 only uses the width - it doesn't care about the height, although it causes the sixth button to be pushed to a new column.

## 1.7.4. 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 its 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">
```

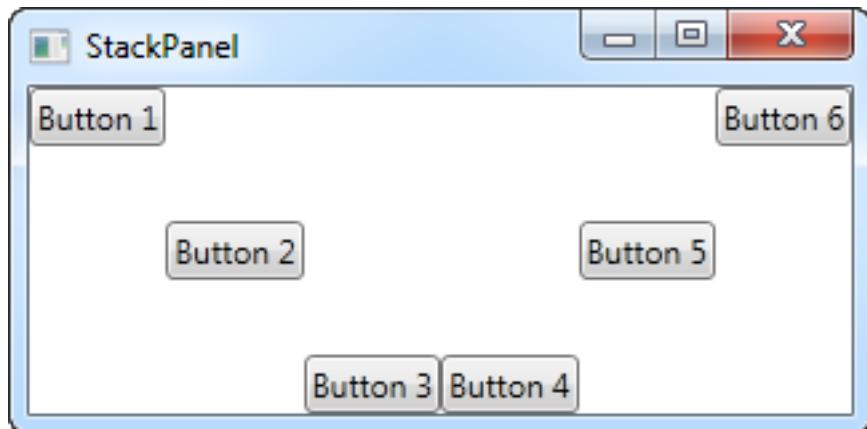


Another thing you will likely notice is that the StackPanel stretches its child control by default. On a vertically aligned StackPanel, like the one in the first example, all child controls get stretched horizontally. On a horizontally aligned StackPanel, all child controls get stretched vertically, 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>
```



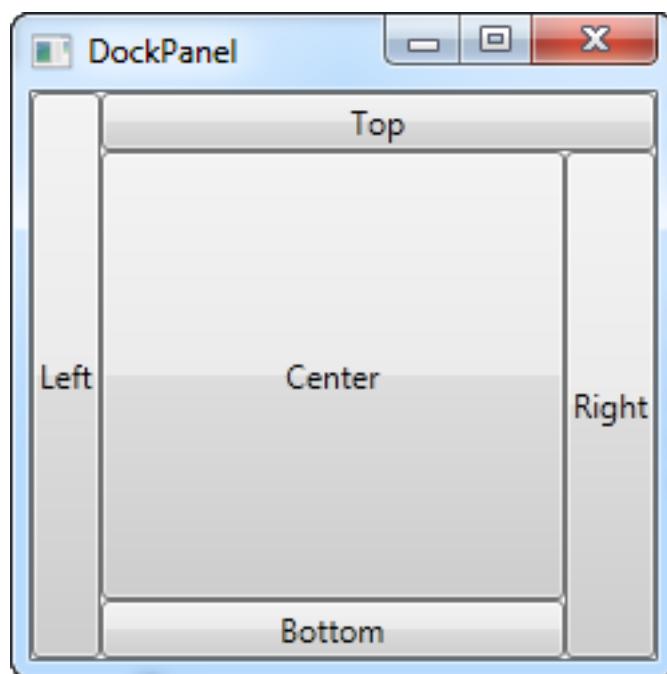
As you can see, the controls still go from top to bottom, but instead of having the same width, each control is aligned to the left, the right or center.

## 1.7.5. 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>
```

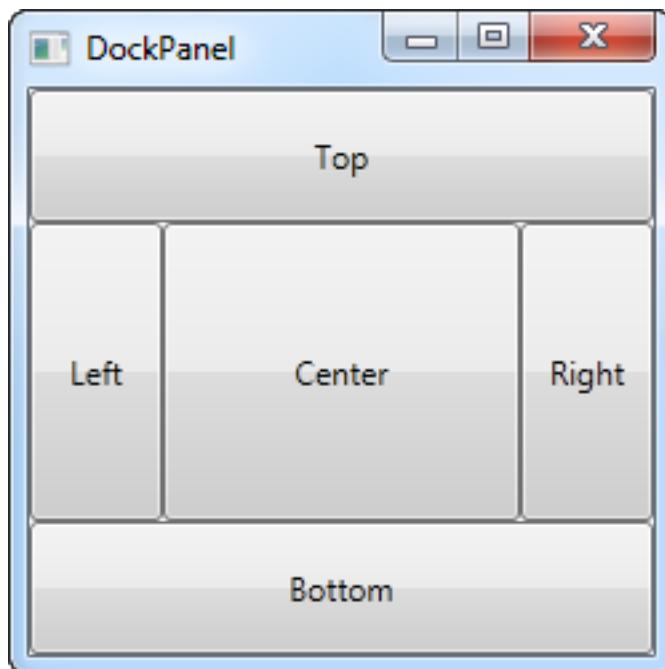


As already mentioned, we don't assign a dock position for the last child, because it automatically centers

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>
```



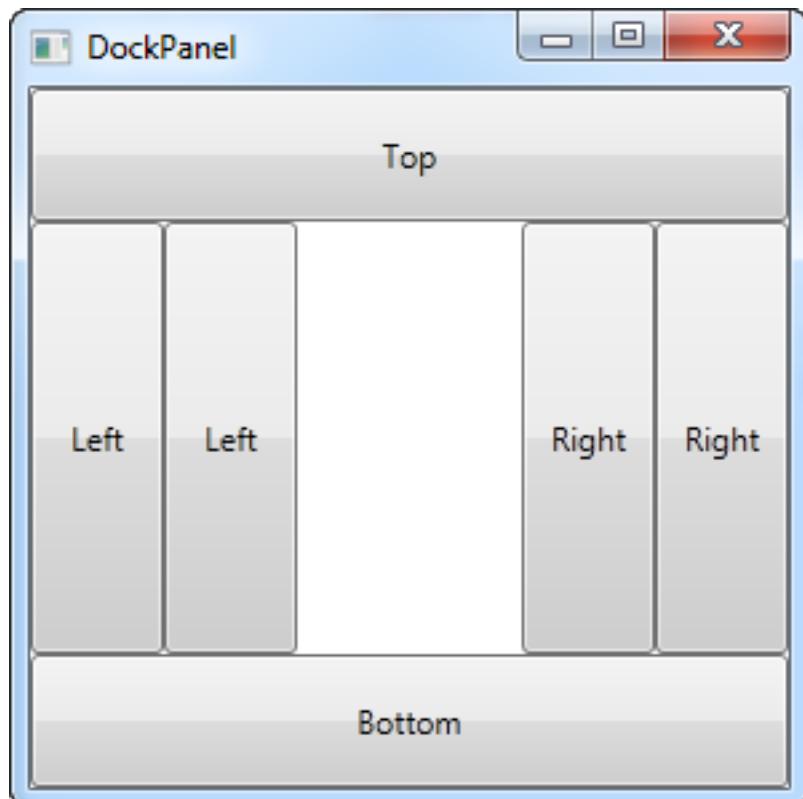
The top and bottom controls now take precedence over the left and right controls, and they're all taking up 50 pixels in either height or width. If you make the window bigger or smaller, you will also see that this static

width/height remains the same no matter what - only the center area increases or decreases in size as you resize the window.

#### 1.7.5.1. 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>
```



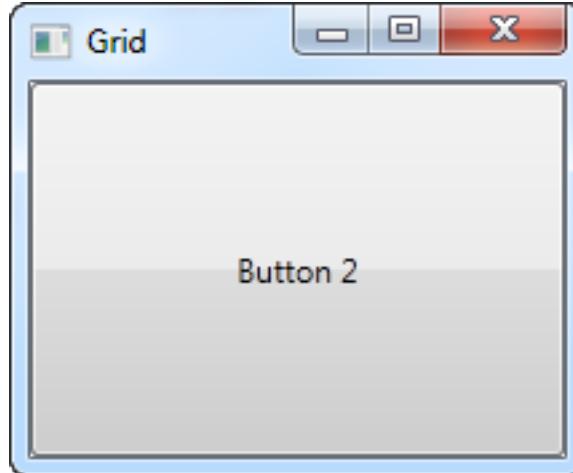
In this example, we dock two controls to the left and two controls to the right, and at the same time, we turn off the LastChildFill property. This leaves us with empty space in the center, which may be preferable in some cases.

## 1.7.6. 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>

```

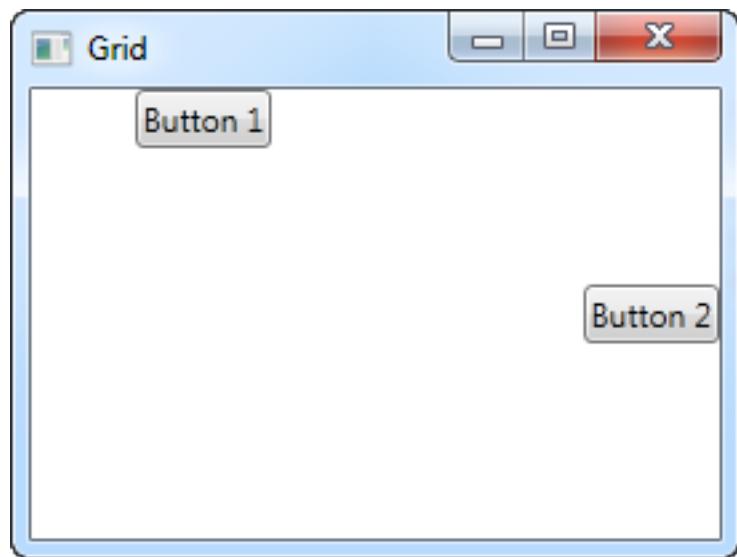


In this example, we have simply divided the available space into two columns, which will share the space equally, using a "star width" (this will be explained later). On the second button, I use a so-called Attached property to place the button in the second column (0 is the first column, 1 is the second 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.

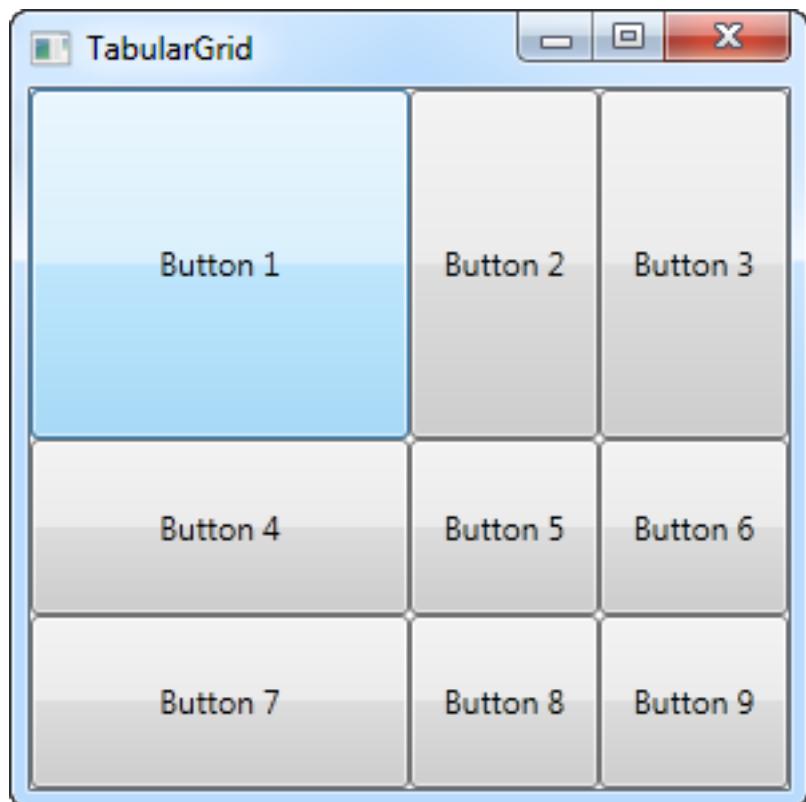
## 1.7.7. The Grid - Rows & columns

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

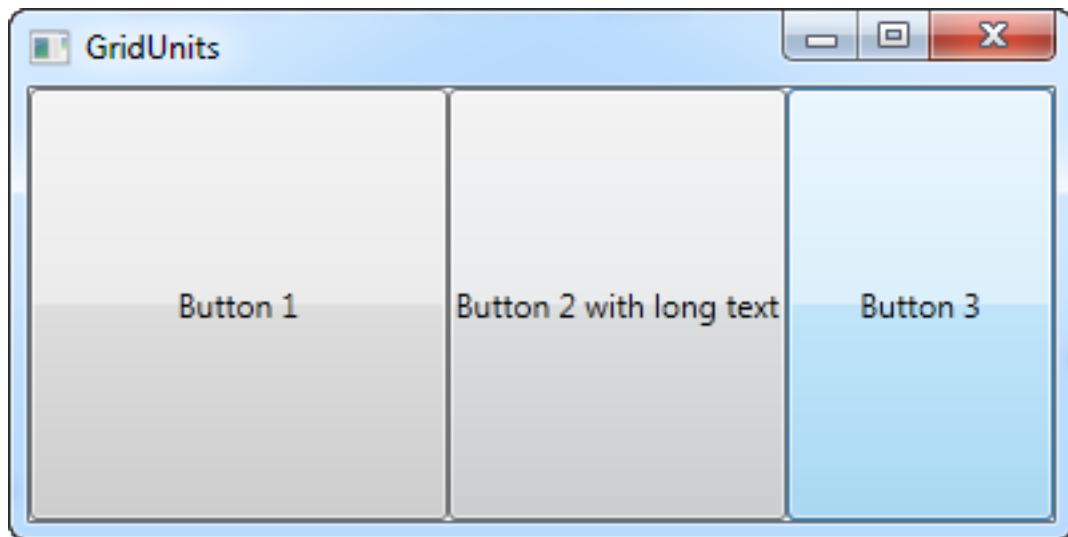


use either the first row or the first column (or both). This is essentially the same as specifying a zero. This saves a bit of typing, but you might prefer to assign them anyway for a better overview - that's totally up to you!

## 1.7.8. 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:

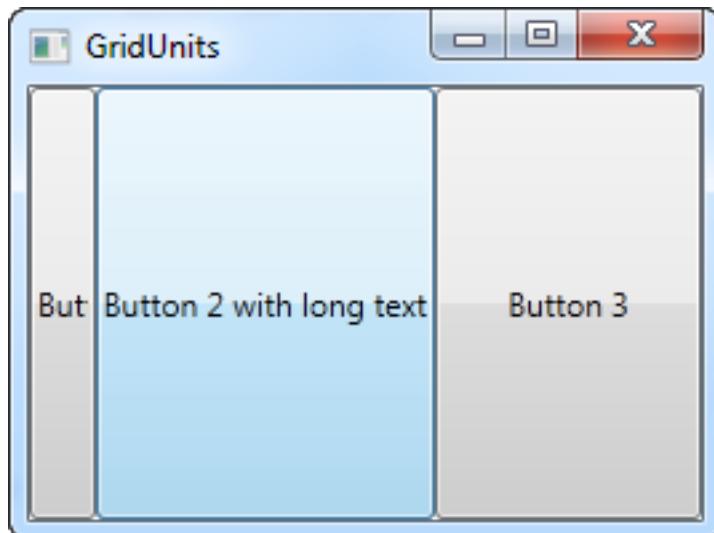
```
<Window x:Class="WpfTutorialSamples.Panels.GridUnits"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="GridUnits" Height="200" Width="400">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition 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.

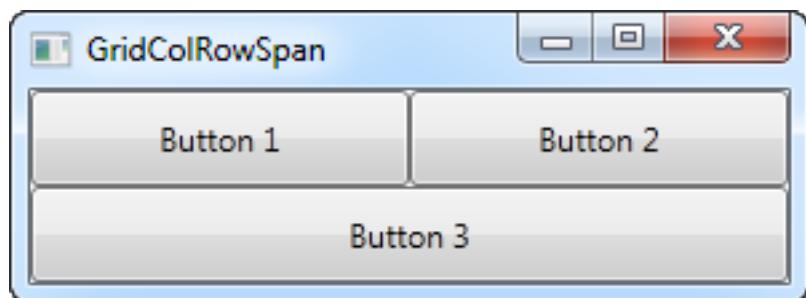
This can be a very useful technique when designing a wide range of dialogs. For instance, consider a simple contact form where the user enters a name, an e-mail address and a comment. The first two fields will usually have a fixed height, while the last one might as well take up as much space as possible, 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.

## 1.7.9. 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.

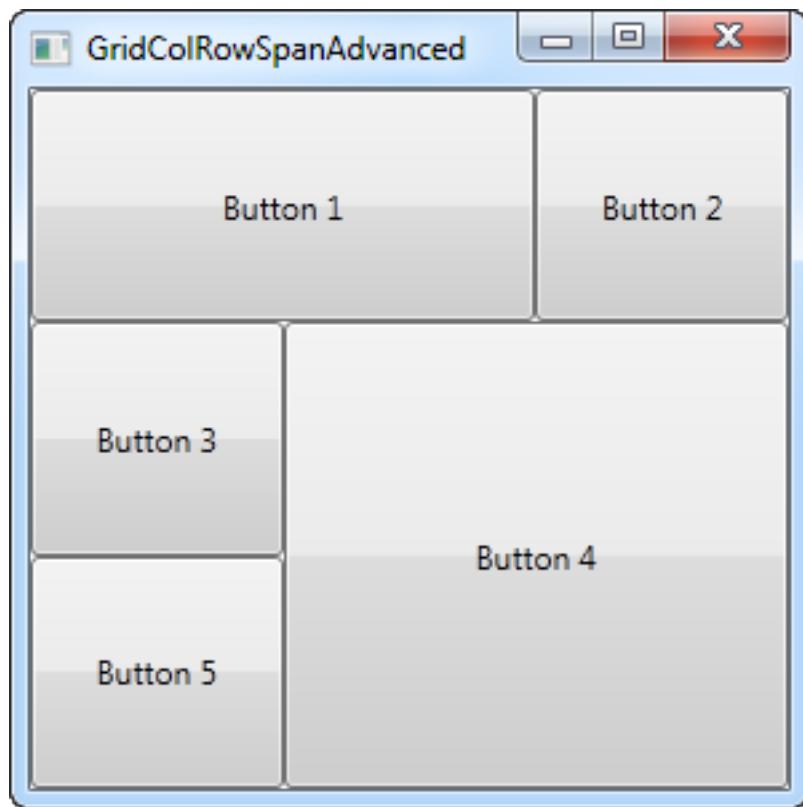
This is all so simple that we could have just used a combination of panels to achieve the same effect, but for just slightly more advanced cases, this is really useful. Let's try something which better shows how

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.



### 1.7.10. 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.

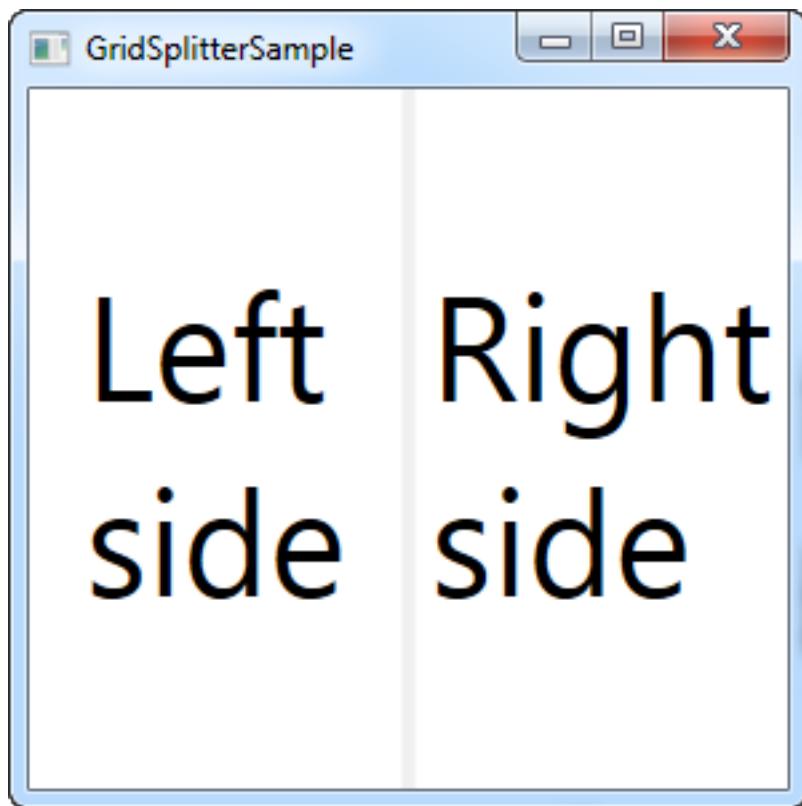
The GridSplitter is used simply by adding it to a column or a row in a Grid, with the proper amount of space for it, e.g. 5 pixels. It will then allow the user to drag it from side to side or up and down, while changing the size of the column or row on each of the sides of it. Here's an example:

```
<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.

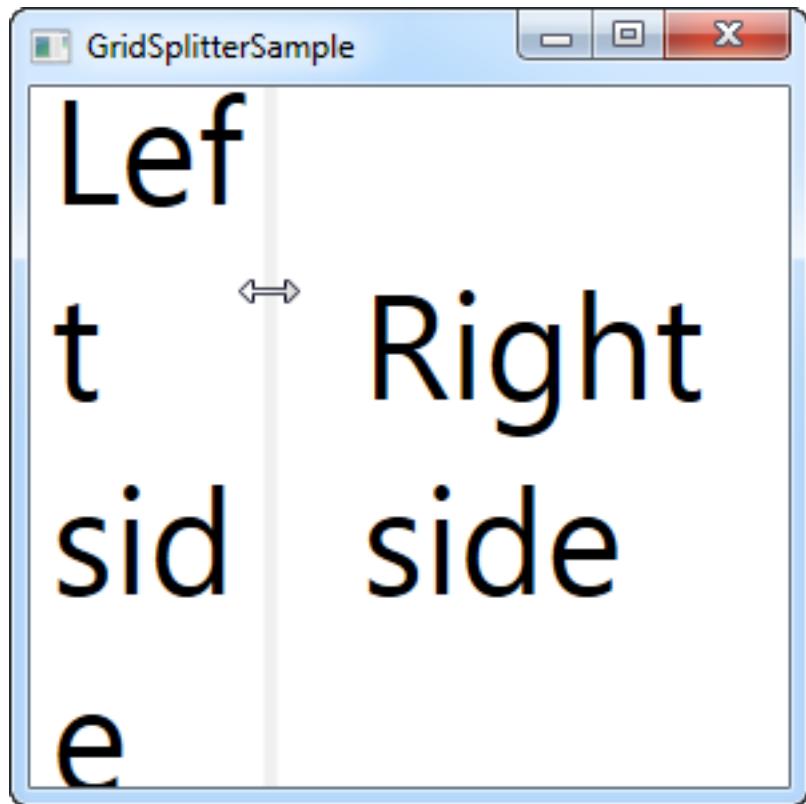
#### 1.7.10.1. Horizontal GridSplitter

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:

```

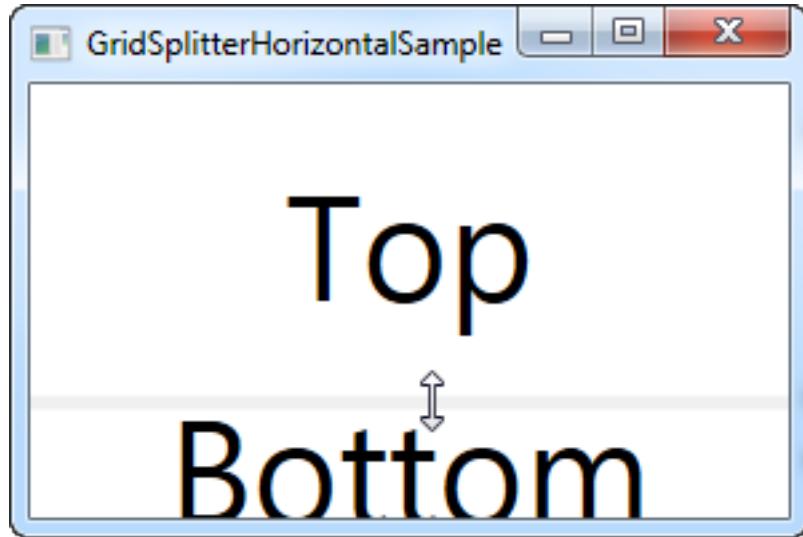
<Window x:Class="WpfTutorialSamples.Panels.GridSplitterHorizontalSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="GridSplitterHorizontalSample" Height="300" Width="300">
    <Grid>

```



```
<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.



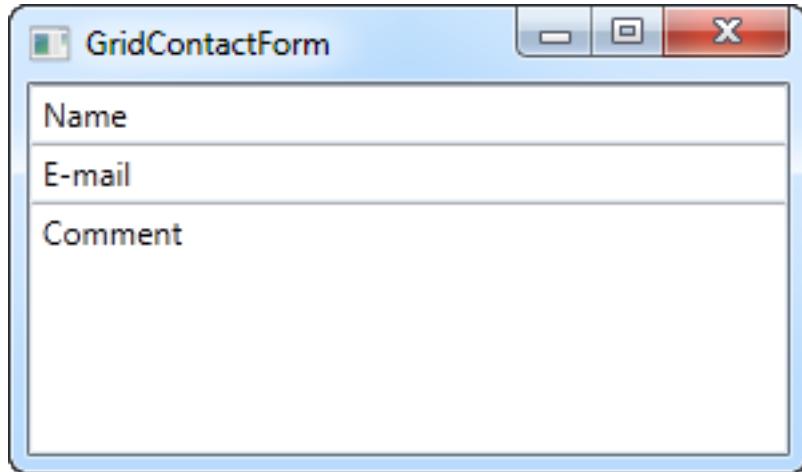
### 1.7.11. Using the Grid: A contact form

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:

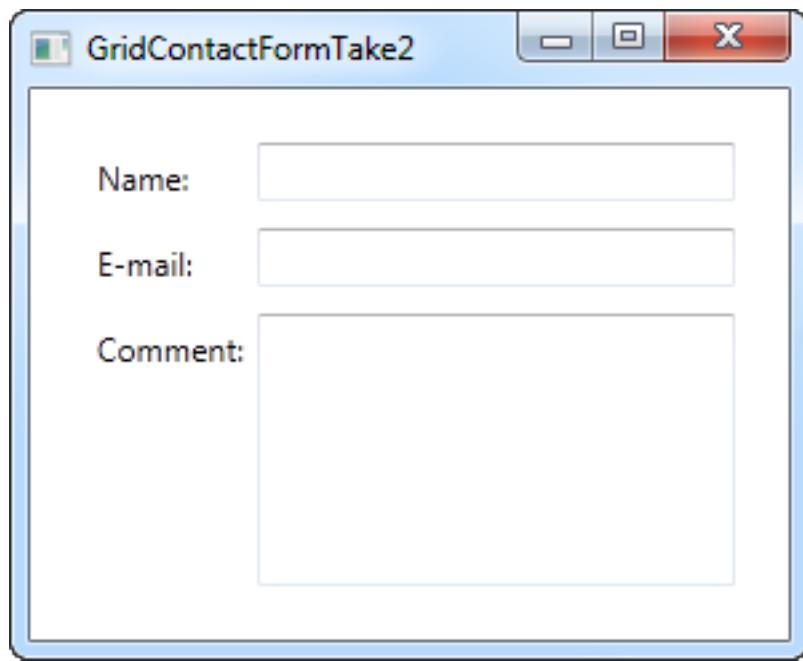
```
<Window x:Class="WpfTutorialSamples.Panels.GridContactForm"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="GridContactForm" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <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.

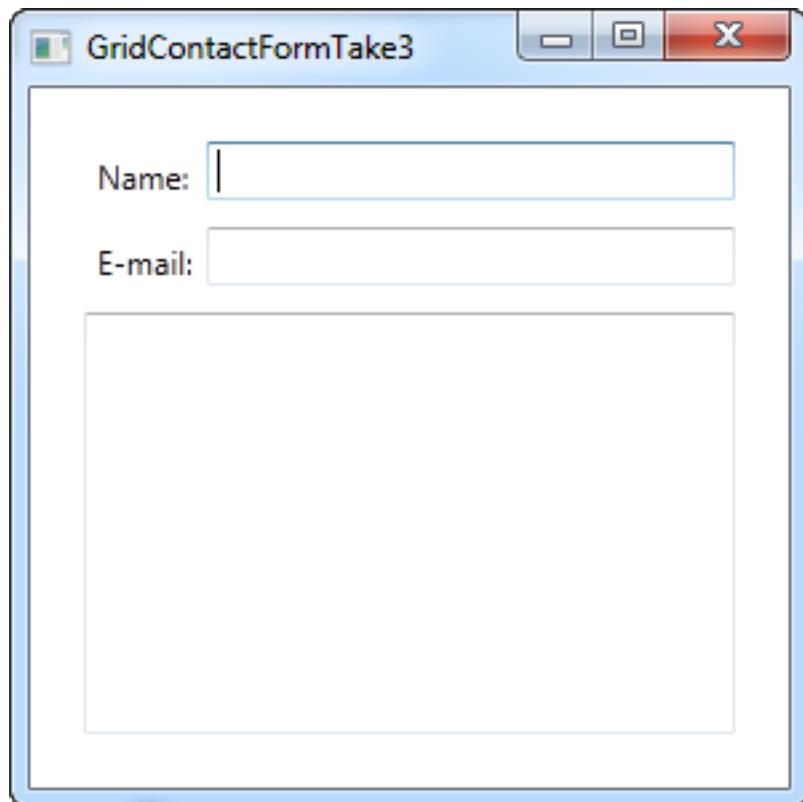
In this very simple example, there are no labels to designate what each of the fields are for. Instead, the explanatory text is inside the TextBox, but this is not generally how a Windows dialog looks. Let's try improving the look and usability a bit:

```
<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.

# 1.8. UserControls & CustomControls

---

## 1.8.1. Introduction

So far in this tutorial, we have only used the built-in controls found in the WPF framework. They will get you a VERY long way, because they are so extremely flexible and can be styled and templated to do almost anything. However, at some point, you will likely benefit from creating your own controls. In other UI frameworks, this can be quite cumbersome, but WPF makes it pretty easy, offering you two ways of accomplishing this task: **UserControls** and **Custom controls**.

### 1.8.1.1. UserControls

A WPF UserControl inherits the UserControl class and acts very much like a WPF Window: You have a XAML file and a Code-behind file. In the XAML file, you can add existing WPF controls to create the look you want and then combine it with code in the Code-behind file, to achieve the functionality you want. WPF will then allow you to embed this collection of functionality in one or several places in your application, allowing you to easily group and re-use functionality across your application(s).

### 1.8.1.2. Custom controls

A Custom control is more low-level than a UserControl. When you create a Custom control, you inherit from an existing class, based on how deep you need to go. In many cases, you can inherit the **Control** class, which other WPF controls inherits from (e.g. the TextBox), but if you need to go even deeper, you can inherit the **FrameworkElement** or even the **UIElement**. The deeper you go, the more control you get and the less functionality is inherited.

The look of the Custom control is usually controlled through styles in a theme file, while the look of the User control will follow the look of the rest of the application. That also highlights one of the major differences between a UserControl and a Custom control: The Custom control can be styled/templated, while a UserControl can't.

### 1.8.1.3. Summary

Creating re-usable controls in WPF is very easy, especially if you take the UserControl approach. In the next article, we'll look into just how easy it is to create a UserControl and then use it in your own application.

## 1.8.2. Creating & using a UserControl

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

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

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

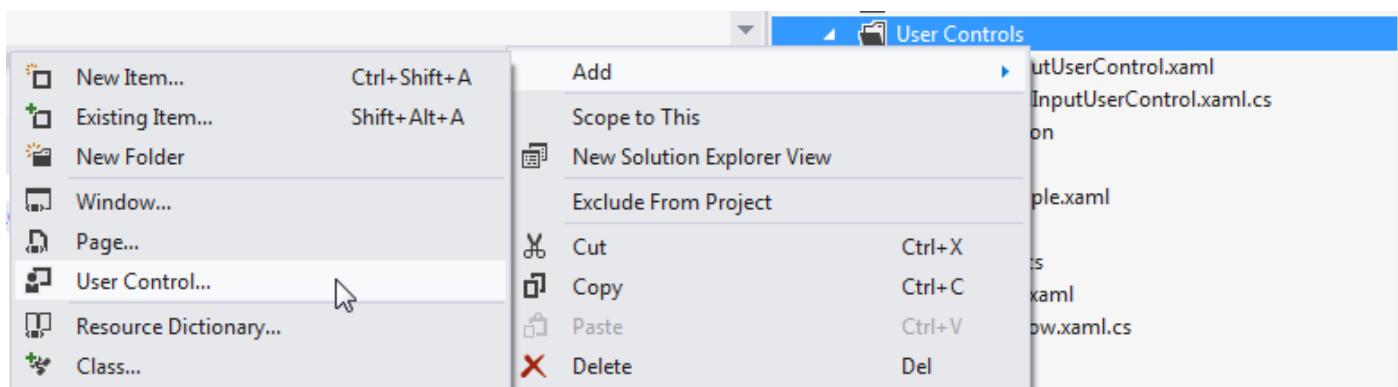
    </Grid>
</UserControl>
```

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

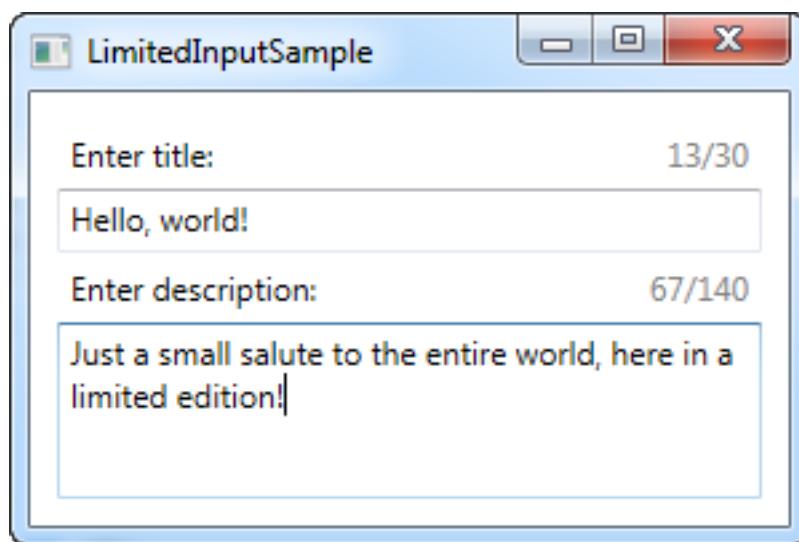
### 1.8.2.1. Creating a User Control

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

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



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



Here's the code for the user control itself:

```
<UserControl x:Class
="WpfTutorialSamples.User_Controls.LimitedInputUserControl"
    xmlns
="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        mc:Ignorable="d"
        d:DesignHeight="300" d:DesignWidth="300">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
```

```

        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <Label Content="{Binding Title}" />
    <Label Grid.Column="1">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding ElementName=txtLimitedInput, Path=Text.Length}" />
            <TextBlock Text="/" />
            <TextBlock Text="{Binding MaxLength}" />
        </StackPanel>
    </Label>
    <TextBox MaxLength="{Binding MaxLength}" Grid.Row="1"
        Grid.ColumnSpan="2" Name="txtLimitedInput"
        ScrollViewer.VerticalScrollBarVisibility="Auto" TextWrapping="Wrap" />
</Grid>
</UserControl>

using System;
using System.Windows.Controls;

namespace WpfTutorialSamples.User_Controls
{
    public partial class LimitedInputUserControl : UserControl
    {
        public LimitedInputUserControl()
        {
            InitializeComponent();
            this.DataContext = this;
        }

        public string Title { get; set; }

        public int MaxLength { get; set; }
    }
}

```

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

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

### 1.8.2.2. Consuming/using the User Control

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

```
xmlns:uc="clr-namespace:WpfTutorialSamples.User_Controls"
```

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

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

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

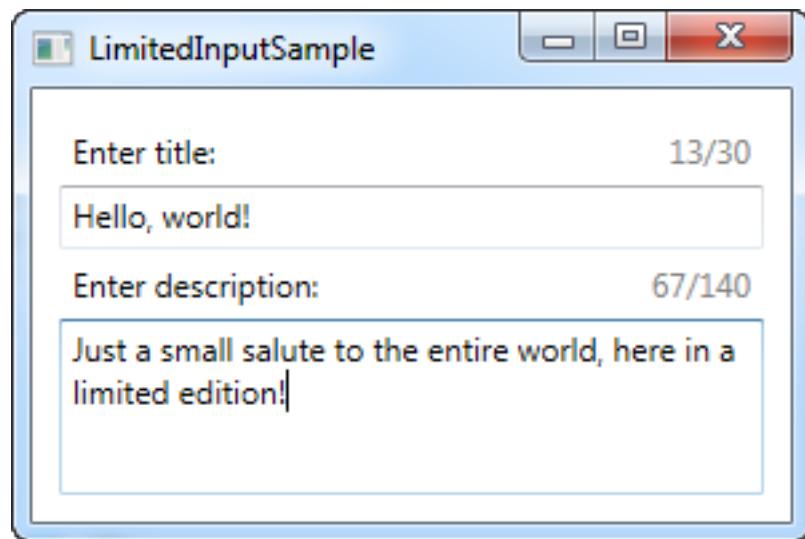
```
<Window x:Class="WpfTutorialSamples.User_Controls.LimitedInputSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:uc="clr-namespace:WpfTutorialSamples.User_Controls"
        Title="LimitedInputSample" Height="200" Width="300">
    <Grid Margin="10">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <uc:LimitedInputUserControl Title="Enter title:" MaxLength="30" Height="50" />
        <uc:LimitedInputUserControl Title="Enter description:" MaxLength="140" Grid.Row="1" />

    </Grid>
</Window>
```

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

this:



#### 1.8.2.3. Summary

Placing commonly used interfaces and functionality in User Controls is highly recommended, and as you can see from the above example, they are very easy to create and use.

# 1.9. Data binding

---

## 1.9.1. Introduction to WPF data binding

[Wikipedia](#) describes the concept of data binding very well:

*Data binding is general technique that binds two data/information sources together and maintains synchronization of data.*

With WPF, Microsoft has put data binding in the front seat and once you start learning WPF, you will realize that it's an important aspect of pretty much everything you do. If you come from the world of WinForms, then the huge focus on data binding might scare you a bit, but once you get used to it, you will likely come to love it, as it makes a lot of things cleaner and easier to maintain.

Data binding in WPF is the preferred way to bring data from your code to the UI layer. Sure, you can set properties on a control manually or you can populate a ListBox by adding items to it from a loop, but the cleanest and purest WPF way is to add a binding between the source and the destination UI element.

### 1.9.1.1. Summary

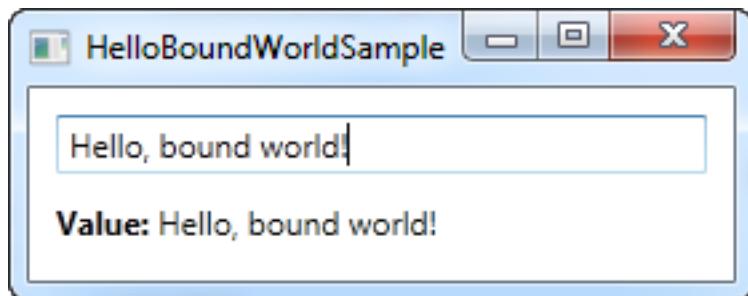
In the next chapter, we'll look into a simple example where data binding is used and after that, we'll talk some more about all the possibilities. The concept of data binding is included pretty early in this tutorial, because it's such an integral part of using WPF, which you will see once you explore the rest of the chapters, where it's used almost all of the time.

However, the more theoretical part of data binding might be too heavy if you just want to get started building a simple WPF application. In that case I suggest that you have a look at the "Hello, bound world!" article to get a glimpse of how data binding works, and then save the rest of the data binding articles for later, when you're ready to get some more theory.

## 1.9.2. Hello, bound world!

Just like we started this tutorial with the classic "Hello, world!" example, we'll show you how easy it is to use data binding in WPF with a "Hello, bound world!" example. Let's jump straight into it and then I'll explain it afterwards:

```
<Window x:Class="WpfTutorialSamples.DataBindings.HelloBoundWorldSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="HelloBoundWorldSample" Height="110" Width="280">
    <StackPanel Margin="10">
        <TextBox Name="txtValue" />
        <WrapPanel Margin="0,10">
            <TextBlock Text="Value: " FontWeight="Bold" />
            <TextBlock Text="{Binding Path=Text,
ElementName=txtValue}" />
        </WrapPanel>
    </StackPanel>
</Window>
```



This simple example shows how we bind the value of the TextBlock to match the Text property of the TextBox. As you can see from the screenshot, the TextBlock is automatically updated when you enter text into the TextBox. In a non-bound world, this would require us to listen to an event on the TextBox and then update the TextBlock each time the text changes, but with data binding, this connection can be established just by using markup.

### 1.9.2.1. The syntax of a Binding

All the magic happens between the curly braces, which in XAML encapsulates a Markup Extension. For data binding, we use the Binding extension, which allows us to describe the binding relationship for the Text property. In its most simple form, a binding can look like this:

{Binding}

This simply returns the current data context (more about that later). This can definitely be useful, but in the

most common situations, you would want to bind a property to another property on the data context. A binding like that would look like this:

```
{Binding Path=NameOfProperty}
```

The Path notes the property that you want to bind to, however, since Path is the default property of a binding, you may leave it out if you want to, like this:

```
{Binding NameOfProperty}
```

You will see many different examples, some of them where Path is explicitly defined and some where it's left out. In the end it's really up to you though.

A binding has many other properties though, one of them being the ElementName which we use in our example. This allows us to connect directly to another UI element as the source. Each property that we set in the binding is separated by a comma:

```
{Binding Path=Text, ElementName=txtValue}
```

#### 1.9.2.2. Summary

This was just a glimpse of all the binding possibilities of WPF. In the next chapters, we'll discover more of them, to show you just how powerful data binding is.

### 1.9.3. Using the DataContext

The `DataContext` property is the default source of your bindings, unless you specifically declare another source, like we did in the previous chapter with the `ElementName` property. It's defined on the `FrameworkElement` class, which most UI controls, including the WPF Window, inherits from. Simply put, it allows you to specify a basis for your bindings

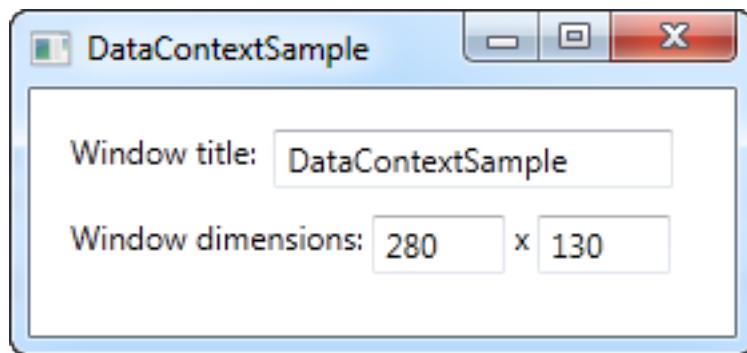
There's no default source for the `DataContext` property (it's simply null from the start), but since a `DataContext` is inherited down through the control hierarchy, you can set a `DataContext` for the Window itself and then use it throughout all of the child controls. Let's try illustrating that with a simple example:

```
<Window x:Class="WpfTutorialSamples.DataBinding.DataContextSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataContextSample" Height="130" Width="280">
    <StackPanel Margin="15">
        <WrapPanel>
            <TextBlock Text="Window title: " />
            <TextBox Text="{Binding Title,
UpdateSourceTrigger=PropertyChanged}" Width="150" />
        </WrapPanel>
        <WrapPanel Margin="0,10,0,0">
            <TextBlock Text="Window dimensions: " />
            <TextBox Text="{Binding Width}" Width="50" />
            <TextBlock Text=" x " />
            <TextBox Text="{Binding Height}" Width="50" />
        </WrapPanel>
    </StackPanel>
</Window>

using System;
using System.Windows;

namespace WpfTutorialSamples.DataBinding
{
    public partial class DataContextSample : Window
    {
        public DataContextSample()
        {
            InitializeComponent();
            this.DataContext = this;
        }
    }
}
```

```
    }  
}  
}
```



The Code-behind for this example only adds one line of interesting code: After the standard InitializeComponent() call, we assign the "this" reference to the DataContext, which basically just tells the Window that we want itself to be the data context.

In the XAML, we use this fact to bind to several of the Window properties, including Title, Width and Height. Since the window has a DataContext, which is passed down to the child controls, we don't have to define a source on each of the bindings - we just use the values as if they were globally available.

Try running the example and resize the window - you will see that the dimension changes are immediately reflected in the textboxes. You can also try writing a different title in the first textbox, but you might be surprised to see that this change is not reflected immediately. Instead, you have to move the focus to another control before the change is applied. Why? Well, that's the subject for the next chapter.

#### 1.9.3.1. Summary

Using the DataContext property is like setting the basis of all bindings down through the hierarchy of controls. This saves you the hassle of manually defining a source for each binding, and once you really start using data bindings, you will definitely appreciate the time and typing saved.

However, this doesn't mean that you have to use the same DataContext for all controls within a Window. Since each control has its own DataContext property, you can easily break the chain of inheritance and override the DataContext with a new value. This allows you to do stuff like having a global DataContext on the window and then a more local and specific DataContext on e.g. a panel holding a separate form or something along those lines.

## 1.9.4. Data binding via Code-behind

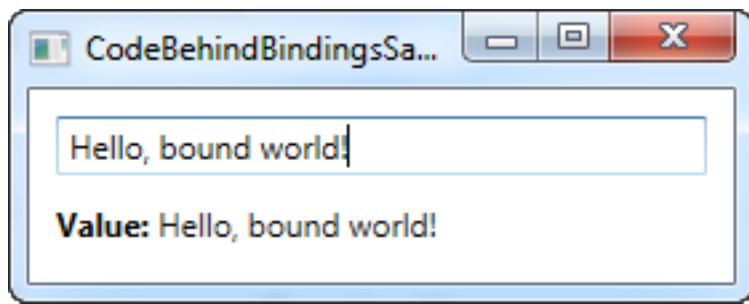
As we saw in the previous data binding examples, defining a binding by using XAML is very easy, but for certain cases, you may want to do it from Code-behind instead. This is pretty easy as well and offers the exact same possibilities as when you're using XAML. Let's try the "Hello, bound world" example, but this time create the required binding from Code-behind:

```
<Window x:Class
        ="WpfTutorialSamples.DataBindings.CodeBehindBindingsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CodeBehindBindingsSample" Height="110" Width="280">
    <StackPanel Margin="10">
        <TextBox Name="txtValue" />
        <WrapPanel Margin="0,10">
            <TextBlock Text="Value: " FontWeight="Bold" />
            <TextBlock Name="lblValue" />
        </WrapPanel>
    </StackPanel>
</Window>

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;

namespace WpfTutorialSamples.DataBindings
{
    public partial class CodeBehindBindingsSample : Window
    {
        public CodeBehindBindingsSample()
        {
            InitializeComponent();

            Binding binding = new Binding("Text");
            binding.Source = txtValue;
            lblValue.SetBinding(TextBlock.TextProperty, binding);
        }
    }
}
```



It works by creating a Binding instance. We specify the path we want directly in the constructor, in this case "Text", since we want to bind to the Text property. We then specify a **Source**, which for this example should be the TextBox control. Now WPF knows that it should use the TextBox as the source control, and that we're specifically looking for the value contained in its Text property.

In the last line, we use the SetBinding method to combine our newly created Binding object with the destination/target control, in this case the TextBlock (lblValue). The **SetBinding()** method takes two parameters, one that tells which dependency property that we want to bind to, and one that holds the binding object that we wish to use.

#### 1.9.4.1. Summary

As you can see, creating bindings in C# code is easy, and perhaps a bit easier to grasp for people new to data bindings, when compared to the syntax used for creating them inline in XAML. Which method you use is up to you though - they both work just fine.

## 1.9.5. The UpdateSourceTrigger property

In the previous article we saw how changes in a TextBox was not immediately sent back to the source. Instead, the source was updated only after focus was lost on the TextBox. This behavior is controlled by a property on the binding called **UpdateSourceTrigger**. It defaults to the value "Default", which basically means that the source is updated based on the property that you bind to. As of writing, all properties except for the Text property, is updated as soon as the property changes (PropertyChanged), while the Text property is updated when focus on the destination element is lost (LostFocus).

Default is, obviously, the default value of the UpdateSourceTrigger. The other options are **PropertyChanged**, **LostFocus** and **Explicit**. The first two has already been described, while the last one simply means that the update has to be pushed manually through to occur, using a call to UpdateSource on the Binding.

To see how all of these options work, I have updated the example from the previous chapter to show you all of them:

```
<Window x:Class="WpfTutorialSamples.DataBindings.DataContextSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataContextSample" Height="130" Width="310">
    <StackPanel Margin="15">
        <WrapPanel>
            <TextBlock Text="Window title: " />
            <TextBox Name="txtWindowTitle" Text="{Binding Title,
UpdateSourceTrigger=Explicit}" Width="150" />
            <Button Name="btnUpdateSource" Click
="btnUpdateSource_Click" Margin="5,0" Padding="5,0">*</Button>
        </WrapPanel>
        <WrapPanel Margin="0,10,0,0">
            <TextBlock Text="Window dimensions: " />
            <TextBox Text="{Binding Width,
UpdateSourceTrigger=LostFocus}" Width="50" />
            <TextBlock Text=" x " />
            <TextBox Text="{Binding Height,
UpdateSourceTrigger=PropertyChanged}" Width="50" />
        </WrapPanel>
    </StackPanel>
</Window>

using System;
using System.Windows;
```

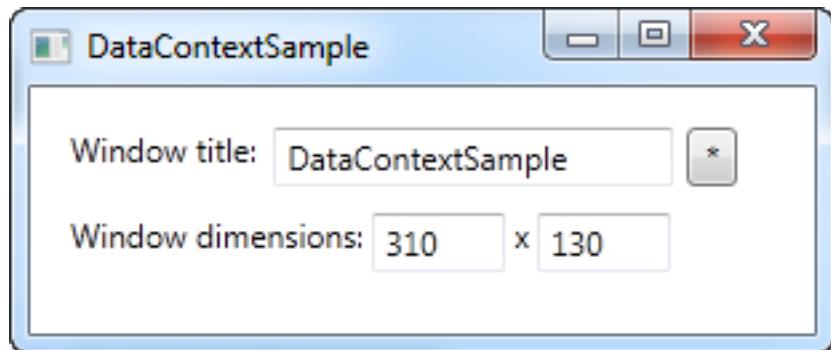
```

using System.Windows.Controls;
using System.Windows.Data;

namespace WpfTutorialSamples.DataBindings
{
    public partial class DataContextSample : Window
    {
        public DataContextSample()
        {
            InitializeComponent();
            this.DataContext = this;
        }

        private void btnUpdateSource_Click(object sender,
RoutedEventArgs e)
        {
            BindingExpression binding =
txtWindowTitle.GetBindingExpression(TextBox.TextProperty);
            binding.UpdateSource();
        }
    }
}

```



As you can see, each of the three textboxes now uses a different **UpdateSourceTrigger**.

The first one is set to **Explicit**, which basically means that the source won't be updated unless you manually do it. For that reason, I have added a button next to the TextBox, which will update the source value on demand. In the Code-behind, you will find the Click handler, where we use a couple of lines of code to get the binding from the destination control and then call the UpdateSource() method on it.

The second TextBox uses the **LostFocus** value, which is actually the default for a Text binding. It means that the source value will be updated each time the destination control loses focus.

The third and last TextBox uses the **PropertyChanged** value, which means that the source value will be updated each time the bound property changes, which it does in this case as soon as the text changes.

Try running the example on your own machine and see how the three textboxes act completely different: The first value doesn't update before you click the button, the second value isn't updated until you leave the TextBox, while the third value updates automatically on each keystroke, text change etc.

#### 1.9.5.1. Summary

The UpdateSourceTrigger property of a binding controls how and when a changed value is sent back to the source. However, since WPF is pretty good at controlling this for you, the default value should suffice for most cases, where you will get the best mix of a constantly updated UI and good performance.

For those situations where you need more control of the process, this property will definitely help though. Just make sure that you don't update the source value more often than you actually need to. If you want the full control, you can use the **Explicit** value and then do the updates manually, but this does take a bit of the fun out of working with data bindings.

## 1.9.6. Responding to changes

So far in this tutorial, we have mostly created bindings between UI elements and existing classes, but in real life applications, you will obviously be binding to your own data objects. This is just as easy, but once you start doing it, you might discover something that disappoints you: Changes are not automatically reflected, like they were in previous examples. As you will learn in this article, you need just a bit of extra work for this to happen, but fortunately, WPF makes this pretty easy.

### 1.9.6.1. Responding to data source changes

There are two different scenarios that you may or may not want to handle when dealing with data source changes: Changes to the list of items and changes in the bound properties in each of the data objects. How to handle them may vary, depending on what you're doing and what you're looking to accomplish, but WPF comes with two very easy solutions that you can use: The **ObservableCollection** and the **INotifyPropertyChanged** interface.

**The following example will show you why we need these two things:**

```
<Window x:Class="WpfTutorialSamples.DataBindings.ChangeNotificationSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ChangeNotificationSample" Height="150" Width="300">
    <DockPanel Margin="10">
        <StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">
            <Button Name="btnAddUser" Click="btnAddUser_Click">Add user</Button>
            <Button Name="btnChangeUser" Click="btnChangeUser_Click" Margin="0,5">Change user</Button>
            <Button Name="btnDeleteUser" Click="btnDeleteUser_Click">Delete user</Button>
        </StackPanel>
        <ListBox Name="lbUsers" DisplayMemberPath="Name" />
    </DockPanel>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples.DataBindings
{
```

```

public partial class ChangeNotificationSample : Window
{
    private List<User> users = new List<User>();

    public ChangeNotificationSample()
    {
        InitializeComponent();

        users.Add(new User() { Name = "John Doe" });
        users.Add(new User() { Name = "Jane Doe" });

        lbUsers.ItemsSource = users;
    }

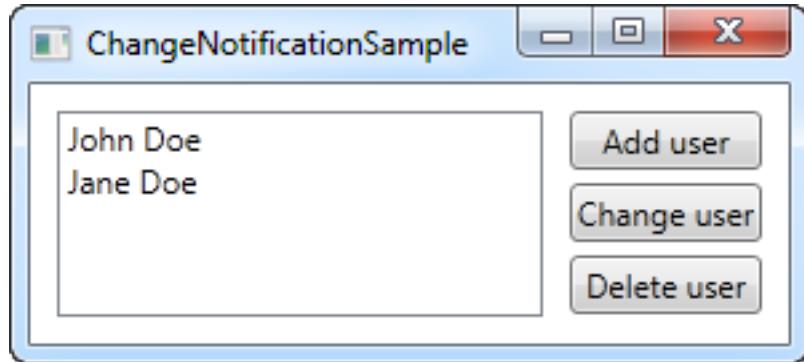
    private void btnAddUser_Click(object sender, RoutedEventArgs e)
    {
        users.Add(new User() { Name = "New user" });
    }

    private void btnChangeUser_Click(object sender,
RoutedEventArgs e)
    {
        if(lbUsers.SelectedItem != null)
            (lbUsers.SelectedItem as User).Name = "Random Name";
    }

    private void btnDeleteUser_Click(object sender,
RoutedEventArgs e)
    {
        if(lbUsers.SelectedItem != null)
            users.Remove(lbUsers.SelectedItem as User);
    }
}

public class User
{
    public string Name { get; set; }
}

```



Try running it for yourself and watch how even though you add something to the list or change the name of one of the users, **nothing in the UI is updated**. The example is pretty simple, with a User class that will keep the name of the user, a ListBox to show them in and some buttons to manipulate both the list and its contents. The ItemsSource of the list is assigned to a quick list of a couple of users that we create in the window constructor. The problem is that none of the buttons seems to work. Let's fix that, in two easy steps.

#### 1.9.6.2. Reflecting changes in the list data source

The first step is to get the UI to respond to changes in the list source (ItemsSource), like when we add or delete a user. What we need is a list that notifies any destinations of changes to its content, and fortunately, WPF provides a type of list that will do just that. It's called ObservableCollection, and you use it much like a regular List<T>, with only a few differences.

In the final example, which you will find below, we have simply replaced the List<User> with an ObservableCollection<User> - that's all it takes! This will make the Add and Delete button work, but it won't do anything for the "Change name" button, because the change will happen on the bound data object itself and not the source list - the second step will handle that scenario though.

#### 1.9.6.3. Reflecting changes in the data objects

The second step is to let our custom User class implement the INotifyPropertyChanged interface. By doing that, our User objects are capable of alerting the UI layer of changes to its properties. This is a bit more cumbersome than just changing the list type, like we did above, but it's still one of the simplest way to accomplish these automatic updates.

#### 1.9.6.4. The final and working example

With the two changes described above, we now have an example that WILL reflect changes in the data source. It looks like this:

```
<Window x:Class="WpfTutorialSamples.DataBindings.ChangeNotificationSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ChangeNotificationSample" Height="135" Width="300">
<DockPanel Margin="10">
    <StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">
        <Button Name="btnAddUser" Click="btnAddUser_Click">Add
user</Button>
        <Button Name="btnChangeUser" Click="btnChangeUser_Click"
Margin="0,5">Change user</Button>
        <Button Name="btnDeleteUser" Click="btnDeleteUser_Click">
Delete user</Button>
    </StackPanel>
    <ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>
</DockPanel>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.ComponentModel;
using System.Collections.ObjectModel;

namespace WpfTutorialSamples.DataBindings
{
    public partial class ChangeNotificationSample : Window
    {
        private ObservableCollection<User> users = new
ObservableCollection<User>();

        public ChangeNotificationSample()
        {
            InitializeComponent();

            users.Add(new User() { Name = "John Doe" });
            users.Add(new User() { Name = "Jane Doe" });

            lbUsers.ItemsSource = users;
        }

        private void btnAddUser_Click(object sender, RoutedEventArgs
e)
    }
}

```

```

{
    users.Add(new User() { Name = "New user" });
}

private void btnChangeUser_Click(object sender,
RoutedEventArgs e)
{
    if(lbUsers.SelectedItem != null)
        (lbUsers.SelectedItem as User).Name = "Random Name";
}

private void btnDeleteUser_Click(object sender,
RoutedEventArgs e)
{
    if(lbUsers.SelectedItem != null)
        users.Remove(lbUsers.SelectedItem as User);
}
}

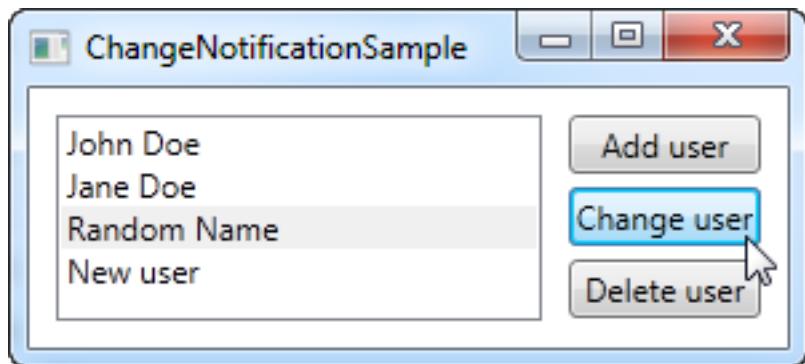
public class User : INotifyPropertyChanged
{
    private string name;
    public string Name {
        get { return this.name; }
        set
        {
            if(this.name != value)
            {
                this.name = value;
                this.NotifyPropertyChanged("Name");
            }
        }
    }
}

public event PropertyChangedEventHandler PropertyChanged;

public void NotifyPropertyChanged(string propName)
{
    if(this.PropertyChanged != null)
        this.PropertyChanged(this, new

```

```
PropertyChangedEventArgs(propName) );  
}  
}  
}
```



#### 1.9.6.5. Summary

As you can see, implementing `INotifyPropertyChanged` is pretty easy, but it does create a bit of extra code on your classes, and adds a bit of extra logic to your properties. This is the price you will have to pay if you want to bind to your own classes and have the changes reflected in the UI immediately. Obviously you only have to call `NotifyPropertyChanged` in the setter's of the properties that you bind to - the rest can remain the way they are.

The `ObservableCollection` on the other hand is very easy to deal with - it simply requires you to use this specific list type in those situations where you want changes to the source list reflected in a binding destination.

## 1.9.7. Value conversion with `IValueConverter`

So far we have used some simple data bindings, where the sending and receiving property was always compatible. However, you will soon run into situations where you want to use a bound value of one type and then present it slightly differently.

### 1.9.7.1. When to use a value converter

Value converters are very frequently used with data bindings. Here are some basic examples:

- You have a numeric value but you want to show zero values in one way and positive numbers in another way
- You want to check a CheckBox based on a value, but the value is a string like "yes" or "no" instead of a Boolean value
- You have a file size in bytes but you wish to show it as bytes, kilobytes, megabytes or gigabytes based on how big it is

These are some of the simple cases, but there are many more. For instance, you may want to check a checkbox based on a Boolean value, but you want it reversed, so that the CheckBox is checked if the value is false and not checked if the value is true. You can even use a converter to generate an image for an `ImageSource`, based on the value, like a green sign for true or a red sign for false - the possibilities are pretty much endless!

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

### 1.9.7.2. Implementing a simple value converter

As mentioned, a WPF value converter needs to implement the `IValueConverter` interface, or alternatively, the `IMultiValueConverter` interface (more about that one later). Both interfaces just requires you to implement two methods: `Convert()` and `ConvertBack()`. As the name implies, these methods will be used to convert the value to the destination format and then back again.

Let's implement a simple converter which takes a string as input and then returns a Boolean value, as well as the other way around. If you're new to WPF, and you likely are since you're reading this tutorial, then you might not know all of the concepts used in the example, but don't worry, they will all be explained after the code listings:

```
<Window x:Class="WpfTutorialSamples.DataBindings.ConverterSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WpfTutorialSamples.DataBindings"
        Title="ConverterSample" Height="140" Width="250">
```

```

<Window.Resources>
    <local:YesNoToBooleanConverter x:Key="YesNoToBooleanConverter" />
</Window.Resources>
<StackPanel Margin="10">
    <TextBox Name="txtValue" />
    <WrapPanel Margin="0,10">
        <TextBlock Text="Current value is: " />
        <TextBlock Text="{Binding ElementName=txtValue, Path=Text, Converter={StaticResource YesNoToBooleanConverter}}"/>
    <TextBlock>
        </WrapPanel>
        <CheckBox IsChecked="{Binding ElementName=txtValue, Path=Text, Converter={StaticResource YesNoToBooleanConverter}} Content="Yes" />
    </StackPanel>
</Window>

using System;
using System.Windows;
using System.Windows.Data;

namespace WpfTutorialSamples.DataBindings
{
    public partial class ConverterSample : Window
    {
        public ConverterSample()
        {
            InitializeComponent();
        }
    }

    public class YesNoToBooleanConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
        {
            switch(value.ToString().ToLower())
            {
                case "yes":
                case "oui":

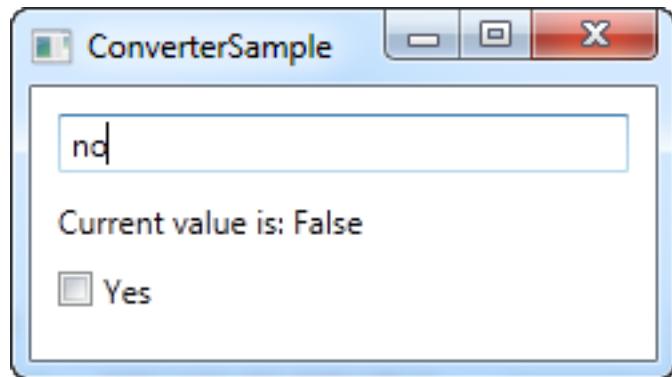
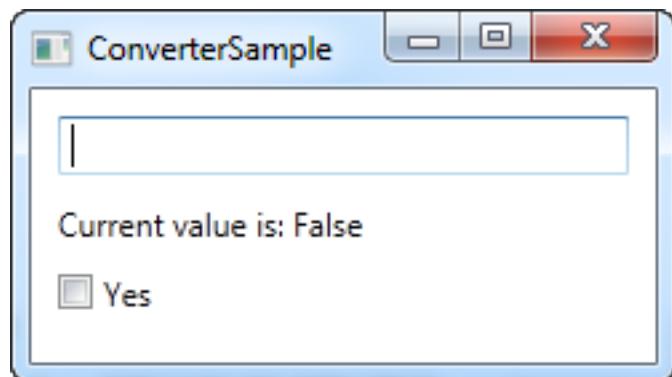
```

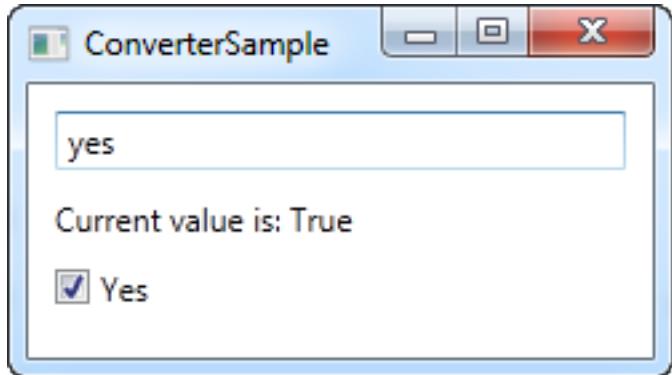
```

        return true;
    case "no":
    case "non":
        return false;
    }
    return false;
}

public object ConvertBack(object value, Type targetType,
object parameter, System.Globalization.CultureInfo culture)
{
    if(value is bool)
    {
        if((bool)value == true)
            return "yes";
        else
            return "no";
    }
    return "no";
}
}
}

```





#### 1.9.7.3. Code-behind

So, let's start from the back and then work our way through the example. We have implemented a converter in the Code-behind file called `YesNoToBooleanConverter`. As advertised, it just implements the two required methods, called `Convert()` and `ConvertBack()`. The `Convert()` method assumes that it receives a string as the input (the `value` parameter) and then converts it to a Boolean true or false value, with a fallback value of false. For fun, I added the possibility to do this conversion from French words as well.

The `ConvertBack()` method obviously does the opposite: It assumes an input value with a Boolean type and then returns the English word "yes" or "no" in return, with a fallback value of "no".

You may wonder about the additional parameters that these two methods take, but they're not needed in this example. We'll use them in one of the next chapters, where they will be explained.

#### 1.9.7.4. XAML

In the XAML part of the program, we start off by declaring an instance of our converter as a resource for the window. We then have a `TextBox`, a couple of `TextBlocks` and a `CheckBox` control and this is where the interesting things are happening: We bind the value of the `TextBox` to the `TextBlock` and the `CheckBox` control and using the `Converter` property and our own converter reference, we juggle the values back and forth between a string and a Boolean value, depending on what's needed.

If you try to run this example, you will be able to change the value in two places: By writing "yes" in the `TextBox` (or any other value, if you want false) or by checking the `CheckBox`. No matter what you do, the change will be reflected in the other control as well as in the `TextBlock`.

#### 1.9.7.5. Summary

This was an example of a simple value converter, made a bit longer than needed for illustrational purposes. In the next chapter we'll look into a more advanced example, but before you go out and write your own converter, you might want to check if WPF already includes one for the purpose. As of writing, there are more than 20 built-in converters that you may take advantage of, but you need to know their name. I found the following list which might come in handy for you:

<http://stackoverflow.com/questions/505397/built-in-wpf-i-valueconverters>



## 1.9.8. The StringFormat property

As we saw in the previous chapters, the way to manipulate the output of a binding before it is shown is typically through the use of a converter. The cool thing about the converters is that they allow you to convert any data type into a completely different data type. However, for more simple usage scenarios, where you just want to change the way a certain value is shown and not necessarily convert it into a different type, the **StringFormat** property might very well be enough.

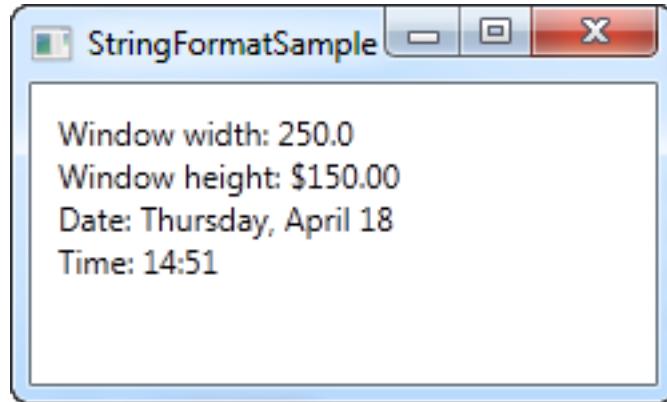
Using the **StringFormat** property of a binding, you lose some of the flexibility you get when using a converter, but in return, it's much simpler to use and doesn't involve the creation of a new class in a new file.

The **StringFormat** property does exactly what the name implies: It formats the output string, simply by calling the `String.Format` method. Sometimes an example says more than a thousand words, so before I hit that word count, let's jump straight into an example:

```
<Window x:Class="WpfTutorialSamples.DataBindings.StringFormatSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="StringFormatSample" Height="150" Width="250"
        Name="wnd">
    <StackPanel Margin="10">
        <TextBlock Text="{Binding ElementName=wnd, Path=ActualWidth,
StringFormat=Window width: {0:#,.0}}"/>
        <TextBlock Text="{Binding ElementName=wnd, Path=ActualHeight,
StringFormat=Window height: {0:C}}"/>
        <TextBlock Text="{Binding Source={x:Static
system:DateTime.Now}, StringFormat=Date: {0:dddd, MMMM dd}}"/>
        <TextBlock Text="{Binding Source={x:Static
system:DateTime.Now}, StringFormat=Time: {0:HH:mm}}"/>
    </StackPanel>
</Window>
```

The first couple of `TextBlock`'s gets their value by binding to the parent `Window` and getting its width and height. Through the **StringFormat** property, the values are formatted. For the width, we specify a custom formatting string and for the height, we ask it to use the currency format, just for fun. The value is saved as a double type, so we can use all the same format specifiers as if we had called `double.ToString()`. You can find a list of them here: <http://msdn.microsoft.com/en-us/library/dwhawy9k.aspx>

Also notice how I can include custom text in the **StringFormat** - this allows you to pre/post-fix the bound



value with text as you please. When referencing the actual value inside the format string, we surround it by a set of curly braces, which includes two values: A reference to the value we want to format (value number 0, which is the first possible value) and the format string, separated by a colon.

For the last two values, we simply bind to the current date (`DateTime.Now`) and the output it first as a date, in a specific format, and then as the time (hours and minutes), again using our own, pre-defined format.

You can read more about `DateTime` formatting here: <http://msdn.microsoft.com/en-us/library/az4se3k1.aspx>

#### 1.9.8.1. Formatting without extra text

Please be aware that if you specify a format string that doesn't include any custom text, which all of the examples above does, then you need to add an extra set of curly braces, when defining it in XAML. The reason is that WPF may otherwise confuse the syntax with the one used for Markup Extensions. Here's an example:

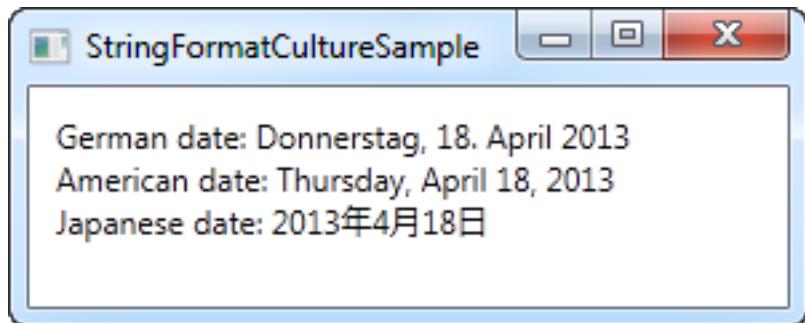
```
<Window x:Class="WpfTutorialSamples.DataBindings.StringFormatSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="StringFormatSample" Height="150" Width="250"
        Name="wnd">
    <WrapPanel Margin="10">
        <TextBlock Text="Width: " />
        <TextBlock Text="{Binding ElementName=wnd, Path=ActualWidth,
StringFormat={}{0:#,##.0}}" />
    </WrapPanel>
</Window>
```

#### 1.9.8.2. Using a specific Culture

If you need to output a bound value in accordance with a specific culture, that's no problem. The `Binding` will use the language specified for the parent element, or you can specify it directly for the binding, using

the ConverterCulture property. Here's an example:

```
<Window x:Class
    ="WpfTutorialSamples.DataBindings.StringFormatCultureSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="StringFormatCultureSample" Height="120" Width="300">
    <StackPanel Margin="10">
        <TextBlock Text="{Binding Source={x:Static
system:DateTime.Now}, ConverterCulture='de-DE', StringFormat=German
date: {0:D}}" />
        <TextBlock Text="{Binding Source={x:Static
system:DateTime.Now}, ConverterCulture='en-US', StringFormat=American
date: {0:D}}" />
        <TextBlock Text="{Binding Source={x:Static
system:DateTime.Now}, ConverterCulture='ja-JP', StringFormat=Japanese
date: {0:D}}" />
    </StackPanel>
</Window>
```



It's pretty simple: By combining the StringFormat property, which uses the D specifier (Long date pattern) and the ConverterCulture property, we can output the bound values in accordance with a specific culture. Pretty nifty!

## 1.9.9. Debugging data bindings

Since data bindings are evaluated at runtime, and no exceptions are thrown when they fail, a bad binding can sometimes be very hard to track down. These problems can occur in several different situations, but a common issue is when you try to bind to a property that doesn't exist, either because you remembered its name wrong or because you simply misspelled it. Here's an example:

```
<Window x:Class
        ="WpfTutorialSamples.DataBindings.DataBindBindingDebuggingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataBindingDebuggingSample" Height="100" Width="200">
    <Grid Margin="10" Name="pnlMain">
        <TextBlock Text="{Binding NonExistingProperty,
ElementName=pnlMain}" />
    </Grid>
</Window>
```

### 1.9.9.1. The Output window

The first place you will want to look is the Visual Studio Output window. It should be at the bottom of your Visual Studio window, or you can activate it by using the [Ctrl+Alt+O] shortcut. There will be loads of output from the debugger, but somewhere you should find a line like this, when running the above example:

*System.Windows.Data Error: 40 : BindingExpression path error: 'NonExistingProperty' property not found on 'object' 'Grid' (Name='pnlMain').*

*BindingExpression:Path=NonExistingProperty; DataItem='Grid' (Name='pnlMain'); target element is 'TextBlock' (Name=""); target property is 'Text' (type 'String')*

This might seem a bit overwhelming, mainly because no linebreaks are used in this long message, but the important part is this:

*'NonExistingProperty' property not found on 'object' "Grid" (Name='pnlMain').*

It tells you that you have tried to use a property called "NonExistingProperty" on an object of the type Grid, with the name pnlMain. That's actually pretty concise and should help you correct the name of the property or bind to the real object, if that's the problem.

### 1.9.9.2. Adjusting the trace level

The above example was easy to fix, because it was clear to WPF what we were trying to do and why it

didn't work. Consider this next example though:

```
<Window x:Class
        ="WpfTutorialSamples.DataBindings.DataBindingsample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataBindingDebuggingSample" Height="100" Width="200">
    <Grid Margin="10">
        <TextBlock Text="{Binding Title}" />
    </Grid>
</Window>
```

I'm trying to bind to the property "Title", but on which object? As stated in the article on data contexts, WPF will use the `DataContext` property on the `TextBlock` here, which may be inherited down the control hierarchy, but in this example, I forgot to assign a data context. This basically means that I'm trying to get a property on a NULL object. WPF will gather that this might be a perfectly valid binding, but that the object just hasn't been initialized yet, and therefore it won't complain about it. If you run this example and look in the **Output** window, you won't see any binding errors.

However, for the cases where this is not the behavior that you're expecting, there is a way to force WPF into telling you about all the binding problems it runs into. It can be done by setting the `TraceLevel` on the `PresentationTraceSources` object, which can be found in the `System.Diagnostics` namespace:

```
<Window x:Class
        ="WpfTutorialSamples.DataBindings.DataBindingsample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:diag="clr-
        namespace:System.Diagnostics;assembly=WindowsBase"
        Title="DataBindingDebuggingSample" Height="100" Width="200">
    <Grid Margin="10">
        <TextBlock Text="{Binding Title,
        diag:PresentationTraceSources.TraceLevel=High}" />
    </Grid>
</Window>
```

Notice that I have added a reference to the `System.Diagnostics` namespace in the top, and then used the `diag:PresentationTraceSources` property on the binding. WPF will now give you loads of information about this specific binding in the **Output** window:

```
System.Windows.Data Warning: 55 : Created BindingExpression  
(hash=2902278) for Binding (hash=52760599)  
System.Windows.Data Warning: 57 : Path: 'Title'  
System.Windows.Data Warning: 59 : BindingExpression (hash=2902278):  
Default mode resolved to OneWay  
System.Windows.Data Warning: 60 : BindingExpression (hash=2902278):  
Default update trigger resolved to PropertyChanged  
System.Windows.Data Warning: 61 : BindingExpression (hash=2902278):  
Attach to System.Windows.Controls.TextBlock.Text (hash=18876224)  
System.Windows.Data Warning: 66 : BindingExpression (hash=2902278):  
Resolving source  
System.Windows.Data Warning: 69 : BindingExpression (hash=2902278):  
Found data context element: TextBlock (hash=18876224) (OK)  
System.Windows.Data Warning: 70 : BindingExpression (hash=2902278):  
DataContext is null  
System.Windows.Data Warning: 64 : BindingExpression (hash=2902278):  
Resolve source deferred  
System.Windows.Data Warning: 66 : BindingExpression (hash=2902278):  
Resolving source  
System.Windows.Data Warning: 69 : BindingExpression (hash=2902278):  
Found data context element: TextBlock (hash=18876224) (OK)  
System.Windows.Data Warning: 70 : BindingExpression (hash=2902278):  
DataContext is null  
System.Windows.Data Warning: 66 : BindingExpression (hash=2902278):  
Resolving source  
System.Windows.Data Warning: 69 : BindingExpression (hash=2902278):  
Found data context element: TextBlock (hash=18876224) (OK)  
System.Windows.Data Warning: 70 : BindingExpression (hash=2902278):  
DataContext is null  
System.Windows.Data Warning: 66 : BindingExpression (hash=2902278):  
Resolving source  
System.Windows.Data Warning: 69 : BindingExpression (hash=2902278):  
Found data context element: TextBlock (hash=18876224) (OK)  
System.Windows.Data Warning: 70 : BindingExpression (hash=2902278):  
DataContext is null  
System.Windows.Data Warning: 66 : BindingExpression (hash=2902278):  
Resolving source (last chance)  
System.Windows.Data Warning: 69 : BindingExpression (hash=2902278):  
Found data context element: TextBlock (hash=18876224) (OK)  
System.Windows.Data Warning: 77 : BindingExpression (hash=2902278):
```

```

Activate with root item <null>
System.Windows.Data Warning: 105 : BindingExpression (hash=2902278):
Item at level 0 is null - no accessor
System.Windows.Data Warning: 79 : BindingExpression (hash=2902278):
TransferValue - got raw value {DependencyProperty.UnsetValue}
System.Windows.Data Warning: 87 : BindingExpression (hash=2902278):
TransferValue - using fallback/default value ''
System.Windows.Data Warning: 88 : BindingExpression (hash=2902278):
TransferValue - using final value ''

```

By reading through the list, you can actually see the entire process that WPF goes through to try to find a proper value for your `TextBlock` control. Several times you will see it being unable to find a proper `DataContext`, and in the end, it uses the default `{DependencyProperty.UnsetValue}` which translates into an empty string.

#### 1.9.9.3. Using the real debugger

The above trick can be great for diagnosing a bad binding, but for some cases, it's easier and more pleasant to work with the real debugger. Bindings doesn't natively support this, since they are being handled deep inside of WPF, but using a Converter, like shown in a previous article, you can actually jump into this process and step through it. You don't really need a Converter that does anything useful, you just need a way into the binding process, and a dummy converter will get you there:

```

<Window x:Class
        ="WpfTutorialSamples.DataBindings.DataBindingsDebuggingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:self="clr-namespace:WpfTutorialSamples.DataBindings"
        Title="DataBindingDebuggingSample" Name="wnd" Height="100"
        Width="200">
    <Window.Resources>
        <self:DebugDummyConverter x:Key="DebugDummyConverter" />
    </Window.Resources>
    <Grid Margin="10">
        <TextBlock Text="{Binding Title, ElementName=wnd,
        Converter={StaticResource DebugDummyConverter}}" />
    </Grid>
</Window>

using System;
using System.Windows;

```

```

using System.Windows.Data;
using System.Diagnostics;

namespace WpfTutorialSamples.DataBindings
{
    public partial class DataBindingDebuggingSample : Window
    {
        public DataBindingDebuggingSample()
        {
            InitializeComponent();
        }
    }

    public class DebugDummyConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
        {
            Debugger.Break();
            return value;
        }

        public object ConvertBack(object value, Type targetType,
object parameter, System.Globalization.CultureInfo culture)
        {
            Debugger.Break();
            return value;
        }
    }
}

```

In the Code-behind file, we define a DebugDummyConverter. In the Convert() and ConvertBack() methods, we call Debugger.Break(), which has the same effect as setting a breakpoint in Visual Studio, and then return the value that was given to us untouched.

In the markup, we add a reference to our converter in the window resources and then we use it in our binding. In a real world application, you should define the converter in a file of its own and then add the reference to it in App.xaml, so that you may use it all over the application without having to create a new reference to it in each window, but for this example, the above should do just fine.

If you run the example, you will see that the debugger breaks as soon as WPF tries to fetch the value for the title of the window. You can now inspect the values given to the Convert() method, or even change

them before proceeding, using the standard debugging capabilities of Visual Studio.

If the debugger never breaks, it means that the converter is not used. This usually indicates that you have an invalid binding expression, which can be diagnosed and fixed using the methods described in the start of this article. The dummy-converter trick is only for testing valid binding expressions.

# 1.10. Commands

---

## 1.10.1. Introduction to WPF Commands

In a previous chapter of this tutorial, we talked about how to handle events, e.g. when the user clicks on a button or a menu item. In a modern user interface, it's typical for a function to be reachable from several places though, invoked by different user actions.

For instance, if you have a typical interface with a main menu and a set of toolbars, an action like New or Open might be available in the menu, on the toolbar, in a context menu (e.g. when right clicking in the main application area) and from a keyboard shortcut like Ctrl+N and Ctrl+O.

Each of these actions needs to perform what is typically the exact same piece of code, so in a WinForms application, you would have to define an event for each of them and then call a common function. With the above example, that would lead to at least three event handlers and some code to handle the keyboard shortcut. Not an ideal situation.

### 1.10.1.1. Commands

With WPF, Microsoft is trying to remedy that with a concept called commands. It allows you to define actions in one place and then refer to them from all your user interface controls like menu items, toolbar buttons and so on. WPF will also listen for keyboard shortcuts and pass them along to the proper command, if any, making it the ideal way to offer keyboard shortcuts in an application.

Commands also solve another hassle when dealing with multiple entrances to the same function. In a WinForms application, you would be responsible for writing code that could disable user interface elements when the action was not available. For instance, if your application was able to use a clipboard command like Cut, but only when text was selected, you would have to manually enable and disable the main menu item, the toolbar button and the context menu item each time text selection changed.

With WPF commands, this is centralized. With one method you decide whether or not a given command can be executed, and then WPF toggles all the subscribing interface elements on or off automatically. This makes it so much easier to create a responsive and dynamic application!

### 1.10.1.2. Command bindings

Commands don't actually do anything by themselves. At the root, they consist of the ICommand interface, which only defines an event and two methods: Execute() and CanExecute(). The first one is for performing the actual action, while the second one is for determining whether the action is currently available. To perform the actual action of the command, you need a link between the command and your code and this is where the CommandBinding comes into play.

A CommandBinding is usually defined on a Window or a UserControl, and holds a reference to the Command that it handles, as well as the actual event handlers for dealing with the Execute() and CanExecute() events of the Command.

### 1.10.1.3. Pre-defined commands

You can of course implement your own commands, which we'll look into in one of the next chapters, but to make it easier for you, the WPF team has defined over 100 commonly used commands that you can use. They have been divided into 5 categories, called ApplicationCommands, NavigationCommands, MediaCommands, EditingCommands and ComponentCommands. Especially ApplicationCommands contains commands for a lot of very frequently used actions like New, Open, Save and Cut, Copy and Paste.

### 1.10.1.4. Summary

Commands help you to respond to a common action from several different sources, using a single event handler. It also makes it a lot easier to enable and disable user interface elements based on the current availability and state. This was all theory, but in the next chapters we'll discuss how commands are used and how you define your own custom commands.

## 1.10.2. Using WPF commands

In the previous article, we discussed a lot of theory about what commands are and how they work. In this chapter, we'll look into how you actually use commands, by assigning them to user interface elements and creating command bindings that links it all together.

We'll start off with a very simple example:

```
<Window x:Class="WpfTutorialSamples.Commands.UsingCommandsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="UsingCommandsSample" Height="100" Width="200">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.New" Executed
        ="NewCommand_Executed" CanExecute="NewCommand_CanExecute" />
    </Window.CommandBindings>

    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" >
        <Button Command="ApplicationCommands.New">New</Button>
    </StackPanel>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;

namespace WpfTutorialSamples.Commands
{
    public partial class UsingCommandsSample : Window
    {
        public UsingCommandsSample()
        {
            InitializeComponent();
        }

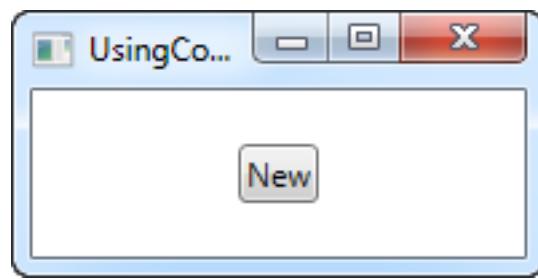
        private void NewCommand_CanExecute(object sender,
        CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
```

```

        }

    private void NewCommand_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    MessageBox.Show("The New command was invoked");
}
}
}

```



We define a command binding on the Window, by adding it to its CommandBindings collection. We specify that Command that we wish to use (the New command from the ApplicationCommands), as well as two event handlers. The visual interface consists of a single button, which we attach the command to using the **Command** property.

In Code-behind, we handle the two events. The **CanExecute** handler, which WPF will call when the application is idle to see if the specific command is currently available, is very simple for this example, as we want this particular command to be available all the time. This is done by setting the **CanExecute** property of the event arguments to true.

The **Executed** handler simply shows a message box when the command is invoked. If you run the sample and press the button, you will see this message. A thing to notice is that this command has a default keyboard shortcut defined, which you get as an added bonus. Instead of clicking the button, you can try to press Ctrl+N on your keyboard - the result is the same.

#### 1.10.2.1. Using the CanExecute method

In the first example, we implemented a CanExecute event that simply returned true, so that the button would be available all the time. However, this is of course not true for all buttons - in many cases, you want the button to be enabled or disabled depending on some sort of state in your application.

A very common example of this is the toggling of buttons for using the Windows Clipboard, where you want the Cut and Copy buttons to be enabled only when text is selected, and the Paste button to only be enabled when text is present in the clipboard. This is exactly what we'll accomplish in this example:

```
<Window x:Class="WpfTutorialSamples.Commands.CommandCanExecuteSample"
```

```

    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CommandCanExecuteSample" Height="200" Width="250">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Cut" CanExecute
    ="CutCommand_CanExecute" Executed="CutCommand_Executed" />
        <CommandBinding Command="ApplicationCommands.Paste" CanExecute
    ="PasteCommand_CanExecute" Executed="PasteCommand_Executed" />
    </Window.CommandBindings>
    <DockPanel>
        <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut" Width="60">_Cut
</Button>
            <Button Command="ApplicationCommands.Paste" Width="60"
Margin="3,0">_Paste</Button>
        </WrapPanel>
        <TextBox AcceptsReturn="True" Name="txtEditor" />
    </DockPanel>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;

namespace WpfTutorialSamples.Commands
{
    public partial class CommandCanExecuteSample : Window
    {
        public CommandCanExecuteSample()
        {
            InitializeComponent();
        }

        private void CutCommand_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = (txtEditor != null) &&
(txtEditor.SelectionLength > 0);
        }
    }
}

```

```

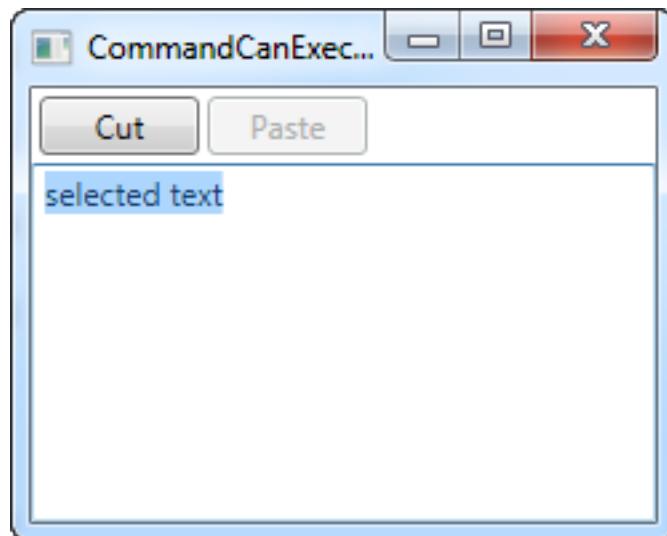
    }

    private void CutCommand_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    txtEditor.Cut();
}

    private void PasteCommand_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
    e.CanExecute = Clipboard.ContainsText();
}

    private void PasteCommand_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    txtEditor.Paste();
}
}

```



So, we have this very simple interface with a couple of buttons and a TextBox control. The first button will cut to the clipboard and the second one will paste from it.

In Code-behind, we have two events for each button: One that performs the actual action, which name ends with `_Executed`, and then the `CanExecute` events. In each of them, you will see that I apply some logic to decide whether or not the action can be executed and then assign it to the return value `CanExecute` on the EventArgs.

The cool thing about this is that you don't have to call these methods to have your buttons updated - WPF does it automatically when the application has an idle moment, making sure that your interface remains updated all the time.

### 1.10.2.2. Default command behavior and CommandTarget

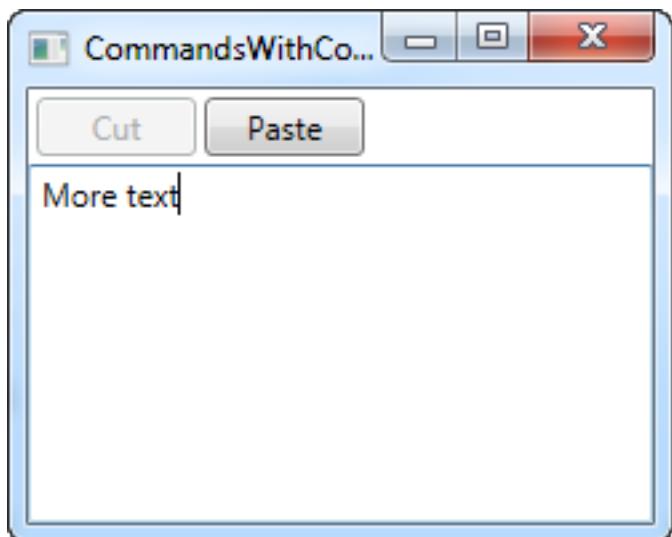
As we saw in the previous example, handling a set of commands can lead to quite a bit of code, with a lot of being method declarations and very standard logic. That's probably why the WPF team decided to handle some of it for you. In fact, we could have avoided all of the Code-behind in the previous example, because a WPF TextBox can automatically handle common commands like Cut, Copy, Paste, Undo and Redo.

WPF does this by handling the Executed and CanExecute events for you, when a text input control like the TextBox has focus. You are free to override these events, which is basically what we did in the previous example, but if you just want the basic behavior, you can let WPF connect the commands and the TextBox control and do the work for you. Just see how much simpler this example is:

```
<Window x:Class
        ="WpfTutorialSamples.Commands.CommandsWithCommandTargetSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CommandsWithCommandTargetSample" Height="200" Width
        ="250">
    <DockPanel>
        <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut" CommandTarget=
{Binding ElementName=txtEditor}" Width="60">_Cut</Button>
            <Button Command="ApplicationCommands.Paste" CommandTarget
        ="{Binding ElementName=txtEditor}" Width="60" Margin="3,0">_Paste</
        Button>
        </WrapPanel>
        <TextBox AcceptsReturn="True" Name="txtEditor" />
    </DockPanel>
</Window>
```

No Code-behind code needed for this example - WPF deals with all of it for us, but only because we want to use these specific commands for this specific control. The TextBox does the work for us.

Notice how I use the **CommandTarget** properties on the buttons, to bind the commands to our TextBox control. This is required in this particular example, because the WrapPanel doesn't handle focus the same way e.g. a Toolbar or a Menu would, but it also makes pretty good sense to give the commands a target.



### 1.10.2.3. Summary

Dealing with commands is pretty straight forward, but does involve a bit extra markup and code. The reward is especially obvious when you need to invoke the same action from multiple places though, or when you use built-in commands that WPF can handle completely for you, as we saw in the last example.

### 1.10.3. Implementing a custom WPF Command

In the previous chapter, we looked at various ways of using commands already defined in WPF, but of course, you can implement your own commands as well. It's pretty simple, and once you've done it, you can use your own commands just like the ones defined in WPF.

The easiest way to start implementing your own commands is to have a *static* class that will contain them. Each command is then added to this class as static fields, allowing you to use them in your application. Since WPF, for some strange reason, doesn't implement an Exit/Quit command, I decided to implement one for our custom commands example. It looks like this:

```
<Window x:Class="WpfTutorialSamples.Commands.CustomCommandSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:self="clr-namespace:WpfTutorialSamples.Commands"
        Title="CustomCommandSample" Height="150" Width="200">
    <Window.CommandBindings>
        <CommandBinding Command="self:CustomCommands.Exit" CanExecute
        ="ExitCommand_CanExecute" Executed="ExitCommand_Executed" />
    </Window.CommandBindings>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Menu>
            <MenuItem Header="File">
                <MenuItem Command="self:CustomCommands.Exit" />
            </MenuItem>
        </Menu>
        <StackPanel Grid.Row="1" HorizontalAlignment="Center"
        VerticalAlignment="Center">
            <Button Command="self:CustomCommands.Exit">Exit</Button>
        </StackPanel>
    </Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;
```

```

namespace WpfTutorialSamples.Commands
{
    public partial class CustomCommandSample : Window
    {
        public CustomCommandSample()
        {
            InitializeComponent();
        }

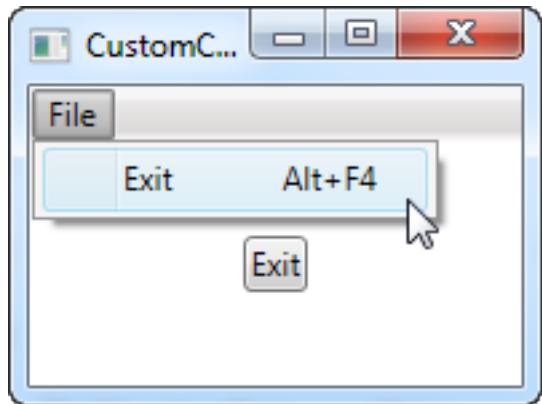
        private void ExitCommand_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }

        private void ExitCommand_Executed(object sender,
ExecutedRoutedEventArgs e)
        {
            Application.Current.Shutdown();
        }
    }

    public static class CustomCommands
    {
        public static readonly RoutedUICommand Exit = new
RoutedUICommand
        (
            "Exit",
            "Exit",
            typeof(CustomCommands),
            new InputGestureCollection()
            {
                new KeyGesture(Key.F4, ModifierKeys.Alt)
            }
        );

        //Define more commands here, just like the one above
    }
}

```



In the markup, I've defined a very simple interface with a menu and a button, both of them using our new, custom Exit command. This command is defined in Code-behind, in our own **CustomCommands** class, and then referenced in the CommandBindings collection of the window, where we assign the events that it should use to execute/check if it's allowed to execute.

All of this is just like the examples in the previous chapter, except for the fact that we're referencing the command from our own code (using the "self" namespace defined in the top) instead of a built-in command.

In Code-behind, we respond to the two events for our command: One event just allows the command to execute all the time, since that's usually true for an exit/quit command, and the other one calls the **Shutdown** method that will terminate our application. All very simple.

As already explained, we implement our Exit command as a field on a static CustomCommands class. There are several ways of defining and assigning properties on the commands, but I've chosen the more compact approach (it would be even more compact if placed on the same line, but I've added line breaks here for readability) where I assign all of it through the constructor. The parameters are the text/label of the command, the name of the command, the owner type and then an InputGestureCollection, allowing me to define a default shortcut for the command (Alt+F4).

#### 1.10.3.1. Summary

Implementing custom WPF commands is almost as easy as consuming the built-in commands, and it allows you to use commands for every purpose in your application. This makes it very easy to re-use actions in several places, as shown in the example of this chapter.

## 1.11. Dialogs

### 1.11.1. The MessageBox

WPF offers several dialogs for your application to utilize, but the simplest one is definitely the MessageBox. Its sole purpose is to show a message to the user, and then offer one or several ways for the user to respond to the message.

The MessageBox is used by calling the static Show() method, which can take a range of different parameters, to be able to look and behave the way you want it to. We'll be going through all the various forms in this article, with each variation represented by the MessageBox.Show() line and a screenshot of the result. **In the end of the article, you can find a complete example which lets you test all the variations.**

In its simplest form, the MessageBox just takes a single parameter, which is the message to be displayed:

```
MessageBox.Show( "Hello, world!" );
```



#### 1.11.1.1. MessageBox with a title

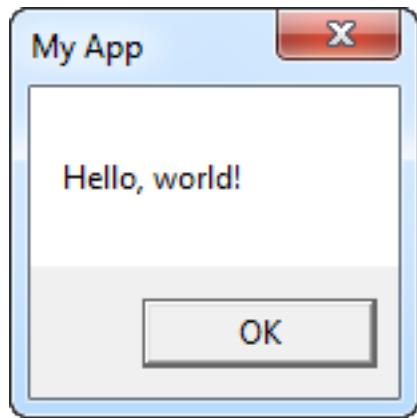
The above example might be a bit too bare minimum - a title on the window displaying the message would probably help. Fortunately, the second and optional parameter allows us to specify the title:

```
MessageBox.Show( "Hello, world!" , "My App" );
```

#### 1.11.1.2. MessageBox with extra buttons

By default, the MessageBox only has the one Ok button, but this can be changed, in case you want to ask your user a question and not just show a piece of information. Also notice how I use multiple lines in this message, by using a line break character (\n):

```
MessageBox.Show( "This MessageBox has extra options.\n\nHello, world?" ,
```



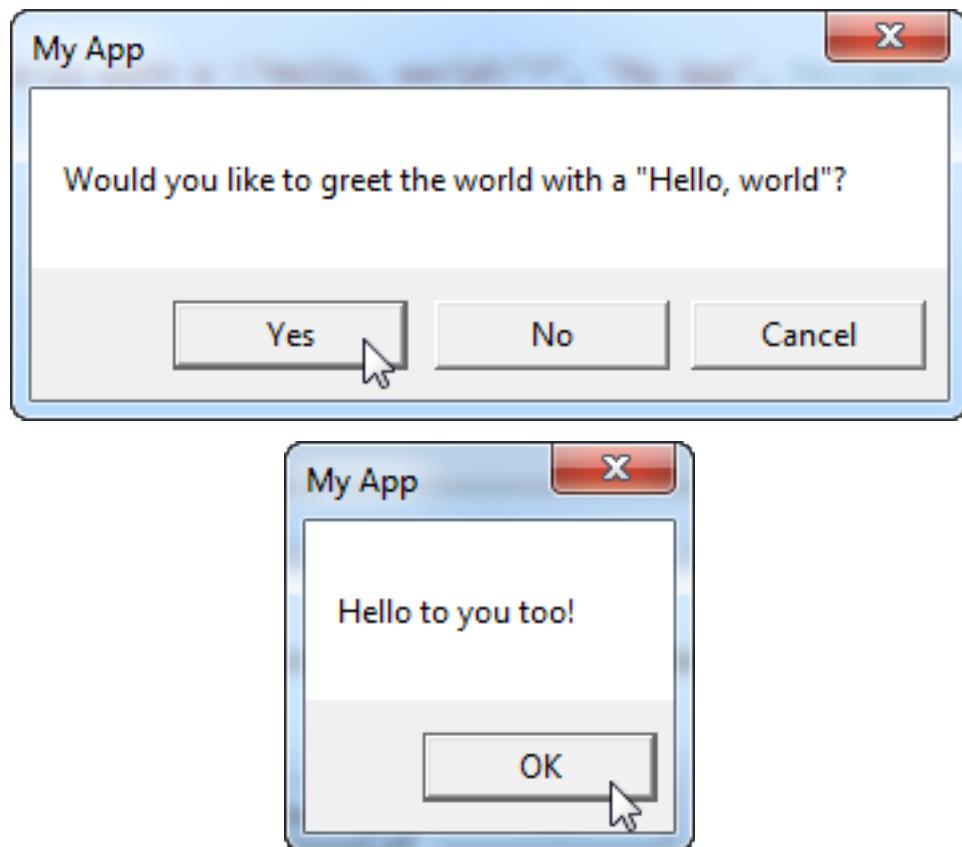
```
"My App", MessageBoxButtons.YesNoCancel);
```

You control which buttons are displayed by using a value from the `MessageBoxButton` enumeration - in this case, a Yes, No and Cancel button is included. The following values, which should be self-explanatory, can be used:

- OK
- OKCancel
- YesNoCancel
- YesNo

Now with multiple choices, you need a way to be able to see what the user chose, and fortunately, the `MessageBox.Show()` method always returns a value from the `MessageBoxResult` enumeration that you can use. Here's an example:

```
MessageBoxResult result = MessageBox.Show("Would you like to greet the  
world with a \"Hello, world\"?", "My App",  
MessageBoxButton.YesNoCancel);  
  
switch(result)  
{  
    case MessageBoxResult.Yes:  
        MessageBox.Show("Hello to you too!", "My App");  
        break;  
    case MessageBoxResult.No:  
        MessageBox.Show("Oh well, too bad!", "My App");  
        break;  
    case MessageBoxResult.Cancel:  
        MessageBox.Show("Nevermind then...", "My App");  
        break;  
}
```

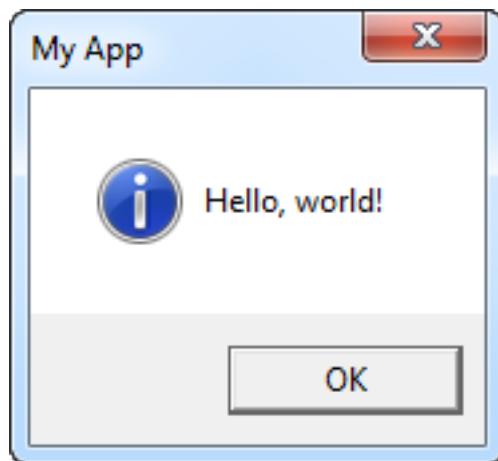


By checking the result value of the `MessageBox.Show()` method, you can now react to the user choice, as seen in the code example as well as on the screenshots.

#### 1.11.1.3. MessageBox with an icon

The `MessageBox` has the ability to show a pre-defined icon to the left of the text message, by using a fourth parameter:

```
MessageBox.Show( "Hello, world!" , "My App" , MessageBoxButtons.OK ,  
MessageBoxImage.Information );
```



Using the **MessageBoxImage** enumeration, you can choose between a range of icons for different

situations. Here's the complete list:

- Asterisk
- Error
- Exclamation
- Hand
- Information
- None
- Question
- Stop
- Warning

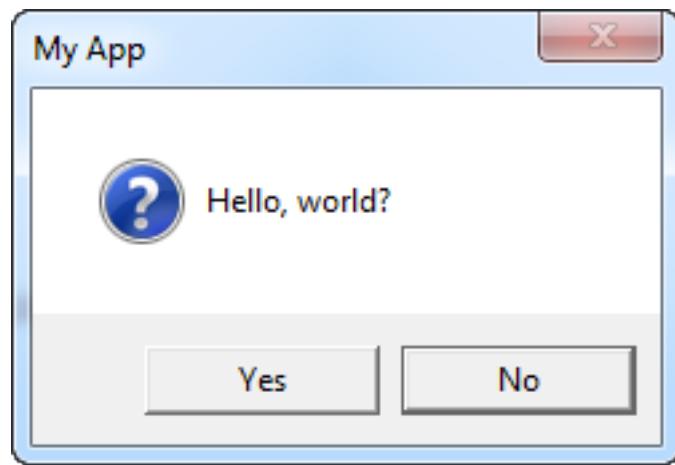
The names should say a lot about how they look, but feel free to experiment with the various values or have a look at this MSDN article, where each value is explained and even illustrated:

<http://msdn.microsoft.com/en-us/library/system.windows.messageboximage.aspx>

#### 1.11.1.4. MessageBox with a default option

The MessageBox will select a button as the default choice, which is then the button invoked in case the user just presses Enter once the dialog is shown. For instance, if you display a MessageBox with a "Yes" and a "No" button, "Yes" will be the default answer. You can change this behavior using a fifth parameter to the MessageBox.Show() method though:

```
MessageBox.Show( "Hello, world?", "My App", MessageBoxButtons.YesNo,  
MessageBoxImage.Question, MessageBoxIcon.No );
```



Notice on the screenshot how the "No" button is slightly elevated, to visually indicate that it is selected and will be invoked if the **Enter** or **Space** button is pressed.

#### 1.11.1.5. The complete example

As promised, here's the complete example used in this article:

```

<Window x:Class="WpfTutorialSamples.Dialogs.MessageBoxSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MessageBoxSample" Height="250" Width="300">
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" >
        <StackPanel.Resources>
            <Style TargetType="Button">
                <Setter Property="Margin" Value="0,0,0,10" />
            </Style>
        </StackPanel.Resources>
        <Button Name="btnSimpleMessageBox" Click
        ="btnSimpleMessageBox_Click">Simple MessageBox</Button>
        <Button Name="btnMessageBoxWithTitle" Click
        ="btnMessageBoxWithTitle_Click">MessageBox with title</Button>
        <Button Name="btnMessageBoxWithButtons" Click
        ="btnMessageBoxWithButtons_Click">MessageBox with buttons</Button>
        <Button Name="btnMessageBoxWithResponse" Click
        ="btnMessageBoxWithResponse_Click">MessageBox with response</Button>
        <Button Name="btnMessageBoxWithIcon" Click
        ="btnMessageBoxWithIcon_Click">MessageBox with icon</Button>
        <Button Name="btnMessageBoxWithDefaultChoice" Click
        ="btnMessageBoxWithDefaultChoice_Click">MessageBox with default choice</
        Button>
    </StackPanel>
</Window>

using System;
using System.Windows;

namespace WpfTutorialSamples.Dialogs
{
    public partial class MessageBoxSample : Window
    {
        public MessageBoxSample()
        {
            InitializeComponent();
        }
}

```

```

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

    private void btnMessageBoxWithTitle_Click(object sender,
RoutedEventArgs e)
{
    MessageBox.Show("Hello, world!", "My App");
}

    private void btnMessageBoxWithButtons_Click(object sender,
RoutedEventArgs e)
{
    MessageBox.Show("This MessageBox has extra
options.\n\nHello, world?", "My App", MessageBoxButton.YesNoCancel);
}

    private void btnMessageBoxWithResponse_Click(object sender,
RoutedEventArgs e)
{
    MessageBoxResult result = MessageBox.Show("Would you like
to greet the world with a \"Hello, world\"?", "My App",
MessageBoxButton.YesNoCancel);
    switch(result)
    {
        case MessageBoxResult.Yes:
            MessageBox.Show("Hello to you too!", "My App");
            break;
        case MessageBoxResult.No:
            MessageBox.Show("Oh well, too bad!", "My App");
            break;
        case MessageBoxResult.Cancel:
            MessageBox.Show("Nevermind then...", "My App");
            break;
    }
}

    private void btnMessageBoxWithIcon_Click(object sender,

```

```
RoutedEventArgs e)
{
    MessageBox.Show( "Hello, world!", "My App",
MessageBoxButton.OK, MessageBoxIcon.Information);
}

private void btnMessageBoxWithDefaultChoice_Click(object
sender, RoutedEventArgs e)
{
    MessageBox.Show( "Hello, world?", "My App",
MessageBoxButton.YesNo, MessageBoxIcon.Question, MessageBoxResult.No);
}
}
```

## 1.11.2. The OpenFileDialog

Whenever you open or save a file in almost any Windows application, you will see roughly the same dialogs for doing that. The reason is of course that these dialogs are a part of the Windows API and therefore also accessible to developers on the Windows platform.

For WPF, you will find standard dialogs for both opening and saving files in the **Microsoft.Win32** namespace. In this article we'll focus on the **OpenFileDialog** class, which makes it very easy to display a dialog for opening one or several files.

### 1.11.2.1. Simple OpenFileDialog example

Let's start off by using the OpenFileDialog without any extra options, to load a file to a TextBox control:

```
<Window x:Class="WpfTutorialSamples.Dialogs.OpenFileDialogSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="OpenFileDialogSample" Height="300" Width="300">
    <DockPanel Margin="10">
        <WrapPanel HorizontalAlignment="Center" DockPanel.Dock="Top"
Margin="0,0,0,10">
            <Button Name="btnOpenFile" Click="btnOpenFile_Click">
                Open file</Button>
        </WrapPanel>
        <TextBox Name="txtEditor" />
    </DockPanel>
</Window>

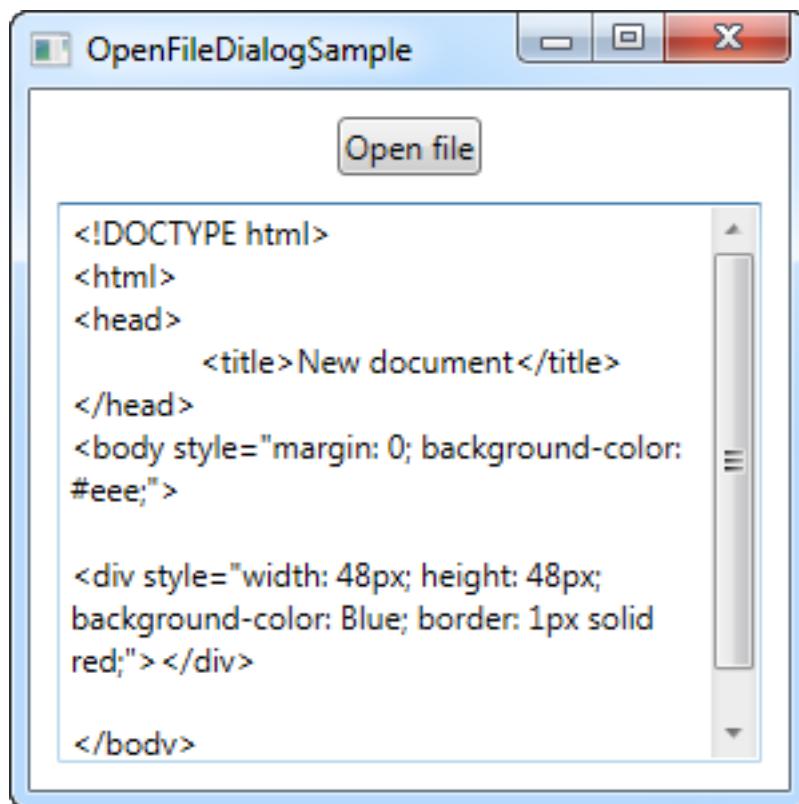
using System;
using System.IO;
using System.Windows;
using Microsoft.Win32;

namespace WpfTutorialSamples.Dialogs
{
    public partial class OpenFileDialogSample : Window
    {
        public OpenFileDialogSample()
        {
            InitializeComponent();
        }
    }
}
```

```

private void btnOpenFile_Click(object sender, RoutedEventArgs
e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    if(openFileDialog.ShowDialog() == true)
        txtEditor.Text =
File.ReadAllText(openFileDialog.FileName);
}
}

```

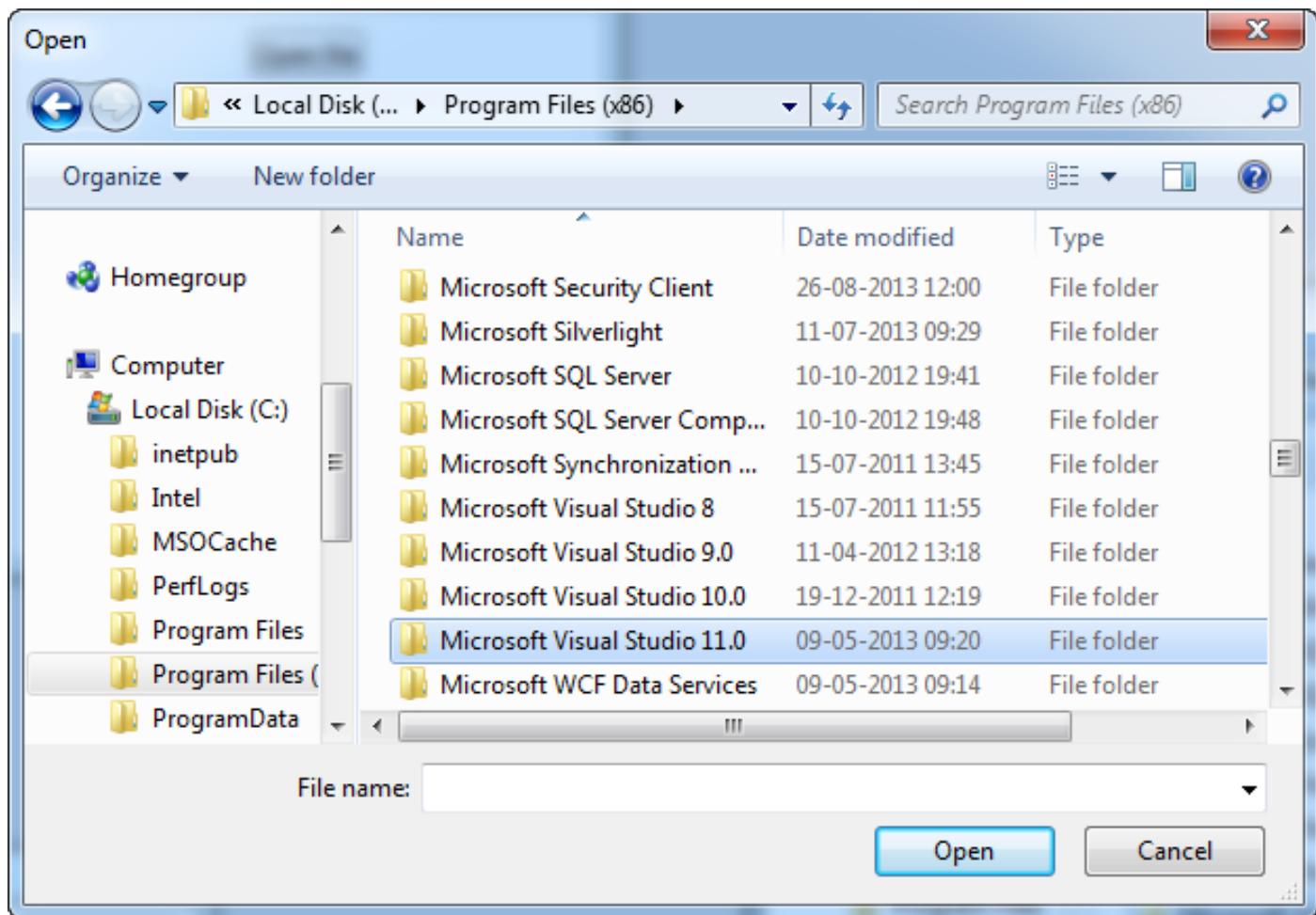


Once you click the Open file button, the OpenFileDialog will be instantiated and shown. Depending on which version of Windows you're using and the theme selected, it will look something like this:

The ShowDialog() will return a nullable boolean value, meaning that it can be either false, true or null. If the user selects a file and presses "Open", the result is True, and in that case, we try to load the file into the TextBox control. We get the complete path of the selected file by using the **FileName** property of the OpenFileDialog.

### 1.11.2.2. Filter

Normally when you want your user to open a file in your application, you want to limit it to one or a couple of file types. For instance, Word mostly opens Word file (with the extension .doc or .docx) and Notepad



mostly open text files (with the extension .txt).

You can specify a filter for your `OpenFileDialog` to indicate to the user which types of file they should be opening in your application, as well as limiting the files shown for a better overview. This is done with the `Filter` property, which we can add to the above example, right after initializing the dialog, like this:

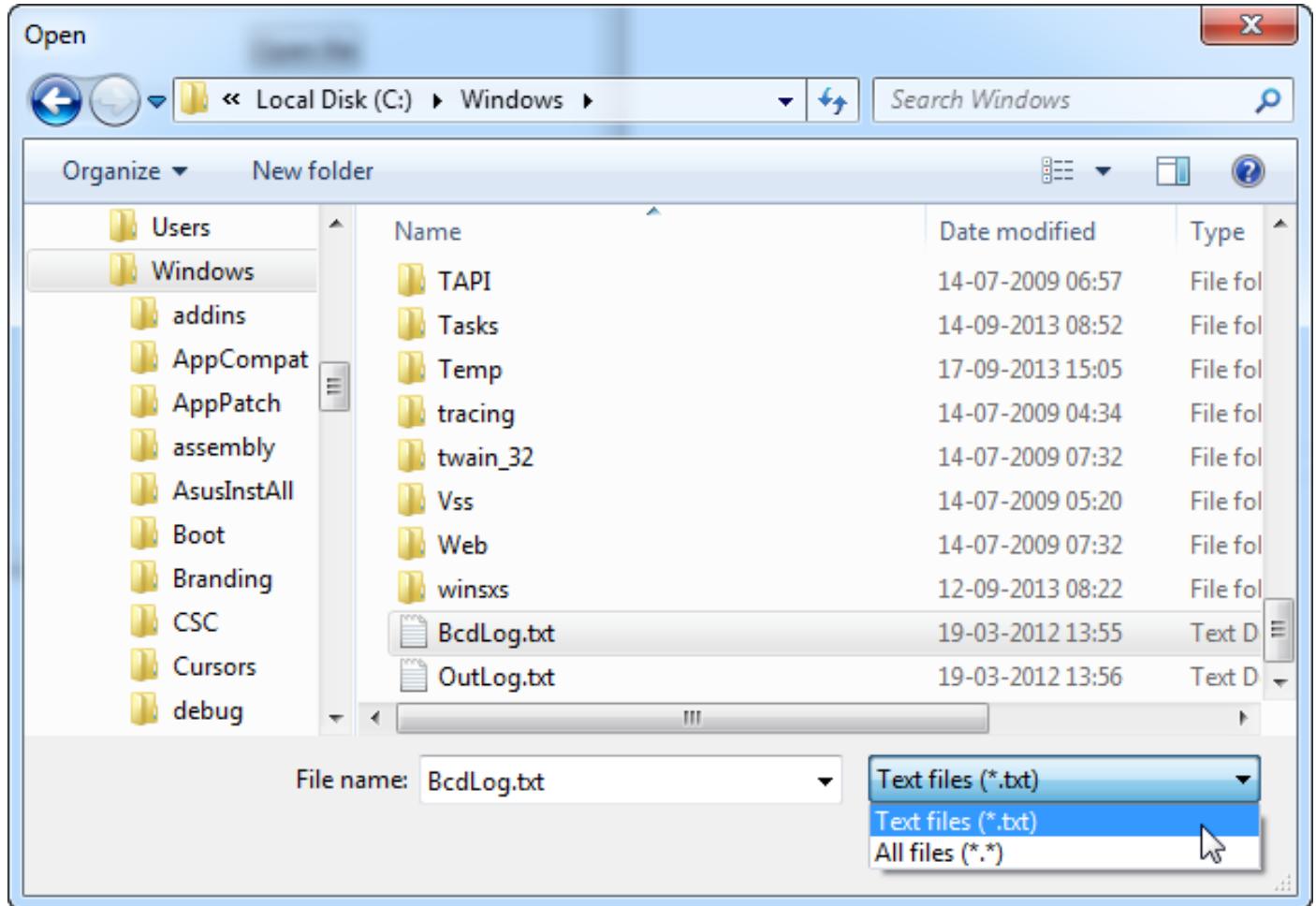
```
openFileDialog.Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*";
```

Here's the result:

Notice how the dialog now has a combo box for selecting the file types, and that the files shown are limited to ones with the extension(s) specified by the selected file type.

The format for specifying the filter might look a bit strange at first sight, but it works by specifying a human-readable version of the desired file extension(s) and then one for the computer to easily parse, separated with a pipe (|) character. If you want more than one file type, as we do in the above example, each set of information are also separated with a pipe character.

So to sum up, the following part means that we want the file type to be named "Text files (\*.txt)" (the extension in the parenthesis is a courtesy to the user, so they know which extension(s) are included) and the second part tells the dialog to show files with a .txt extension:



Text files (\*.txt) | \*.txt

Each file type can of course have multiple extensions. For instance, image files could be specified as both JPEG and PNG files, like this:

```
openFileDialog.Filter = "Image files (*.png;*.jpeg) | *.png;*.jpeg|All files (*.*) | *.*";
```

Simply separate each extension with a semicolon in the second part (the one for the computer) - in the first part, you can format it the way you want to, but most developers seem to use the same notation for both parts, as seen in the example above.

### 1.11.2.3. Setting the initial directory

The initial directory used by the OpenFileDialog is decided by Windows, but by using the **InitialDirectory** property, you can override it. You will usually set this value to a user specified directory, the application directory or perhaps just to the directory last used. You can set it to a path in a string format, like this:

```
openFileDialog.InitialDirectory = @"c:\temp\";
```

If you want to use one of the special folders on Windows, e.g. the Desktop, My Documents or the Program Files directory, you have to take special care, since these may vary from each version of Windows and also

be dependent on which user is logged in. The .NET framework can help you though, just use the Environment class and its members for dealing with special folders:

```
openFileDialog.InitialDirectory =  
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

In this case, I get the path for the My Documents folder, but have a look at the SpecialFolder enumeration - it contains values for a lot of interesting paths. For a full list, please see this [MSDN article](#).

#### 1.11.2.4. Multiple files

If your application supports multiple open files, or you simply want to use the OpenFileDialog to select more than one file at a time, you need to enable the **Multiselect** property. In the next example, we've done just that, and as a courtesy to you, dear reader, we've also applied all the techniques mentioned above, including filtering and setting the initial directory:

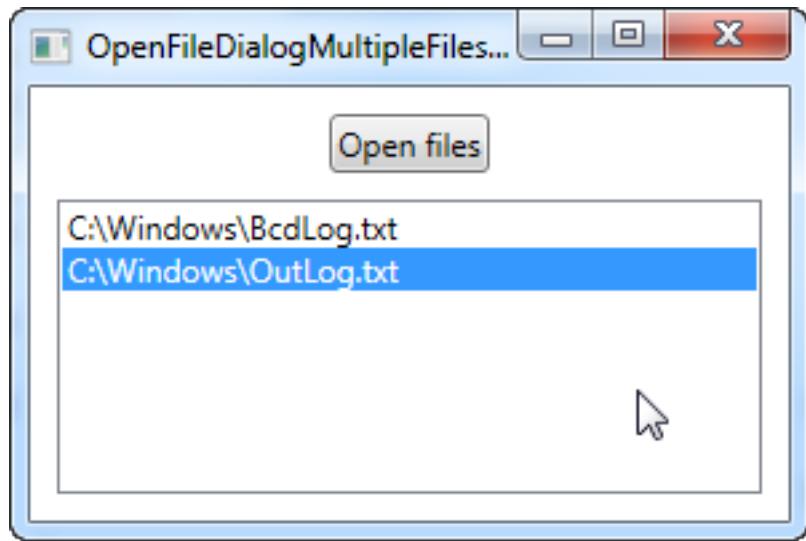
```
<Window x:Class  
="WpfTutorialSamples.Dialogs.OpenFileDialogMultipleFilesSample"  
    xmlns  
="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Title="OpenFileDialogMultipleFilesSample" Height="300" Width  
="300">>  
    <DockPanel Margin="10">  
        <WrapPanel HorizontalAlignment="Center" DockPanel.Dock="Top"  
Margin="0,0,0,10">  
            <Button Name="btnOpenFile" Click="btnOpenFiles_Click">  
                Open files</Button>  
        </WrapPanel>  
        <ListBox Name="lbFiles" />  
    </DockPanel>  
</Window>  
  
using System;  
using System.IO;  
using System.Windows;  
using Microsoft.Win32;  
  
namespace WpfTutorialSamples.Dialogs  
{  
    public partial class OpenFileDialogMultipleFilesSample : Window  
    {
```

```

public OpenFileDialogMultipleFilesSample()
{
    InitializeComponent();
}

private void btnOpenFiles_Click(object sender, RoutedEventArgs
e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Multiselect = true;
    openFileDialog.Filter = "Text files (*.txt)|*.txt|All
files (*.*)|*.*";
    openFileDialog.InitialDirectory =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
    if(openFileDialog.ShowDialog() == true)
    {
        foreach(string filename in openFileDialog.FileNames)
            lbFiles.Items.Add(Path.GetFileName(filename));
    }
}
}

```



If you test this code, you will see that you can now select multiple files in the same directory, by holding down either **Ctrl** or **Shift** and clicking with the mouse. Once accepted, this example simply adds the filenames to the **ListBox** control, by looping through the **FileNames** property.

#### 1.11.2.5. Summary

As you can see, using the OpenFileDialog in WPF is very easy and really takes care of a lot of work for you. Please be aware that to reduce the amount of code lines, no exception handling is done in these examples. When working with files and doing IO tasks in general, you should always look out for exceptions, as they can easily occur due to a locked file, non-existing path or related problems.

### 1.11.3. The SaveFileDialog

The SaveFileDialog will help you select a location and a filename when you wish to save a file. It works and looks much like the OpenFileDialog which we used in the previous article, with a few subtle differences. Just like the OpenFileDialog, the SaveFileDialog is a wrapper around a common Windows dialog, meaning that your users will see roughly the same dialog whether they initiate it in your application or e.g. in Notepad.

#### 1.11.3.1. Simple SaveFileDialog example

To kick things off, let's begin with a very simple example on using the SaveFileDialog:

```
<Window x:Class="WpfTutorialSamples.Dialogs.SaveFileDialogSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SaveFileDialogSample" Height="300" Width="300">
    <DockPanel Margin="10">
        <WrapPanel HorizontalAlignment="Center" DockPanel.Dock="Top"
Margin="0,0,0,10">
            <Button Name="btnSaveFile" Click="btnSaveFile_Click">
                Save file</Button>
        </WrapPanel>
        <TextBox Name="txtEditor" TextWrapping="Wrap" AcceptsReturn
=True" ScrollViewer.VerticalScrollBarVisibility="Auto" />
    </DockPanel>
</Window>

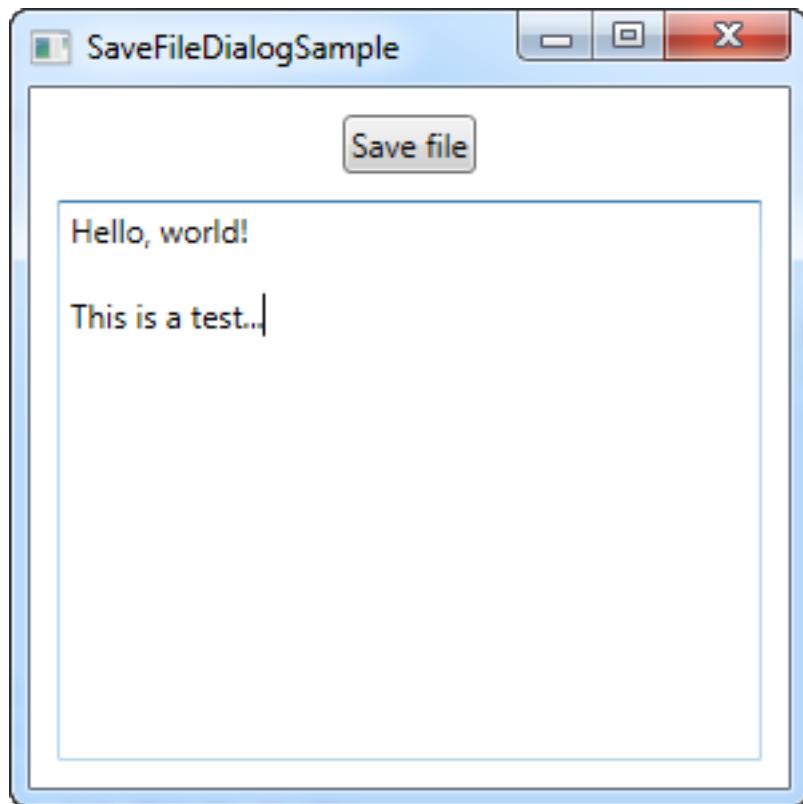
using System;
using System.IO;
using System.Windows;
using Microsoft.Win32;

namespace WpfTutorialSamples.Dialogs
{
    public partial class SaveFileDialogSample : Window
    {
        public SaveFileDialogSample()
        {
            InitializeComponent();
        }
    }
}
```

```

private void btnSaveFile_Click(object sender, RoutedEventArgs
e)
{
    SaveFileDialog saveFileDialog = new SaveFileDialog();
    if(saveFileDialog.ShowDialog() == true)
        File.WriteAllText(saveFileDialog.FileName,
txtEditor.Text);
}
}

```

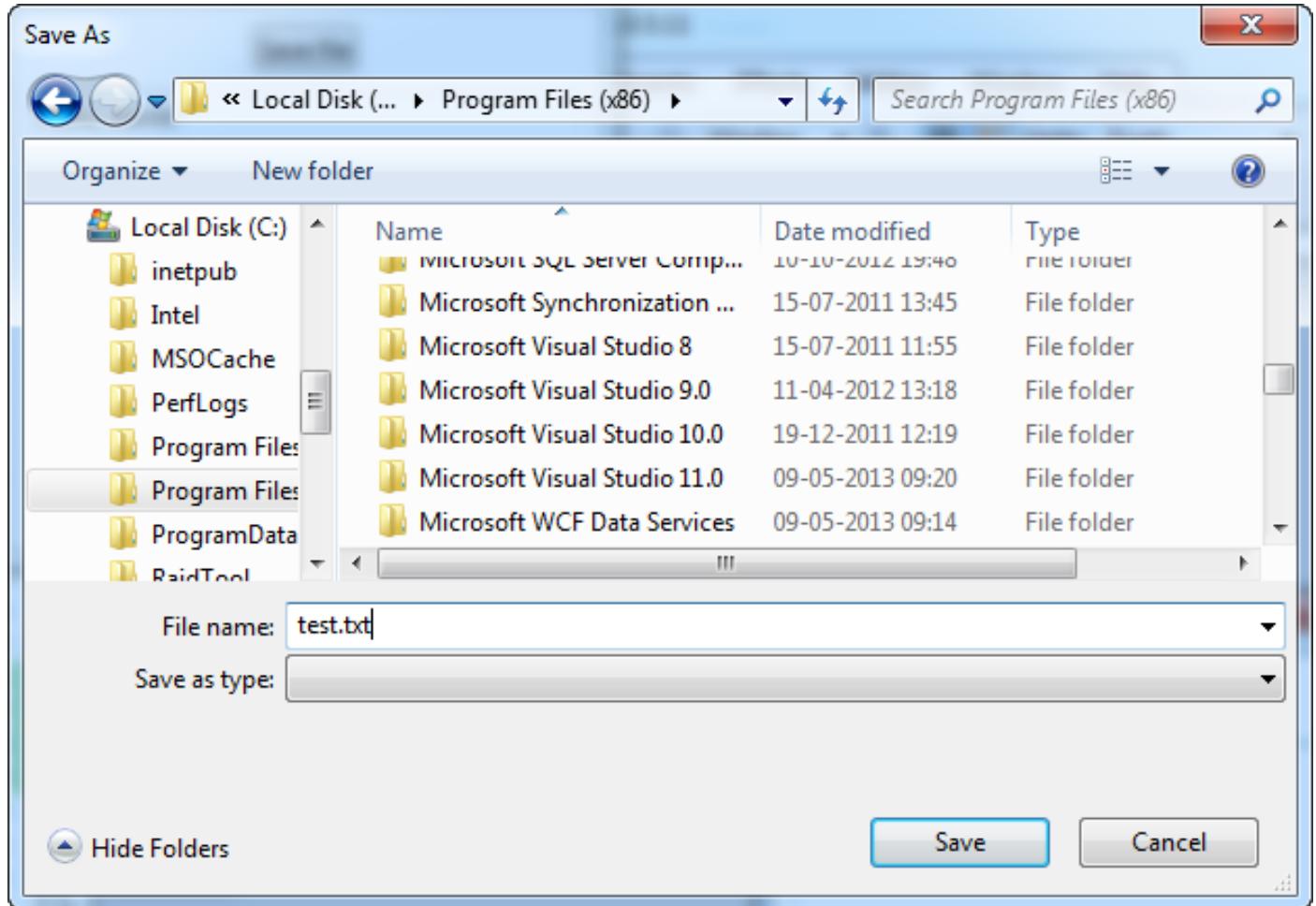


As you can see, it's mostly about instantiating the **SaveFileDialog** and then calling the **ShowDialog()** method. If it returns true, we use the **FileName** property (which will contain the selected path as well as the user entered file name) as the path to write our contents to.

If you click the save button, you should see a dialog like this, depending on the version of Windows you're using:

### 1.11.3.2. Filter

As you can see from the first example, I manually added a .txt extension to my desired filename, mainly because the "Save as type" combo box is empty. Just like for the OpenFileDialog, this box is controlled through the **Filter** property, and it's also used in the exact same way.



```
saveFileDialog.Filter = "Text file (*.txt)|*.txt|C# file (*.cs)|*.cs";
```

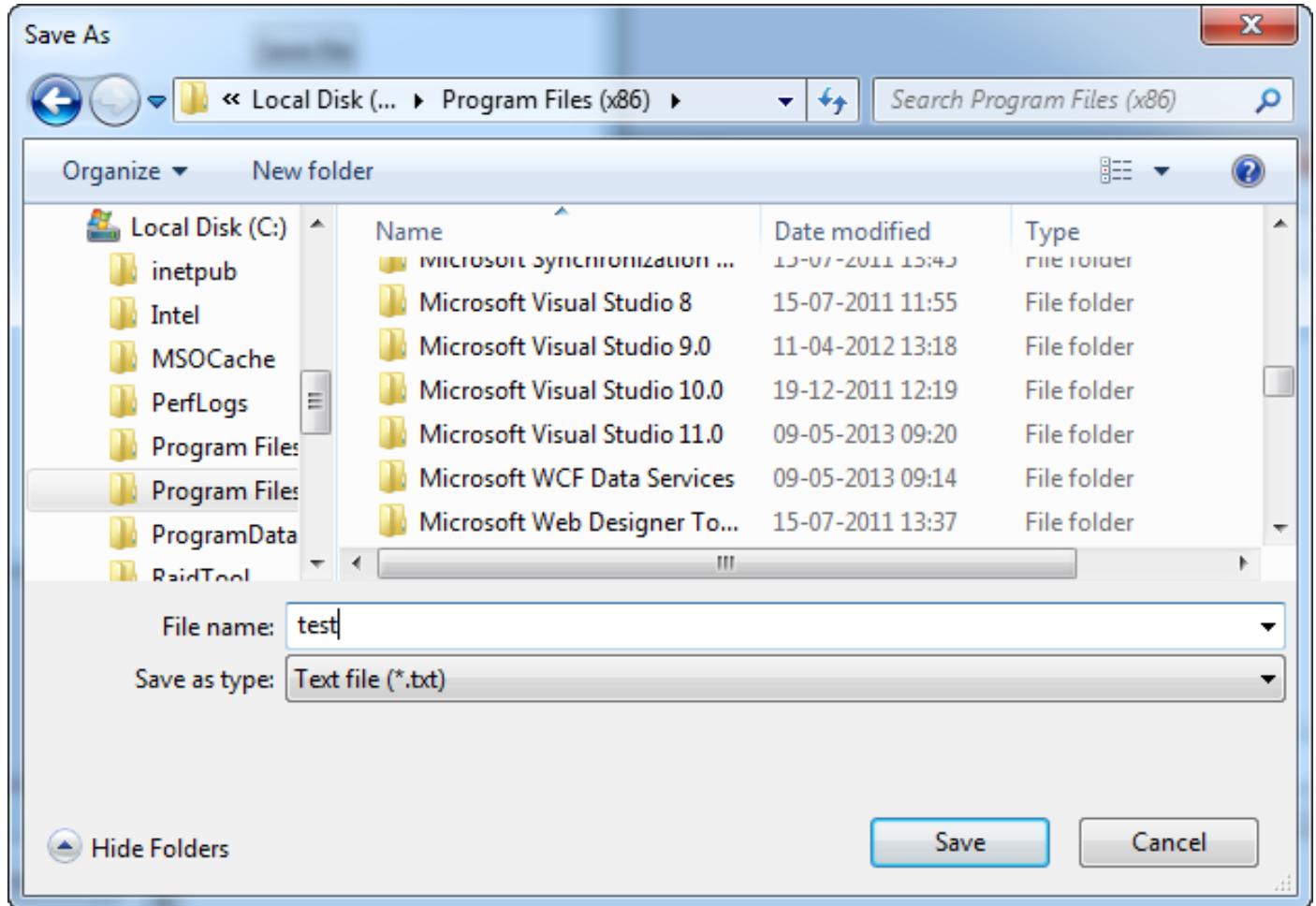
For more details about the format of the Filter property, please see the previous article on the [OpenFileDialog](#), where it's explained in details.

With a filter like the above, the resulting SaveFileDialog will look like this instead:

With that in place, you can write filenames without specifying the extension - it will be taken from the selected file type in the filter combo box instead. This also indicates to the user which file formats your application supports, which is of course important.

#### 1.11.3.3. Setting the initial directory

The initial directory used by the SaveFileDialog is decided by Windows, but by using the **InitialDirectory** property, you can override it. You will usually set this value to a user specified directory, the application directory or perhaps just to the directory last used. You can set it to a path in a string format, like this:



```
saveFileDialog.InitialDirectory = @"c:\temp\";
```

If you want to use one of the special folders on Windows, e.g. the Desktop, My Documents or the Program Files directory, you have to take special care, since these may vary from each version of Windows and also depend on which user is logged in. The .NET framework can help you though, just use the Environment class and its members for dealing with special folders:

```
saveFileDialog.InitialDirectory =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

In this case, I get the path for the My Documents folder, but have a look at the SpecialFolder enumeration - it contains values for a lot of interesting paths. For a full list, please see this [MSDN article](#).

#### 1.11.3.4. Options

Besides the options already mentioned in this article, I want to draw your attention to the following properties, which will help you tailor the SaveFileDialog to your needs:

**AddExtension** - defaults to true and determines if the SaveFileDialog should automatically append an extension to the filename, if the user omits it. The extension will be based on the selected filter, unless that's not possible, in which case it will fall back to the **DefaultExt** property (if specified). If you want your application to be able to save files without file extensions, you may have to disable this option.

**OverwritePrompt** - defaults to true and determines if the SaveFileDialog should ask for a confirmation if the user enters a file name which will result in an existing file being overwritten. You will normally want to leave this option enabled except in very special situations.

**Title** - you may override this property if you want a custom title on your dialog. It defaults to "Save As" or the localized equivalent and the property is also valid for the OpenFileDialog.

**ValidateNames** - defaults to true and unless it's disabled, it will ensure that the user enters only valid Windows file names before allowing the user to continue.

## 1.11.4. The other dialogs

Windows Forms comes with a range of dialogs which we haven't talked about in this tutorial yet, for the simple reason that they don't exist in WPF. The most important one is definitely the [FolderBrowserDialog](#), which lets the user select a folder within the file system, but other dialogs missing in WPF include the [ColorDialog](#), the [FontDialog](#), the [PrintPreviewDialog](#) and the [PageSetupDialog](#).

This can be a real problem for WPF developers, since re-implementing these dialogs would be a huge task. Fortunately, WPF and WinForms can be mixed, simply by referencing the **System.Windows.Forms** assembly, but since WPF uses different base types for both colors and dialogs, this is not always a viable solution. It is however an easy solution if you just need the FolderBrowserDialog, since it only deals with folder paths as simple strings, but some purists would argue that mixing WPF and WinForms is never the way to go.

A better way to go, if you don't want to reinvent the wheel yourself, might be to use some of the work created by other developers. Here are a couple of links for article which offers a solution to some of the missing dialogs:

- [A FontDialog alternative for WPF](#)
- [A ColorDialog alternative for WPF](#)

In the end, you should choose the solution which fits the requirements of your application best.

## 1.11.5. Creating a custom input dialog

In the last couple of articles, we've looked at using the built-in dialogs of WPF, but creating your own is almost just as easy. In fact, you really just need to create a Window, place the required controls in it and then show it.

However, there are a few things that you should remember when creating dialogs, to ensure that your application acts like other Windows applications. In this article, we'll create a very simple dialog to ask the user a question and then return the answer, while discussing the various good practices that you should follow.

### 1.11.5.1. Designing the dialog

For this particular dialog, I just wanted a Label telling the user which information we need from him/her, a TextBox for entering the answer, and then the usual Ok and Cancel buttons. I decided to add an icon to the dialog as well, for good looks. Here's the end result:



And here's the code for the dialog:

```
<Window x:Class="WpfTutorialSamples.DialogsInputDialogSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Input" SizeToContent="WidthAndHeight"
        WindowStartupLocation="CenterScreen"
        ContentRendered="Window_ContentRendered">
    <Grid Margin="15">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
```

```

        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Image Source
=/WpfTutorialSamples/component/Images/question32.png" Width="32" Height
="32" Grid.RowSpan="2" Margin="20,0" />

    <Label Name="lblQuestion" Grid.Column="1">Question:</Label>
    <TextBox Name="txtAnswer" Grid.Column="1" Grid.Row="1"
MinWidth="250">Answer</TextBox>

    <WrapPanel Grid.Row="2" Grid.ColumnSpan="2"
HorizontalAlignment="Right" Margin="0,15,0,0">
        <Button IsDefault="True" Name="btnDialogOk" Click
="btnDialogOk_Click" MinWidth="60" Margin="0,0,10,0">_Ok</Button>
        <Button IsCancel="True" MinWidth="60">_Cancel</Button>
    </WrapPanel>
</Grid>
</Window>

using System;
using System.Windows;

namespace WpfTutorialSamples.Dialogs
{
    public partial class InputDialogSample : Window
    {
        public InputDialogSample(string question, string defaultAnswer
= " ")
        {
            InitializeComponent();
            lblQuestion.Content = question;
            txtAnswer.Text = defaultAnswer;
        }

        private void btnDialogOk_Click(object sender, RoutedEventArgs
e)
        {
            this.DialogResult = true;
        }
    }
}

```

```

        }

    private void Window_ContentRendered(object sender, EventArgs
e)
{
    txtAnswer.SelectAll();
    txtAnswer.Focus();
}

public string Answer
{
    get { return txtAnswer.Text; }
}
}
}

```

The code is pretty simple, but here are the things that you should pay special attention to:

#### 1.11.5.2. XAML

In the **XAML** part, I've used a Grid for layout of the controls - nothing fancy here. I've removed the Width and Height properties of the Window and instead set it to automatically resize to match the content - this makes sense in a dialog, so you don't have to fine tune the size to make everything look alright. Instead, use margins and minimum sizes to ensure that things look the way you want them to, while still allowing the user to resize the dialog.

Another property which I've changed on the Window is the **WindowStartupLocation** property. For a dialog like this, and probably for most other non-main windows, you should change this value to CenterScreen or CenterOwner, to change the default behavior where your window will appear in a position decided by Windows, unless you manually specify **Top** and **Left** properties for it.

Also pay special attention to the two properties I've used on the dialog buttons: **IsCancel** and **IsDefault**. IsCancel tells WPF that if the user clicks this button, the **DialogResult** of the Window should be set to *false* which will also close the window. This also ensures that the user can press the **Esc** key on their keyboard to close the window, something that should always be possible in a Windows dialog.

The **IsDefault** property gives focus to the Ok button and also ensures that if the user presses the Enter key on their keyboard, this button is activated. An event handler is needed to set the DialogResult for this though, as described later.

#### 1.11.5.3. Code-behind

In **Code-behind**, I changed the constructor to take two parameters, with one being optional. This allows us to place the question and the default answer, if provided, into the designated UI controls.

The Ok button has an event handler which ensures that the special `DialogResult` property of the `Window` is set to `true` when clicked, to signal to the initiator of the dialog that the user accepted the entered value. We don't have one for the Cancel button, because WPF handles this for us when we set the `IsCancel` property to true, as described above.

To give focus to the `TextBox` upon showing the dialog, I've subscribed to the `ContentRendered` event, where I select all the text in the control and then give focus. If I just wanted to give focus, I could have used the `FocusManager.FocusedElement` attached property on the `Window`, but in this case, I also want to select the text, to allow the user to instantly overwrite the answer provided by default (if any).

A last detail is the `Answer` property which I've implemented. It simply gives access to the entered value of the `TextBox` control, but it's good practice to provide a property with the return value(s) of the dialog, instead of directly accessing controls from outside the window. This also allows you to influence the return value before returning it, if needed.

#### 1.11.5.4. Using the dialog

With all the above in place, we're now ready to actually use our dialog. It's a very simple task, so I've created a small application for testing it. Here's the code:

```
<Window x:Class="WpfTutorialSamples.DialogsInputDialogAppSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="InputDialogAppSample" Height="150" Width="300">
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <TextBlock>Hello, world. My name is:</TextBlock>
        <TextBlock Name="lblName" Margin="0,10" TextAlignment="Center"
        FontWeight="Bold">[No name entered]</TextBlock>
        <Button Name="btnEnterName" Click="btnEnterName_Click">Enter
        name...</Button>
    </StackPanel>
</Window>

using System;
using System.Windows;

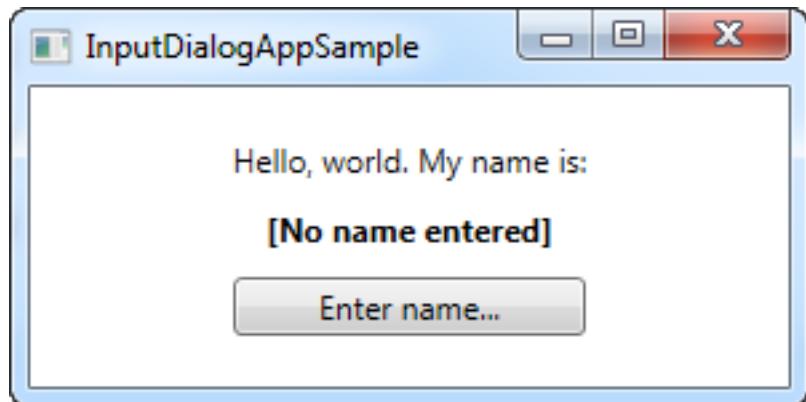
namespace WpfTutorialSamples.Dialogs
{
    public partial class InputDialogAppSample : Window
    {
        public InputDialogAppSample()
        {
```

```

    {
        InitializeComponent();
    }

    private void btnEnterName_Click(object sender, RoutedEventArgs
e)
    {
        InputDialogSample inputDialog = new InputDialogSample(
"Please enter your name:", "John Doe");
        if(inputDialog.ShowDialog() == true)
            lblName.Text = inputDialog.Answer;
    }
}

```



There's nothing special to it - just a couple of `TextBlock` controls and a `Button` for invoking the dialog. In the `Click` event handler, we instantiate the `InputDialogSample` window, providing a question and a default answer, and then we use the `ShowDialog()` method to show it - you should always use `ShowDialog()` method and not just `Show()` for a modal dialog like this.

If the result of the dialog is true, meaning that the user has activated the Ok button either by clicking it or pressing Enter, the result is assigned to the name Label. That's all there is to it!

## 1.12. Common interface controls

---

### 1.12.1. The WPF Menu control

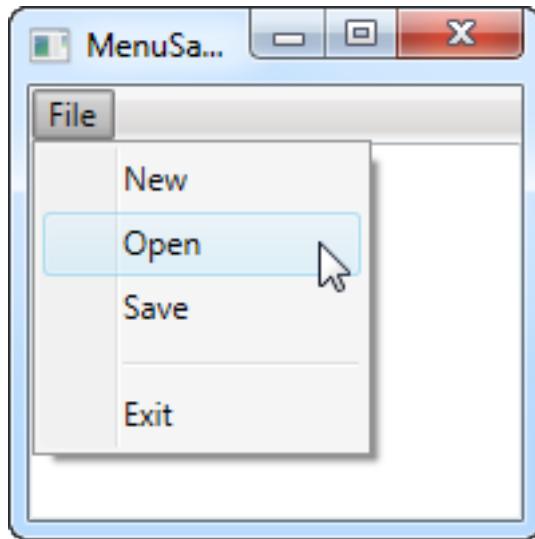
One of the most common parts of a Windows application is the menu, sometimes referred to as the main menu because only one usually exists in the application. The menu is practical because it offers a lot of options, using only very little space, and even though Microsoft is pushing the Ribbon as a replacement for the good, old menu and toolbars, they definitely still have their place in every good developer's toolbox.

WPF comes with a fine control for creating menus called... `Menu`. Adding items to it is very simple - you simply add `MenuItem` elements to it, and each `MenuItem` can have a range of sub-items, allowing you to create hierarchical menus as you know them from a lot of Windows applications. Let's jump straight to an example where we use the `Menu`:

```
<Window x:Class="WpfTutorialSamples.Common_interface_controls.MenuSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MenuSample" Height="200" Width="200">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="_File">
                <MenuItem Header="_New" />
                <MenuItem Header="_Open" />
                <MenuItem Header="_Save" />
                <Separator />
                <MenuItem Header="_Exit" />
            </MenuItem>
        </Menu>
        <TextBox AcceptsReturn="True" />
    </DockPanel>
</Window>
```

As in most Windows applications, my menu is placed in the top of the window, but in keeping with the enormous flexibility of WPF, you can actually place a `Menu` control wherever you like, and in any width or height that you may desire.

I have defined a single top-level item, with 4 child items and a separator. I use the `Header` property to define the label of the item, and you should notice the underscore before the first character of each label. It tells WPF to use that character as the accelerator key, which means that the user can press the Alt key



followed by the given character, to activate the menu item. This works all the way from the top-level item and down the hierarchy, meaning that in this example I could press **Alt**, then **F** and then **N**, to activate the *New* item.

#### 1.12.1.1. Icons and checkboxes

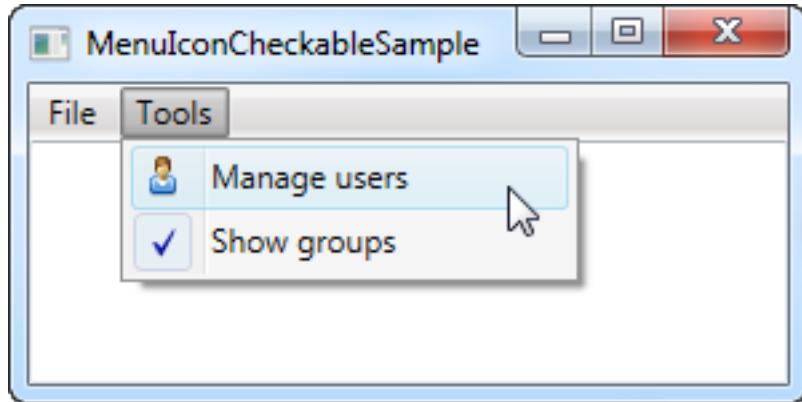
Two common features of a menu item is the icon, used to more easily identify the menu item and what it does, and the ability to have checkable menu items, which can toggle a specific feature on and off. The WPF MenuItem supports both, and it's very easy to use:

```
<Window x:Class
    ="WpfTutorialSamples.Common_interface_controls.MenuIconCheckableSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MenuIconCheckableSample" Height="150" Width="300">
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="_File">
                <MenuItem Header="_Exit" />
            </MenuItem>
            <MenuItem Header="_Tools">
                <MenuItem Header="_Manage users">
                    <MenuItem.Icon>
                        <Image Source
                            ="/WpfTutorialSamples;component/Images/user.png" />
                    </MenuItem.Icon>
                </MenuItem>
                <MenuItem Header="_Show groups" IsCheckable="True"
                    IsChecked="True" />
            
```

```

    </MenuItem>
</Menu>
<TextBox AcceptsReturn="True" />
</DockPanel>
</Window>

```



For this example I've created a secondary top-level item, where I've added two items: One with an icon defined, using the **Icon** property with a standard Image control inside of it, and one where we use the **IsCheckable** property to allow the user to check and uncheck the item. I even used the **.IsChecked** property to have it checked by default. From Code-behind, this is the same property that you can read to know whether a given menu item is checked or not.

#### 1.12.1.2. Handling clicks

When the user clicks on a menu item, you will usually want something to happen. The easiest way is to simply add a click event handler to the MenuItem, like this:

```
<MenuItem Header="_New" Click="mnuNew_Click" />
```

In Code-behind you will then need to implement the mnuNew\_Click method, like this:

```
private void mnuNew_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show( "New" );
}
```

This will suffice for the more simple applications, or when prototyping something, but the WPF way is to use a Command for this.

#### 1.12.1.3. Keyboard shortcuts and Commands

You can easily handle the Click event of a menu item like we did above, but the more common approach is to use WPF commands. There's a lot of theory on using and creating commands, so they have their own category of articles here on the site, but for now, I can tell you that they have a couple of advantages when

used in WPF, especially in combination with a Menu or a Toolbar.

First of all, they ensure that you can have the same action on a toolbar, a menu and even a context menu, without having to implement the same code in multiple places. They also make the handling of keyboard shortcuts a whole lot easier, because unlike with WinForms, WPF is not listening for keyboard shortcuts automatically if you assign them to e.g. a menu item - you will have to do that manually.

However, when using commands, WPF is all ears and will respond to keyboard shortcuts automatically. The text (Header) of the menu item is also set automatically (although you can overwrite it if needed), and so is the InputGestureText, which shows the user which keyboard shortcut can be used to invoke the specific menu item. Let's jump straight to an example of combining the Menu with WPF commands:

```
<Window x:Class
        ="WpfTutorialSamples.Common_interface_controls.MenuWithCommandsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MenuWithCommandsSample" Height="200" Width="300">
    <Window.CommandBindings>
        <CommandBinding Command="New" CanExecute
        ="NewCommand_CanExecute" Executed="NewCommand_Executed" />
    </Window.CommandBindings>
    <DockPanel>
        <Menu DockPanel.Dock="Top">
            <MenuItem Header="_File">
                <MenuItem Command="New" />
                <Separator />
                <MenuItem Header="_Exit" />
            </MenuItem>
            <MenuItem Header="_Edit">
                <MenuItem Command="Cut" />
                <MenuItem Command="Copy" />
                <MenuItem Command="Paste" />
            </MenuItem>
        </Menu>
        <TextBox AcceptsReturn="True" Name="txtEditor" />
    </DockPanel>
</Window>

using System;
using System.Windows;
```

```

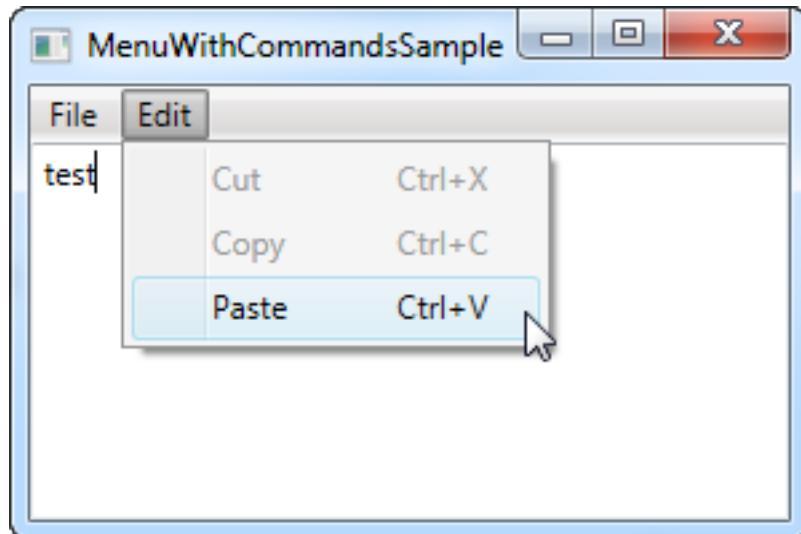
using System.Windows.Input;

namespace WpfTutorialSamples.Common_interface_controls
{
    public partial class MenuWithCommandsSample : Window
    {
        public MenuWithCommandsSample()
        {
            InitializeComponent();
        }

        private void NewCommand_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }

        private void NewCommand_Executed(object sender,
ExecutedRoutedEventArgs e)
        {
            txtEditor.Text = "";
        }
    }
}

```



It might not be completely obvious, but by using commands, we just got a whole bunch of things for free: Keyboard shortcuts, text and **InputGestureText** on the items and WPF automatically enables/disables the items depending on the active control and its state. In this case, Cut and Copy are disabled because no

text is selected, but Paste is enabled, because my clipboard is not empty!

And because WPF knows how to handle certain commands in combination with certain controls, in this case the Cut/Copy/Paste commands in combination with a text input control, we don't even have to handle their Execute events - they work right out of the box! We do have to handle it for the **New** command though, since WPF has no way of guessing what we want it to do when the user activates it. This is done with the **CommandBindings** of the Window, all explained in detail in the chapter on commands.

#### 1.12.1.4. Summary

Working with the WPF Menu control is both easy and fast, making it simple to create even complex menu hierarchies, and when combining it with WPF commands, you get so much functionality for free.

## 1.12.2. The WPF ContextMenu

A context menu, often referred to as a popup or pop-up menu, is a menu which is shown upon certain user actions, usually a right-click with the mouse on a specific control or window. Contextual menus are often used to offer functionality that's relevant within a single control.

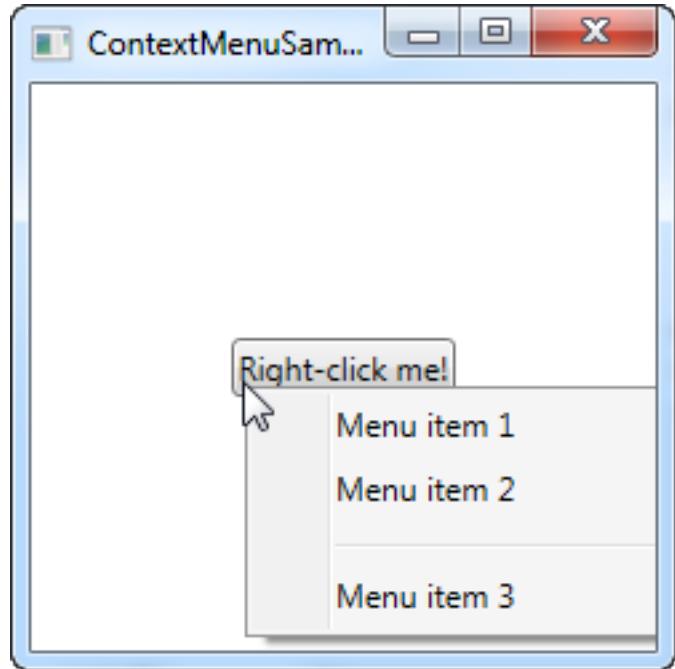
WPF comes with a `ContextMenu` control and because it's almost always tied to a specific control, that's also usually how you add it to the interface. This is done through the `ContextMenu` property, which all controls exposes (it comes from the `FrameworkElement` which most WPF controls inherits from). Consider the next example to see how it's done:

```
<Window x:Class
        ="WpfTutorialSamples.Common_interface_controls.ContextMenuSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ContextMenuSample" Height="250" Width="250">
    <Grid>
        <Button Content="Right-click me!" VerticalAlignment="Center"
        HorizontalAlignment="Center">
            <Button.ContextMenu>
                <ContextMenu>
                    <MenuItem Header="Menu item 1" />
                    <MenuItem Header="Menu item 2" />
                    <Separator />
                    <MenuItem Header="Menu item 3" />
                </ContextMenu>
            </Button.ContextMenu>
        </Button>
    </Grid>
</Window>
```

If you've already read the chapter on the regular menu, you will soon realize that the `ContextMenu` works exactly the same way, and no wonder, since they both inherit the `MenuBase` class. Just like we saw in the examples on using the regular Menu, you can of course add Click events to these items to handle when the user clicks on them, but a more WPF-suitable way is to use Commands.

### 1.12.2.1. ContextMenu with Commands and icons

In this next example, I'm going to show you two key concepts when using the `ContextMenu`: The usage of WPF Commands, which will provide us with lots of functionality including a Click event handler, a text and a shortcut text, simply by assigning something to the `Command` property. I will also show you to use icons on your `ContextMenu` items. Have a look:

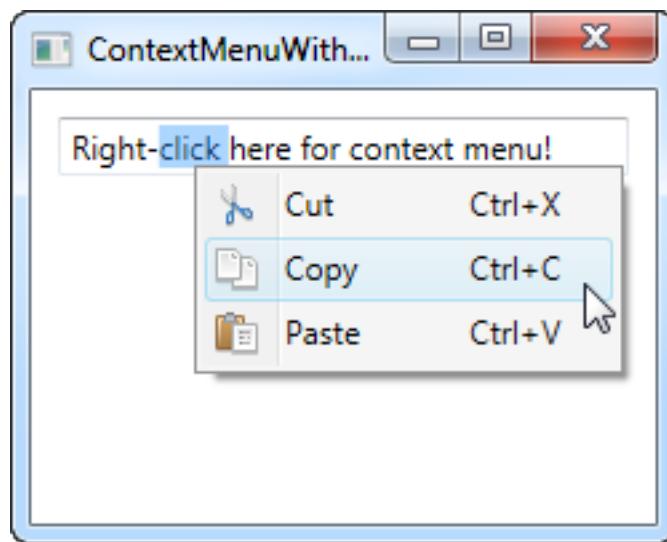


```
<Window x:Class
        ="WpfTutorialSamples.Common_interface_controls.ContextMenuItemCommandsSa
mple"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ContextMenuWithCommandsSample" Height="200" Width="250"
>
    <StackPanel Margin="10">
        <TextBox Text="Right-click here for context menu!">
            <TextBox.ContextMenu>
                <ContextMenu>
                    <MenuItem Command="Cut">
                        <MenuItem.Icon>
                            <Image Source
=" /WpfTutorialSamples;component/Images/cut.png" />
                        </MenuItem.Icon>
                    </MenuItem>
                    <MenuItem Command="Copy">
                        <MenuItem.Icon>
                            <Image Source
=" /WpfTutorialSamples;component/Images/copy.png" />
                        </MenuItem.Icon>
                    </MenuItem>
                    <MenuItem Command="Paste">
                        <MenuItem.Icon>
```

```

        <Image Source
      ="/WpfTutorialSamples/component/Images/paste.png" />
        </MenuItem.Icon>
      </MenuItem>
    </ContextMenu>
  </TextBox.ContextMenu>
</TextBox>
</StackPanel>
</Window>

```



Try running the example and see for yourself how much functionality we get for free by assigning commands to the items. Also notice how fairly simple it is to use icons on the menu items of the ContextMenu.

#### 1.12.2.2. Invoke ContextMenu from Code-behind

So far, the ContextMenu has been invoked when right-clicking on the control to which it belongs. WPF does this for us automatically, when we assign it to the **ContextMenu** property. However, in some situations, you might very well want to invoke it manually from code. This is pretty easy as well, so let's re-use the first example to demonstrate it with:

```

<Window x:Class
  ="WpfTutorialSamples.Common_interface_controls.ContextMenuManuallyInvoke
dSample"
  xmlns
  ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ContextMenuManuallyInvokedSample" Height="250" Width
  ="250">

```

```

<Window.Resources>
    <ContextMenu x:Key="cmButton">
        <MenuItem Header="Menu item 1" />
        <MenuItem Header="Menu item 2" />
        <Separator />
        <MenuItem Header="Menu item 3" />
    </ContextMenu>
</Window.Resources>
<Grid>
    <Button Content="Click me!" VerticalAlignment="Center"
HorizontalAlignment="Center" Click="Button_Click" />
</Grid>
</Window>

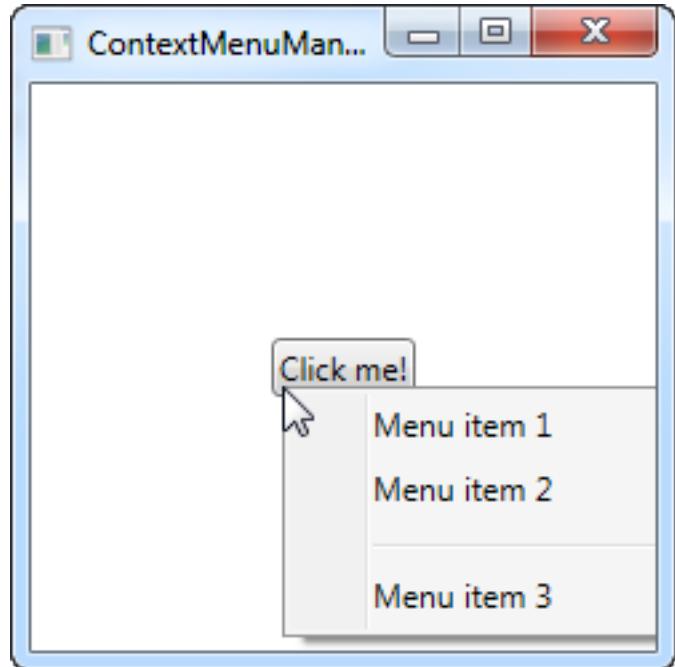
using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfTutorialSamples.Common_interface_controls
{
    public partial class ContextMenuManuallyInvokedSample : Window
    {
        public ContextMenuManuallyInvokedSample()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            ContextMenu cm = this.FindResource("cmButton") as
ContextMenu;
            cm.PlacementTarget = sender as Button;
            cm.IsOpen = true;
        }
    }
}

```

The first thing you should notice is that I've moved the ContextMenu away from the button. Instead, I've added it as a resource of the Window, to make it available from all everywhere within the Window. This also makes it a lot easier to find when we need to show it.



The Button now has a Click event handler, which I handle in Code-behind. From there, I simply find the ContextMenu instance within the window resources and then I do two things: I set its PlacementTarget property, which tells WPF which element it should calculate the position based on, and then I set the IsOpen to true, to open the menu. That's all you need!

### 1.12.3. The WPF ToolBar control

The toolbar is a row of commands, usually sitting right below the main menu of a standard Windows application. This could in fact be a simple panel with buttons on it, but by using the WPF ToolBar control, you get some extra goodies like automatic overflow handling and the possibility for the end-user to re-position your toolbars.

A WPF ToolBar is usually placed inside of a ToolBarTray control. The ToolBarTray will handle stuff like placement and sizing, and you can have multiple ToolBar controls inside of the ToolBarTray element. Let's try a pretty basic example, to see what it all looks like:

```
<Window x:Class
        ="WpfTutorialSamples.Common_interface_controls.ToolbarSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ToolbarSample" Height="200" Width="300">
    <Window.CommandBindings>
        <CommandBinding Command="New" CanExecute
        ="CommonCommandBinding_CanExecute" />
        <CommandBinding Command="Open" CanExecute
        ="CommonCommandBinding_CanExecute" />
        <CommandBinding Command="Save" CanExecute
        ="CommonCommandBinding_CanExecute" />
    </Window.CommandBindings>
    <DockPanel>
        <ToolBarTray DockPanel.Dock="Top">
            <ToolBar>
                <Button Command="New" Content="New" />
                <Button Command="Open" Content="Open" />
                <Button Command="Save" Content="Save" />
            </ToolBar>
            <ToolBar>
                <Button Command="Cut" Content="Cut" />
                <Button Command="Copy" Content="Copy" />
                <Button Command="Paste" Content="Paste" />
            </ToolBar>
        </ToolBarTray>
        <TextBox AcceptsReturn="True" />
    </DockPanel>
</Window>
```

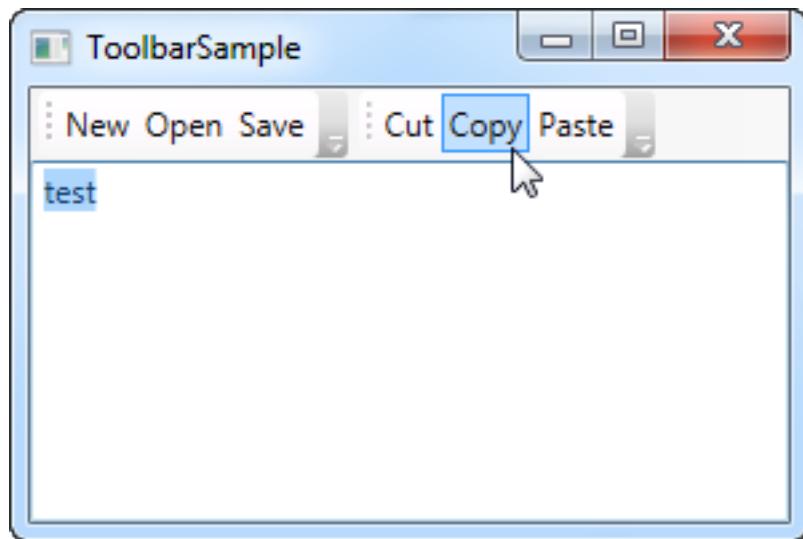
```

using System;
using System.Windows;
using System.Windows.Input;

namespace WpfTutorialSamples.Common_interface_controls
{
    public partial class ToolbarSample : Window
    {
        public ToolbarSample()
        {
            InitializeComponent();
        }

        private void CommonCommandBinding_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
    }
}

```



Notice how I use commands for all the buttons. We discussed this in the previous chapter and using commands definitely gives us some advantages. Take a look at the Menu chapter, or the articles on commands, for more information.

In this example, I add a ToolBarTray to the top of the screen, and inside of it, twoToolBar controls. Each contains some buttons and we use commands to give them their behavior. In Code-behind, I make sure to handle the CanExecute event of the first three buttons, since that's not done automatically by WPF, contrary to the Cut, Copy and Paste commands, which WPF is capable of fully handling for us.

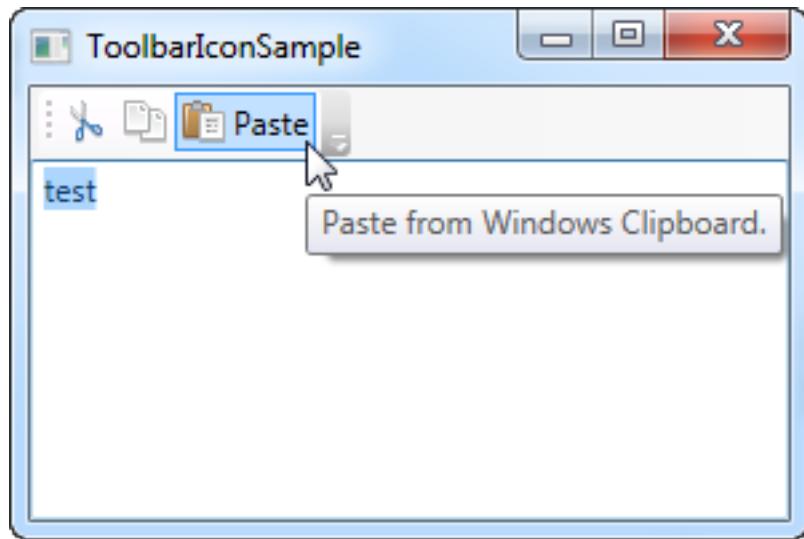
Try running the example and place the cursor over the left part of one of the toolbars (the dotted area). If you click and hold your left mouse button, you can now re-position the toolbar, e.g. below the other or even make them switch place.

### 1.12.3.1. Images

While text on the toolbar buttons is perfectly okay, the normal approach is to have icons or at least a combination of an icon and a piece of text. Because WPF uses regular Button controls, adding icons to the toolbar items is very easy. Just have a look at this next example, where we do both:

```
<Window x:Class
    ="WpfTutorialSamples.Common_interface_controls.ToolbarIconSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ToolbarIconSample" Height="200" Width="300">
    <DockPanel>
        <ToolBarTray DockPanel.Dock="Top">
            <ToolBar>
                <Button Command="Cut" ToolTip="Cut selection to Windows Clipboard.">
                    <Image Source
                ="/WpfTutorialSamples;component/Images/cut.png" />
                </Button>
                <Button Command="Copy" ToolTip="Copy selection to Windows Clipboard.">
                    <Image Source
                ="/WpfTutorialSamples;component/Images/copy.png" />
                </Button>
                <Button Command="Paste" ToolTip="Paste from Windows Clipboard.">
                    <StackPanel Orientation="Horizontal">
                        <Image Source
                    ="/WpfTutorialSamples;component/Images/paste.png" />
                        <TextBlock Margin="3,0,0,0">Paste</TextBlock>
                    </StackPanel>
                </Button>
            </ToolBar>
        </ToolBarTray>
        <TextBox AcceptsReturn="True" />
    </DockPanel>
```

```
</Window>
```



By specifying an **Image** control as the Content of the first two buttons, they will be icon based instead of text based. On the third button, I combine an **Image** control and a **TextBlock** control inside of a **StackPanel**, to achieve both icon and text on the button, a commonly used technique for buttons which are extra important or with a less obvious icon.

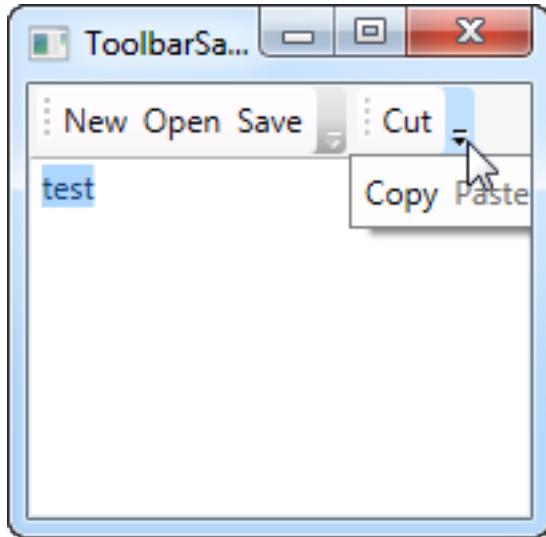
Notice how I've used the **ToolTip** property on each of the buttons, to add an explanatory text. This is especially important for those buttons with only an icon, because the purpose of the button might not be clear from only looking at the icon. With the **ToolTip** property, the user can hover the mouse over the button to get a description of what it does, as demonstrated on the screenshot.

#### 1.12.3.2. Overflow

As already mentioned, a very good reason for using the **ToolBar** control instead of just a panel of buttons, is the automatic overflow handling. It means that if there's no longer enough room to show all of the buttons on the toolbar, WPF will put them in a menu accessible by clicking on the arrow to the right of the toolbar. You can see how it works on this screenshot, which shows the first example, but with a smaller window, thereby leaving less space for the toolbars:

WPF even allows you to decide which items are suitable for overflow hiding and which should always be visible. Usually, when designing a toolbar, some items are less important than the others and some of them you might even want to have in the overflow menu all the time, no matter if there's space enough or not.

This is where the attached property **ToolBar.OverflowMode** comes into play. The default value is **IfNeeded**, which simply means that a toolbar item is put in the overflow menu if there's not enough room for it. You may use **Always** or **Never** instead, which does exactly what the names imply: Puts the item in the overflow menu all the time or prevents the item from ever being moved to the overflow menu. Here's an example on how to assign this property:



```
<ToolBar>
    <Button Command="Cut" Content="Cut" ToolBar.OverflowMode="Always" />
    <Button Command="Copy" Content="Copy" ToolBar.OverflowMode="AsNeeded" />
    <Button Command="Paste" Content="Paste" ToolBar.OverflowMode="Never" />
</ToolBar>
```

### 1.12.3.3. Position

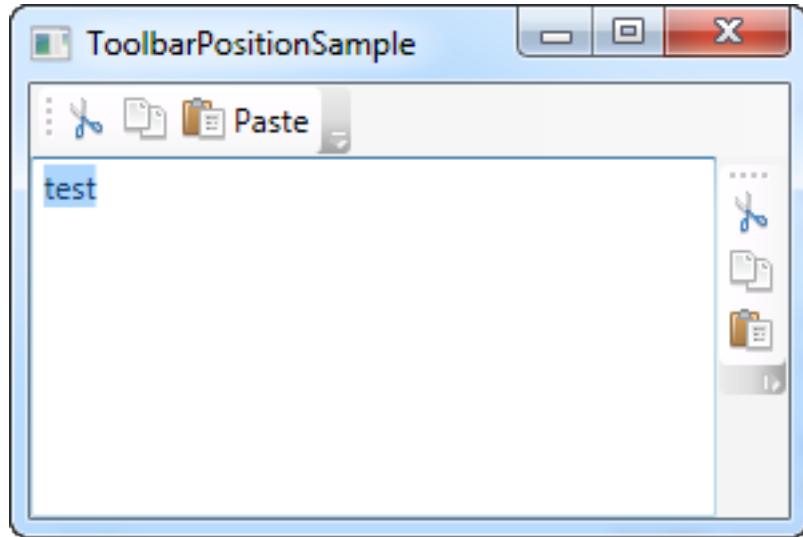
While the most common position for the toolbar is indeed in the top of the screen, toolbars can also be found in the bottom of the application window or even on the sides. The WPF ToolBar of course supports all of this, and while the bottom placed toolbar is merely a matter of docking to the bottom of the panel instead of the top, a vertical toolbar requires the use of the **Orientation** property of the ToolBar tray. Allow me to demonstrate with an example:

```
<Window x:Class="WpfTutorialSamples.Common_interface_controls.ToolbarPositionSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ToolbarPositionSample" Height="200" Width="300">
    <DockPanel>
        <ToolBarTray DockPanel.Dock="Top">
            <ToolBar>
                <Button Command="Cut" ToolTip="Cut selection to Windows Clipboard.">
                    <Image Source="/WpfTutorialSamples;component/Images/cut.png" />
                </Button>
            </ToolBar>
        </ToolBarTray>
    </DockPanel>
</Window>
```

```

        </Button>
        <Button Command="Copy" ToolTip="Copy selection to
Windows Clipboard.">
            <Image Source
            =" /WpfTutorialSamples;component/Images/copy.png" />
        </Button>
        <Button Command="Paste" ToolTip="Paste from Windows
Clipboard.">
            <StackPanel Orientation="Horizontal">
                <Image Source
                =" /WpfTutorialSamples;component/Images/paste.png" />
                <TextBlock Margin="3,0,0,0">Paste</
TextBlock>
            </StackPanel>
        </Button>
    </ToolBar>
</ToolBarTray>
<ToolBarTray DockPanel.Dock="Right" Orientation="Vertical">
    <ToolBar>
        <Button Command="Cut" ToolTip="Cut selection to
Windows Clipboard.">
            <Image Source
            =" /WpfTutorialSamples;component/Images/cut.png" />
        </Button>
        <Button Command="Copy" ToolTip="Copy selection to
Windows Clipboard.">
            <Image Source
            =" /WpfTutorialSamples;component/Images/copy.png" />
        </Button>
        <Button Command="Paste" ToolTip="Paste from Windows
Clipboard.">
            <Image Source
            =" /WpfTutorialSamples;component/Images/paste.png" />
        </Button>
    </ToolBar>
</ToolBarTray>
<TextBox AcceptsReturn="True" />
</DockPanel>
</Window>

```



The trick here lies in the combination of the **DockPanel.Dock** property, that puts the ToolBarTray to the right of the application, and the **Orientation** property, that changes the orientation from horizontal to vertical. This makes it possible to place toolbars in pretty much any location that you might think of.

#### 1.12.3.4. Custom controls on the ToolBar

As you have seen on all of the previous examples, we use regular WPF Button controls on the toolbars. This also means that you can place pretty much any other WPF control on the toolbars, with no extra effort. Of course, some controls work better on a toolbar than others, but controls like the ComboBox and TextBox are commonly used on the toolbars in e.g. older versions of Microsoft Office, and you can do the same on your own WPF toolbars.

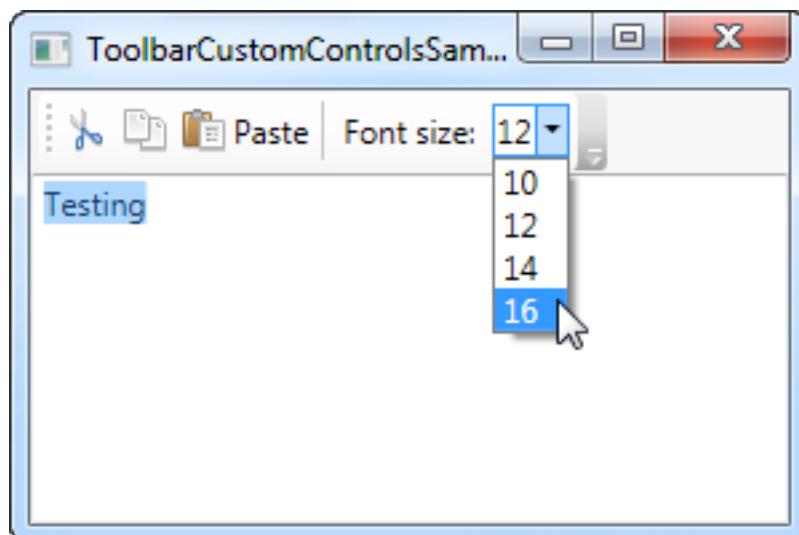
Another thing introduced in this example is the Separator element, which simply creates a separator between two sets of toolbar items. As you can see from the example, it's very easy to use!

```
<Window x:Class="WpfTutorialSamples.Common_interface_controls.ToolbarCustomControlsSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ToolbarCustomControlsSample" Height="200" Width="300">
    <DockPanel>
        <ToolBarTray DockPanel.Dock="Top">
            <ToolBar>
                <Button Command="Cut" ToolTip="Cut selection to Windows Clipboard.">
                    <Image Source="/WpfTutorialSamples;component/Images/cut.png" />
                </Button>
```

```

        <Button Command="Copy" ToolTip="Copy selection to
Windows Clipboard.">
            <Image Source
            =" /WpfTutorialSamples;component/Images/copy.png" />
        </Button>
        <Button Command="Paste" ToolTip="Paste from Windows
Clipboard.">
            <StackPanel Orientation="Horizontal">
                <Image Source
                =" /WpfTutorialSamples;component/Images/paste.png" />
                <TextBlock Margin="3,0,0,0">Paste</
TextBlock>
            </StackPanel>
        </Button>
        <Separator />
        <Label>Font size:</Label>
        <ComboBox>
            <ComboBoxItem>10</ComboBoxItem>
            <ComboBoxItem IsSelected="True">12</
ComboBoxItem>
            <ComboBoxItem>14</ComboBoxItem>
            <ComboBoxItem>16</ComboBoxItem>
        </ComboBox>
    </ToolBar>
</ToolBarTray>
<TextBox AcceptsReturn="True" />
</DockPanel>
</Window>

```



### 1.12.3.5. Summary

Creating interfaces with toolbars is very easy in WPF, with the flexible ToolBar control. You can do things that previously required 3rd party toolbar controls and you can even do it without much extra effort.

## 1.12.4. The WPF StatusBar control

With the top of the application window usually occupied by the main menu and/or toolbars, described in previous chapters, the bottom part of the window is usually the home of the status bar. The status bar is used to show various information about the current state of the application, like cursor position, word count, progress of tasks and so on. Fortunately for us, WPF comes with a nice StatusBar control, making it very easy to add status bar functionality to your applications.

Let's start off with a very basic example:

```
<Window x:Class
    ="WpfTutorialSamples.Common_interface_controls.StatusBarSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StatusBarSample" Height="150" Width="300">
    <DockPanel>
        <StatusBar DockPanel.Dock="Bottom">
            <StatusBarItem>
                <TextBlock Name="lblCursorPosition" />
            </StatusBarItem>
        </StatusBar>
        <TextBox AcceptsReturn="True" Name="txtEditor"
SelectionChanged="txtEditor_SelectionChanged" />
    </DockPanel>
</Window>

using System;
using System.Windows;

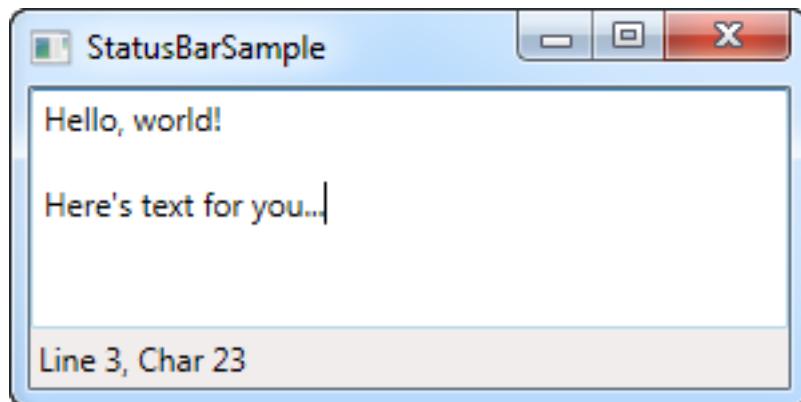
namespace WpfTutorialSamples.Common_interface_controls
{
    public partial class StatusBarSample : Window
    {
        public StatusBarSample()
        {
            InitializeComponent();
        }

        private void txtEditor_SelectionChanged(object sender,
RoutedEventArgs e)
        {
    
```

```

        int row =
txtEditor.GetLineIndexFromCharacterIndex(txtEditor.CaretIndex);
        int col = txtEditor.CaretIndex -
txtEditor.GetCharacterIndexFromLineIndex(row);
        lblCursorPosition.Text = "Line " + (row + 1) + ", Char "
+ (col + 1);
    }
}

```



It's all very simple - a **TextBlock** control that shows the current cursor position, just like in pretty much any other application that allows you to edit text. In this very basic form, the **StatusBar** could just as easily have been a panel with a set of controls on it, but the real advantage of the **StatusBar** comes when we need to divide it into several areas of information.

#### 1.12.4.1. Advanced StatusBar example

Let's try a more advanced example of using the **StatusBar**. The first thing we want to do is to make the **StatusBar** use another panel for the layout. By default, it uses the **DockPanel**, but when we want a more complex layout, with columns that adjusts its width in a certain way and aligned content, the **Grid** is a much better choice.

We'll divide the **Grid** into three areas, with the left and right one having a fixed width and the middle column automatically taking up the remaining space. We'll also add columns in between for **Separator** controls. Here's how it looks now:

```

<Window x:Class
    ="WpfTutorialSamples.Common_interface_controls.StatusBarAdvancedSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StatusBarAdvancedSample" Height="150" Width="400">

```

```

<DockPanel>
    <StatusBar DockPanel.Dock="Bottom">
        <StatusBar.ItemsPanel>
            <ItemsPanelTemplate>
                <Grid>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="100" />
                        <ColumnDefinition Width="Auto" />
                        <ColumnDefinition Width="*" />
                        <ColumnDefinition Width="Auto" />
                        <ColumnDefinition Width="100" />
                    </Grid.ColumnDefinitions>
                </Grid>
            </ItemsPanelTemplate>
        </StatusBar.ItemsPanel>
        <StatusBarItem>
            <TextBlock Name="lblCursorPosition" />
        </StatusBarItem>
        <Separator Grid.Column="1" />
        <StatusBarItem Grid.Column="2">
            <TextBlock Text="c:\path\of\current\file.txt" />
        </StatusBarItem>
        <Separator Grid.Column="3" />
        <StatusBarItem Grid.Column="4">
            <ProgressBar Value="50" Width="90" Height="16" />
        </StatusBarItem>
    </StatusBar>
    <TextBox AcceptsReturn="True" Name="txtEditor"
SelectionChanged="txtEditor_SelectionChanged" />
</DockPanel>
</Window>

using System;
using System.Windows;

namespace WpfTutorialSamples.Common_interface_controls
{
    public partial class StatusBarAdvancedSample : Window
    {
        public StatusBarAdvancedSample()

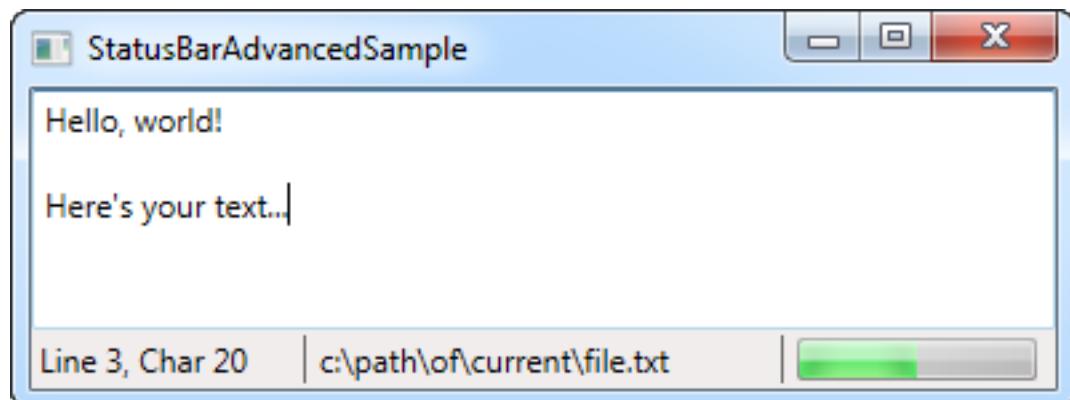
```

```

    {
        InitializeComponent();
    }

    private void txtEditor_SelectionChanged(object sender,
RoutedEventArgs e)
    {
        int row =
txtEditor.GetLineIndexFromCharacterIndex(txtEditor.CaretIndex);
        int col = txtEditor.CaretIndex -
txtEditor.GetCharacterIndexFromLineIndex(row);
        lblCursorPosition.Text = "Line " + (row + 1) + ", Char "
+ (col + 1);
    }
}

```



As you can see, I've added a bit of sample information, like the fake filename in the middle column and the progress bar to the right, showing a static value for now. You could easily make this work for real though, and it gives a pretty good idea on what you can do with the StatusBar control.

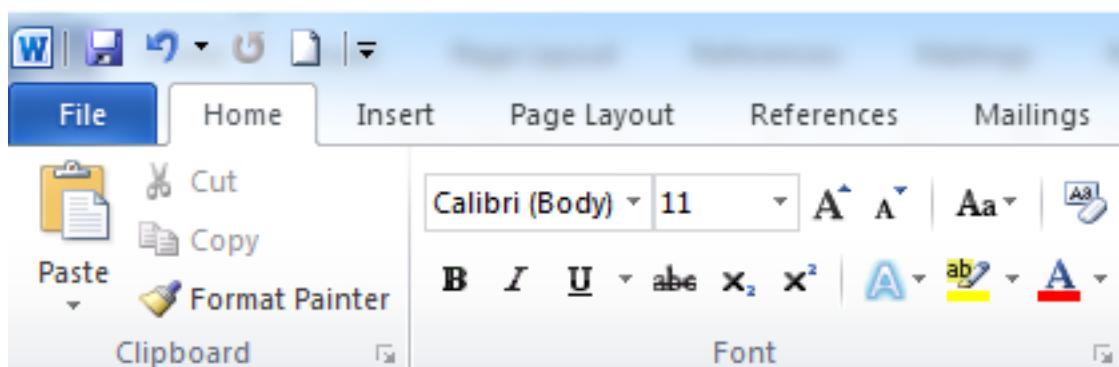
#### 1.12.4.2. Summary

Once again, WPF makes it easy to get standard Windows functionality, in this case the StatusBar, integrated into your applications.

You can even place other controls than the ones used in these examples, like buttons, combo boxes and so on, but please be aware that since the StatusBar doesn't apply any special rendering to these controls when hosting them, it might not look as you would expect it to for controls in a status bar. This can be handled with custom styling if you need it though, a subject discussed elsewhere in this tutorial.

## 1.12.5. The Ribbon control

The Ribbon interface was invented by Microsoft and first used in Office 2007. It combines the original menu and toolbar(s) into one control, with various functions grouped into tabs and groups. The most important purpose was to make it easier for the user to discover all the functionality, instead of hiding it in long menus. The Ribbon also allows for prioritization of functionality, with the ability to use different sizes of buttons.



WPF doesn't come with a built-in Ribbon control, but Microsoft has released one that you can download and use for free, as long as you promise to follow their implementation guide when using it. You can read much more about it at [MSDN](#), where you'll also find a [download link](#) for the Ribbon control.

### 1.12.5.1. Summary

You can download and use a Microsoft created Ribbon control, but it's not yet a part of the .NET framework by default. Once it becomes an integrated part of the framework, we'll dig into it here at this tutorial. In the meantime, if you're looking for a more complete Ribbon implementation, you might want to look at some 3rd party alternatives - there are plenty of them, from some of the big WPF control vendors.

## 1.13. Rich Text controls

---

### 1.13.1. Introduction to WPF Rich Text controls

In other UI frameworks like WinForms, displaying large amounts of richly formatted text has been somewhat of a problem. Sure, you could load a file into a RichTextBox or you could create a WebBrowser object and load a local or remote web page, but specifying larger amounts of rich text in design-time wasn't really possible. It seems that Microsoft wanted to remedy that in WPF and even go beyond just simple viewing of the text.

The FlowDocument does indeed render rich text, and that even includes images, lists and tables, and elements can be floated, adjusted and so on, and using a FlowDocument, you can specify rich text in design-time as if it were HTML (thanks to XAML) and have it rendered directly in your WPF application.

The FlowDocument doesn't stand alone. Instead, it uses one of several built-in wrappers, which controls how the FlowDocument is laid out and whether the content can be edited by the user or not. WPF includes three controls for rendering a FlowDocument in read-only mode, which all has easy support for zooming and printing:

**FlowDocumentScrollView** - the simplest wrapper around a FlowDocument, which simply displays the document as one long document of text which you can scroll in.

**FlowDocumentPageViewer** - this wrapper will automatically split your document into pages, which the user can navigate back and forth between.

**FlowDocumentReader** - a combination of the *FlowDocumentScrollView* and the *FlowDocumentPageViewer*, which will let the user decide between the two rendering modes. It also offers the ability AND the interface to search in the document.

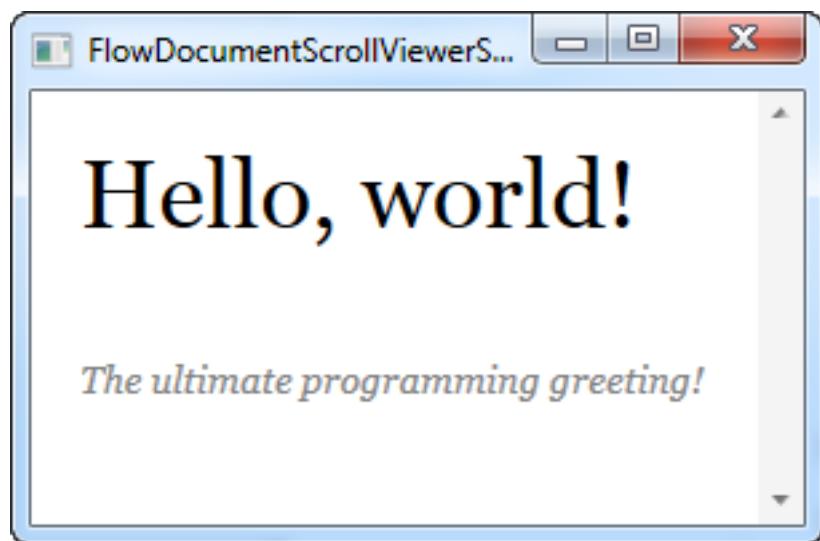
The FlowDocument is normally read-only, but put it inside of a **RichTextBox** control (described later in this tutorial) and you can now edit the text, much like in real word processors like Microsoft Word.

Read on through the next chapters, where we'll discuss all the wrappers that you can use with a FlowDocument, both read-only and editable. After that, we'll look into all of the possibilities you have when creating rich documents using the FlowDocument, including tables, lists, images and much more.

## 1.13.2. The FlowDocumentScrollViewer control

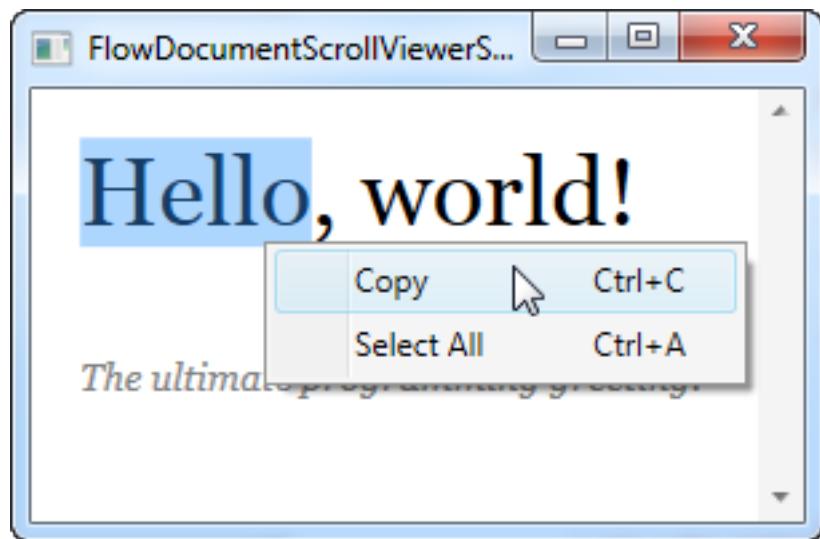
In the range of FlowDocument wrappers, discussed in the introduction, the FlowDocumentScrollViewer is the simplest one. It simply allows the users to scroll to long documents, using regular scrollbars. Since this is our first meeting with the FlowDocument used in any form, we'll start off with a basic "Hello World!" example, and besides the use of FlowDocumentScrollViewer, this article will also cover several concepts common between all of the wrappers. Here's the first example:

```
<Window x:Class
        ="WpfTutorialSamples.Rich_text_controls.FlowDocumentScrollViewerSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FlowDocumentScrollViewerSample" Height="200" Width
        ="300">
    <Grid>
        <FlowDocumentScrollViewer>
            <FlowDocument>
                <Paragraph FontSize="36">Hello, world!</Paragraph>
                <Paragraph FontStyle="Italic" TextAlignment="Left"
                FontSize="14" Foreground="Gray">The ultimate programming greeting!<
                Paragraph>
            </FlowDocument>
        </FlowDocumentScrollViewer>
    </Grid>
</Window>
```



Notice how easy it was to specify the text, using simple markup tags, in this case the **Paragraph** tag. Now you might argue that this could have been achieved with a couple of TextBlock controls, and you would be absolutely right, but even with an extremely basic example like this, you get a bit of added functionality for

free: You can select the text and copy it to the clipboard. It'll look like this:



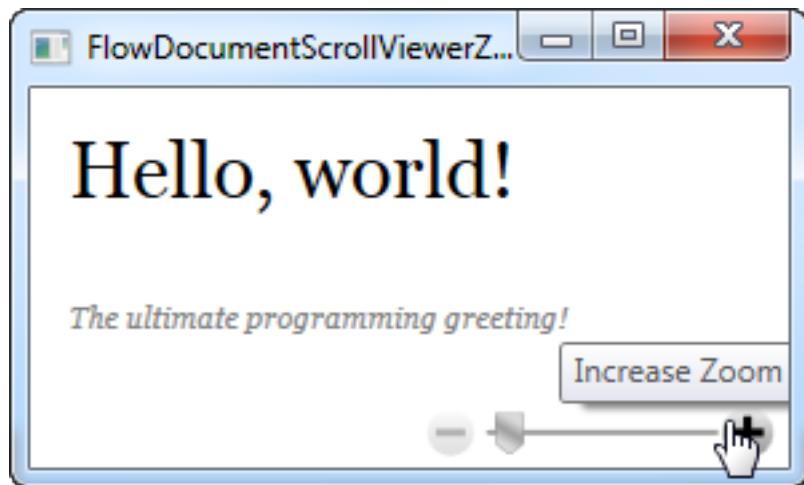
#### 1.13.2.1. Zooming and scrollbar visibility

As previously mentioned, all of the FlowDocument wrappers support zooming out of the box. With the example above, you can simply hold down the Ctrl key while using the mouse wheel to zoom in and out. This might not be obvious to your end users though, so you can help them by displaying the built-in toolbar of the FlowDocumentScrollView, which has controls that will allow you to change the zoom level. Just set the **IsToolBarVisible** property to true on the FlowDocumentScrollView, and you're good to go, as you can see in the next example:

```
<Window x:Class
        ="WpfTutorialSamples.Rich_text_controls.FlowDocumentScrollViewZoomSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FlowDocumentScrollViewZoomSample" Height="180" Width
        ="300">
    <Grid>
        <FlowDocumentScrollView IsToolBarVisible="True" Zoom="80"
        ScrollViewer.VerticalScrollBarVisibility="Auto">
            <FlowDocument>
                <Paragraph FontSize="36">Hello, world!</Paragraph>
                <Paragraph FontStyle="Italic" TextAlignment="Left"
                FontSize="14" Foreground="Gray">The ultimate programming greeting!</
                Paragraph>
            </FlowDocument>
        </FlowDocumentScrollView>
    </Grid>

```

```
</Grid>  
</Window>
```



Now the user can control the zoom level using the slider and the buttons in the toolbar below the document. Notice also that we changed the default zoom level, using the **Zoom** property - it defines the zoom level in percentages, so in this case, the text is zoomed out to 80% by default.

The last thing I changed in this example, in comparison to the first one, is the use of the **ScrollView.VerticalScrollBarVisibility** property. By setting it to **Auto**, the scrollbars will be invisible until the content actually goes beyond the available space, which is usually what you want.

#### 1.13.2.2. Text alignment

You may have noticed that I specifically used the **TextAlignment** property in the above examples. That's because the text is rendered justified by default, in a WPF FlowDocument, meaning that each line of text is stretched to cover the entire available width, if needed. As you can see, this can be changed, either on a single paragraph or globally for the entire document by setting the same property on the FlowDocument element.

However, in many situations, justified text makes sense, but it can result in some very bad layout, with very excessive amounts of whitespace on lines where a linebreak is inserted right before a very long word.

The following example will illustrate that, as well as provide a solution that will help remedy the problem. By using the **IsOptimalParagraphEnabled** property in combination with the **IsHyphenationEnabled** property, you will give WPF a better chance of laying out the text in the best possible way.

**IsOptimalParagraphEnabled** allows WPF to look ahead in your text, to see if it would make more sense to break the text in a different position than right at the moment where it runs out of space.

**IsHyphenationEnabled** allows WPF to split your words with a hyphen, if it would allow for a more natural layout of the text.

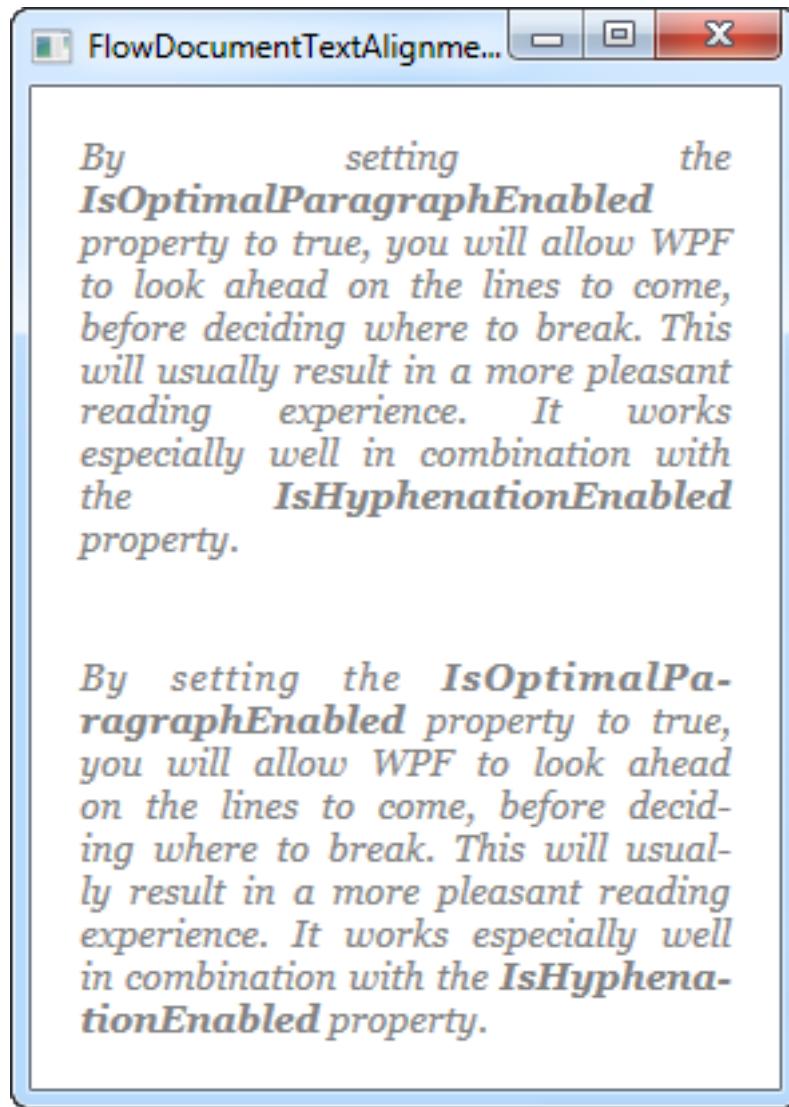
In the next example, I've rendered the same text twice - one without these properties, and one with. The difference is quite obvious:

```

<Window x:Class
        ="WpfTutorialSamples.Rich_text_controls.FlowDocumentTextAlignmentSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FlowDocumentTextAlignmentSample" Height="400" Width
        ="330">
    <StackPanel>
        <FlowDocumentScrollViewer
            ScrollViewer.VerticalScrollBarVisibility="Auto">
            <FlowDocument>
                <Paragraph FontStyle="Italic" FontSize="14"
                    Foreground="Gray">
                    By setting the
                    <Bold>IsOptimalParagraphEnabled</Bold> property
                    to true,
                    you will allow WPF to look ahead on the lines
                    to come, before deciding
                    where to break. This will usually result in a
                    more pleasant reading
                    experience. It works especially well in
                    combination with the
                    <Bold>IsHyphenationEnabled</Bold> property.
                </Paragraph>
            </FlowDocument>
        </FlowDocumentScrollViewer>
        <FlowDocumentScrollViewer
            ScrollViewer.VerticalScrollBarVisibility="Auto">
            <FlowDocument IsOptimalParagraphEnabled="True"
                IsHyphenationEnabled="True">
                <Paragraph FontStyle="Italic" FontSize="14"
                    Foreground="Gray">
                    By setting the <Bold>IsOptimalParagraphEnabled
                </Bold> property to true,
                    you will allow WPF to look ahead on the lines
                    to come, before deciding
                    where to break. This will usually result in a
                    more pleasant reading
                    experience. It works especially well in
                    combination with the

```

```
<Bold>IsHyphenationEnabled</Bold> property.  
</Paragraph>  
</FlowDocument>  
</FlowDocumentScrollViewer>  
</StackPanel>  
</Window>
```



`IsOptimalParagraphEnabled` is not enabled by default because it does require a bit more CPU power when rendering the text, especially if the window is frequently resized. For most situations this shouldn't be a problem though.

If you have a lot of `FlowDocument` instances in your application and you prefer this optimal rendering method, you can enable it on all of your `FlowDocument` instances by specifying a global style that enables it, in your `App.xaml`. Here's an example:

```
<Application x:Class="WpfTutorialSamples.App"  
    xmlns
```

```
= "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    StartupUri="Rich text
controls/FlowDocumentTextAlignmentSample.xaml">
<Application.Resources>
    <Style TargetType="FlowDocument">
        <Setter Property="IsOptimalParagraphEnabled" Value="True" />
        <Setter Property="IsHyphenationEnabled" Value="True" />
    </Style>
</Application.Resources>
</Application>
```

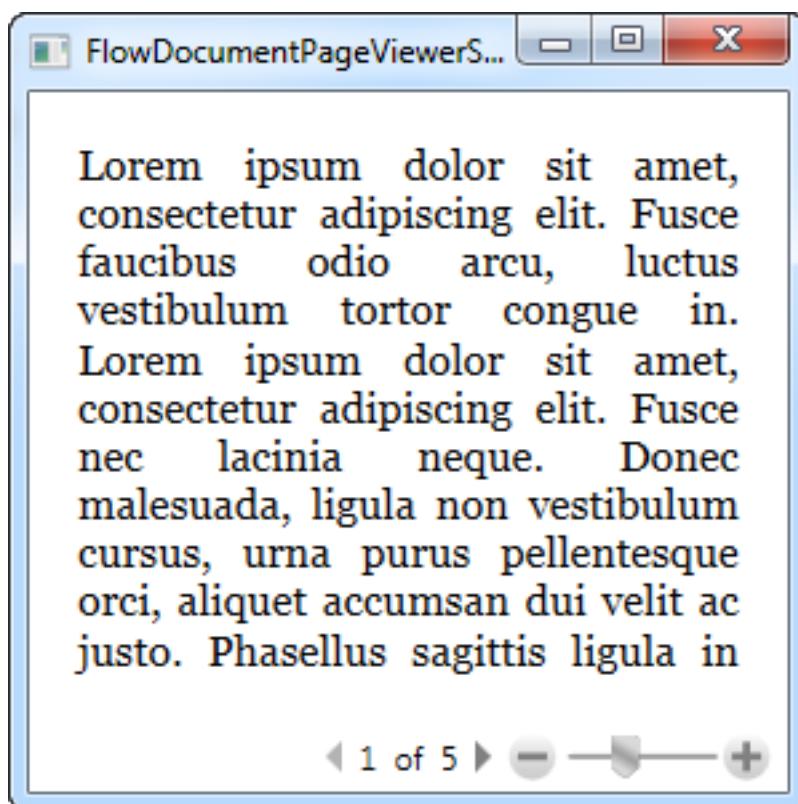
### 1.13.3. The FlowDocumentPageViewer control

In the previous article, we discussed the FlowDocumentScrollView, along with some more general FlowDocument related techniques. In this article, we'll focus on the **FlowDocumentPageViewer** which, instead of just offering a scroll text when the text gets longer than the available space, divides the entire document up into pages. This allows you to navigate from page to page, giving a more book-like reading experience.

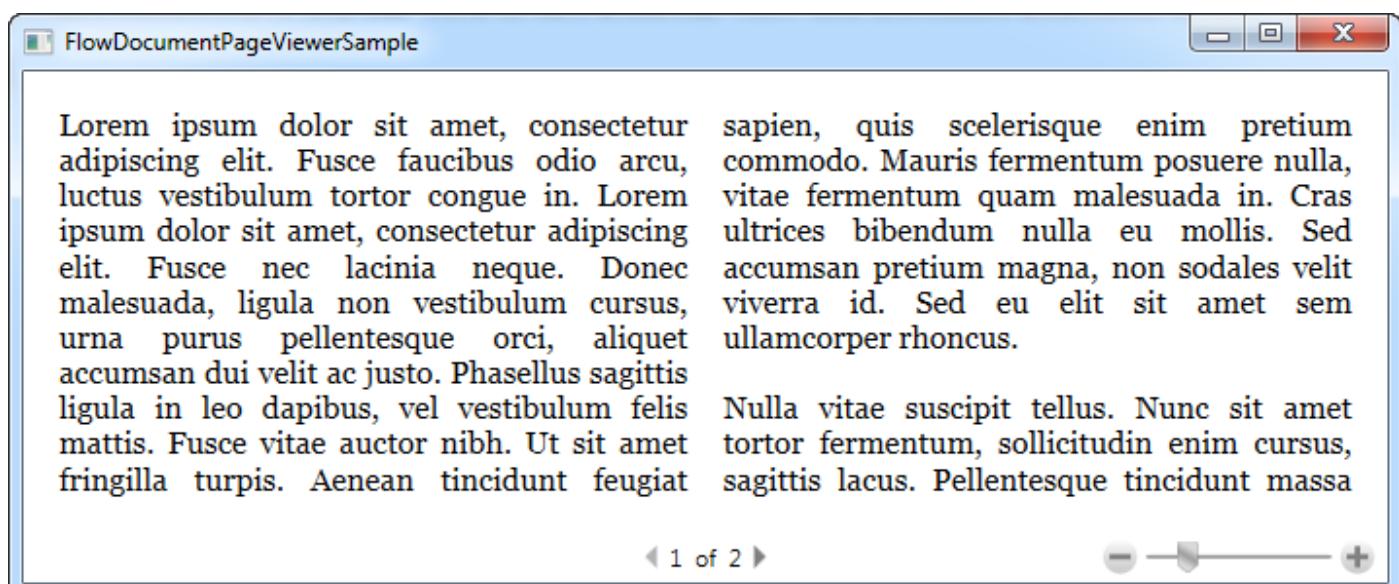
We'll start off with a simple example, where we can see how the FlowDocumentPageViewer control handles our Lorem Ipsum test text:

```
<Window x:Class="WpfTutorialSamples.Rich_text_controls.FlowDocumentPageViewerSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FlowDocumentPageViewerSample" Height="300" Width="300">
    <Grid>
        <FlowDocumentPageViewer>
            <FlowDocument>
                <Paragraph>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce faucibus odio arcu, luctus vestibulum tortor congue in. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce nec lacinia neque. Donec malesuada, ligula non vestibulum cursus, urna purus pellentesque orci, aliquet accumsan dui velit ac justo. Phasellus sagittis ligula in leo dapibus, vel vestibulum felis mattis. Fusce vitae auctor nibh. Ut sit amet fringilla turpis. Aenean tincidunt feugiat sapien, quis scelerisque enim pretium commodo. Mauris fermentum posuere nulla, vitae fermentum quam malesuada in. Cras ultrices bibendum nulla eu mollis. Sed accumsan pretium magna, non sodales velit viverra id. Sed eu elit sit amet sem ullamcorper rhoncus.</Paragraph>
                <Paragraph>Nulla vitae suscipit tellus. Nunc sit amet tortor fermentum, sollicitudin enim cursus, sagittis lacus. Pellentesque tincidunt massa nisl, nec tempor nulla consequat a. Proin pharetra neque vel dolor congue, at condimentum arcu varius. Sed vel luctus enim. Curabitur eleifend dui et arcu faucibus, sit amet vulputate libero suscipit. Vestibulum ultrices nisi id metus ultrices, eu ultricies ligula rutrum. Phasellus rhoncus aliquam pretium. Quisque in nunc erat. Etiam mollis turpis cursus, sagittis felis vel, dignissim risus. Ut at est nec tellus lobortis venenatis. Fusce elit mi, gravida sed tortor at, faucibus interdum felis. Phasellus porttitor dolor in nunc pellentesque, eu hendrerit nulla porta. Vestibulum cursus placerat
```

```
elit. Nullam malesuada dictum venenatis. Interdum et malesuada fames ac  
ante ipsum primis in faucibus.</Paragraph>  
</FlowDocument>  
</FlowDocumentPageViewer>  
</Grid>  
</Window>
```



Notice how the long text is cut off, and in the bottom, you can navigate between pages. This is not all that the FlowDocumentPageViewer will do for you though - just look what happens when we make the window wider:



Instead of just stretching the text indefinitely, the FlowDocumentPageViewer now divides your text up into columns, to prevent lines from becoming too long. Besides looking good, this also increases the readability, as texts with very long lines are harder to read. The page count is of course automatically adjusted, bringing the amount of pages down from 5 to 2.

The FlowDocument class has a range of properties that will allow you to control how and when they are used. Using them is simple, but a complete example goes beyond the scope of this tutorial. Instead, have a look at this MSDN article, where several properties are used in a nice example: [How to: Use FlowDocument Column-Separating Attributes](#).

#### 1.13.3.1. Searching

As you're about to see in the next chapter, the FlowDocumentReader wrapper supports searching right out of the box, with search controls in the toolbar and everything. However, all of the three read-only FlowDocument wrappers which will be discussed in this tutorial does in fact support searching, it just has to be manually invoked for the first two (FlowDocumentScrollView and FlowDocumentPageViewer).

All three viewers support the **Ctrl+F** keyboard shortcut for initiating a search, but if you want this to be accessible from e.g. a button as well, you just have to call the Find() method. Here's an example:

```
<Window x:Class="WpfTutorialSamples.Rich_text_controls.FlowDocumentSearchSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FlowDocumentSearchSample" Height="300" Width="580">
    <DockPanel>
        <WrapPanel DockPanel.Dock="Top">
            <Button Name="btnSearch" Click="btnSearch_Click">Search</Button>
        </WrapPanel>
        <FlowDocumentPageViewer Name="fdViewer">
            <FlowDocument>
                <Paragraph>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce faucibus odio arcu, luctus vestibulum tortor congue in. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce nec lacinia neque. Donec malesuada, ligula non vestibulum cursus, urna purus pellentesque orci, aliquet accumsan dui velit ac justo. Phasellus sagittis ligula in leo dapibus, vel vestibulum felis mattis. Fusce vitae auctor nibh. Ut sit amet fringilla turpis. Aenean tincidunt feugiat sapien, quis scelerisque enim pretium commodo. Mauris fermentum posuere nulla, vitae fermentum quam malesuada in. Cras ultrices bibendum nulla eu mollis. Sed accumsan pretium magna, non sodales velit viverra
```

```

id. Sed eu elit sit amet sem ullamcorper rhoncus.</Paragraph>
    <Paragraph>Nulla vitae suscipit tellus. Nunc sit
amet tortor fermentum, sollicitudin enim cursus, sagittis lacus.
Pellentesque tincidunt massa nisl, nec tempor nulla consequat a. Proin
pharetra neque vel dolor congue, at condimentum arcu varius. Sed vel
luctus enim. Curabitur eleifend dui et arcu faucibus, sit amet vulputate
libero suscipit. Vestibulum ultrices nisi id metus ultrices, eu
ultricies ligula rutrum. Phasellus rhoncus aliquam pretium. Quisque in
nunc erat. Etiam mollis turpis cursus, sagittis felis vel, dignissim
risus. Ut at est nec tellus lobortis venenatis. Fusce elit mi, gravida
sed tortor at, faucibus interdum felis. Phasellus porttitor dolor in
nunc pellentesque, eu hendrerit nulla porta. Vestibulum cursus placerat
elit. Nullam malesuada dictum venenatis. Interdum et malesuada fames ac
ante ipsum primis in faucibus.</Paragraph>
    </FlowDocument>
</FlowDocumentPageViewer>
</DockPanel>
</Window>

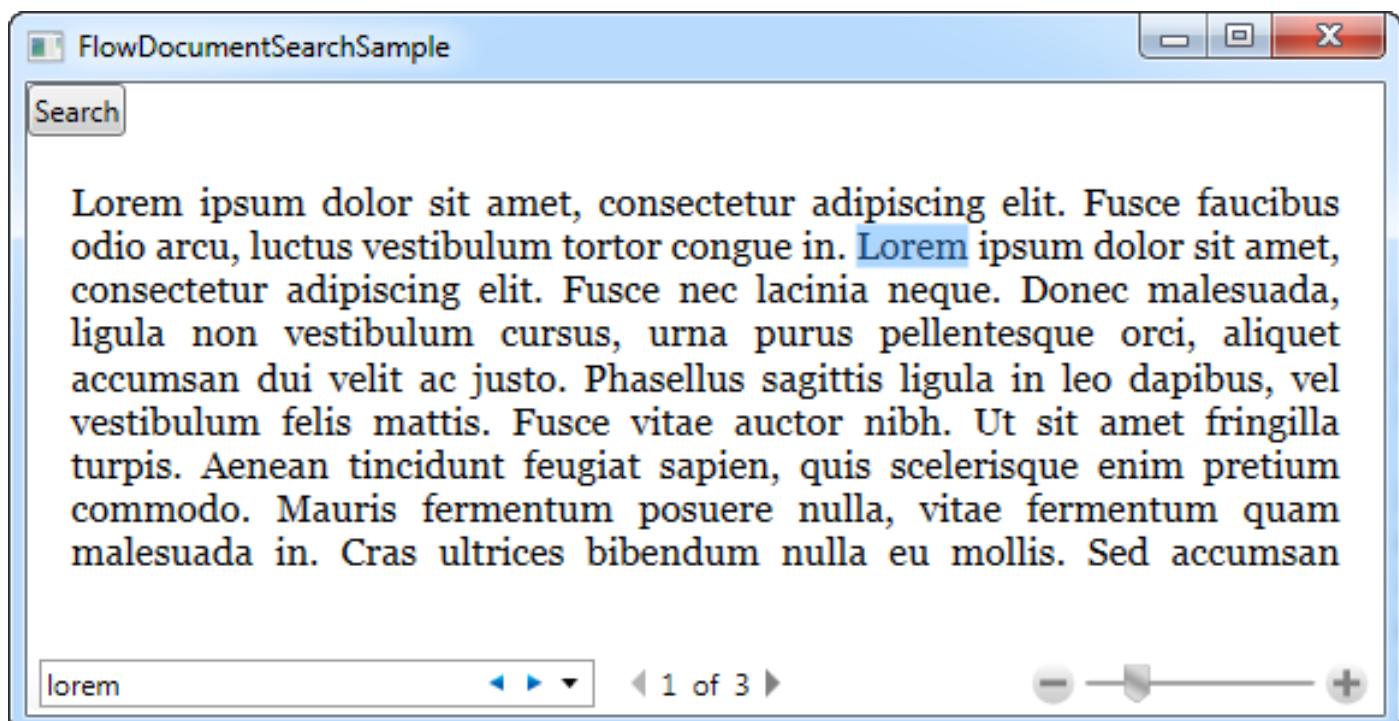
using System;
using System.Windows;

namespace WpfTutorialSamples.Rich_text_controls
{
    public partial class FlowDocumentSearchSample : Window
    {
        public FlowDocumentSearchSample()
        {
            InitializeComponent();
        }

        private void btnSearch_Click(object sender, RoutedEventArgs e)
        {
            fdViewer.Find();
        }
    }
}

```

Simply press our dedicated **Search** button or the keyboard shortcut (Ctrl+F) and you have search functionality in the FlowDocumentPageViewer. As mentioned, this works for both

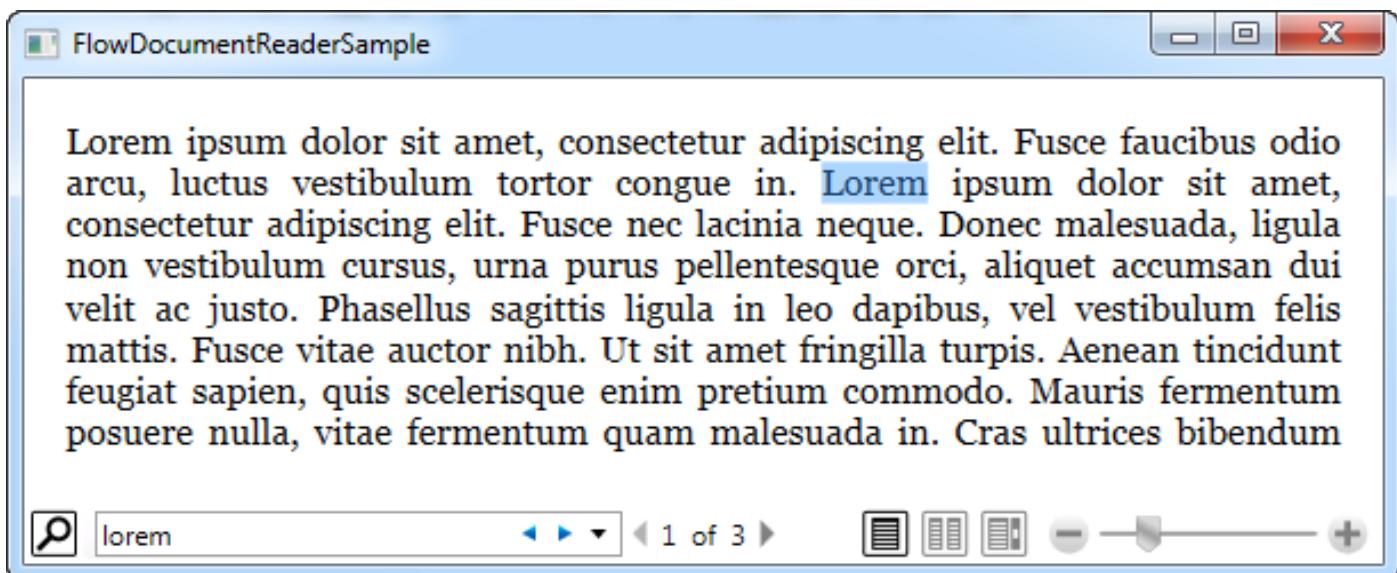


FlowDocumentScrollView and FlowDocumentPageViewer (FlowDocumentPageReader has a search button by default), **but make sure that the search box has enough horizontal room on the toolbar - otherwise you won't see it when you invoke the Find() command!**

#### 1.13.4. The FlowDocumentReader control

The **FlowDocumentReader** is definitely the most advanced read-only wrapper that you can place around a **FlowDocument**. It offers buttons that allows the end user to toggle between the rendering modes offered by the **FlowDocumentScrollView** and the **FlowDocumentPageViewer**, as well as out-of-the-box document searching and of course controls for zooming in and out.

All of this functionality also makes the **FlowDocumentReader** the heaviest of the three read-only wrappers, but this should hardly be an issue with most regularly sized documents. Here's an example of how the **FlowDocumentReader** might look:



This screenshot is taken in the page-based view, which is the default. You can switch between the view modes using the buttons to the left of the zoom controls. In the left part of the toolbar, you have the controls for searching through the document, as I have done here on the screenshot.

Here's the code that will give you the above result:

```
<Window x:Class="WpfTutorialSamples.Rich_text_controls.FlowDocumentReaderSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FlowDocumentReaderSample" Height="250" Width="550">
    <Grid>
        <FlowDocumentReader>
            <FlowDocument>
                <Paragraph>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce faucibus odio arcu, luctus vestibulum tortor congue in. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
```

Fusce nec lacinia neque. Donec malesuada, ligula non vestibulum cursus, urna purus pellentesque orci, aliquet accumsan dui velit ac justo. Phasellus sagittis ligula in leo dapibus, vel vestibulum felis mattis. Fusce vitae auctor nibh. Ut sit amet fringilla turpis. Aenean tincidunt feugiat sapien, quis scelerisque enim pretium commodo. Mauris fermentum posuere nulla, vitae fermentum quam malesuada in. Cras ultrices bibendum nulla eu mollis. Sed accumsan pretium magna, non sodales velit viverra id. Sed eu elit sit amet sem ullamcorper rhoncus.

<Paragraph> Nulla vitae suscipit tellus. Nunc sit amet tortor fermentum, sollicitudin enim cursus, sagittis lacus. Pellentesque tincidunt massa nisl, nec tempor nulla consequat a. Proin pharetra neque vel dolor congue, at condimentum arcu varius. Sed vel luctus enim. Curabitur eleifend dui et arcu faucibus, sit amet vulputate libero suscipit. Vestibulum ultrices nisi id metus ultrices, eu ultricies ligula rutrum. Phasellus rhoncus aliquam pretium. Quisque in nunc erat. Etiam mollis turpis cursus, sagittis felis vel, dignissim risus. Ut at est nec tellus lobortis venenatis. Fusce elit mi, gravida sed tortor at, faucibus interdum felis. Phasellus porttitor dolor in nunc pellentesque, eu hendrerit nulla porta. Vestibulum cursus placerat elit. Nullam malesuada dictum venenatis. Interdum et malesuada fames ac ante ipsum primis in faucibus.

</FlowDocument>  
</FlowDocumentReader>  
</Grid>

</Window>

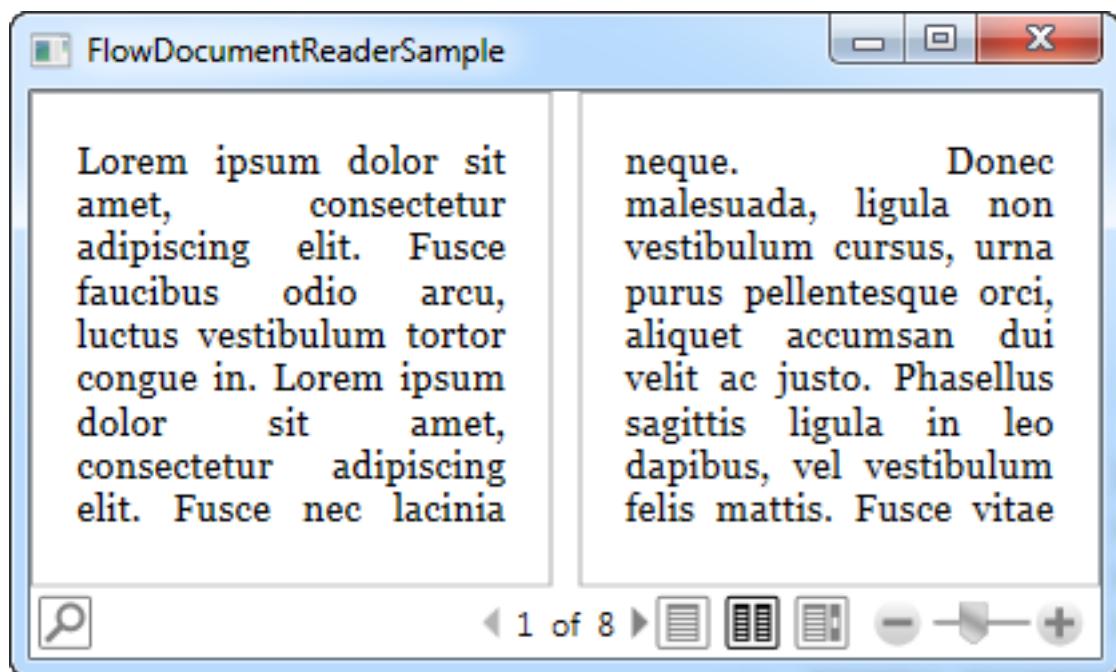
This markup will result in a window as seen on the screenshot above. Here's a screenshot where we've gone into the two-page mode as well as reduced the zoom a bit:

The FlowDocumentReader has a range of properties that can help you in controlling how it works. Here's an incomplete list of some of the most important ones:

**ViewingMode** - controls the initial viewing mode. The default is **Page**, but you can change that into **Scroll** or **TwoPage**, if you want another default view. This can still be changed by the user, unless specifically disabled.

**IsFindEnabled** - gives you the ability to disable searching in the document. If disabled, the search button will be removed from the toolbar.

**IsTwoPageViewEnabled**, **IsPageViewEnabled** and **IsScrollViewEnabled** - allows you to turn off a specific viewing mode for the reader. When set to false, this mode is no longer available for the reader and



the button is removed from the toolbar.

**Zoom** - allows you to set the default zoom level. The standard is 100%, but you can change this by using the `Zoom` property.

#### 1.13.4.1. Summary

We've now been through all the choices for a read-only `FlowDocument` wrapper, and as you can probably see, which one to choose really depends on the task at hand.

If you just want simple `FlowDocument` rendering with a scrollbar you should go with the `FlowDocumentScrollView` - it's simple and is the least space and resource consuming of the three. If you want a paged view, go with the `FlowDocumentPageViewer`, unless you want your user to be able to switch between the modes and be able to quickly search, in which case you should use the `FlowDocumentReader`.

## 1.13.5. Creating a FlowDocument from Code-behind

So far, we've been creating our FlowDocument's directly in XAML. Representing a document in XAML makes sense, because XAML is so much like HTML, which is used all over the Internet to create pages of information. However, this obviously doesn't mean that you can't create FlowDocument's from Code-behind - you absolutely can, since every element is represented by a class that you can instantiate and add with good, old C# code.

As a bare minimum example, here's our "Hello, world!" example from one of the first articles, created from Code-behind instead of XAML:

```
<Window x:Class
        ="WpfTutorialSamples.Rich_text_controls.CodeBehindFlowDocumentSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CodeBehindFlowDocumentSample" Height="200" Width="300">
    <Grid>
        <FlowDocumentScrollViewer Name="fdViewer" />
    </Grid>
</Window>

using System;
using System.Windows;
using System.Windows.Documents;
using System.Windows.Media;

namespace WpfTutorialSamples.Rich_text_controls
{
    public partial class CodeBehindFlowDocumentSample : Window
    {
        public CodeBehindFlowDocumentSample()
        {
            InitializeComponent();

            FlowDocument doc = new FlowDocument();

            Paragraph p = new Paragraph(new Run("Hello, world!"));
            p.FontSize = 36;
            doc.Blocks.Add(p);

            p = new Paragraph(new Run("The ultimate programming

```

```

greeting!"));

    p.FontSize = 14;
    p.FontStyle = FontStyles.Italic;
    p.TextAlignment = TextAlignment.Left;
    p.Foreground = Brushes.Gray;
    doc.Blocks.Add(p);

    fdViewer.Document = doc;
}
}
}

```



When compared to the small amount of XAML required to achieve the exact same thing, this is hardly impressive:

```

<FlowDocument>
    <Paragraph FontSize="36">Hello, world!</Paragraph>
    <Paragraph FontStyle="Italic" TextAlignment="Left" FontSize="14"
Foreground="Gray">The ultimate programming greeting!</Paragraph>
</FlowDocument>

```

That's beside the point here though - sometimes it just makes more sense to handle stuff from Code-behind, and as you can see, it's definitely possible.

## 1.13.6. Advanced FlowDocument content

As I already mentioned, the text presentation capabilities of WPF and the FlowDocument is very rich - you can do almost anything, and this includes stuff like lists, images and even tables. So far, we've used very basic examples of FlowDocument content, but in this article, we'll finally do a more comprehensive example.

The XAML code for the next example might look a bit overwhelming, but notice how simple it actually is - just like HTML, you can format text simply by placing them in styled paragraphs. Now have a look at the XAML. A screenshot of the result will follow directly after it:

```
<Window x:Class
        ="WpfTutorialSamples.Rich_text_controls.ExtendedFlowDocumentSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ExtendedFlowDocumentSample" Height="550" Width="500">
    <Grid>
        <FlowDocumentScrollView>
            <FlowDocument>
                <Paragraph>
                    <Image Source="http://www.wpf-
tutorial.com/images/logo.png" Width="90" Height="90" Margin="0,0,30,0" />
                    <Run FontSize="120">WPF</Run>
                </Paragraph>

                <Paragraph>
                    WPF, which stands for
                    <Bold>Windows Presentation Foundation</Bold>,
                    is Microsoft's latest approach to a GUI
                    framework, used with the .NET framework.
                    Some advantages include:
                </Paragraph>

                <List>
                    <ListItem>
                        <Paragraph>
                            It's newer and thereby more in tune
                            with current standards
                        </Paragraph>
                    </ListItem>
                </List>
            </FlowDocument>
        </FlowDocumentScrollView>
    </Grid>
</Window>
```

```

<ListItem>
    <Paragraph>
        Microsoft is using it for a lot of
new applications, e.g. Visual Studio
    </Paragraph>
</ListItem>
<ListItem>
    <Paragraph>
        It's more flexible, so you can do
more things without having to write or buy new controls
    </Paragraph>
</ListItem>
</List>

<Table CellSpacing="0">
    <TableRowGroup>
        <TableRow Background="Gainsboro"
FontWeight="Bold">
            <TableCell></TableCell>
            <TableCell>
                <Paragraph TextAlignment="Right">
>WinForms</Paragraph>
            </TableCell>
            <TableCell>
                <Paragraph TextAlignment="Right">
>WPF</Paragraph>
            </TableCell>
        </TableRow>
    </TableRowGroup>

    <TableRowGroup>
        <TableRow>
            <TableCell Background="Gainsboro"
FontWeight="Bold">
                <Paragraph>Lines of code</
Paragraph>
            </TableCell>
            <TableCell>
                <Paragraph TextAlignment="Right">
>1.718.000</Paragraph>

```

```

        </TableCell>
        <TableCell>
            <Paragraph TextAlignment="Right">
>1.542.000</Paragraph>
            </TableCell>
        </TableRow>
    </TableRowGroup>
    <TableRowGroup>
        <TableRow>
            <TableCell Background="Gainsboro">
                <Paragraph>Developers</Paragraph>
            </TableCell>
            <TableCell>
                <Paragraph TextAlignment="Right">
            </Paragraph>
        </TableRow>
    </TableRowGroup>
    <Table>
        <Row>
            <Cell>
                <Paragraph TextAlignment="Right">
            </Paragraph>
        </Cell>
        <Cell>
            <Paragraph TextAlignment="Right">
            </Paragraph>
        </Cell>
    </Row>
</Table>
<Paragraph Foreground="Silver" FontStyle="Italic">A
table of made up WinForms/WPF numbers</Paragraph>
</FlowDocument>
</FlowDocumentScrollView>
</Grid>
</Window>

```

I'm not going to go too much into details about each of the tags - hopefully they should make sense as they are.

As you can see, including lists, images and tables are pretty easy, but in fact, you can include any WPF control inside of your FlowDocument. Using the BlockUIContainer element you get access to all controls that would otherwise only be available inside of a window. Here's an example:

```
<Window x:Class
```

**WPF**, which stands for **Windows Presentation Foundation**, is Microsoft's latest approach to a GUI framework, used with the .NET framework. Some advantages include:

- It's newer and thereby more in tune with current standards
- Microsoft is using it for a lot of new applications, e.g. Visual Studio
- It's more flexible, so you can do more things without having to write or buy new controls

	<b>WinForms</b>	<b>WPF</b>
<b>Lines of code</b>	1.718.000	1.542.000
<b>Developers</b>	633.000	981.000

*A table of made up WinForms/WPF numbers*

```
= "WpfTutorialSamples.Rich_text_controls.BlockUIContainerSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:self="clr-
namespace:WpfTutorialSamples.Rich_text_controls"
    Title="BlockUIContainerSample" Height="275" Width="300">
<Window.Resources>
    <x:Array x:Key="UserArray" Type="{x:Type self:User}">
```

```

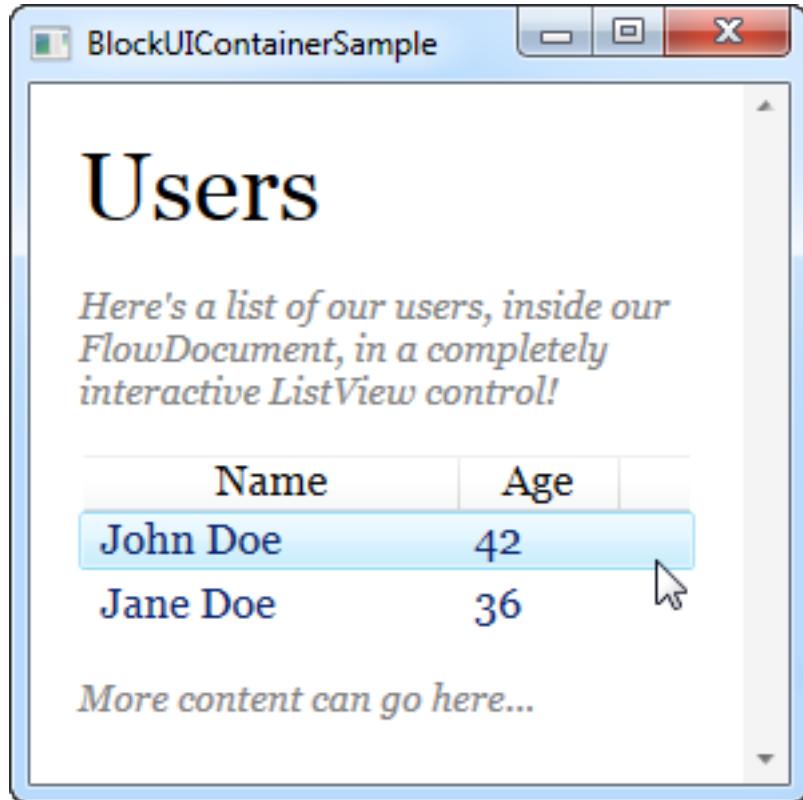
        <self:User Name="John Doe" Age="42" />
        <self:User Name="Jane Doe" Age="36" />
    </x:Array>
</Window.Resources>
<Grid>
    <FlowDocumentScrollView>
        <FlowDocument>
            <Paragraph FontSize="36" Margin="0">Users</Paragraph>
        >
            <Paragraph FontStyle="Italic" TextAlignment="Left" FontSize="14" Foreground="Gray">Here's a list of our users, inside our FlowDocument, in a completely interactive ListView control!</Paragraph>
            <BlockUIContainer>
                <ListView BorderThickness="0" ItemsSource="{StaticResource UserArray}">
                    <ListView.View>
                        <GridView>
                            <GridViewColumn Header="Name" DisplayMemberBinding="{Binding Name}" Width="150" />
                            <GridViewColumn Header="Age" DisplayMemberBinding="{Binding Age}" Width="75" />
                        </GridView>
                    </ListView.View>
                </ListView>
            </BlockUIContainer>
            <Paragraph FontStyle="Italic" TextAlignment="Left" FontSize="14" Foreground="Gray">More content can go here...</Paragraph>
        </FlowDocument>
    </FlowDocumentScrollView>
</Grid>
</Window>

```

Now we have a FlowDocument with a ListView inside of it, and as you can see from the screenshot, the ListView works just like it normally would, including selections etc. Pretty cool!

#### 1.13.6.1. Summary

By using the techniques described in the two examples of this article, pretty much anything is possible, when creating FlowDocument documents. It's excellent for presenting visual information to the end-user, as seen in many of the expensive reporting suites.



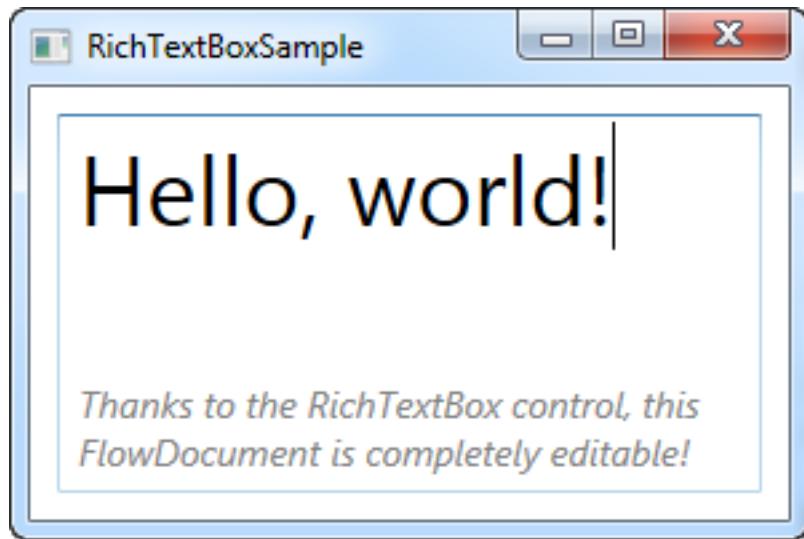
### 1.13.7. The RichTextBox control

So far, we've only looked at the read-only wrappers for the `FlowDocument`, but WPF also includes a control which makes a `FlowDocument` editable: The `RichTextBox` control.

You can add a `RichTextBox` directly to the window, without any content - in that case, it will automatically create a `FlowDocument` instance that you will be editing. Alternatively, you can wrap a `FlowDocument` instance with the `RichTextBox` and thereby control the initial content. It could look like this:

```
<Window x:Class="WpfTutorialSamples.Rich_text_controls.RichTextBoxSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="RichTextBoxSample" Height="200" Width="300">
    <Grid>
        <RichTextBox Margin="10">
            <FlowDocument>
                <Paragraph FontSize="36">Hello, world!</Paragraph>
                <Paragraph FontStyle="Italic" TextAlignment="Left" FontSize="14" Foreground="Gray">Thanks to the RichTextBox control, this FlowDocument is completely editable!</Paragraph>
            </FlowDocument>
        </RichTextBox>
    </Grid>
```

```
</Window>
```



With this example, you can start editing your rich text content straight away. However, now that the content is no longer read-only, it's obviously interesting how you can manipulate the text, as well as work with the selection. We'll look into that right now.

Another interesting aspect is of course working with the various formatting possibilities - we'll look into that in the next article, where we actually implement a small, but fully functional rich text editor.

#### 1.13.7.1. Working with text and selection

Because the RichTextBox uses a FlowDocument internally, and because the rich text format is obviously more complicated than plain text, working with text and selections are not quite as easy as for the WPF TextBox control.

The next example will provide show off a range of functionality that works with the text and/or selection in the RichTextBox control:

```
<Window x:Class  
       ="WpfTutorialSamples.Rich_text_controls.RichTextBoxTextSelectionSample"  
       xmlns  
       ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       Title="RichTextBoxTextSelectionSample" Height="300" Width  
       ="400">  
    <DockPanel>  
        <WrapPanel DockPanel.Dock="Top">  
            <Button Name="btnGetText" Click="btnGetText_Click">Get  
text</Button>  
            <Button Name="btnSetText" Click="btnSetText_Click">Set
```

```

text</Button>
    <Button Name="btnGetSelectedText" Click
="btnGetSelectedText_Click">Get sel. text</Button>
    <Button Name="btnSetSelectedText" Click
="btnSetSelectedText_Click">Replace sel. text</Button>
</WrapPanel>
<TextBox DockPanel.Dock="Bottom" Name="txtStatus" />
<RichTextBox Name="rtbEditor" SelectionChanged
="rtbEditor_SelectionChanged">
    <FlowDocument>
        <Paragraph FontSize="36">Hello, world!</Paragraph>
        <Paragraph FontStyle="Italic" TextAlignment="Left"
FontSize="14" Foreground="Gray">Thanks to the RichTextBox control, this
FlowDocument is completely editable!</Paragraph>
    </FlowDocument>
</RichTextBox>
</DockPanel>
</Window>

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;

namespace WpfTutorialSamples.Rich_text_controls
{
    public partial class RichTextBoxTextSelectionSample : Window
    {
        public RichTextBoxTextSelectionSample()
        {
            InitializeComponent();
        }

        private void btnGetText_Click(object sender, RoutedEventArgs
e)
        {
            TextRange textRange = new
TextRange(rtbEditor.Document.ContentStart,
rtbEditor.Document.ContentEnd);
            MessageBox.Show(textRange.Text);
        }
    }
}

```

```

    }

    private void btnSetText_Click(object sender, RoutedEventArgs e)
    {
        TextRange textRange = new
TextRange(rtbEditor.Document.ContentStart,
rtbEditor.Document.ContentEnd);
        textRange.Text = "Another world, another text!";
    }

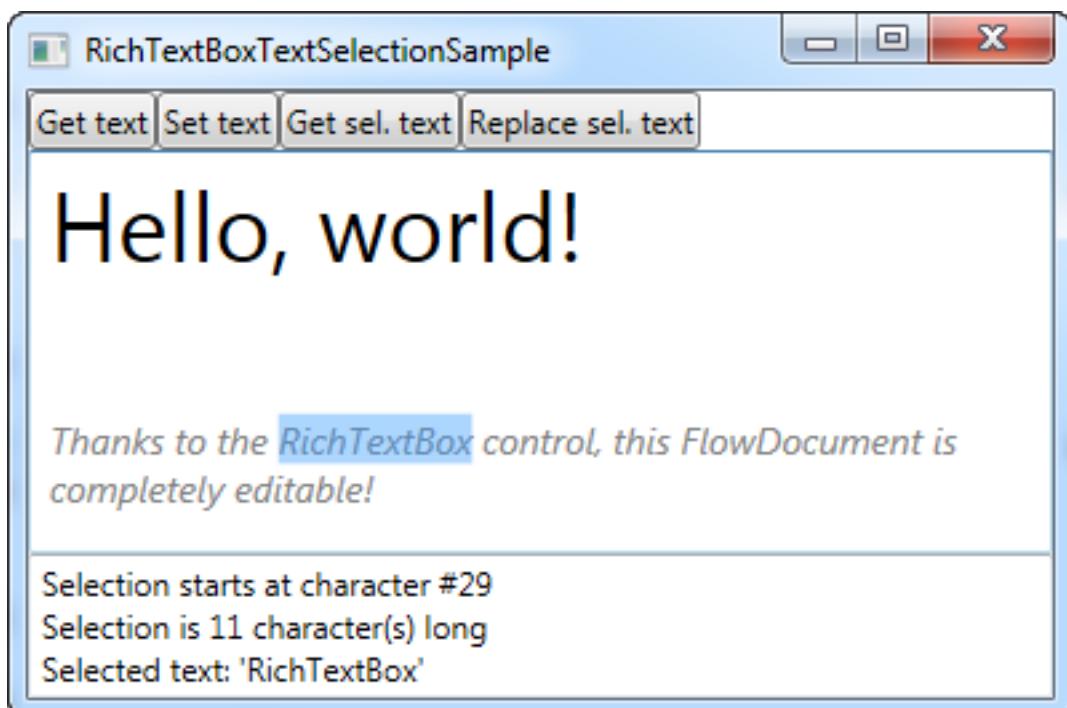
    private void btnGetSelectedText_Click(object sender,
RoutedEventArgs e)
{
    MessageBox.Show(rtbEditor.Selection.Text);
}

    private void btnSetSelectedText_Click(object sender,
RoutedEventArgs e)
{
    rtbEditor.Selection.Text = "[Replaced text]";
}

    private void rtbEditor_SelectionChanged(object sender,
RoutedEventArgs e)
{
    TextRange tempRange = new
TextRange(rtbEditor.Document.ContentStart, rtbEditor.Selection.Start);
        txtStatus.Text = "Selection starts at character #" +
tempRange.Text.Length + Environment.NewLine;
        txtStatus.Text += "Selection is " +
rtbEditor.Selection.Text.Length + " character(s) long" +
Environment.NewLine;
        txtStatus.Text += "Selected text: '" +
rtbEditor.Selection.Text + "'";
    }
}
}

```

As you can see, the markup consists of a panel of buttons, a RichTextBox and a TextBox in the bottom, to



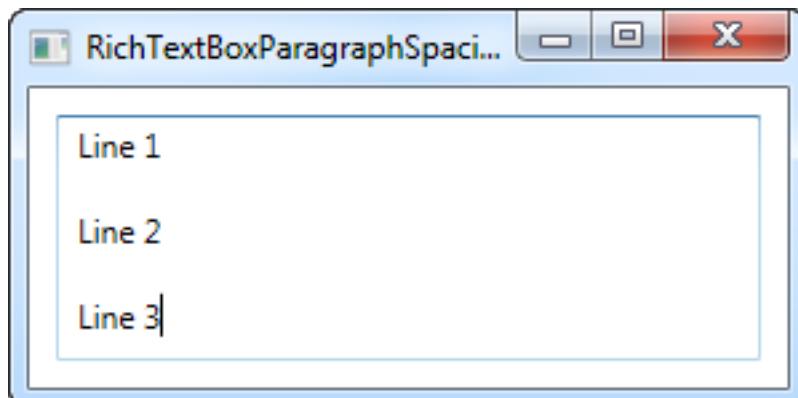
show the current selection status. Each of the four available buttons will work with the RichTextBox by either getting or setting/replacing text, to show you how that's done.

In Code-behind, we handle the four buttons click events, as well as the SelectionChanged event for the RichTextBox, which allows us to show statistics about the current selection.

Pay special attention to the fact that instead of accessing a text property directly on the RichTextBox, as we would do with a regular TextBox, we're using TextRange objects with TextPointer's from the RichTextBox to obtain text from the control or the selection in the control. This is simply how it works with RichTextBox, which, as already mentioned, doesn't work like a regular TextBox in several aspects.

#### 1.13.7.2. Paragraph spacing

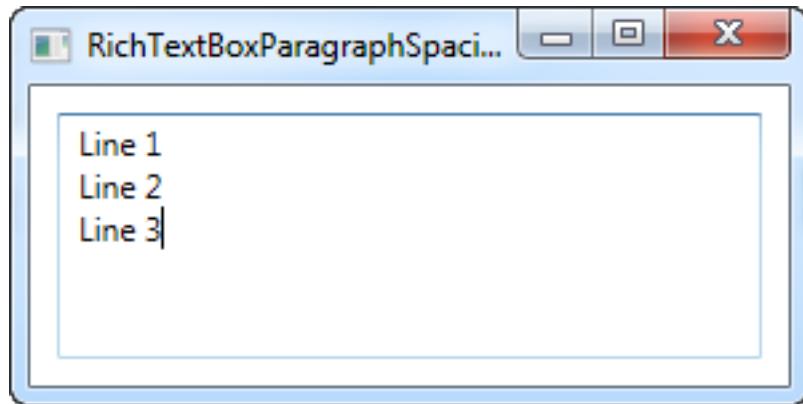
Another thing you may have noticed when working with the RichTextBox, is the fact that when you press Enter to start a new paragraph, this paragraph will leave an empty line between the old and the new paragraph. Allow me to illustrate with a screenshot, where I've entered three lines of text, each just separated by a single Enter key press:



This is normal behavior for a text editor working in paragraphs, but depending on how and where you use the RichTextBox, it might be confusing for your users that a single Enter press results in such a large amount of space between the lines.

Fortunately, it's very easy to fix. The extra spaces comes from the fact that paragraphs have a default margin bigger than zero, so fixing it is as simple as changing this property, which we can do with a style, like this:

```
<Window x:Class="WpfTutorialSamples.Rich_text_controls.RichTextBoxParagraphSpacingSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="RichTextBoxParagraphSpacingSample" Height="150" Width="300">
    <Grid>
        <RichTextBox Margin="10">
            <RichTextBox.Resources>
                <Style TargetType="{x:Type Paragraph}">
                    <Setter Property="Margin" Value="0" />
                </Style>
            </RichTextBox.Resources>
        </RichTextBox>
    </Grid>
</Window>
```



Now the lines don't have extra space around them, and if you want, you can place the style in the window or even in App.xaml, if you want it to work for more than just a single RichTextBox.

#### 1.13.7.3. Summary

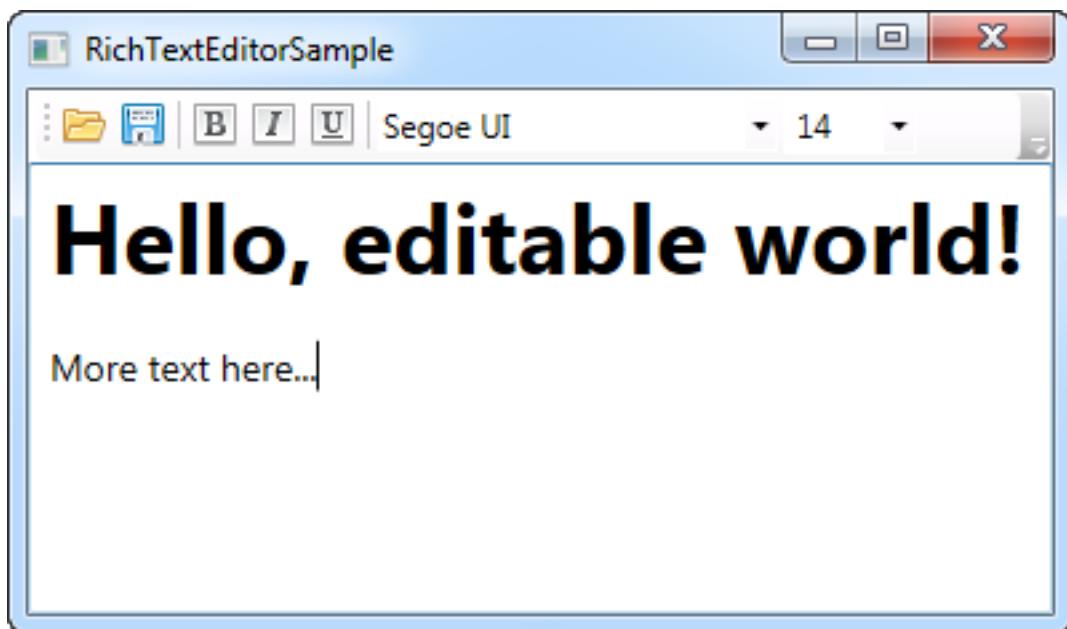
The RichTextBox is easy to use, has lots of features straight out of the box, and can easily be used if you wish to create a fully featured rich text editor. In the next article, we'll have a look at doing just that! This will also get us around important subjects such as loading and saving text from a RichTextBox and how to affect formatting of text in the control.

## 1.13.8. How-to: Creating a Rich Text Editor

This is another how-to article, inspired by how cool the RichTextBox control is and how easy it makes it to create a small but very capable rich text editor - think Windows Wordpad! While WPF makes this really easy for us, there will be a bit more XAML and C# code than usual, but that's okay. We'll go through the interesting sections one at a time, and then **in the end, I will show you the entire code listing.**

In this article, we'll be using lots of controls and techniques that we've used in other parts of the tutorial, so the explanations won't be too detailed. In case you need to freshen up on parts of it, you can always go back for the fully detailed descriptions.

As a start, let's have a look at what we're going for. This should be the final result:



### 1.13.8.1. Interface

The interface consists of a ToolBar control with buttons and combo boxes on it. There are buttons for loading and saving a document, buttons for controlling various font weight and style properties, and then two combo boxes for controlling the font family and size.

Below the toolbar is the RichTextBox control, where all the editing will be done.

### 1.13.8.2. Commands

The first thing you might notice is the use of WPF Commands, which we've already discussed previously in this article. We use the **Open** and **Save** commands from the ApplicationCommands class to load and save the document, and we use the ToggleBold, ToggleItalic and ToggleUnderline commands from the EditingCommands class for our style related buttons.

The advantage of using Commands is once again obvious, since the RichTextBox control already implements the ToggleBold, ToggleItalic and ToggleUnderline commands. This means that we don't have

to write any code for them to work. Just hook them up to the designated button and it works:

```
<ToggleButton Command="EditingCommands.ToggleBold" Name="btnBold">
    <Image Source="/WpfTutorialSamples/component/Images/text_bold.png"
Width="16" Height="16" />
</ToggleButton>
```

We also get keyboard shortcuts for free - press Ctrl+B to activate ToggleBold, Ctrl+I to activate ToggleItalic and Ctrl+U to activate ToggleUnderline.

Notice that I'm using a **ToggleButton** instead of a regular **Button** control. I want the button to be checkable, if the selection is currently bold, and that's supported through the `IsChecked` property of the `ToggleButton`. Unfortunately, WPF has no way of handling this part for us, so we need a bit of code to update the various button and combo box states. More about that later.

The Open and Save commands can't be handled automatically, so we'll have to do that as usual, with a `CommandBinding` for the Window and then an event handler in Code-behind:

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open" Executed
="Open_Executed" />
    <CommandBinding Command="ApplicationCommands.Save" Executed
="Save_Executed" />
</Window.CommandBindings>
```

I'll show you the implementation later in this article.

#### 1.13.8.3. Font family and size

To show and change the font family and size, we have a couple of combo boxes. They are populated with the system font families as well as a selection of possible sizes in the constructor for the window, like this:

```
public RichTextEditorSample()
{
    InitializeComponent();
    cmbFontFamily.ItemsSource = Fonts.SystemFontFamilies.OrderBy(f =>
f.Source);
    cmbFontSize.ItemsSource = new List<double>() { 8, 9, 10, 11, 12,
14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72 };
}
```

Once again, WPF makes it easy for us to get a list of possible fonts, by using the `SystemFontFamilies` property. Since the list of sizes is more of a suggestion, we make that `ComboBox` control editable, so that

the user may enter a custom size:

```
<ComboBox Name="cmbFontFamily" Width="150" SelectionChanged="cmbFontFamily_SelectionChanged" />
<ComboBox Name="cmbFontSize" Width="50" IsEditable="True" TextBoxBase.TextChanged="cmbFontSize_TextChanged" />
```

This also means that we'll be handling changes differently. For the font family ComboBox, we can just handle the SelectionChanged event, while we hook into the TextBoxBase.TextChanged event of the size ComboBox, to handle the fact that the user can both select and manually enter a size.

WPF handles the implementation of the Bold, Italic and Underline commands for us, but for font family and size, we'll have to manually change these values. Fortunately, it's quite easy to do, using the ApplyPropertyValue() method. The above mentioned event handlers look like this.

```
private void cmbFontFamily_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (cmbFontFamily.SelectedItem != null)

        rtbEditor.Selection.ApplyPropertyValue(Inline.FontFamilyProperty,
        cmbFontFamily.SelectedItem);
}

private void cmbFontSize_TextChanged(object sender, TextChangedEventArgs e)
{
    rtbEditor.Selection.ApplyPropertyValue(Inline.FontSizeProperty,
    cmbFontSize.Text);
}
```

Nothing too fancy here - we simply pass on the selected/entered value to the ApplyPropertyValue() method, along with the property that we wish to change.

#### 1.13.8.4. Updating the state

As previously mentioned, WPF handles the Bold, Italic and Underline commands for us, but we have to manually update the state of their buttons, since this is not a part of the Commands functionality. However, that's okay, since we also have to update the two combo boxes to reflect the current font family and size.

We want to update the state as soon as the cursor moves and/or the selection changes, and for that, the **SelectionChanged** event of the RichTextBox is perfect. Here's how we handle it:

```

private void rtbEditor_SelectionChanged(object sender, RoutedEventArgs e)
{
    object temp =
    rtbEditor.Selection.GetValue(Inline.FontWeightProperty);
    btnBold.IsChecked = (temp != DependencyProperty.UnsetValue) &&
    (temp.Equals(FontWeights.Bold));
    temp =
    rtbEditor.Selection.GetValue(Inline.FontStyleProperty);
    btnItalic.IsChecked = (temp != DependencyProperty.UnsetValue) &&
    (temp.Equals(FontStyles.Italic));
    temp =
    rtbEditor.Selection.GetValue(Inline.TextDecorationsProperty);
    btnUnderline.IsChecked = (temp != DependencyProperty.UnsetValue) &&
    (temp.Equals(TextDecorations.Underline));

    temp =
    rtbEditor.Selection.GetValue(Inline.FontFamilyProperty);
    cmbFontFamily.SelectedItem = temp;
    temp =
    rtbEditor.Selection.GetValue(Inline.FontSizeProperty);
    cmbFontSize.Text = temp.ToString();
}

```

Quite a bit of lines, but the actual job only requires a couple of lines - we just repeat them with a small variation to update each of the three buttons, as well as the two combo boxes.

The principle is quite simple. For the buttons, we use the `GetValue()` method to get the current value for a given text property, e.g. the `FontWeight`, and then we update the `.IsChecked` property depending on whether the returned value is the same as the one we're looking for or not.

For the combo boxes, we do the same thing, but instead of setting an `.IsChecked` property, we set the `SelectedItem` or `Text` properties directly with the returned values.

#### 1.13.8.5. Loading and saving a file

When handling the Open and Save commands, we use some very similar code:

```

private void Open_Executed(object sender, ExecutedRoutedEventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.Filter = "Rich Text Format (*.rtf)|*.rtf|All files (*.*)|*.*";

```

```

    if(dlg.ShowDialog() == true)
    {
        FileStream fileStream = new FileStream(dlg.FileName,
        FileMode.Open);
        TextRange range = new
        TextRange(rtbEditor.Document.ContentStart,
        rtbEditor.Document.ContentEnd);
        range.Load(fileStream, DataFormats.Rtf);
    }
}

private void Save_Executed(object sender, ExecutedRoutedEventArgs e)
{
    SaveFileDialog dlg = new SaveFileDialog();
    dlg.Filter = "Rich Text Format (*.rtf)|*.rtf|All files (*.*)|*.*";
    if(dlg.ShowDialog() == true)
    {
        FileStream fileStream = new FileStream(dlg.FileName,
        FileMode.Create);
        TextRange range = new
        TextRange(rtbEditor.Document.ContentStart,
        rtbEditor.Document.ContentEnd);
        range.Save(fileStream, DataFormats.Rtf);
    }
}

```

An OpenFileDialog or SaveFileDialog is used to specify the location and filename, and then the text is either loaded or saved by using a TextRange object, which we obtain directly from the RichTextBox, in combination with a FileStream, which provides the access to the physical file. The file is loaded and saved in the RTF format, but you can specify one of the other format types if you want your editor to support other formats, e.g. plain text.

#### 1.13.8.6. The complete example

Here's the code for the entire application - first the XAML and then the C# Code-behind:

```

<Window x:Class
        ="WpfTutorialSamples.Rich_text_controls.RichTextBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="RichTextEditorSample" Height="300" Width="400">

```

```

<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open" Executed
="Open_Executed" />
    <CommandBinding Command="ApplicationCommands.Save" Executed
="Save_Executed" />
</Window.CommandBindings>
<DockPanel>
    <ToolBar DockPanel.Dock="Top">
        <Button Command="ApplicationCommands.Open">
            <Image Source
="/WpfTutorialSamples;component/Images/folder.png" Width="16" Height
="16" />
        </Button>
        <Button Command="ApplicationCommands.Save">
            <Image Source
="/WpfTutorialSamples;component/Images/disk.png" Width="16" Height="16"
/>
        </Button>
        <Separator />
        <ToggleButton Command="EditingCommands.ToggleBold" Name
="btnBold">
            <Image Source
="/WpfTutorialSamples;component/Images/text_bold.png" Width="16" Height
="16" />
        </ToggleButton>
        <ToggleButton Command="EditingCommands.ToggleItalic" Name
="btnItalic">
            <Image Source
="/WpfTutorialSamples;component/Images/text_italic.png" Width="16"
Height="16" />
        </ToggleButton>
        <ToggleButton Command="EditingCommands.ToggleUnderline"
Name="btnUnderline">
            <Image Source
="/WpfTutorialSamples;component/Images/text_underline.png" Width="16"
Height="16" />
        </ToggleButton>
        <Separator />
        <ComboBox Name="cmbFontFamily" Width="150"
SelectionChanged="cmbFontFamily_SelectionChanged" />
    </ToolBar>

```

```

        <ComboBox Name="cmbFontSize" Width="50" IsEditable="True"
TextBoxBase.TextChanged="cmbFontSize_TextChanged" />
    </ToolBar>
    <RichTextBox Name="rtbEditor" SelectionChanged
="rtbEditor_SelectionChanged" />
</DockPanel>
</Window>

using System;
using System.Linq;
using System.Collections.Generic;
using System.IO;
using System.Windows;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using Microsoft.Win32;
using System.Windows.Controls;

namespace WpfTutorialSamples.Rich_text_controls
{
    public partial class RichTextEditorSample : Window
    {
        public RichTextEditorSample()
        {
            InitializeComponent();
            cmbFontFamily.ItemsSource =
Fonts.SystemFontFamilies.OrderBy(f => f.Source);
            cmbFontSize.ItemsSource = new List<double>() { 8, 9, 10,
11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, 72 };
        }

        private void rtbEditor_SelectionChanged(object sender,
RoutedEventArgs e)
        {
            object temp =
rtbEditor.Selection.GetProperty(Inline.FontWeightProperty);
            btnBold.IsChecked = (temp !=
DependencyProperty.UnsetValue) && (temp.Equals(FontWeights.Bold));
            temp =

```

```

rtbEditor.Selection.GetPropertyValue(Inline.FontStyleProperty);
    btnItalic.IsChecked = (temp != DependencyProperty.UnsetValue) && (temp.Equals(FontStyles.Italic));
    temp =
rtbEditor.Selection.GetPropertyValue(Inline.TextDecorationsProperty);
    btnUnderline.IsChecked = (temp != DependencyProperty.UnsetValue) && (temp.Equals(TextDecorations.Underline));

    temp =
rtbEditor.Selection.GetPropertyValue(Inline.FontFamilyProperty);
    cmbFontFamily.SelectedItem = temp;
    temp =
rtbEditor.Selection.GetPropertyValue(Inline.FontSizeProperty);
    cmbFontSize.Text = temp.ToString();
}

private void Open_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.Filter = "Rich Text Format (*.rtf)|*.rtf|All files (*.*)|*.*";
    if(dlg.ShowDialog() == true)
    {
        FileStream fileStream = new FileStream(dlg.FileName,
 FileMode.Open);
        TextRange range = new TextRange(rtbEditor.Document.ContentStart,
 rtbEditor.Document.ContentEnd);
        range.Load(fileStream, DataFormats.Rtf);
    }
}

private void Save_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    SaveFileDialog dlg = new SaveFileDialog();
    dlg.Filter = "Rich Text Format (*.rtf)|*.rtf|All files (*.*)|*.*";
}

```

```

        if(dlg.ShowDialog() == true)
    {
        FileStream fileStream = new FileStream(dlg.FileName,
        FileMode.Create);
        TextRange range = new
        TextRange(rtbEditor.Document.ContentStart,
        rtbEditor.Document.ContentEnd);
        range.Save(fileStream, DataFormats.Rtf);
    }
}

private void cmbFontFamily_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    if(cmbFontFamily.SelectedItem != null)

rtbEditor.Selection.ApplyPropertyValue(Inline.FontFamilyProperty,
cmbFontFamily.SelectedItem);
}

private void cmbFontSize_TextChanged(object sender,
TextChangedEventArgs e)
{

rtbEditor.Selection.ApplyPropertyValue(Inline.FontSizeProperty,
cmbFontSize.Text);
}
}

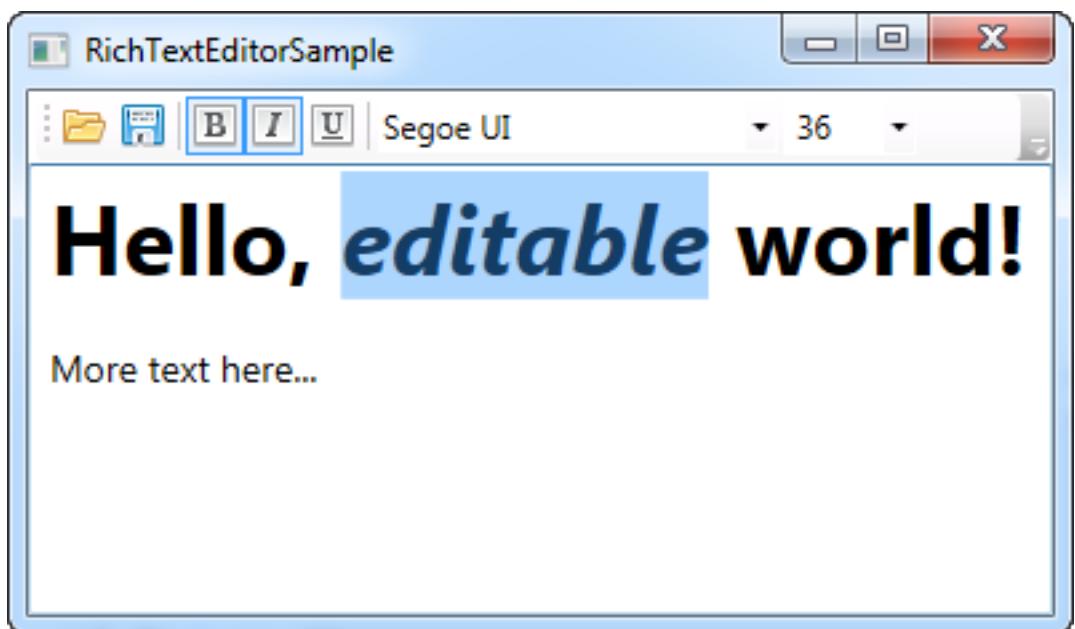
```

And here's another screenshot where we've selected some text. Notice how the toolbar controls reflects the state of the current selection:

#### 1.13.8.7. Summary

As you can see, implementing a rich text editor in WPF is very simple, especially because of the excellent RichTextBox control. If you want, you can easily extend this example with stuff like text alignment, colors, lists and even tables.

Please be aware that while the above example should work just fine, there's absolutely no exception handling or checking at all, to keep the amount of code to a minimum. There are several places which



could easily throw an exception, like the font size combo box, where one could cause an exception by entering a non-numeric value. You should of course check all of this and handle possible exceptions if you wish to expand on this example for your work.

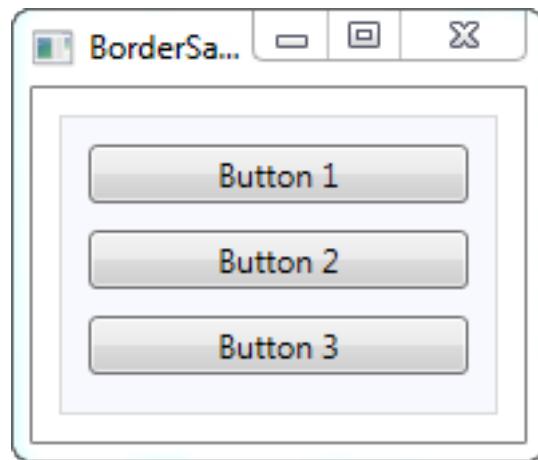
## 1.14. Misc. controls

### 1.14.1. The Border control

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

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

```
<Window x:Class="WpfTutorialSamples.Misc_controls.BorderSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BorderSample" Height="170" Width="200">
    <Grid Margin="10">
        <Border Background="GhostWhite" BorderBrush="Gainsboro"
BorderThickness="1">
            <StackPanel Margin="10">
                <Button>Button 1</Button>
                <Button Margin="0,10">Button 2</Button>
                <Button>Button 3</Button>
            </StackPanel>
        </Border>
    </Grid>
</Window>
```

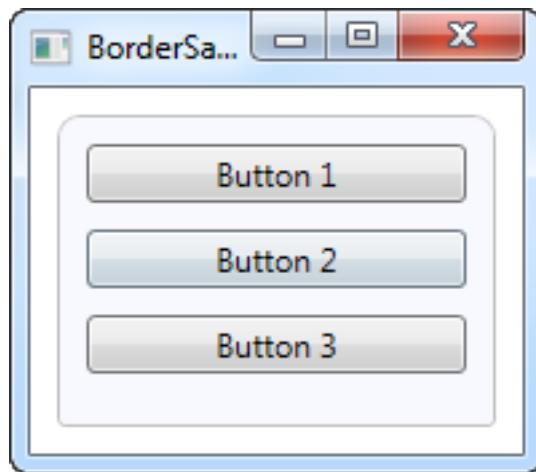


The Border is completely lookless until you define either a background or a border brush and thickness, so that's what I've done here, using the **Background**, **BorderBrush** and **BorderThickness** properties.

### 1.14.1.1. Border with round corners

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

```
<Window x:Class="WpfTutorialSamples.Misc_controls.BorderSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BorderSample" Height="175" Width="200">
    <Grid Margin="10">
        <Border Background="GhostWhite" BorderBrush="Silver"
BorderThickness="1" CornerRadius="8,8,3,3">
            <StackPanel Margin="10">
                <Button>Button 1</Button>
                <Button Margin="0,10">Button 2</Button>
                <Button>Button 3</Button>
            </StackPanel>
        </Border>
    </Grid>
</Window>
```



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

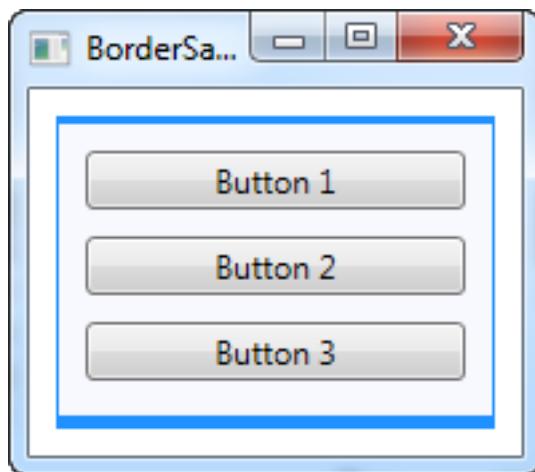
### 1.14.1.2. Border color/thickness

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

```

<Window x:Class="WpfTutorialSamples.Misc_controls.BorderSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BorderSample" Height="175" Width="200">
    <Grid Margin="10">
        <Border Background="GhostWhite" BorderBrush="DodgerBlue"
BorderThickness="1,3,1,5">
            <StackPanel Margin="10">
                <Button>Button 1</Button>
                <Button Margin="0,10">Button 2</Button>
                <Button>Button 3</Button>
            </StackPanel>
        </Border>
    </Grid>
</Window>

```



#### 1.14.1.3. Border background

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

```

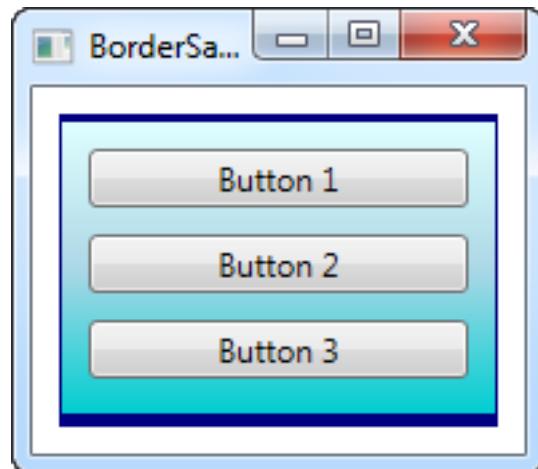
<Window x:Class="WpfTutorialSamples.Misc_controls.BorderSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BorderSample" Height="175" Width="200">
    <Grid Margin="10">
        <Border BorderBrush="Navy" BorderThickness="1,3,1,5">

```

```

<Border.Background>
    <LinearGradientBrush StartPoint="0.5,0" EndPoint
    ="0.5,1">
        <GradientStop Color="LightCyan" Offset="0.0" />
        <GradientStop Color="LightBlue" Offset="0.5" />
        <GradientStop Color="DarkTurquoise" Offset
    ="1.0" />
    </LinearGradientBrush>
</Border.Background>
<StackPanel Margin="10">
    <Button>Button 1</Button>
    <Button Margin="0,10">Button 2</Button>
    <Button>Button 3</Button>
</StackPanel>
</Border>
</Grid>
</Window>

```

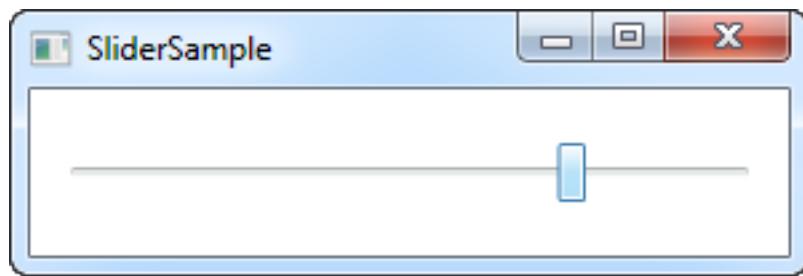


In this case, I've specified a **LinearGradientBrush** to be used for the background of the Border and then a more fitting border color. The LinearGradientBrush might not have the most obvious syntax, so I will explain that in a later chapter, including other brush types, but for now, you can try my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

## 1.14.2. The Slider control

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

```
<Window x:Class="WpfTutorialSamples.Misc_controls.SliderSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SliderSample" Height="100" Width="300">
    <StackPanel VerticalAlignment="Center" Margin="10">
        <Slider Maximum="100" />
    </StackPanel>
</Window>
```

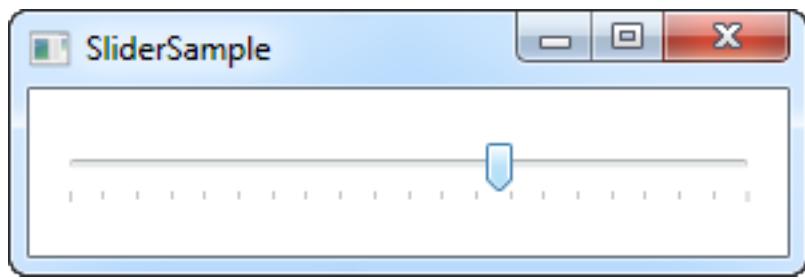


This will allow the end-user to select a value between 0 and 100 by dragging the button (referred to as the thumb) along the line.

### 1.14.2.1. Ticks

In the example, I have dragged the thumb beyond the middle, but it's obviously hard to see the exact value. One way to remedy this is to turn on ticks, which are small markers shown on the line to give a better indication on how far the thumb is. Here's an example:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.SliderSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SliderSample" Height="100" Width="300">
    <StackPanel VerticalAlignment="Center" Margin="10">
        <Slider Maximum="100" TickPlacement="BottomRight"
        TickFrequency="5" />
    </StackPanel>
</Window>
```



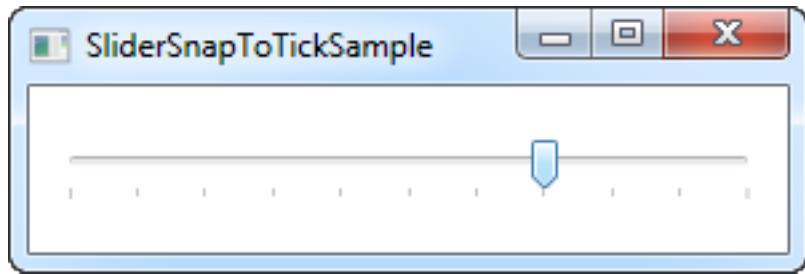
I turn on the tick markers by giving the **TickPlacement** property another value than None, which is the default. In my example, I want the ticks placed below the line, but you can use **TopLeft** or even **Both** as possible values, to change this.

Also notice my use of the **TickFrequency** property. It defaults to 1, but in an example where the range of possible values goes from 0 to 100, this will result in 100 tick markers, which will have to be fitted into the limited space. In a case like this, it makes sense to raise the **TickFrequency** to something that will make it look less crowded.

#### 1.14.2.2. Snapping to ticks

If you have a look at the screenshot above, you will see that the thumb is between ticks. This makes sense, since there are five values between each tick, as specified by the **TickFrequency** property. Also, the value of the Slider control is in fact by default a double, meaning that the value can (and will likely) be a non-integer. We can change this by using the **IsSnapToTickEnabled** property, like in the below example:

```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.SliderSnapToTickSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SliderSnapToTickSample" Height="100" Width="300">
    <StackPanel VerticalAlignment="Center" Margin="10">
        <Slider Maximum="100" TickPlacement="BottomRight"
        TickFrequency="10" IsSnapToTickEnabled="True" />
    </StackPanel>
</Window>
```



Notice that I've changed the **TickFrequency** to 10, and then enabled the **IsSnapToTickEnabled** property.

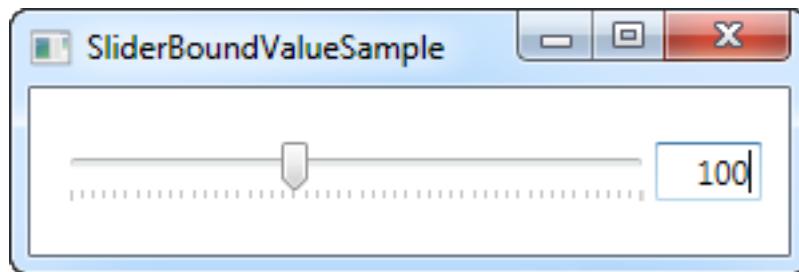
This ensures that the thumb can only be placed directly on a tick value, so for this example, it can only be 0, 10, 20, 30, 40 and so on.

#### 1.14.2.3. Slider value

So far, we've just used the Slider illustratively, but of course, the actual purpose is to read its current value and use it for something. The Slider has a Value property for that, which you can of course read from Code-behind, or even bind to.

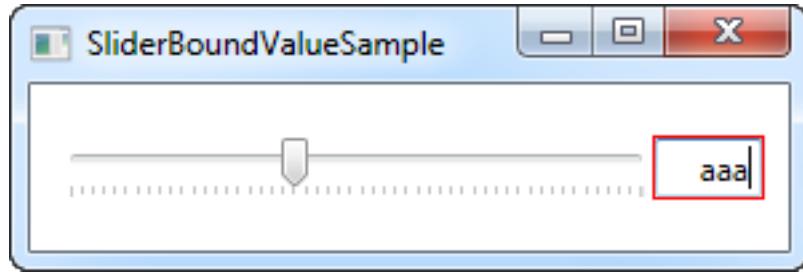
A common scenario in using the Slider is to combine it with a TextBox, which will allow the user to see the currently selected value, as well as changing it by entering a number instead of dragging the Slider thumb. Normally, you would have to subscribe to change events on both the Slider and the TextBox and then update accordingly, but a simple binding can do all of that for us:

```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.SliderBoundValueSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SliderBoundValueSample" Height="100" Width="300">
    <DockPanel VerticalAlignment="Center" Margin="10">
        <TextBox Text="{Binding ElementName=slValue, Path=Value,
        UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right"
        TextAlignment="Right" Width="40" />
        <Slider Maximum="255" TickPlacement="BottomRight"
        TickFrequency="5" IsSnapToTickEnabled="True" Name="slValue" />
    </DockPanel>
</Window>
```



Now you can change the value by using either the Slider or by entering a value in the TextBox, and it will be immediately reflected in the other control. As an added bonus, we get simple validation as well, without any extra work, like if we try to enter a non-numeric value in the TextBox:

#### 1.14.2.4. Responding to changed values



Of course, while bindings are very cool for a lot of purposes, you still may want to respond to changes in the Slider value from your Code-behind. Fortunately for us, the Slider comes with a ValueChanged event which will help us with that. To illustrate this, I've created a more complex sample with three sliders, where we change the Red, Green and Blue (RGB) values of a color:

```
<Window x:Class
    ="WpfTutorialSamples.Misc_controls.SliderValueChangedSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SliderValueChangedSample" Height="200" Width="300">
    <StackPanel Margin="10" VerticalAlignment="Center">
        <DockPanel VerticalAlignment="Center" Margin="10">
            <Label DockPanel.Dock="Left" FontWeight="Bold">R:</Label>
            <TextBox Text="{Binding ElementName=slColorR, Path=Value,
UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right"
TextAlignment="Right" Width="40" />
            <Slider Maximum="255" TickPlacement="BottomRight"
TickFrequency="5" IsSnapToTickEnabled="True" Name="slColorR"
ValueChanged="ColorSlider_ValueChanged" />
        </DockPanel>

        <DockPanel VerticalAlignment="Center" Margin="10">
            <Label DockPanel.Dock="Left" FontWeight="Bold">G:</Label>
            <TextBox Text="{Binding ElementName=slColorG, Path=Value,
UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right"
TextAlignment="Right" Width="40" />
            <Slider Maximum="255" TickPlacement="BottomRight"
TickFrequency="5" IsSnapToTickEnabled="True" Name="slColorG"
ValueChanged="ColorSlider_ValueChanged" />
        </DockPanel>

        <DockPanel VerticalAlignment="Center" Margin="10">
            <Label DockPanel.Dock="Left" FontWeight="Bold">B:</Label>
            <TextBox Text="{Binding ElementName=slColorB, Path=Value,
UpdateSourceTrigger=PropertyChanged}" DockPanel.Dock="Right"
TextAlignment="Right" Width="40" />
            <Slider Maximum="255" TickPlacement="BottomRight"
TickFrequency="5" IsSnapToTickEnabled="True" Name="slColorB"
ValueChanged="ColorSlider_ValueChanged" />
        </DockPanel>
    </StackPanel>
</Window>
```

```

UpdateSourceTrigger=PropertyChanged} " DockPanel.Dock="Right"
TextAlignment="Right" Width="40" />
    <Slider Maximum="255" TickPlacement="BottomRight"
TickFrequency="5" IsSnapToTickEnabled="True" Name="slColorB"
ValueChanged="ColorSlider_ValueChanged" />

```

```

    </DockPanel>
</StackPanel>
</Window>

using System;
using System.Windows;
using System.Windows.Media;

namespace WpfTutorialSamples.Misc_controls
{
    public partial class SliderValueChangedSample : Window
    {
        public SliderValueChangedSample()
        {
            InitializeComponent();
        }

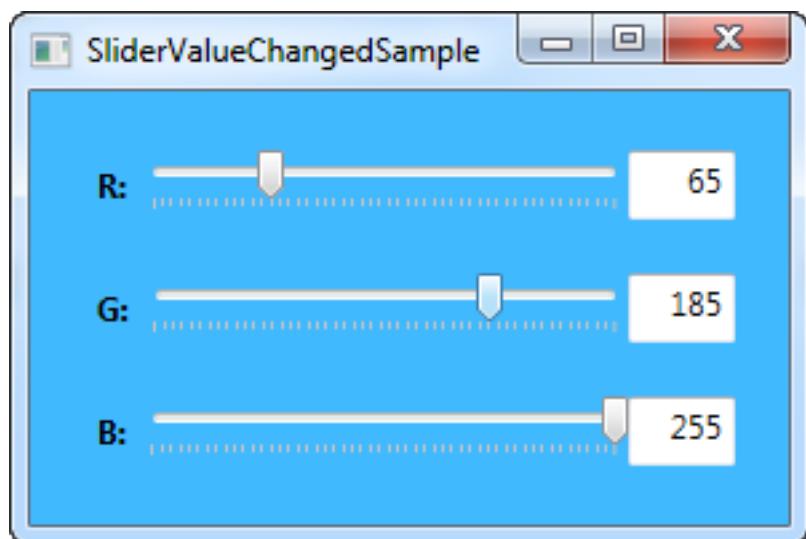
        private void ColorSlider_ValueChanged(object sender,
RoutedEventArgs<double> e)
        {
            Color color = Color.FromRgb((byte)slColorR.Value, (byte)
)slColorG.Value, (byte)slColorB.Value);
            this.Background = new SolidColorBrush(color);
        }
    }
}

```

In the XAML part of the code, we have three DockPanels, each with a Label, a Slider and a TextBox control. Just like before, the Text property of the TextBox controls have been bound to the Value of the Slider.

Each slider subscribes to the same **ValueChanged** event, in which we create a new Color instance, based on the currently selected values and then uses this color to create a new SolidColorBrush for the Background property of the Window.

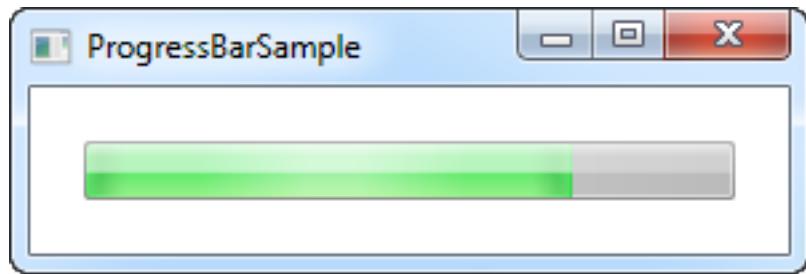
All in all, this is a pretty good example of what the Slider control can be used for.



### 1.14.3. The ProgressBar control

WPF comes with a handy control for displaying progress, called the **ProgressBar**. It works by setting a minimum and maximum value and then incrementing a value, which will give a visual indication on how far in the process you currently are. Here's a very basic example to demonstrate it with:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.ProgressBarSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ProgressBarSample" Height="100" Width="300">
    <Grid Margin="20">
        <ProgressBar Minimum="0" Maximum="100" Value="75" />
    </Grid>
</Window>
```



In this case, I've used a pretty standard approach of showing progress as a percentage (between 0 and 100%), giving it an initial value of 75. Another approach is to use actual minimum and maximum values from a list of tasks you're performing. For instance, if you loop through a collected list of files while checking each of them, you can set the Minimum property to 0, the Maximum to the amount of files in your list, and then just increment as you loop through it.

The **ProgressBar** is, just like other standard WPF controls, rendered to match the visual style of the operating system. Here on Windows 7, it has a nice animated gradient, as seen on the screenshot.

#### 1.14.3.1. Showing progress while performing a lengthy task

The above example illustrates how simple it is to use a **ProgressBar**, but normally you would of course want to show the progress of some actual work and not just a static value.

In most situations you will use the **ProgressBar** to show progress for some heavy/lengthy task, and this is where most new programmers run into a very common problem: If you do a piece of heavy work on the UI thread, while trying to simultaneously update e.g. a **ProgressBar** control, you will soon realize that you can't do both, at the same time, on the same thread. Or to be more clear, you can, but the **ProgressBar** won't actually show each update to the progress before the task is completed, which pretty much renders it useless.

To illustrate, you can try the following example:

```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.ProgressBarTaskOnUiThread"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ProgressBarTaskOnUiThread" Height="100" Width="300"
        ContentRendered="Window_ContentRendered">
    <Grid Margin="20">
        <ProgressBar Minimum="0" Maximum="100" Name="pbStatus" />
    </Grid>
</Window>

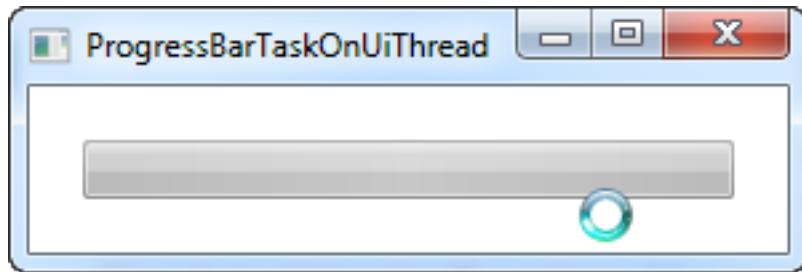
using System;
using System.Threading;
using System.Windows;

namespace WpfTutorialSamples.Misc_controls
{
    public partial class ProgressBarTaskOnUiThread : Window
    {
        public ProgressBarTaskOnUiThread()
        {
            InitializeComponent();
        }

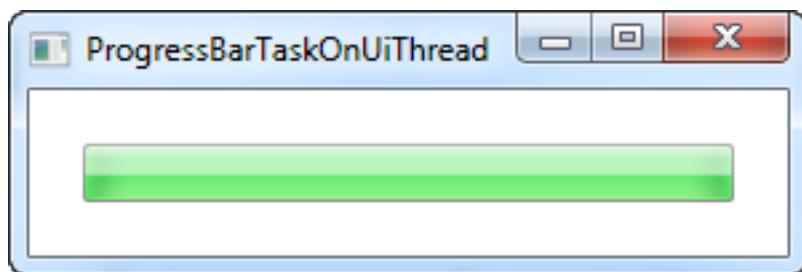
        private void Window_ContentRendered(object sender, EventArgs e)
        {
            for(int i = 0; i < 100; i++)
            {
                pbStatus.Value++;
                Thread.Sleep(100);
            }
        }
    }
}
```

A very basic example, where, as soon as the window is ready, we do a loop from 0 to 100 and in each iteration, we increment the value of the ProgressBar. Any modern computer can do this faster than you can

blink your eyes, so I've added a delay to each iteration of 100 milliseconds. Unfortunately, as I already described, nothing will happen. This is how it looks in the middle of the process:



Notice that the cursor indicates that something is happening, yet the ProgressBar still looks like it did at the start (empty). As soon as the loop, which represents our lengthy task, is done, the ProgressBar will look like this:



That really didn't help your users see the progress! Instead, we have to perform the task on a worker thread and then push updates to the UI thread, which will then be able to immediately process and visually show these updates. An excellent tool for handling this job is the `BackgroundWorker` class, which we talk much more about elsewhere in this tutorial. Here's the same example as above, but this time using a `BackgroundWorker`:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.ProgressBarTaskOnWorkerThread"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ProgressBarTaskOnWorkerThread" Height="100" Width="300"
        ContentRendered="Window_ContentRendered">
    <Grid Margin="20">
        <ProgressBar Minimum="0" Maximum="100" Name="pbStatus" />
    </Grid>
</Window>

using System;
using System.ComponentModel;
using System.Threading;
```

```

using System.Windows;

namespace WpfTutorialSamples.Misc_controls
{
    public partial class ProgressBarTaskOnWorkerThread : Window
    {
        public ProgressBarTaskOnWorkerThread()
        {
            InitializeComponent();
        }

        private void Window_ContentRendered(object sender, EventArgs e)
        {
            BackgroundWorker worker = new BackgroundWorker();
            worker.WorkerReportsProgress = true;
            worker.DoWork += worker_DoWork;
            worker.ProgressChanged += worker_ProgressChanged;

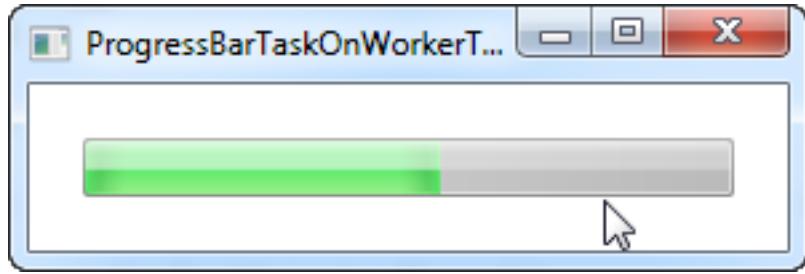
            worker.RunWorkerAsync();
        }

        void worker_DoWork(object sender, DoWorkEventArgs e)
        {
            for(int i = 0; i < 100; i++)
            {
                (sender as BackgroundWorker).ReportProgress(i);
                Thread.Sleep(100);
            }
        }

        void worker_ProgressChanged(object sender,
ProgressChangedEventArgs e)
        {
            pbStatus.Value = e.ProgressPercentage;
        }
    }
}

```

As you can see on the screenshot, the progress is now updated all the way through the task, and as the



cursor indicates, no hard work is being performed on the UI thread, which means that you can still interact with the rest of the interface.

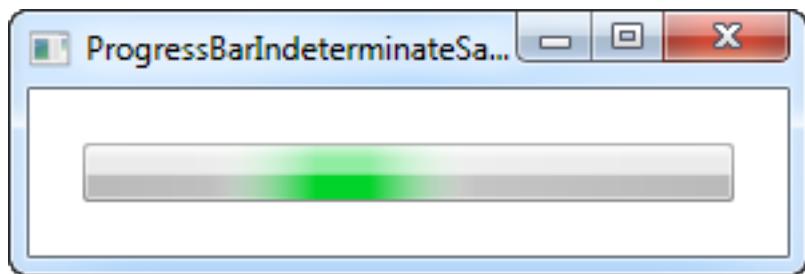
Please be aware that while the `BackgroundWorker` does help a lot with multithreading related problems, there are still some things you should be aware of, so please have a look at the `BackgroundWorker` articles in this tutorial before doing anything more advanced than a scenario like the one above.

#### 1.14.3.2. Indeterminate

For some tasks, expressing the progress as a percentage is not possible or you simply don't know how long it will take. For those situations, the indeterminate progress bar has been invented, where an animation lets the user know that something is happening, while indicating that the running time can't be determined.

The WPF `ProgressBar` supports this mode through the use of the `IsIndeterminate` property, which we'll show you in the next example:

```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.ProgressBarIndeterminateSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ProgressBarIndeterminateSample" Height="100" Width
        ="300">
    <Grid Margin="20">
        <ProgressBar Minimum="0" Maximum="100" Name="pbStatus"
        IsIndeterminate="True" />
    </Grid>
</Window>
```

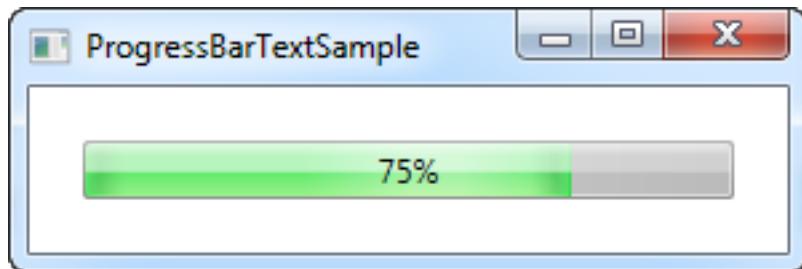


Notice that the green progress indicator is not anchored to either of the sides - instead it floats freely from start to finish and then it starts all over again.

#### 1.14.3.3. ProgressBar with text

One thing that I really missed from the standard WPF ProgressBar is the ability to show a text representation of the progress as well as the progress bar. Fortunately for us, the flexibility of WPF makes this really easy for us to accomplish. Here's an example:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.ProgressBarTextSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ProgressBarTextSample" Height="100" Width="300">
    <Grid Margin="20">
        <ProgressBar Minimum="0" Maximum="100" Value="75" Name
        ="pbStatus" />
        <TextBlock Text="{Binding ElementName=pbStatus, Path=Value,
        StringFormat={}{0:0}%}" HorizontalAlignment="Center" VerticalAlignment
        ="Center" />
    </Grid>
</Window>
```



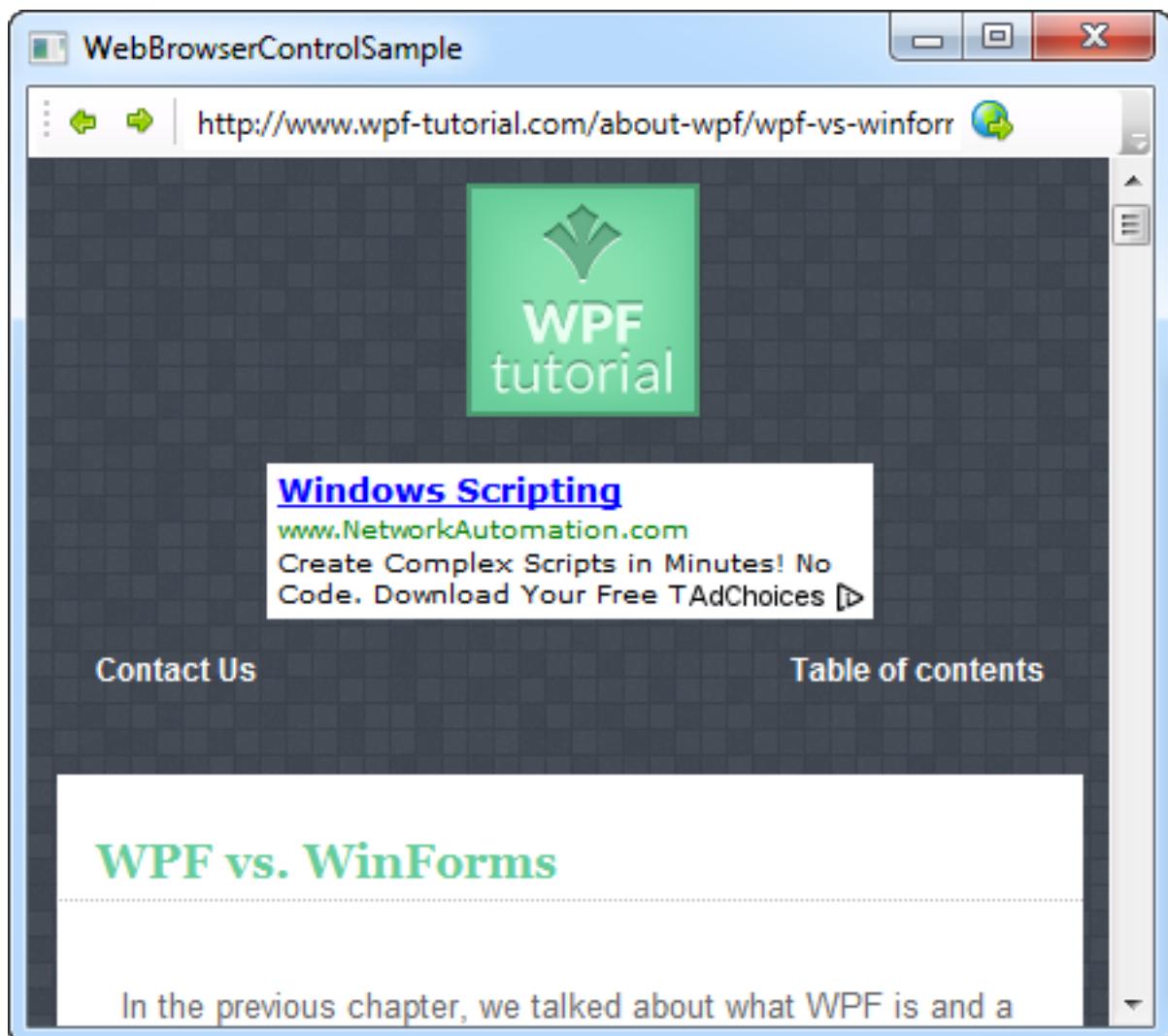
We accomplish the above by putting the ProgressBar and the TextBlock showing the percentage inside of the same Grid, without specifying any rows or columns. This will render the TextBlock on top of the ProgressBar, which is exactly what we want here, because the TextBlock has a transparent background by default.

We use a binding to make sure that the TextBlock show the same value as the ProgressBar. Notice the special **StringFormat** syntax, which allows us to show the value with a percentage sign postfix - it might look a bit strange, but please see the **StringFormat** article of this tutorial for more information on it.

## 1.14.4. The WebBrowser control

WPF comes with a ready to use WebBrowser control, which allows you to host a complete web browser within your application. The WebBrowser control is really just a shell around an ActiveX version of Internet Explorer, but since this is an integrated part of Windows, your application should work on all Windows machines without requiring the installation of additional components.

I've done things a bit differently in this article: Instead of starting off with a very limited example and then adding to it, I've created just one but more complex example. It illustrates how easy you can get a small web browser up and running. It's very basic in its functionality, but you can easily extend it if you want to. Here's how it looks:



So let's have a look at the code:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.WebBrowserControlSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```

        Title="WebBrowserControlSample" Height="300" Width="450">
    <Window.CommandBindings>
        <CommandBinding Command="NavigationCommands.BrowseBack"
CanExecute="BrowseBack_CanExecute" Executed="BrowseBack_Executed" />
        <CommandBinding Command="NavigationCommands.BrowseForward"
CanExecute="BrowseForward_CanExecute" Executed="BrowseForward_Executed" />
    </Window.CommandBindings>
    <DockPanel>
        <ToolBar DockPanel.Dock="Top">
            <Button Command="NavigationCommands.BrowseBack">
                <Image Source
                ="/WpfTutorialSamples;component/Images/arrow_left.png" Width="16" Height
                ="16" />
            </Button>
            <Button Command="NavigationCommands.BrowseForward">
                <Image Source
                ="/WpfTutorialSamples;component/Images/arrow_right.png" Width="16"
Height="16" />
            </Button>
            <Separator />
            <TextBox Name="txtUrl" Width="300" KeyUp="txtUrl_KeyUp" />
            <Button Command="NavigationCommands.GoToPage">
                <Image Source
                ="/WpfTutorialSamples;component/Images/world_go.png" Width="16" Height
                ="16" />
            </Button>
        </ToolBar>
        <WebBrowser Name="wbSample" Navigating="wbSample_Navigating" />
    </WebBrowser>
    </DockPanel>
</Window>

using System;
using System.Windows;
using System.Windows.Input;

```

```

namespace WpfTutorialSamples.Misc_controls
{
    public partial class WebBrowserControlSample : Window
    {
        public WebBrowserControlSample()
        {
            InitializeComponent();
            wbSample.Navigate("http://www.wpf-tutorial.com");
        }

        private void txtUrl_KeyUp(object sender, KeyEventArgs e)
        {
            if(e.Key == Key.Enter)
                wbSample.Navigate(txtUrl.Text);
        }

        private void wbSample_Navigating(object sender,
System.Windows.Navigation.NavigatingCancelEventArgs e)
        {
            txtUrl.Text = e.Uri.OriginalString;
        }

        private void BrowseBack_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = ((wbSample != null) &&
(wbSample.CanGoBack));
        }

        private void BrowseBack_Executed(object sender,
ExecutedRoutedEventArgs e)
        {
            wbSample.GoBack();
        }

        private void BrowseForward_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = ((wbSample != null) &&
(wbSample.CanGoForward));
        }
    }
}

```

```

        }

    private void BrowseForward_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    wbSample.GoForward();
}

private void GoToPage_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}

private void GoToPage_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    wbSample.Navigate(txtUrl.Text);
}

}
}

```

The code might seem a bit overwhelming at first, but if you take a second look, you'll realize that there's a lot of repetition in it.

Let's start off by talking about the **XAML** part. Notice that I'm using several concepts discussed elsewhere in this tutorial, including the ToolBar control and WPF commands. The ToolBar is used to host a couple of buttons for going backward and forward. After that, we have an address bar for entering and showing the current URL, along with a button for navigating to the entered URL.

Below the toolbar, we have the actual WebBrowser control. As you can see, using it only requires a single line of XAML - in this case we subscribe to the **Navigating** event, which occurs as soon as the WebBrowser starts navigating to a URL.

In **Code-behind**, we start off by navigating to a URL already in the constructor of the Window, to have something to show immediately instead of a blank control. We then have the **txtUrl\_KeyUp** event, in which we check to see if the user has hit Enter inside of the address bar - if so, we start navigating to the entered URL.

The **wbSample\_Navigating** event makes sure that the address bar is updated each time a new navigation starts. This is important because we want it to show the current URL no matter if the user initiated the navigation by entering a new URL or by clicking a link on the webpage.

The last part of the Code-behind is simple handling of our commands: Two for the back and forward buttons, where we use the CanGoBack and CanGoForward to decide whether they can execute, and the GoBack and GoForward methods to do the actual work. This is very standard when dealing with WPF commands, as described in the commands section of this tutorial.

For the last command, we allow it to always execute and when it does, we use the Navigate() method once again.

#### 1.14.4.1. Summary

As you can see, hosting and using a complete webbrowser inside of your application becomes very easy with the WebBrowser control. However, you should be aware that the WPF version of WebBrowser is a bit limited when compared to the WinForms version, but for basic usage and navigation, it works fine.

If you wish to use the WinForms version instead, you may do so using the WindowsFormsHost, which is explained elsewhere in this tutorial.

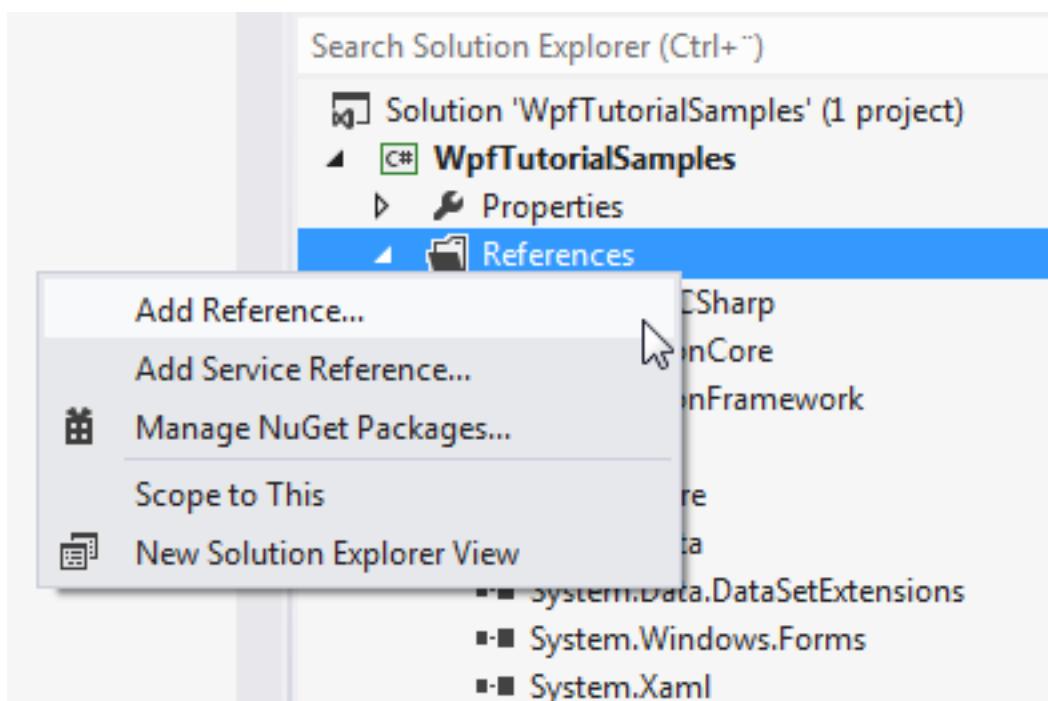
## 1.14.5. The WindowsFormsHost control

WPF and WinForms are two distinct UI frameworks, both created by Microsoft. WPF is meant as a more modern alternative to WinForms, which was the first .NET UI framework. To lighten the transition between the two, Microsoft has made sure that WinForms controls may still be used inside of a WPF application. This is done with the WindowsFormsHost, which we'll discuss in this article.

To use the WindowsFormsHost and controls from WinForms, you need to add a reference to the following assemblies in your application:

- WindowsFormsIntegration
- System.Windows.Forms

In Visual Studio, this is done by right-clicking the "References" node in your project and selecting "**Add reference**":

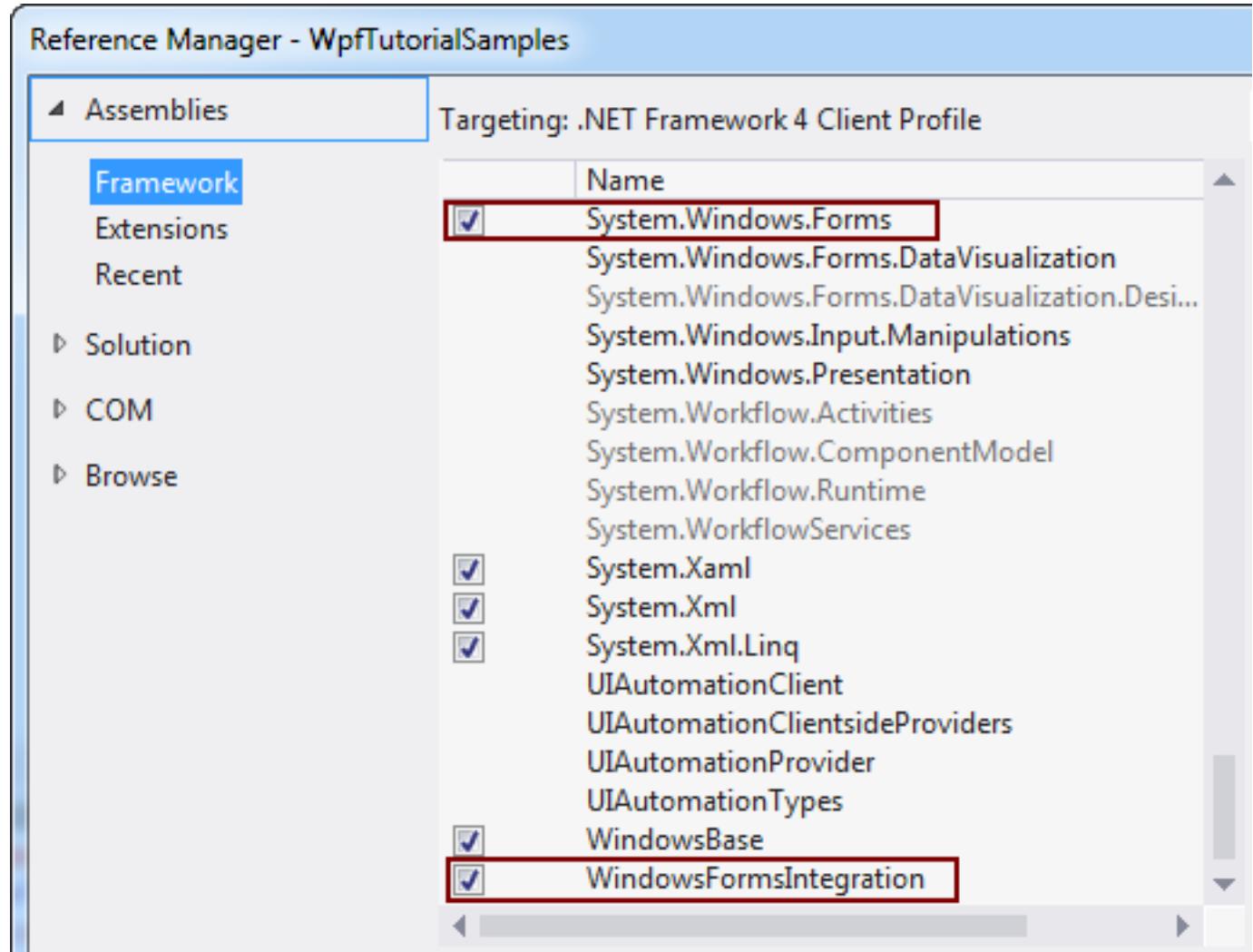


In the dialog that pops up, you should select "**Assemblies**" and then check the two assemblies that we need to add:

### 1.14.5.1. Using the WinForms WebBrowser control

In a previous article, we used the WPF WebBrowser control to create a small web browser. However, as stated in that article, the WPF WebBrowser control is a bit limited when compared to the WinForms version. There are many examples on things easily done with the WinForms version, which are either harder or impossible to do with the WPF version.

A small example is the **DocumentTitle** property and corresponding **DocumentTitleChanged** event, which makes it easy to get and update the title of the window to match the title of the current webpage. We'll use



this as an excuse to test out the WinForms version right here in our WPF application:

```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.WindowsFormsHostSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wf="clr-
        namespace:System.Windows.Forms;assembly=System.Windows.Forms"
        Title="WindowsFormsHostSample" Height="350" Width="450">
    <Grid>
        <WindowsFormsHost Name="wfhSample">
            <WindowsFormsHost.Child>
                <wf:WebBrowser DocumentTitleChanged
                    ="wbWinForms_DocumentTitleChanged" />
            </WindowsFormsHost.Child>
        </WindowsFormsHost>
    </Grid>
```

```

</Window>

using System;
using System.Windows;

namespace WpfTutorialSamples.Misc_controls
{
    public partial class WindowsFormsHostSample : Window
    {
        public WindowsFormsHostSample()
        {
            InitializeComponent();
            (wfhSample.Child as
System.Windows.Forms.WebBrowser).Navigate("http://www.wpf-tutorial.com");
        }

        private void wbWinForms_DocumentTitleChanged(object sender,
EventArgs e)
        {
            this.Title = (sender as
System.Windows.Forms.WebBrowser).DocumentTitle;
        }
    }
}

```

Pay special attention to the line where we add the WinForms namespace to the window, so that we may reference controls from it:

```

xmlns:wf="clr-
namespace:System.Windows.Forms;assembly=System.Windows.Forms"

```

This will allow us to reference WinForms controls using the wf: prefix.

The WindowsFormsHost is fairly simple to use, as you can see. It has a Child property, in which you can define a single WinForms control, much like the WPF Window only holds a single root control. If you need more controls from WinForms inside of your WindowsFormsHost, you can use the **Panel** control from WinForms or any of the other container controls.

The WinForms WebBrowser control is used by referencing the System.Windows.Forms assembly, using the wf prefix, as explained above.



In **Code-behind**, we do an initial call to `Navigate`, to have a visible webpage instead of the empty control on startup. We then handle the **DocumentTitleChanged** event, in which we update the Title property of the Window in accordance with the current **DocumentTitle** value of the WebBrowser control.

Congratulations, you now have a WPF application with a WinForms WebBrowser hosted inside of it.

#### 1.14.5.2. Summary

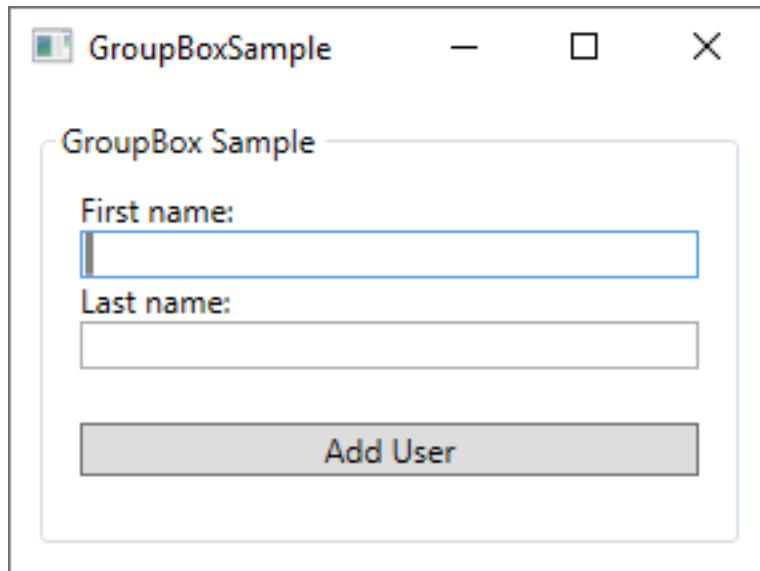
As you can see, using WinForms controls inside of your WPF applications is pretty easy, but the question remains: Is it a good idea?

In general, you may want to avoid it. There are a number of issues that may or may not affect your application (a lot of them are described in this MSDN article: <http://msdn.microsoft.com/en-us/library/aa970911%28v=VS.100%29.aspx>), but a more serious problem is that this kind of UI framework mixing might not be supported in future versions of the .NET framework.

In the end though, the decision is up to you - do you really need the WinForms control or is there a WPF alternative that might work just as well?

## 1.14.6. The GroupBox control

The GroupBox control will allow you to visually group a set of controls together. This could obviously be done using one of the many panels as well, but the GroupBox adds a special type of header and border, which has historically been used a lot within in the Windows operating system. Here's a screenshot of how it might look when you use the **GroupBox** control:



Notice the border around the controls, with the text "GroupBox Sample" placed inside the border line - this is how a GroupBox looks and acts. Using a GroupBox is as simple as adding the tag to your Window and writing something relevant in the **Header** property:

```
<GroupBox Header="GroupBox Sample">  
  
</GroupBox>
```

The GroupBox can only contain a single child element, but that's no problem - just make this one control a Panel, and you are free to add multiple controls to the panel, e.g. to create a dialog like the one displayed above. Here's the full XAML code listing for my example dialog:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.GroupBoxSample"  
       xmlns  
       ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
       xmlns:d="http://schemas.microsoft.com/expressionblend/2008"  
       xmlns:mc="http://schemas.openxmlformats.org/markup-  
compatibility/2006"  
       xmlns:local="clr-namespace:WpfTutorialSamples.Misc_controls"  
       mc:Ignorable="d"  
       Title="GroupBoxSample" Height="220" Width="300">
```

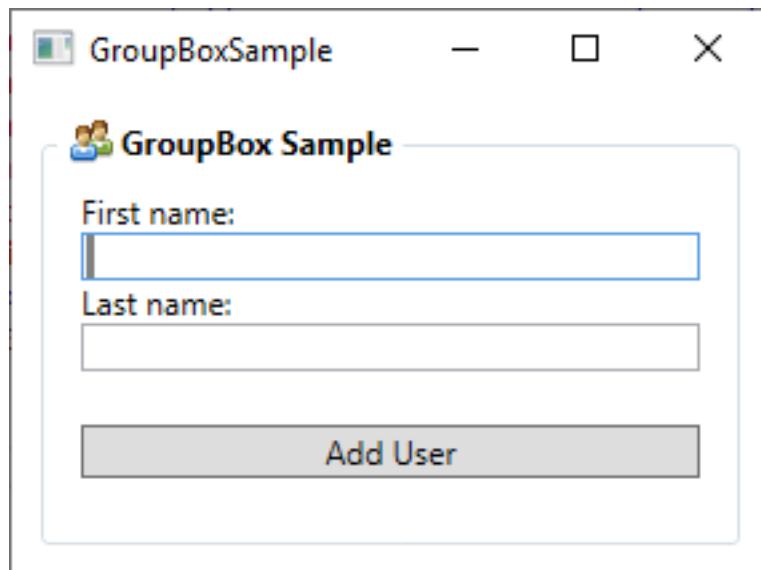
```

<Grid>
    <GroupBox Header="GroupBox Sample" Margin="10" Padding="10">
        <StackPanel>
            <TextBlock>First name:</TextBlock>
            <TextBox />
            <TextBlock>Last name:</TextBlock>
            <TextBox />
            <Button Margin="0, 20">Add User</Button>
        </StackPanel>
    </GroupBox>
</Grid>
</Window>

```

#### 1.14.6.1. GroupBox with custom Header

The Header of a GroupBox is normally just plain, unformatted text, but perhaps you're looking to make it a bit more fancy? No problem, because just like pretty much anything found in the WPF framework, you can just replace the text with one or several other controls. So you can just add a TextBlock control and then change the formatting, e.g. the color of the text. You can even add an image, if you want to, like I have done in this next example:



Now the Header has an image and bold text, and it's so easy to do:

```

<Window x:Class="WpfTutorialSamples.Misc_controls.GroupBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008">

```

```

    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:WpfTutorialSamples.Misc_controls"
    mc:Ignorable="d"
    Title="GroupBoxSample" Height="220" Width="300">
<Grid>
    <GroupBox Margin="10" Padding="10">
        <GroupBox.Header>
            <StackPanel Orientation="Horizontal">
                <Image Source
                    ="/WpfTutorialSamples;component/Images/group.png" Margin="3,0" />
                <TextBlock FontWeight="Bold">GroupBox Sample</TextBlock>
            </StackPanel>
        </GroupBox.Header>
        <StackPanel>
            <TextBlock>First name:</TextBlock>
            <TextBox />
            <TextBlock>Last name:</TextBlock>
            <TextBox />
            <Button Margin="0,20">Add User</Button>
        </StackPanel>
    </GroupBox>
</Grid>
</Window>

```

Notice how I have simply replaced the Header property with a **GroupBox.Header** tag, which then hosts a StackPanel to contain an Image and a TextBlock - with that in place, you have full control of how the Header should look!

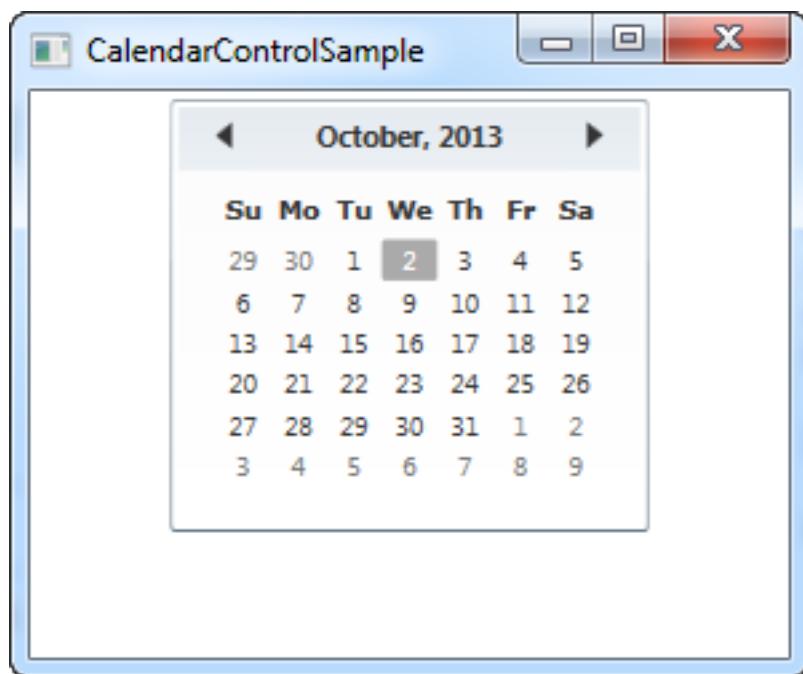
#### 1.14.6.2. Summary

The WPF GroupBox control will make it easy for you to group related controls together, in a way that visually matches the look of especially the Microsoft Windows operating system.

## 1.14.7. The Calendar control

WPF comes with a control for displaying a full calendar right out of the box. It's so simple that you only have to drop it inside your window for a full calendar view, like this:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.CalendarControlSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarControlSample" Height="250" Width="300">
    <Grid>
        <Calendar />
    </Grid>
</Window>
```



Notice how you now get a full list of the dates within the selected month, including the possibility to jump to previous and next months using the arrows in the top of the control. Unless you set a specific date, the current month will be shown and the current date will be marked as selected.

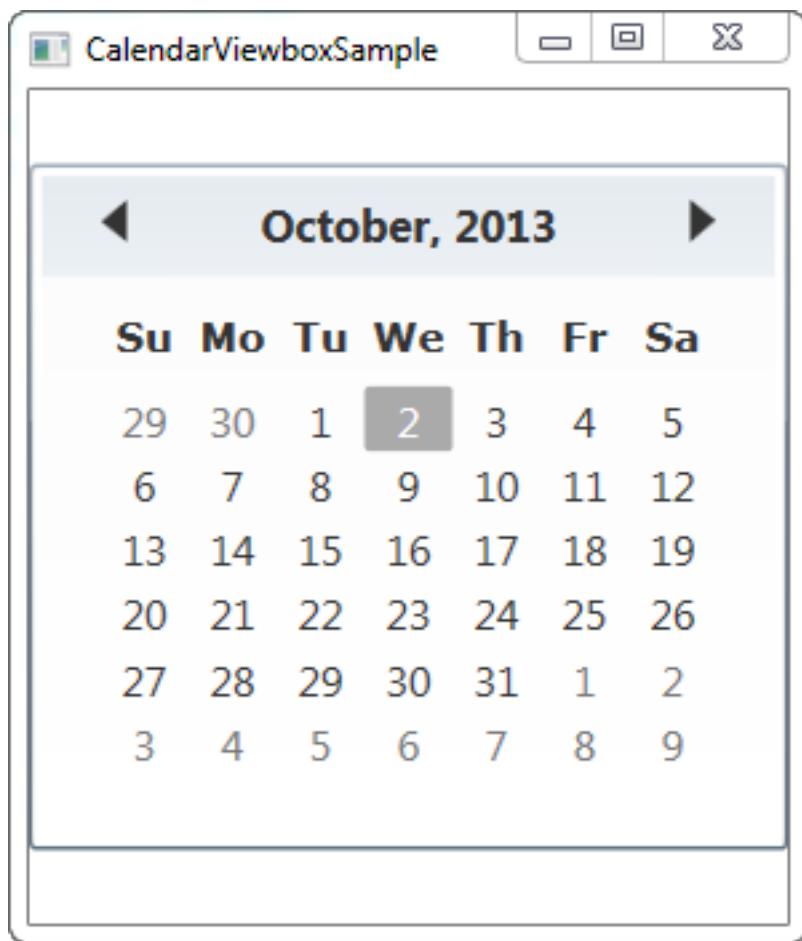
### 1.14.7.1. Calendar size

You will probably notice from our first example that the Calendar doesn't take up all the available space. In fact, even if you give it a large width and height, the actual calendar part will still only take up the amount of space you see on the screenshot, and if you set either of the values very low, the calendar will only be partially visible.

This fixed size behavior is not very typical WPF, where things usually stretch to fill out available space, and it can be a bit annoying to work with if you have a designated amount of space available for the calendar

which you want it to fill out. Fortunately for us, everything in WPF is scalable but in the case of the Calendar control, it needs a bit of help. We'll use the Viewbox control for this purpose:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.CalendarViewboxSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarViewboxSample" Height="350" Width="300">
    <Viewbox>
        <Calendar />
    </Viewbox>
</Window>
```



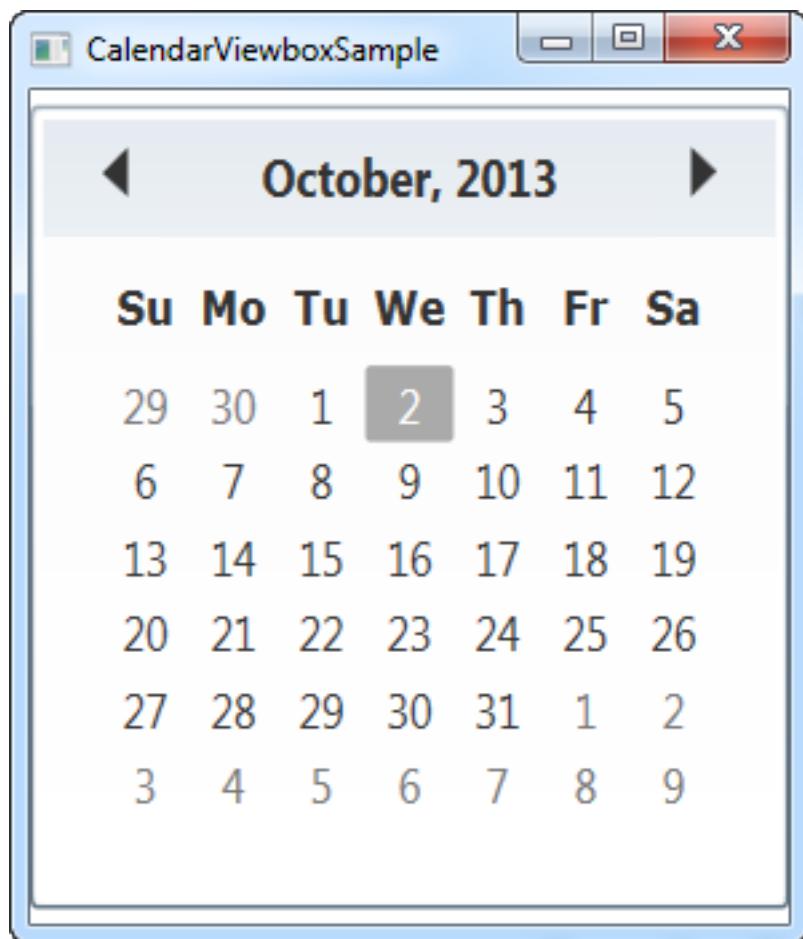
Notice how the Calendar control now scales up to the point where it uses all the available space in the width. The scaling is performed on all parts of the control, including font sizes and border widths.

You will probably also notice that the Calendar control doesn't use up all the available height space. This is noticeable because the window is higher than it is wide and by default, the Viewbox will stretch while maintaining the original aspect ratio. You can easily make it stretch to fill all space in both directions though - simply change the **Stretch** property from its default **Uniform** value to **Fill**:

```

<Window x:Class="WpfTutorialSamples.Misc_controls.CalendarViewboxSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarViewboxSample" Height="350" Width="300">
    <Viewbox Stretch="Fill" StretchDirection="UpOnly">
        <Calendar />
    </Viewbox>
</Window>

```



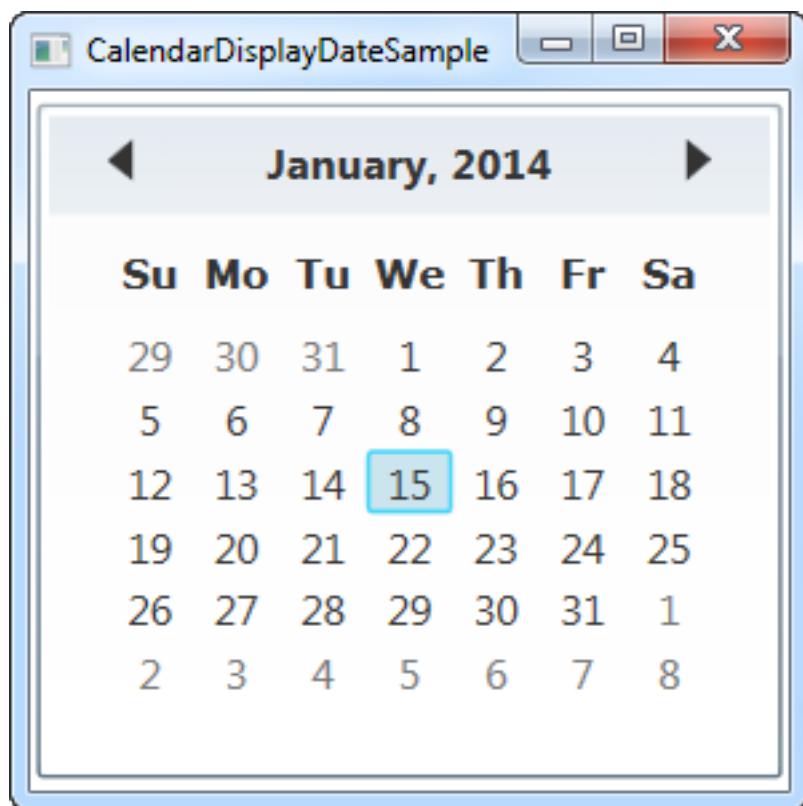
Now it takes up all the available space, in both directions. This is generally not preferable though, since most controls, and this one in particular, will look strange if it gets an abnormal set of dimensions, e.g. 800 pixels high and 300 pixels wide. A **Stretch** mode set to **Uniform** (or left out, as it is the default) is usually the way to go.

I would recommend including the **StretchDirection** property though, as seen in this example. It allows us to specify that the contents should only be scaled up or down, which can be useful. For instance, the Calendar control becomes quite useless below a certain size, where you can no longer see what it is, and to avoid that, you can set the **StretchDirection** to **UpOnly** - the Calendar control will then no longer be scaled below its default size.

### 1.14.7.2. Setting the initial view using DisplayDate

The Calendar control will by default show the current month, but you can change this by using the **DisplayDate** property. Simply set it to a date within the month you wish to start with and it will be reflected in the control:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.CalendarDisplayDateSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarDisplayDateSample" Height="300" Width="300">
    <Viewbox>
        <Calendar DisplayDate="01.01.2014" />
    </Viewbox>
</Window>
```



### 1.14.7.3. Calendar SelectionMode

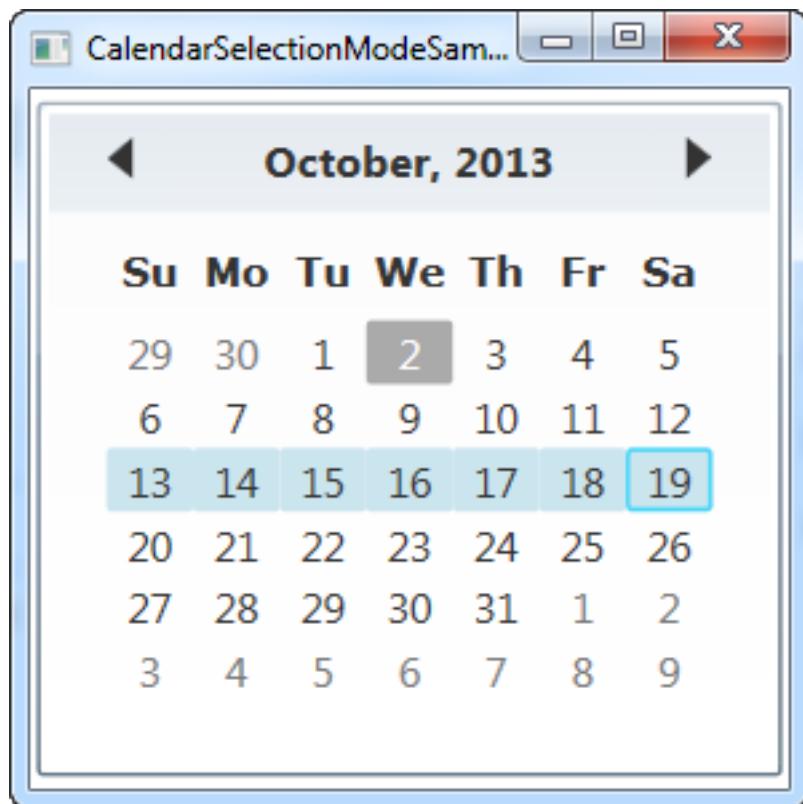
The **SelectionMode** property is interesting. By changing it from its default value, **SingleDate**, you can select multiple dates or ranges of dates. Here's an example:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.CalendarSelectionModeSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```

    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CalendarSelectionModeSample" Height="300" Width="300">
    <Viewbox>
        <Calendar SelectionMode="SingleRange" />
    </Viewbox>
</Window>

```

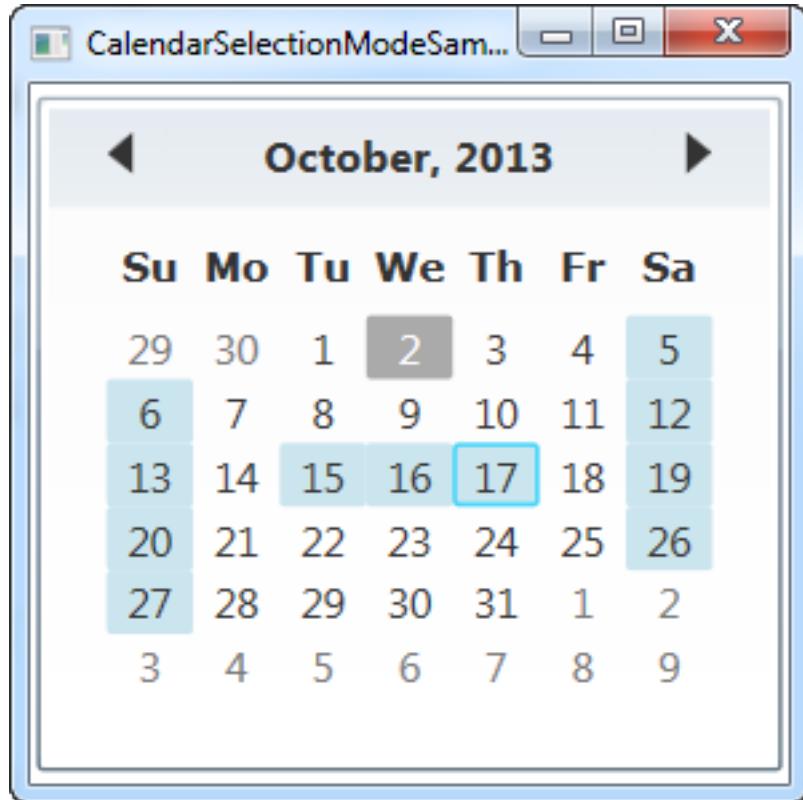


In the **SingleRange** SelectionMode, you can select an entire range of dates, either by holding down the left mouse button and dragging from one date to another or by holding down the Ctrl or Shift keys while clicking several dates, much like multi selection works in all parts of Windows. On the screenshot, I've selected an entire week, from Sunday to Monday, but you can just as easily select dates in the middle of the week and ranges which expands a single week.

SingleRange mode only allows a single range of dates to be selected though, much like the name suggests. This means that you can't select two dates which are not next to each other, and you can't select more than one range. If you want this, you should switch to **MultipleRange** selection:

```
<Calendar SelectionMode="MultipleRange" />
```

With this property, there are really no limits to the dates you can select. In this case, I've selected all the Saturdays, all the Sundays and a couple of week days in between.



Of course, if you don't want the ability to select one or several dates, you can set the **SelectionMode** to **None**.

Now let's discuss how we can work with the selected date(s) of the Calendar control.

#### 1.14.7.4. Working with the selected date

The **SelectedDate** property is all you need if you're only allowing single selections (see the above explanation on selection modes). It allows you to both set and get a currently selected date, from Code-behind as well as through a data binding.

Here's an example where we set the selected date to tomorrow from Code-behind and then use a data binding to read out the selected date to a TextBox control:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.CalendarSelectionSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarSelectionSample" Height="280" Width="220">
    <StackPanel Margin="10">
        <Calendar Name="cldSample" SelectionMode="MultipleRange"
SelectedDate="10.10.2013" />
        <Label>Selected date:</Label>
        <TextBox Text="{Binding ElementName=cldSample,"
```

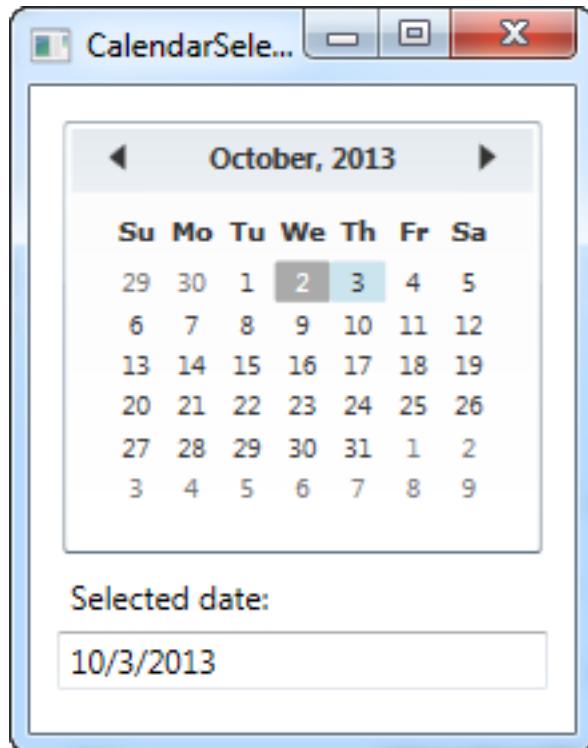
```

Path=SelectedDate, StringFormat=d, UpdateSourceTrigger=PropertyChanged}""
/>
</StackPanel>
</Window>

using System;
using System.Windows;

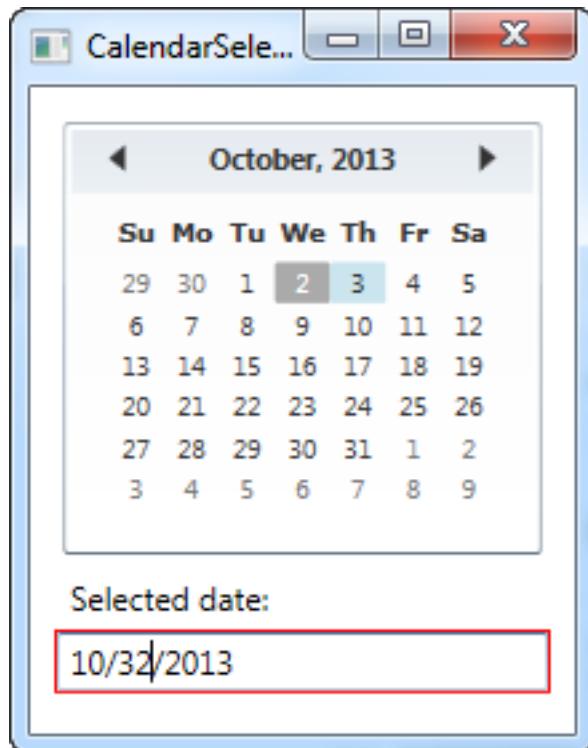
namespace WpfTutorialSamples.Misc_controls
{
    public partial class CalendarSelectionSample : Window
    {
        public CalendarSelectionSample()
        {
            InitializeComponent();
            cldSample.SelectedDate = DateTime.Now.AddDays(1);
        }
    }
}

```



In Code-behind, we simply set the **SelectedDate** property to the current date plus one day, meaning tomorrow. The user can then change this selection by clicking in the Calendar control, and through the data binding established in Text property of the TextBox, this change will automatically be reflected there.

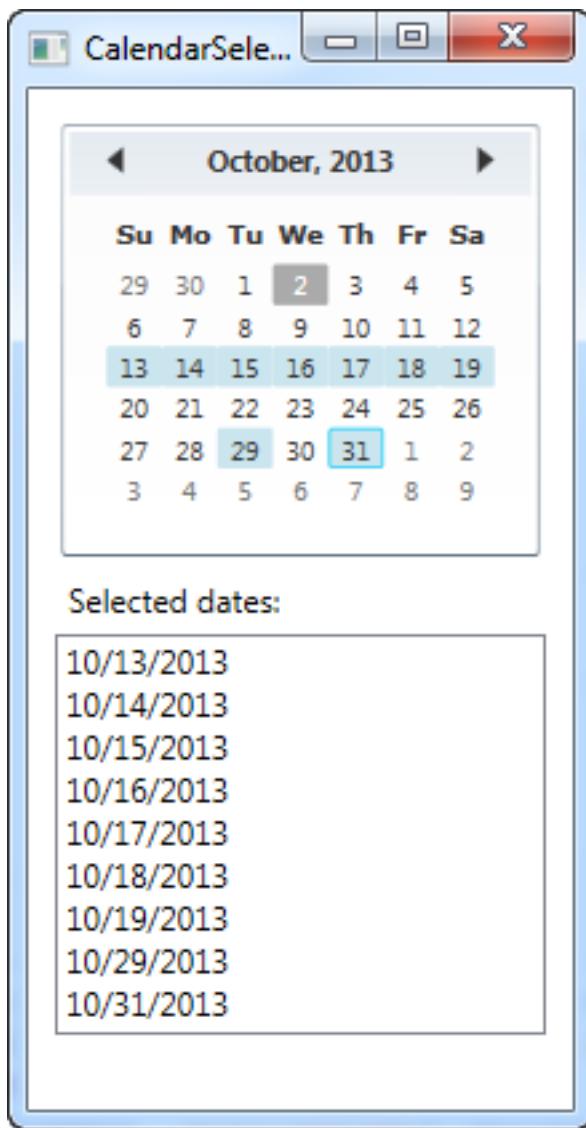
As an added bonus, through the magic of data binding, you can also change the value from the TextBox - just input a valid date and the change will be immediately reflected in the Calendar control. Should you enter a bad date, the automatic binding validation notifies you of the problem:



#### 1.14.7.5. Working with multiple selected dates

If you allow more than one selected date at the time, you won't find the SelectedDate property that useful. Instead, you should use the SelectedDates, which is a collection of currently selected dates in the Calendar control. This property can be accessed from Code-behind or used with a binding, like we do here:

```
<Window x:Class
    ="WpfTutorialSamples.Misc_controls.CalendarSelectedDatesSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarSelectedDatesSample" Height="420" Width="220">
    <StackPanel Margin="10">
        <Calendar Name="cldSample" SelectionMode="MultipleRange" />
        <Label>Selected dates:</Label>
        <ListBox ItemsSource="{Binding ElementName=cldSample,
Path=SelectedDates}" MinHeight="150" />
    </StackPanel>
</Window>
```



With a simple binding like that, we're now able to display a list of the currently selected dates.

If you want to react to dates being changed from Code-behind, you can subscribe to the **SelectedDatesChanged** event of the Calendar control.

#### 1.14.7.6. Blackout dates

Depending on what you use the Calendar control for, you may want to black out certain dates. This could be relevant e.g. in a booking application, where you want to prevent already reserved dates from being selected. The Calendar control supports this right out of the box through the use of the **BlackoutDates** collection, which you can of course use from both XAML and Code-behind:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.CalendarBlockedoutDatesSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarBlockedoutDatesSample" Height="300" Width="300">
```

```

>
<Viewbox>
    <Calendar Name="cldSample" SelectionMode="MultipleRange">
        <Calendar.BlackoutDates>
            <CalendarDateRange Start="10.13.2013" End
="10.19.2013" />
            <CalendarDateRange Start="10.27.2013" End
="10.31.2013" />
        </Calendar.BlackoutDates>
    </Calendar>
</Viewbox>
</Window>

using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfTutorialSamples.Misc_controls
{
    public partial class CalendarBlockedoutDatesSample : Window
    {
        public CalendarBlockedoutDatesSample()
        {
            InitializeComponent();
            cldSample.BlackoutDates.AddDatesInPast();
            cldSample.BlackoutDates.Add(new
CalendarDateRange(DateTime.Today, DateTime.Today.AddDays(1)));
        }
    }
}

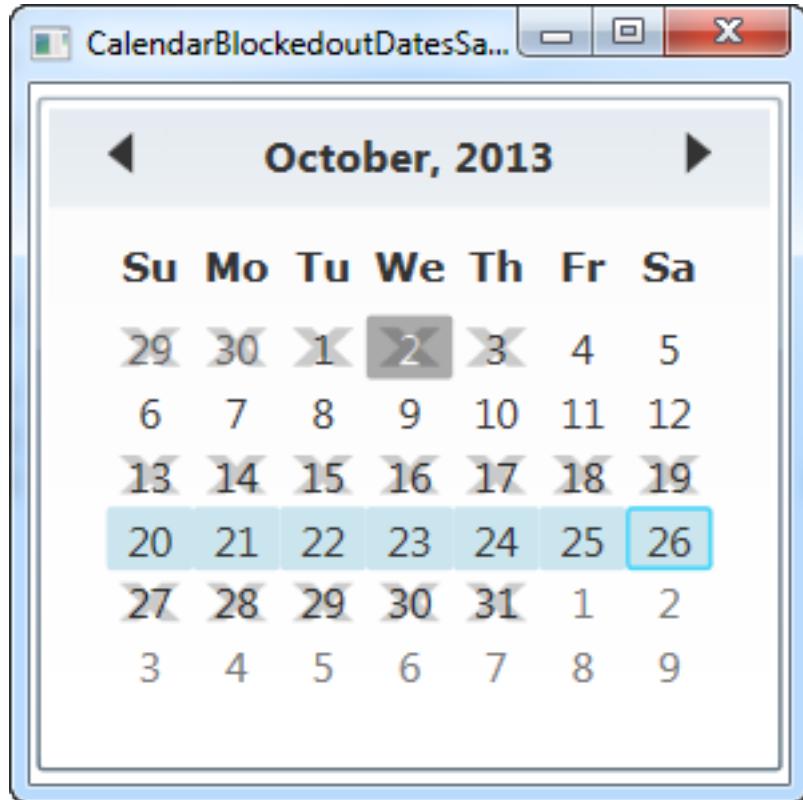
```

In this example, I demonstrate both ways of adding blacked out dates - through XAML and through Code-behind. Both ways works by adding instances of **CalendarDateRange** to the **BlackedoutDates** collection.

In XAML, I'm hardcoding the date ranges (mostly to show you it can be done that way too), while I do something a bit more clever in Code-behind, by first adding all past dates to the collection with a single call to the **AddDatesInPast()** method and then adding a range consisting of today and tomorrow.

#### 1.14.7.7. DisplayMode - showing months or years

The **DisplayMode** property can change the Calendar control from a place where you can select a date to a



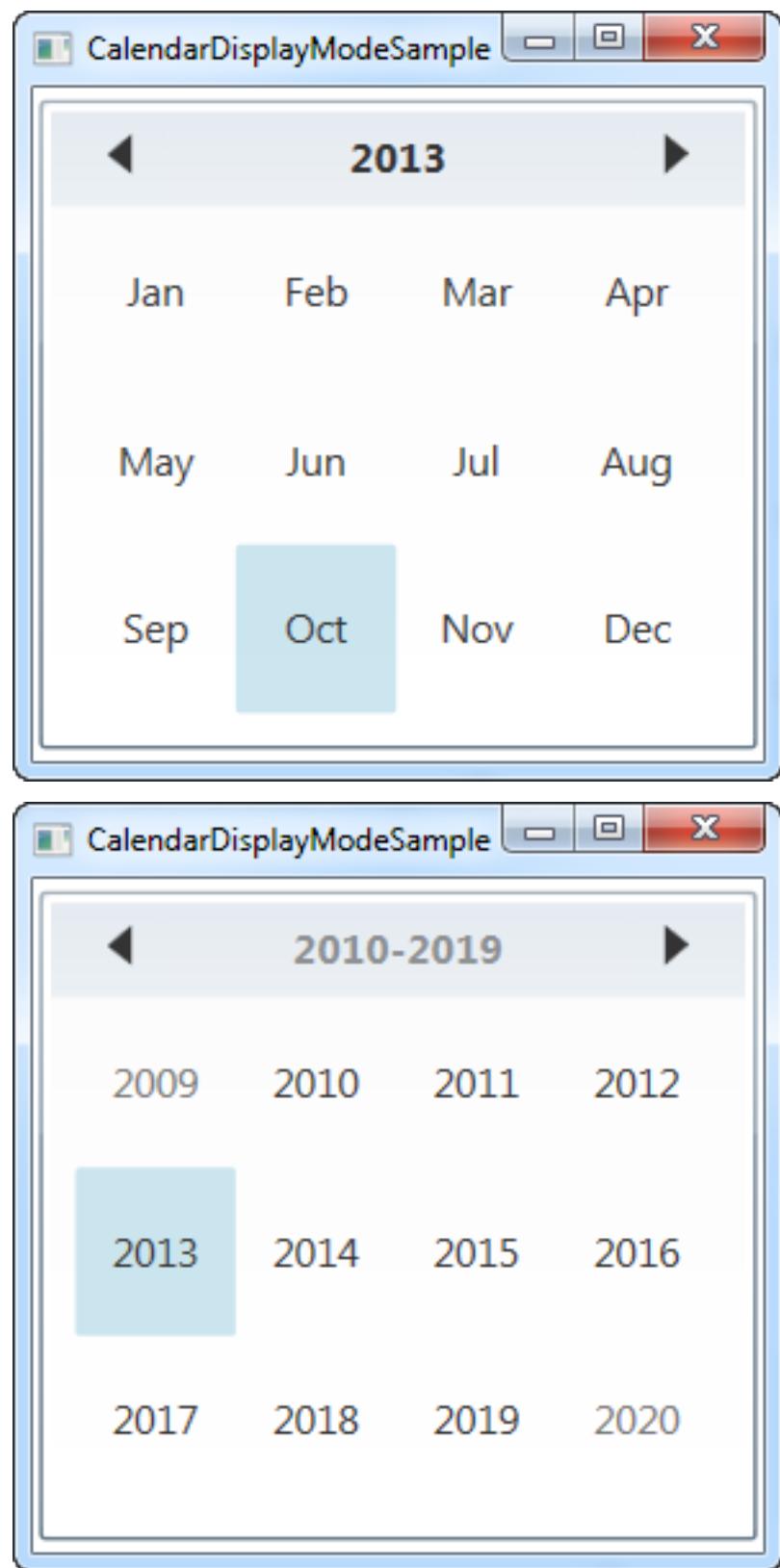
place where you can select a month or even a year. This is done through the **DisplayMode** property, which defaults to Month, which we've used in all the previous examples. Here's how it looks if we change it:

```
<Window x:Class
    ="WpfTutorialSamples.Misc_controls.CalendarDisplayModeSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CalendarDisplayModeSample" Height="300" Width="300">
    <Viewbox>
        <Calendar DisplayMode="Year" />
    </Viewbox>
</Window>
```

By setting the **DisplayMode** to **Year**, we can now select a month of a given year. You can change the year in the top, by using the arrows.

The Calendar control also allows for selecting an entire year, by using the **Decade** value for the **DisplayMode** property:

```
<Calendar DisplayMode="Decade" />
```



#### 1.14.7.8. Summary

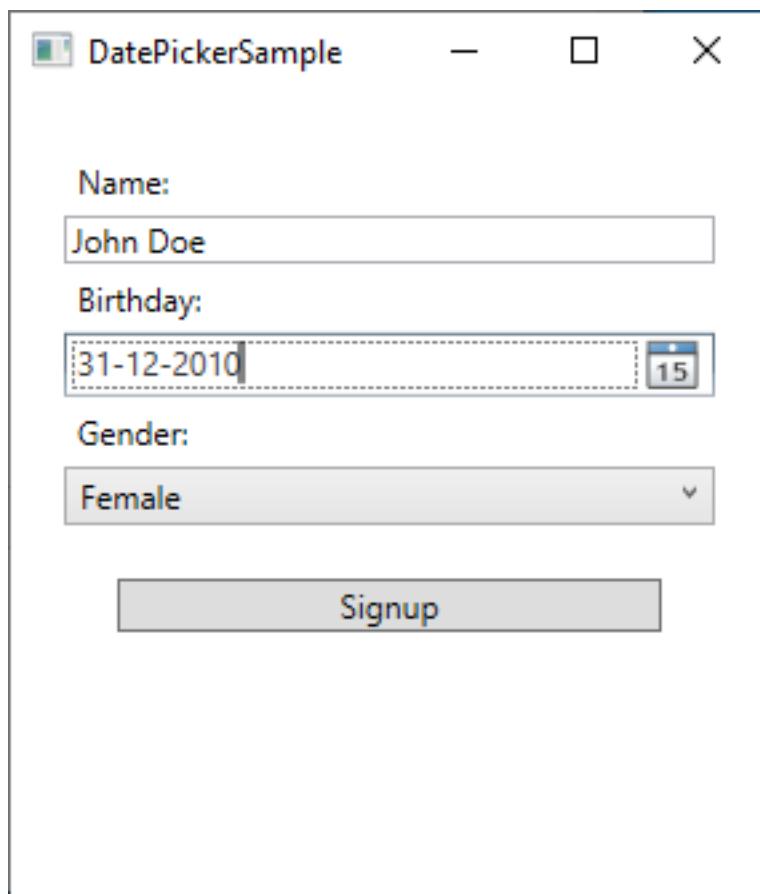
As you can see, the Calendar control is a very versatile control with lots of options and functionality, requiring only a minimum of configuration to use. If you're building an application with any sort of date-related functionality, you will probably be able to use the Calendar control, one way or another.

## 1.14.8. The DatePicker control

Dealing with dates in general can be cumbersome. There are many ways of writing a date, depending on where in the world your user(s) are, so allowing them to enter a date freely in a TextBox is almost never a good idea. Fortunately for us, WPF comes with several controls for dealing with dates.

We already looked at one of these controls, the *Calendar* control, which is great if selecting the date is the primary task of your dialog. However, often you will need to collect a date along with a lot of other information, in a form with multiple input controls like TextBox's, ComboBox's and so on. For a situation like that, you need a date-input control which can blend in with the rest and fit into the layout of a form - in other words, you need the **DatePicker** control!

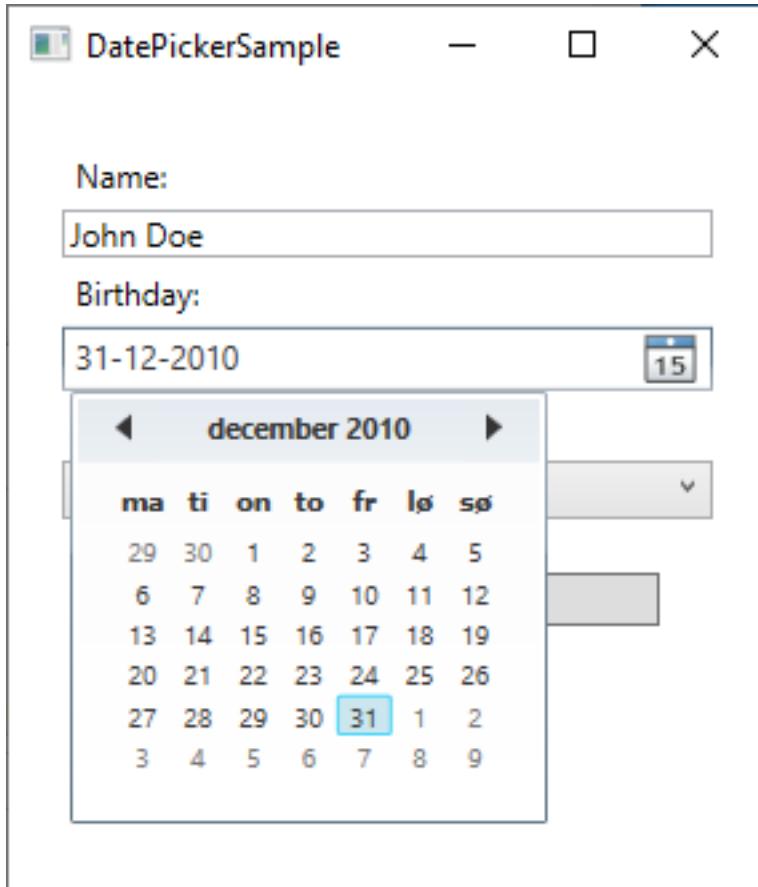
The DatePicker control will be displayed pretty much like a regular TextBox, but with a small button which will bring up a Calendar-view when clicked, allowing your user to select the date. Here's an example of how it could look:



You can then write the date manually or click the small button to select the date from a calendar control:

### 1.14.8.1. Adding a DatePicker control

The DatePicker control works straight out of the box - just add it anywhere in your Window and you're good to go:



```
<DatePicker></DatePicker>
```

Here's the full code listing used to create the example dialog above:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.DatePickerSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
            xmlns:local="clr-namespace:WpfTutorialSamples.Misc_controls"
            mc:Ignorable="d"
            Title="DatePickerSample" Height="300" Width="300">
<StackPanel Margin="20">
    <Label>Name:</Label>
    <TextBox />
    <Label>Birthday:</Label>
    <DatePicker></DatePicker>
    <Label>Gender:</Label>
    <ComboBox>
        <ComboBoxItem>Female</ComboBoxItem>

```

```

<ComboBoxItem>Male</ComboBoxItem>
</ComboBox>
<Button Margin="20">Signup</Button>
</StackPanel>
</Window>

```

### 1.14.8.2. DisplayDate and SelectedDate

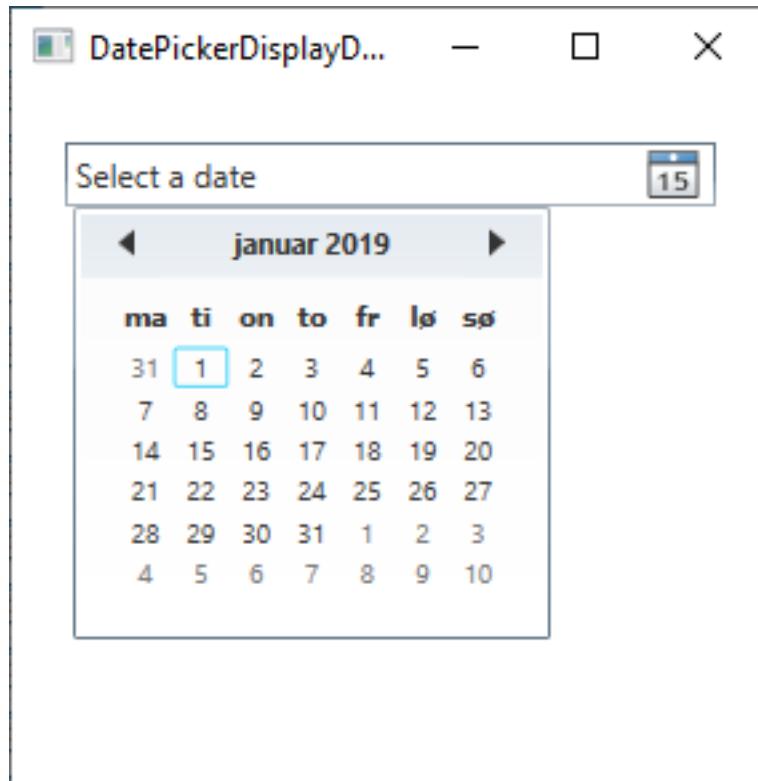
By default, the DatePicker control will not have a date selected - that is left to the user. However, if you need to prefill the control with a date, just use the **SelectedDate** property, like this:

```
<DatePicker SelectedDate="2000-12-31"></DatePicker>
```

The DatePicker will now have a pre-selected date, which the user can choose to override by selecting/entering another date. The **SelectedDate** can also be set from Code-behind, and perhaps more importantly, it can also be read from Code-behind, or you can bind its value to your Model or another control.

Sometimes you might need to start the calendar at a specific date, without actually selecting one for the user. For that, we have the **DisplayDate** property. The default value is the current date, but you can easily change that:

```
<DatePicker Name="dp1" DisplayDate="2019-01-01" />
```



Notice how, when we use the **DisplayDate** property, the calendar starts at the specified date (and highlights

it), but no date is actually selected (as indicated by the "Select a date" text).

#### 1.14.8.3. SelectedDateFormat

Another interesting property is the **SelectedDateFormat**. The default value is **Short**, but if you change it to **Long**, it will be formatted in a slightly more verbose way:

```
<DatePicker SelectedDate="2000-12-31" SelectedDateFormat="Long"></
DatePicker>
```



Whether the Short or the Long format is used, the actual format of the date is decided by the culture of your application. If you don't specifically define a culture for your application, the system settings are used. You will notice from the screenshots of this article that on this computer, the date format is DMY (date-month-year), but this can easily be changed by setting a specific culture. We'll discuss that elsewhere in this tutorial.

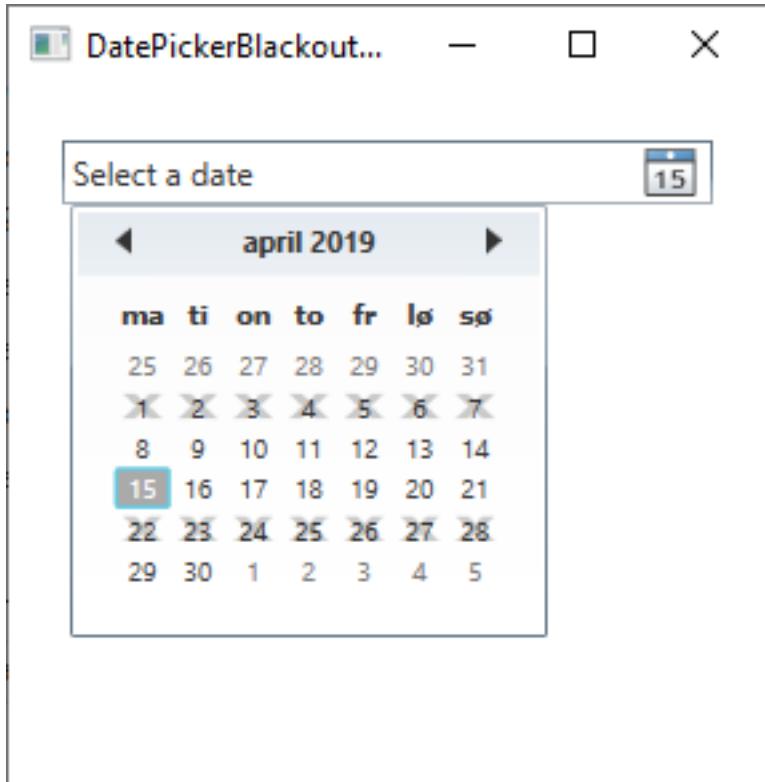
#### 1.14.8.4. Blackout dates

Depending on what you use the DatePicker control for, you may want to black out certain dates. This will prevent the selection of these dates, which will be visually indicated, and could be relevant e.g. in a booking application, where you want to prevent already reserved dates from being selected. The DatePicker control supports this right out of the box through the use of the **BlackoutDates** collection, which you can of course use from both XAML and Code-behind. Here's how to do it with XAML:

```
<DatePicker Name="dp1">
    <DatePicker.BlackoutDates>
        <CalendarDateRange Start="2019-04-01" End="2019-04-07" />
        <CalendarDateRange Start="2019-04-22" End="2019-04-28" />
    </DatePicker.BlackoutDates>
</DatePicker>
```

The result will look like this:

Doing it from Code-behind is just as easy and it has two added benefits: First of all, you can create the date range dynamically, e.g. based on the current date. You can also use the **AddDatesInPast()** method to



automatically exclude all dates in the past. Here's an example:

```
dp1.BlackoutDates.AddDatesInPast();
dp1.BlackoutDates.Add(new CalendarDateRange(DateTime.Now,
DateTime.Now.AddDays(7))');
```

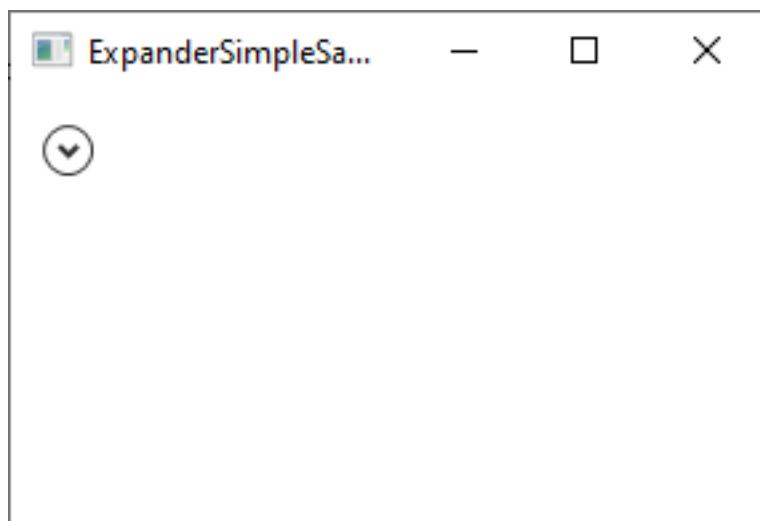
With that in place, all past dates, as well as the next week, will be unavailable for selection.

#### 1.14.8.5. Summary

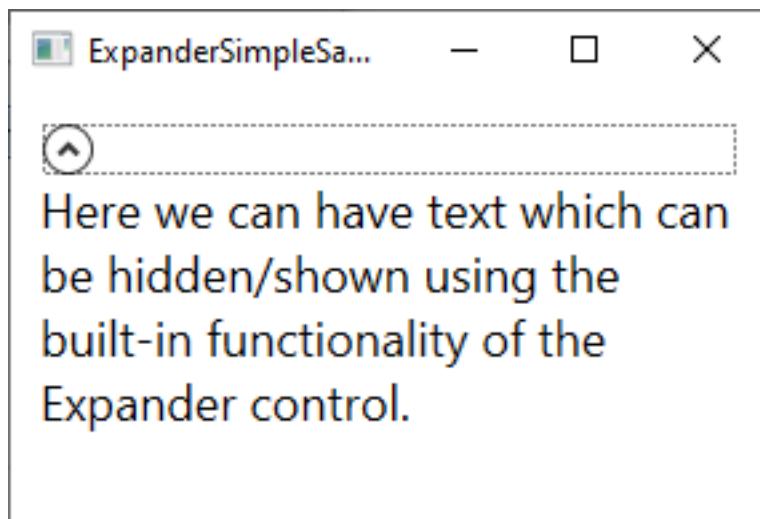
The **DatePicker** control allows the user to specify a valid date, either by writing it in the text box or by selecting it from the built-in calendar widget. If the date is entered manually, it will be validated immediately and only allowed to remain in the text box if it's valid. This will make it much easier for you to create forms which includes dates.

## 1.14.9. The Expander control

The **Expander** control will provide you with the ability to hide/show a piece of content. This would usually be a piece of text, but thanks to the flexibility of WPF, it can be used for any type of mixed content like texts, images and even other WPF controls. To see what I'm talking about, here's an example:



Notice the arrow part - as soon as you click it, the Expander control will expand and reveal its content:



The code for it is of course very simple:

```
<Expander>
<TextBlock TextWrapping="Wrap" FontSize="18">
    Here we can have text which can be hidden/shown using the built-in
    functionality of the Expander control.
</TextBlock>
</Expander>
```

By default, the Expander is NOT expanded and therefore looks like it does on the first screenshot. The user

can expand it by clicking it or you can make it initially expanded by using the **IsExpanded** property:

```
<Expander IsExpanded="True">
```

You can of course also read this property at runtime, if you need to know about the current state of the Expander control.

#### 1.14.9.1. Advanced content

The Content of the Expander can only be one control, like in our first example where we use a TextBlock control, but nothing prevents you from making this e.g. a Panel, which can then hold as many child controls as you want it to. This allows you to host rich content inside your Expander, from text and images to e.g. a ListView or any other WPF control.

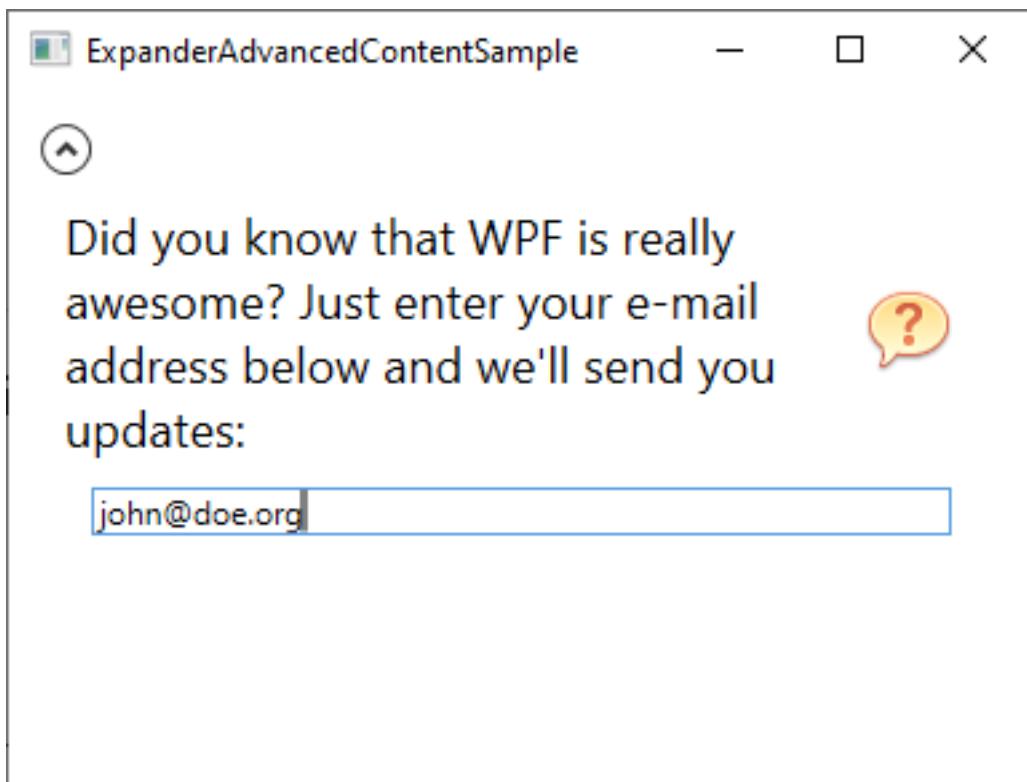
Here's an example of more advanced content, where we use several panels, text and an image and even a TextBox control:

```
<Expander Margin="10">
    <StackPanel Margin="10">
        <DockPanel>
            <Image Source
="~/WpfTutorialSamples/component/Images/question32.png" Width="32" Height
="32" DockPanel.Dock="Right" Margin="10"></Image>
            <TextBlock TextWrapping="Wrap" FontSize="18">
                Did you know that WPF is really awesome? Just
                enter your e-mail address below and we'll send you updates:
            </TextBlock>
        </DockPanel>
        <TextBox Margin="10">john@doe.org</TextBox>
    </StackPanel>
</Expander>
```

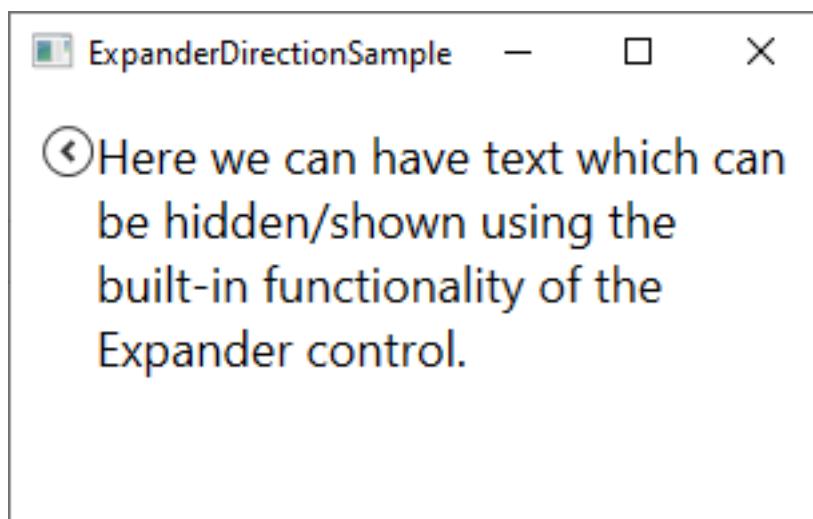
#### 1.14.9.2. ExpandDirection

By default, the Expander control will occupy all available space inside its container control and then expand in accordance with the value of the **ExpandDirection** property, which is set to **Down** as default. You can see this indicated on the screenshots above because the arrow is placed in top of the control and it points up or down based on whether the control has been expanded or not.

If you change the value of the **ExpandDirection** property, it will affect how the Expander control acts and looks. For instance, if you change the value to **Right**, the arrow will be placed on the left side and point to the left/right instead of up/down. Here's an example:



```
<Expander Margin="10" ExpandDirection="Right">
    <TextBlock TextWrapping="Wrap" FontSize="18">
        Here we can have text which can be hidden/shown using the
        built-in functionality of the Expander control.
    </TextBlock>
</Expander>
```

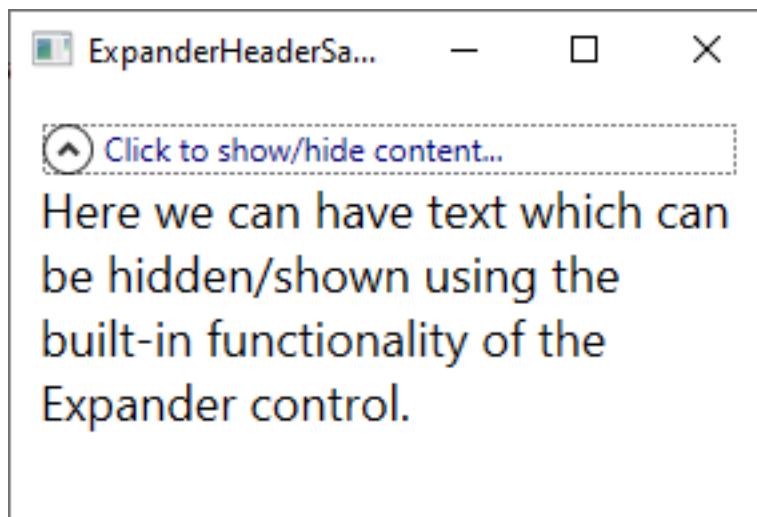


You can of course set this property to **Up** or **Left** as well - if you do so, the button will be placed at the bottom or to the right.

#### 1.14.9.3. Custom header

In all the examples so far, the Expander control is almost look-less, except for the button which is used to show/hide the content - it's drawn as a circular button with an arrow inside. You can easily customize the header-area of the control though, using the **Header** property. Here's an example where we use this property to add an explanatory text next to the button:

```
<Expander Margin="10" Header="Click to show/hide content...">
    <TextBlock TextWrapping="Wrap" FontSize="18">
        Here we can have text which can be hidden/shown using the
        built-in functionality of the Expander control.
    </TextBlock>
</Expander>
```

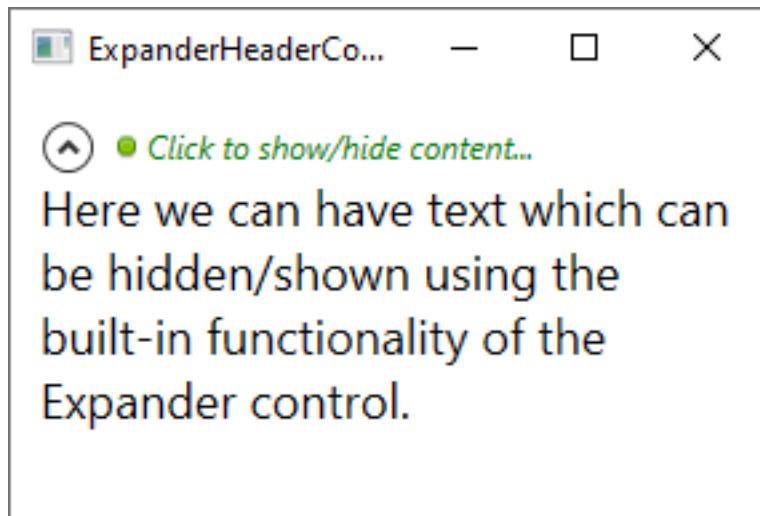


But you don't have to settle for a simple piece of text - the **Header** property will allow you to add controls to it, to create an even more customized look:

```
<Expander Margin="10">
    <Expander.Header>
        <DockPanel VerticalAlignment="Stretch">
            <Image Source
                = "/WpfTutorialSamples;component/Images/bullet_green.png" Height="16"
                DockPanel.Dock="Left" />
            <TextBlock FontStyle="Italic" Foreground="Green">Click to
            show/hide content...</TextBlock>
        </DockPanel>
    </Expander.Header>
    <TextBlock TextWrapping="Wrap" FontSize="18">
        Here we can have text which can be hidden/shown using the
        built-in functionality of the Expander control.
    </TextBlock>
```

```
</Expander>
```

Notice how I simply add a Panel as the content of the **Header** property and inside of that, I can do whatever I want, like adding an Image and a TextBlock control with custom formatting:



#### 1.14.9.4. Summary

The Expander control is a great little helper when you need the ability to hide/show content on demand, and much like any other control in the WPF framework, it's both easy to use and easy to customize.

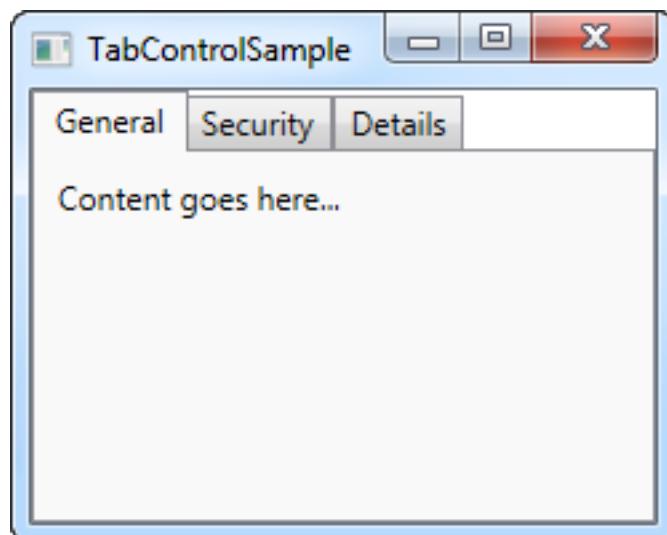
## 1.15. The TabControl

### 1.15.1. Using the WPF TabControl

The WPF TabControl allows you to split your interface up into different areas, each accessible by clicking on the tab header, usually positioned at the top of the control. Tab controls are commonly used in Windows applications and even within Windows' own interfaces, like the properties dialog for files/folders etc.

Just like with most other WPF controls, the TabControl is very easy to get started with. Here's a very basic example:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.TabControlSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TabControlSample" Height="200" Width="250">
    <Grid>
        <TabControl>
            <TabItem Header="General">
                <Label Content="Content goes here..." />
            </TabItem>
            <TabItem Header="Security" />
            <TabItem Header="Details" />
        </TabControl>
    </Grid>
</Window>
```



As you can see, each tab is represented with a **TabItem** element, where the text shown on it is controlled

by the **Header** property. The TabItem element comes from the ContentControl class, which means that you may define a single element inside of it that will be shown if the tab is active (like on the screenshot). I used a Label in this example, but if you want to place more than one control inside of the tab, just use one of the panels with child controls inside of it.

### 1.15.1.1. Customized headers

Once again, WPF proves to be extremely flexible when you want to customize the look of your tabs. Obviously the content can be rendered any way you like it, but so can the tab headers! The Header property can be filled with anything you like, which we'll take advantage of in the next example:

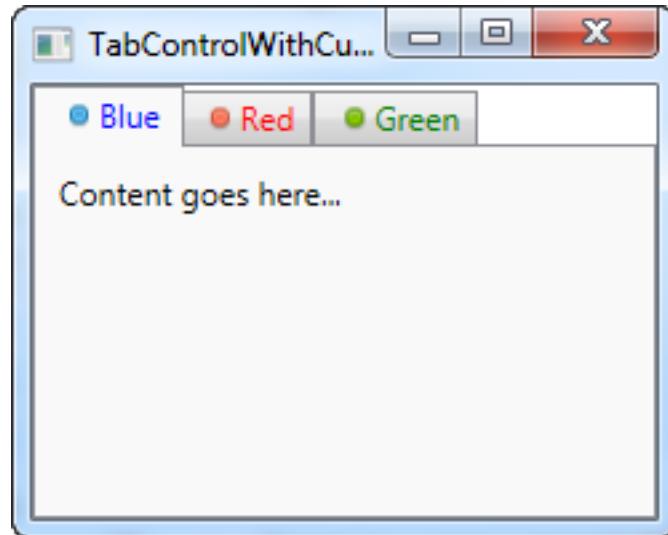
```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.TabControlWithCustomHeadersSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TabControlWithCustomHeadersSample" Height="200" Width
        ="250">
    <Grid>
        <Grid>
            <TabControl>
                <TabItem>
                    <TabItem.Header>
                        <StackPanel Orientation="Horizontal">
                            <Image Source
                                =" /WpfTutorialSamples;component/Images/bullet_blue.png" />
                            <TextBlock Text="Blue" Foreground
                                ="Blue" />
                        </StackPanel>
                    </TabItem.Header>
                    <Label Content="Content goes here..." />
                </TabItem>
                <TabItem>
                    <TabItem.Header>
                        <StackPanel Orientation="Horizontal">
                            <Image Source
                                =" /WpfTutorialSamples;component/Images/bullet_red.png" />
                            <TextBlock Text="Red" Foreground
                                ="Red" />
                        </StackPanel>
                    </TabItem.Header>
                </TabItem>
            </TabControl>
        </Grid>
    </Grid>

```

```

<TabItem>
    <TabItem.Header>
        <StackPanel Orientation="Horizontal">
            <Image Source
                ="/WpfTutorialSamples/component/Images/bullet_green.png" />
            <TextBlock Text="Green" Foreground
                ="Green" />
        </StackPanel>
    </TabItem.Header>
</TabItem>
</TabControl>
</Grid>
</Grid>
</Window>

```



The amount of markup might be a bit overwhelming, but as you can probably see once you dig into it, it's all very simple. Each of the tabs now has a `TabControl.Header` element, which contains a `StackPanel`, which in turn contains an `Image` and a `TextBlock` control. This allows us to have an image on each of the tabs as well as customize the color of the text (we could have made it bold, italic or another size as well).

### 1.15.1.2. Controlling the TabControl

Sometimes you may wish to control which tab is selected programmatically or perhaps get some information about the selected tab. The WPF `TabControl` has several properties which makes this possible, including `SelectedIndex` and `SelectedItem`. In the next example, I've added a couple of buttons to the first example which allows us to control the `TabControl`:

```

<Window x:Class
    ="WpfTutorialSamples.Misc_controls.ControllingTheTabControlSample"

```

```

    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ControllingTheTabControlSample" Height="300" Width
    ="350">>
    <DockPanel>
        <StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom"
Margin="2,5">
            <Button Name="btnPreviousTab" Click
="btnPreviousTab_Click">Prev.</Button>
            <Button Name="btnNextTab" Click="btnNextTab_Click">Next</
Button>
            <Button Name="btnSelectedTab" Click
="btnSelectedTab_Click">Selected</Button>
        </StackPanel>
        <TabControl Name="tcSample">
            <TabItem Header="General">
                <Label Content="Content goes here..." />
            </TabItem>
            <TabItem Header="Security" />
            <TabItem Header="Details" />
        </TabControl>
    </DockPanel>
</Window>

using System;
using System.Windows;
using System.Windows.Controls;

namespace WpfTutorialSamples.Misc_controls
{
    public partial class ControllingTheTabControlSample : Window
    {
        public ControllingTheTabControlSample()
        {
            InitializeComponent();
        }

        private void btnPreviousTab_Click(object sender,

```

```

RoutedEventArgs e)
{
    int newIndex = tcSample.SelectedIndex - 1;
    if(newIndex < 0)
        newIndex = tcSample.Items.Count - 1;
    tcSample.SelectedIndex = newIndex;
}

private void btnNextTab_Click(object sender, RoutedEventArgs e)
{
    int newIndex = tcSample.SelectedIndex + 1;
    if(newIndex >= tcSample.Items.Count)
        newIndex = 0;
    tcSample.SelectedIndex = newIndex;
}

private void btnSelectedTab_Click(object sender,
RoutedEventArgs e)
{
    MessageBox.Show("Selected tab: " + (tcSample.SelectedItem
as TabItem).Header);
}
}
}

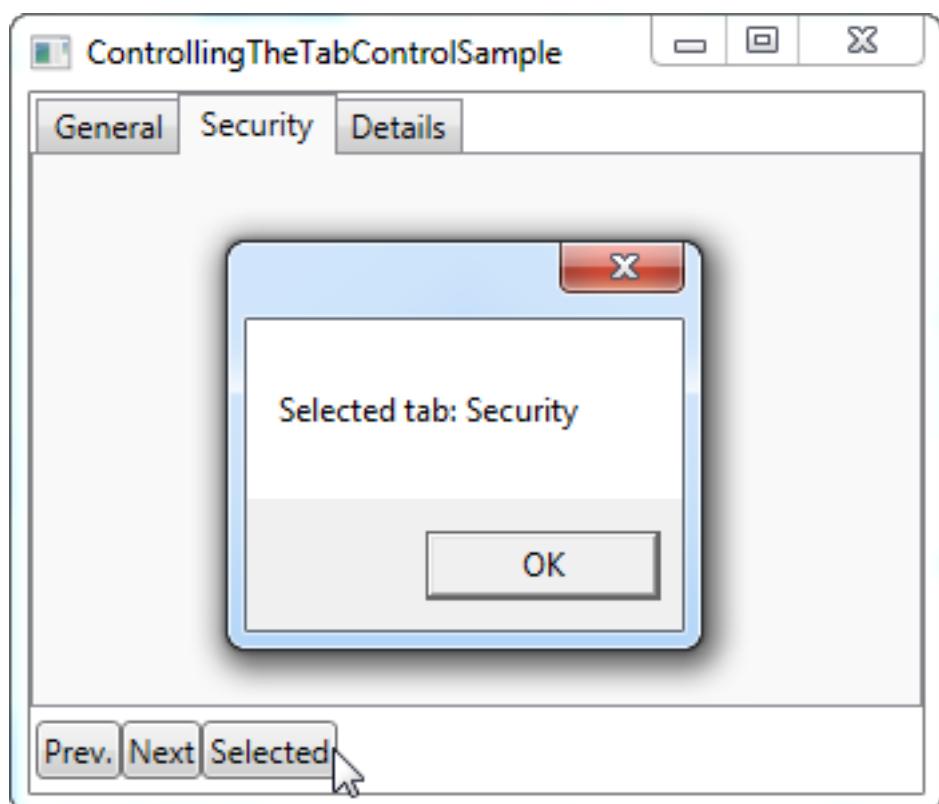
```

As you can see, I've simply added a set of buttons in the lower part of the interface. The first two allows will select the previous or next tab on the control, while the last one will display information about the currently selected tab, as demonstrated on the screenshot.

The first two buttons uses the **SelectedIndex** property to determine where we are and then either subtracts or adds one to that value, making sure that the new index doesn't fall below or above the amount of available items. The third button uses the **SelectedItem** property to get a reference to the selected tab. As you can see, I have to typecast it into the TabItem class to get a hold of the header property, since the SelectedProperty is of the object type by default.

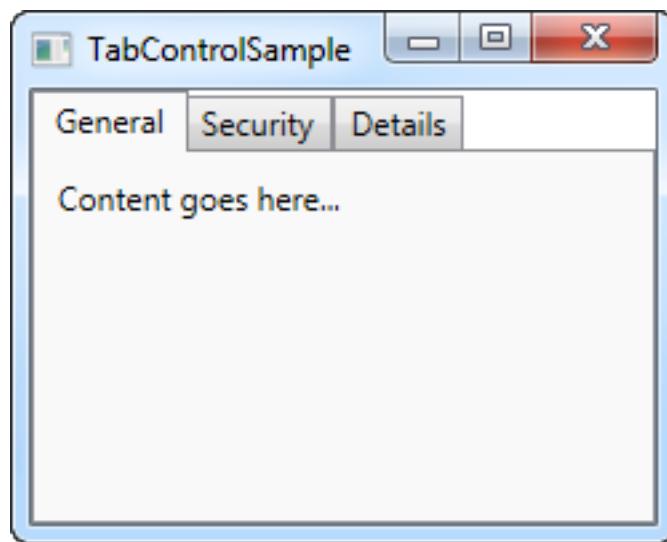
### 1.15.1.3. Summary

The TabControl is great when you need a clear separation in a dialog or when there's simply not enough space for all the controls you want in it. In the next couple of chapters, we'll look into some of the possibilities there are when using the TabControl for various purposes.



## 1.15.2. WPF TabControl: Tab positions

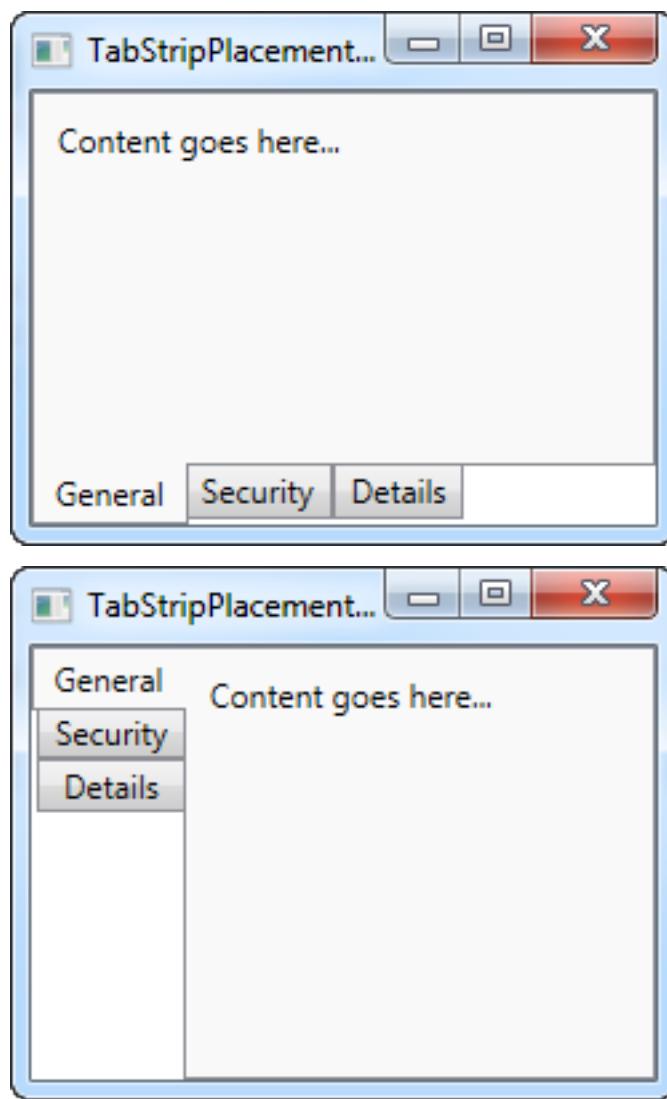
The tabs of a TabControl is usually placed on top of the control, which is also how it will look by default when using the WPF TabControl:



However, using the **TabStripPlacement** property, we can very easily change this:

```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.TabStripPlacementSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TabStripPlacementSample" Height="200" Width="250">
    <Grid>
        <TabControl TabStripPlacement="Bottom">
            <TabItem Header="General">
                <Label Content="Content goes here..." />
            </TabItem>
            <TabItem Header="Security" />
            <TabItem Header="Details" />
        </TabControl>
    </Grid>
</Window>
```

The TabStripPlacement can be set to Top, Bottom, Left and Right. However, if we set it to Left or Right, we get a result like this:



I personally would expect that the tabs to be rotated when placed on one of the sides, so that the tab text becomes vertical instead of horizontal, but the WPF TabControl doesn't do this. Fortunately, we can accomplish this behavior with a small hack:

```
<Window x:Class
        ="WpfTutorialSamples.Misc_controls.TabStripPlacementSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TabStripPlacementSample" Height="200" Width="250"
        UseLayoutRounding="True">
    <Grid>
        <TabControl TabStripPlacement="Left">
            <TabControl.Resources>
                <Style TargetType="{x:Type TabItem}">
                    <Setter Property="HeaderTemplate">
                        <Setter.Value>
```

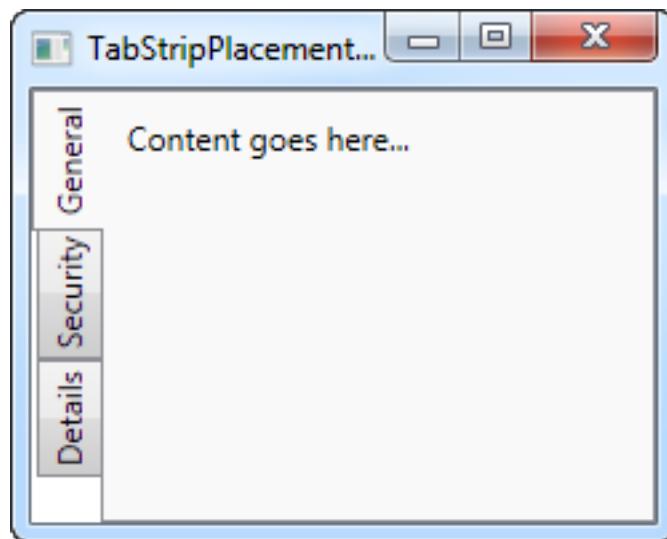
```

        <DataTemplate>
            <ContentPresenter Content="

{TemplateBinding Content}">
                <
ContentPresenter.LayoutTransform>
                    <RotateTransform Angle="

=270" />
                </
ContentPresenter.LayoutTransform>
            </ContentPresenter>
        </DataTemplate>
    </Setter.Value>
</Setter>
<Setter Property="Padding" Value="3" />
</Style>
</TabControl.Resources>
<TabItem Header="General">
    <Label Content="Content goes here..." />
</TabItem>
<TabItem Header="Security" />
<TabItem Header="Details" />
</TabControl>
</Grid>
</Window>

```



If you haven't yet read the chapters on templates or styles, this might seem a bit confusing, but what we do is using a style targeted at the `TabItem` elements, where we override the `HeaderTemplate` and then apply a rotate transform to the tabs. For tabs placed on the left side, we rotate 270 degrees - if placed on the right,

you should only rotate 90 degrees, to make it look correct.

### 1.15.3. WPF TabControl: Styling the TabItems

In one of the previous articles, we discovered how easy it was to customize the tab headers of the WPF TabControl, for instance to add an image or color the text. However, if you wish to go beyond that and directly influence how the tab looks, including shape and borders, you need to override the control template of the TabItem element, and while this is not as straight forward as most other areas of WPF, it's still manageable.

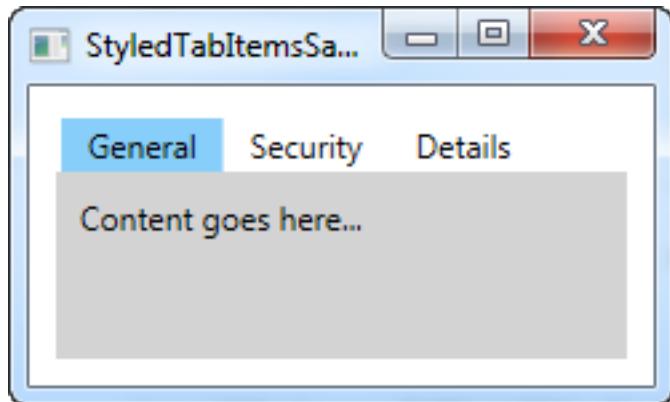
So, if you would like to get full control of how the tabs of your TabControl looks, check out the next example:

```
<Window x:Class="WpfTutorialSamples.Misc_controls.StyledTabItemsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StyledTabItemsSample" Height="150" Width="250">
    <Grid>
        <TabControl Margin="10" BorderThickness="0" Background
        ="LightGray">
            <TabControl.Resources>
                <Style TargetType="TabItem">
                    <Setter Property="Template">
                        <Setter.Value>
                            <ControlTemplate TargetType="TabItem">
                                <>
                                    <Grid Name="Panel">
                                        <ContentPresenter x:Name
                                        ="ContentSite"
                                        VerticalAlignment
                                        ="Center"
                                        HorizontalAlignment
                                        ="Center"
                                        ContentSource="Header"
                                        Margin="10,2"/>
                                    </Grid>
                                    <ControlTemplate.Triggers>
                                        <Trigger Property
                                        ="IsSelected" Value="True">
                                            <Setter TargetName
                                            ="Panel" Property="Background" Value="LightSkyBlue" />
                                        </Trigger>
                                        <Trigger Property
```

```

        ="IsSelected" Value="False">
            <Setter TargetName
        ="Panel" Property="Background" Value="White" />
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
</Setter.Value>
</Setter>
</Style>
</TabControl.Resources>
<TabItem Header="General">
    <Label Content="Content goes here..." />
</TabItem>
<TabItem Header="Security" />
<TabItem Header="Details" />
</TabControl>
</Grid>
</Window>

```



As you can see, this makes the TabControl looks a bit Windows 8'ish, with no borders and a less subtle color to mark the selected tab and no background for the unselected tabs. All of this is accomplished by changing the ControlTemplate, using a Style. By adding a **ContentPresenter** control, we specify where the content of the TabItem should be placed. We also have a couple of triggers, which controls the background color of the tabs based on the **IsSelected** property.

In case you want a less subtle look, it's as easy as changing the template. For instance, you might want a border, but with round corners and a gradient background - no problem! Check out this next example, where we accomplish just that:

```

<Window x:Class
    ="WpfTutorialSamples.Misc_controls.StyledTabItemsWithBorderSample"
    xmlns

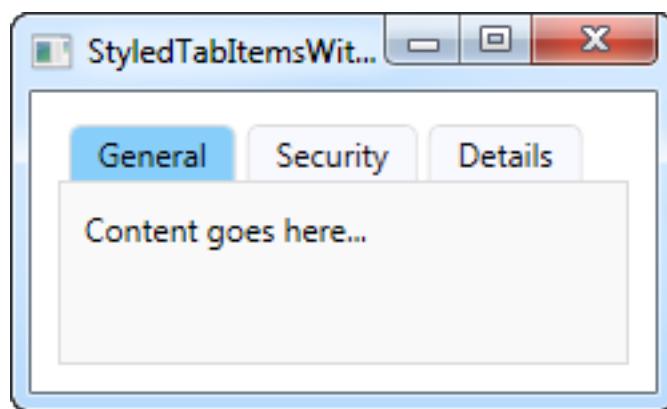
```

```

="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyledTabItemsWithBorderSample" Height="150" Width
="250">
    <Grid>
        <TabControl Margin="10" BorderBrush="Gainsboro">
            <TabControl.Resources>
                <Style TargetType="TabItem">
                    <Setter Property="Template">
                        <Setter.Value>
                            <ControlTemplate TargetType="TabItem">
                                <Border Name="Border"
BorderThickness="1,1,1,0" BorderBrush="Gainsboro" CornerRadius="4,4,0,0"
Margin="2,0">
                                    <ContentPresenter x:Name
="ContentSite"
VerticalAlignment="Center"
HorizontalContentAlignment="Center"
ContentSource="Header"
Margin="10,2"/>
                                    </Border>
                                    <ControlTemplate.Triggers>
                                        <Trigger Property
="IsSelected" Value="True">
                                            <Setter TargetName
="Border" Property="Background" Value="LightSkyBlue" />
                                            </Trigger>
                                        <Trigger Property
="IsSelected" Value="False">
                                            <Setter TargetName
="Border" Property="Background" Value="GhostWhite" />
                                            </Trigger>
                                    </ControlTemplate.Triggers>
                                </ControlTemplate>
                            </Setter.Value>
                        </Setter>
                    </Style>
                </TabControl.Resources>
            </TabControl>
        </Grid>

```

```
</TabControl.Resources>
<TabItem Header="General">
    <Label Content="Content goes here..." />
</TabItem>
<TabItem Header="Security" />
<TabItem Header="Details" />
</TabControl>
</Grid>
</Window>
```



As you can see, I pretty much just added a Border control around the ContentPresenter to achieve this changed look. Hopefully this should demonstrate just how easy it is to get custom styled tabs and how many possibilities there are in this technique.

# 1.16. List controls

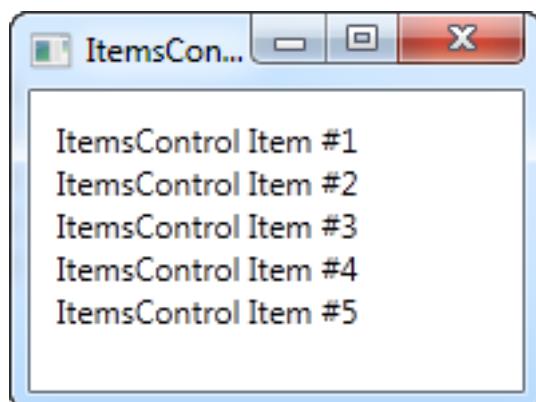
## 1.16.1. The ItemsControl

WPF has a wide range of controls for displaying a list of data. They come in several shapes and forms and vary in how complex they are and how much work they perform for you. The simplest variant is the `ItemsControl`, which is pretty much just a markup-based loop - you need to apply all the styling and templating, but in many cases, that's just what you need.

### 1.16.1.1. A simple `ItemsControl` example

Let's kick off with a very simple example, where we hand-feed the `ItemsControl` with a set of items. This should show you just how simple the `ItemsControl` is:

```
<Window x:Class="WpfTutorialSamples.ItemsControl.ItemsControlSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="ItemsControlSample" Height="150" Width="200">
    <Grid Margin="10">
        <ItemsControl>
            <system:String>ItemsControl Item #1</system:String>
            <system:String>ItemsControl Item #2</system:String>
            <system:String>ItemsControl Item #3</system:String>
            <system:String>ItemsControl Item #4</system:String>
            <system:String>ItemsControl Item #5</system:String>
        </ItemsControl>
    </Grid>
</Window>
```



As you can see, there is nothing that shows that we're using a control for repeating the items instead of just manually adding e.g. 5 TextBlock controls - the ItemsControl is completely lookless by default. If you click on one of the items, nothing happens, because there's no concept of selected item(s) or anything like that.

### 1.16.1.2. ItemsControl with data binding

Of course the ItemsControl is not meant to be used with items defined in the markup, like we did in the first example. Like pretty much any other control in WPF, the ItemsControl is made for data binding, where we use a template to define how our code-behind classes should be presented to the user.

To demonstrate that, I've whipped up an example where we display a TODO list to the user, and to show you just how flexible everything gets once you define your own templates, I've used a ProgressBar control to show you the current completion percentage. First some code, then a screenshot and then an explanation of it all:

```
<Window x:Class
        ="WpfTutorialSamples.ItemsControl.ItemsControlDataBindingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ItemsControlDataBindingSample" Height="150" Width="300"
>
    <Grid Margin="10">
        <ItemsControl Name="icTodoList">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <Grid Margin="0,0,0,5">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="*" />
                            <ColumnDefinition Width="100" />
                        </Grid.ColumnDefinitions>
                        <TextBlock Text="{Binding Title}" />
                        <ProgressBar Grid.Column="1" Minimum="0"
Maximum="100" Value="{Binding Completion}" />
                    </Grid>
                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>
    </Grid>
</Window>

using System;
```

```

using System.Windows;
using System.Collections.Generic;

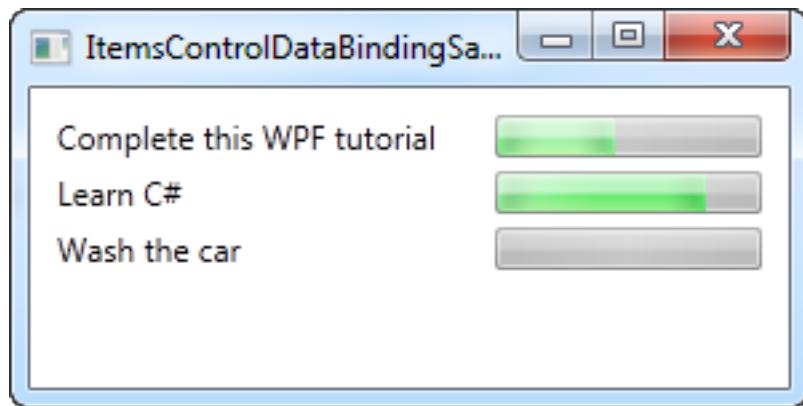
namespace WpfTutorialSamples.ItemsControl
{
    public partial class ItemsControlDataBindingSample : Window
    {
        public ItemsControlDataBindingSample()
        {
            InitializeComponent();

            List<TodoItem> items = new List<TodoItem>();
            items.Add(new TodoItem() { Title = "Complete this WPF tutorial", Completion = 45 });
            items.Add(new TodoItem() { Title = "Learn C#", Completion = 80 });
            items.Add(new TodoItem() { Title = "Wash the car", Completion = 0 });

            icToDoList.ItemsSource = items;
        }
    }

    public class TodoItem
    {
        public string Title { get; set; }
        public int Completion { get; set; }
    }
}

```



The most important part of this example is the template that we specify inside of the ItemsControl, using a

DataTemplate tag inside of the ItemsControl.ItemTemplate. We add a Grid panel, to get two columns: In the first we have a TextBlock, which will show the title of the TODO item, and in the second column we have a ProgressBar control, which value we bind to the Completion property.

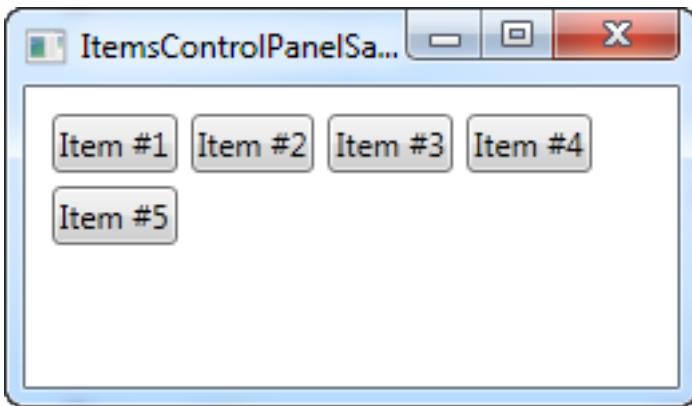
The template now represents a TodoItem, which we declare in the Code-behind file, where we also instantiate a number of them and add them to a list. In the end, this list is assigned to the **ItemsSource** property of our ItemsControl, which then does the rest of the job for us. Each item in the list is displayed by using our template, as you can see from the resulting screenshot.

### 1.16.1.3. The ItemsPanelTemplate property

In the above examples, all items are rendered from top to bottom, with each item taking up the full row. This happens because the ItemsControl throw all of our items into a vertically aligned StackPanel by default. It's very easy to change though, since the ItemsControl allows you to change which panel type is used to hold all the items. Here's an example:

```
<Window x:Class
        ="WpfTutorialSamples.ItemsControl.ItemsControlPanelSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="ItemsControlPanelSample" Height="150" Width="250">
    <Grid Margin="10">
        <ItemsControl>
            <ItemsControl.ItemsPanel>
                <ItemsPanelTemplate>
                    <WrapPanel />
                </ItemsPanelTemplate>
            </ItemsControl.ItemsPanel>
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <Button Content="{Binding}" Margin="0,0,5,5" />
                </DataTemplate>
            </ItemsControl.ItemTemplate>
            <system:String>Item #1</system:String>
            <system:String>Item #2</system:String>
            <system:String>Item #3</system:String>
            <system:String>Item #4</system:String>
            <system:String>Item #5</system:String>
        </ItemsControl>
    </Grid>
```

```
</Window>
```



We specify that the ItemsControl should use a WrapPanel as its template by declaring one in the **ItemsPanelTemplate** property and just for fun, we throw in an ItemTemplate that causes the strings to be rendered as buttons. You can use any of the WPF panels, but some are more useful than others.

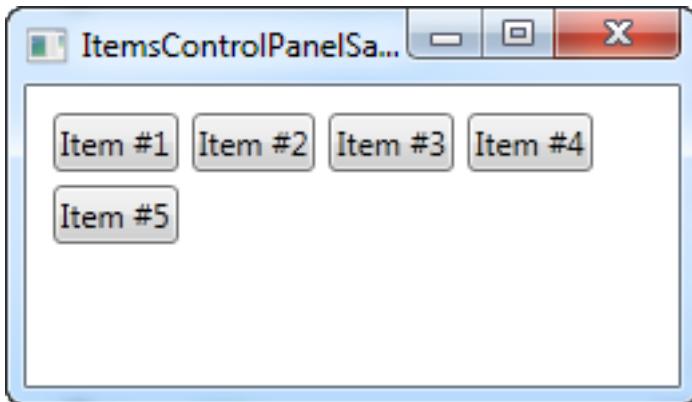
Another good example is the UniformGrid panel, where we can define a number of columns and then have our items neatly shown in equally-wide columns:

```
<Window x:Class
        ="WpfTutorialSamples.ItemsControl.ItemsControlPanelSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="ItemsControlPanelSample" Height="150" Width="250">
    <Grid Margin="10">
        <ItemsControl>
            <ItemsControl.ItemsPanel>
                <ItemsPanelTemplate>
                    <UniformGrid Columns="2" />
                </ItemsPanelTemplate>
            </ItemsControl.ItemsPanel>
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <Button Content="{Binding}" Margin="0,0,5,5" />
                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>
        <system:String>Item #1</system:String>
        <system:String>Item #2</system:String>
        <system:String>Item #3</system:String>
```

```

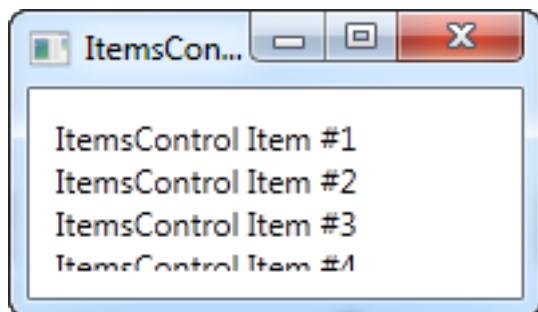
<system:String>Item #4</system:String>
<system:String>Item #5</system:String>
</ItemsControl>
</Grid>
</Window>

```



#### 1.16.1.4. ItemsControl with scrollbars

Once you start using the ItemsControl, you might run into a very common problem: By default, the ItemsControl doesn't have any scrollbars, which means that if the content doesn't fit, it's just clipped. This can be seen by taking our first example from this article and resizing the window:



WPF makes this very easy to solve though. There are a number of possible solutions, for instance you can alter the template used by the ItemsControl to include a ScrollViewer control, but the easiest solution is to simply throw a ScrollViewer around the ItemsControl. Here's an example:

```

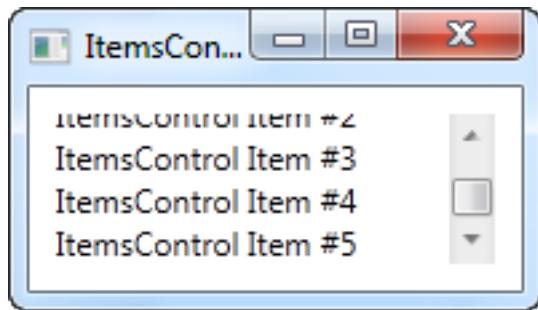
<Window x:Class="WpfTutorialSamples.ItemsControl.ItemsControlSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        Title="ItemsControlSample" Height="150" Width="200">
    <Grid Margin="10">
        <ScrollViewer VerticalScrollBarVisibility="Auto">

```

```


    <ItemsControl>
        <system:String>ItemsControl Item #1</system:String>
        <system:String>ItemsControl Item #2</system:String>
        <system:String>ItemsControl Item #3</system:String>
        <system:String>ItemsControl Item #4</system:String>
        <system:String>ItemsControl Item #5</system:String>
    </ItemsControl>
</ScrollViewer>
</Grid>
</Window>

```



I set the two visibility options to Auto, to make them only visible when needed. As you can see from the screenshot, you can now scroll through the list of items.

#### 1.16.1.5. Summary

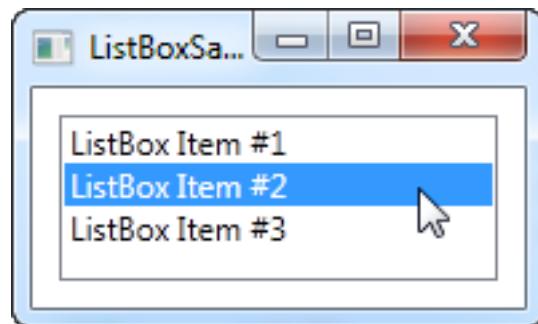
The `ItemsControl` is great when you want full control of how your data is displayed, and when you don't need any of your content to be selectable. If you want the user to be able to select items from the list, then you're better off with one of the other controls, e.g. the `ListBox` or the `ListView`. They will be described in upcoming chapters.

## 1.16.2. The ListBox control

In the last article, we had a look at the ItemsControl, which is probably the simplest list in WPF. The ListBox control is the next control in line, which adds a bit more functionality. One of the main differences is the fact that the ListBox control actually deals with selections, allowing the end-user to select one or several items from the list and automatically giving visual feedback for it.

Here's an example of a very simple ListBox control:

```
<Window x:Class="WpfTutorialSamples.ListBox_control.ListBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListBoxSample" Height="120" Width="200">
    <Grid Margin="10">
        <ListBox>
            <ListBoxItem>ListBox Item #1</ListBoxItem>
            <ListBoxItem>ListBox Item #2</ListBoxItem>
            <ListBoxItem>ListBox Item #3</ListBoxItem>
        </ListBox>
    </Grid>
</Window>
```



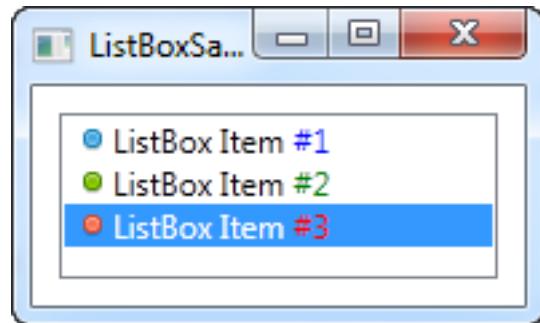
This is as simple as it gets: We declare a ListBox control, and inside of it, we declare three ListBoxItem's, each with its own text. However, since the ListBoxItem is actually a ContentControl, we can define custom content for it:

```
<Window x:Class="WpfTutorialSamples.ListBox_control.ListBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListBoxSample" Height="120" Width="200">
    <Grid Margin="10">
        <ListBox>
```

```

<ListBoxItem>
    <StackPanel Orientation="Horizontal">
        <Image Source
        =" /WpfTutorialSamples;component/Images/bullet_blue.png" />
        <TextBlock>ListBox Item #1</TextBlock>
    </StackPanel>
</ListBoxItem>
<ListBoxItem>
    <StackPanel Orientation="Horizontal">
        <Image Source
        =" /WpfTutorialSamples;component/Images/bullet_green.png" />
        <TextBlock>ListBox Item #2</TextBlock>
    </StackPanel>
</ListBoxItem>
<ListBoxItem>
    <StackPanel Orientation="Horizontal">
        <Image Source
        =" /WpfTutorialSamples;component/Images/bullet_red.png" />
        <TextBlock>ListBox Item #3</TextBlock>
    </StackPanel>
</ListBoxItem>
</ListBox>
</Grid>
</Window>

```



For each of the `ListBoxItem`'s we now add a `StackPanel`, in which we add an `Image` and a `TextBlock`. This gives us full control of the content as well as the text rendering, as you can see from the screenshot, where different colors have been used for each of the numbers.

From the screenshot you might also notice another difference when comparing the `ItemsControl` to the `ListBox`: By default, a border is shown around the control, making it look like an actual control instead of just output.

### 1.16.2.1. Data binding the ListBox

Manually defining items for the ListBox makes for a fine first example, but most of the times, your ListBox controls will be filled with items from a data source using data binding. By default, if you bind a list of items to the ListBox, their ToString() method will be used to represent each item. This is rarely what you want, but fortunately, we can easily declare a template that will be used to render each item.

I have re-used the TODO based example from the ItemsControl article, where we build a cool TODO list using a simple Code-behind class and, in this case, a ListBox control for the visual representation. Here's the example:

```
<Window x:Class
        ="WpfTutorialSamples.ListBox_control.ListBoxDataBindingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListBoxDataBindingSample" Height="150" Width="300">
    <Grid Margin="10">
        <ListBox Name="lbTodoList" HorizontalContentAlignment
        ="Stretch">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Grid Margin="0,2">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="*" />
                            <ColumnDefinition Width="100" />
                        </Grid.ColumnDefinitions>
                        <TextBlock Text="{Binding Title}" />
                        <ProgressBar Grid.Column="1" Minimum="0"
                        Maximum="100" Value="{Binding Completion}" />
                    </Grid>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>

using System;
using System.Windows;
using System.Collections.Generic;

namespace WpfTutorialSamples.ListBox_control
```

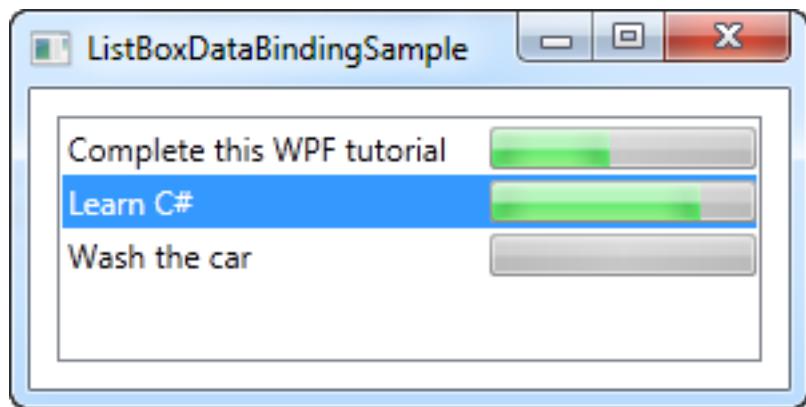
```

{
    public partial class ListBoxDataBindingSample : Window
    {
        public ListBoxDataBindingSample()
        {
            InitializeComponent();
            List<TodoItem> items = new List<TodoItem>();
            items.Add(new TodoItem() { Title = "Complete this WPF tutorial", Completion = 45 });
            items.Add(new TodoItem() { Title = "Learn C#", Completion = 80 });
            items.Add(new TodoItem() { Title = "Wash the car", Completion = 0 });

            lbTodoList.ItemsSource = items;
        }
    }

    public class TodoItem
    {
        public string Title { get; set; }
        public int Completion { get; set; }
    }
}

```

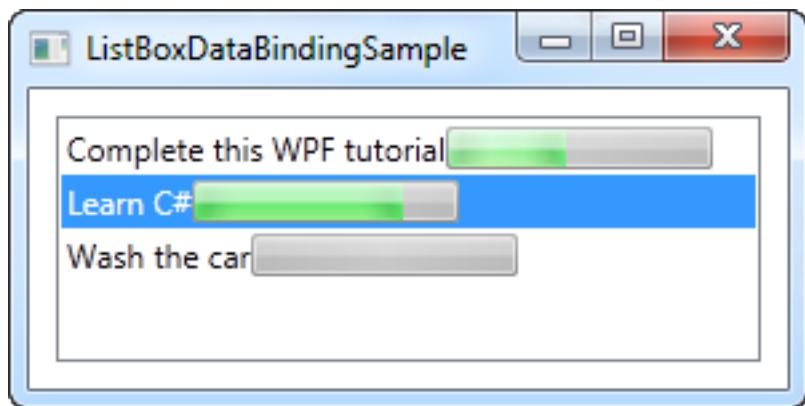


All the magic happens in the ItemTemplate that we have defined for the ListBox. In there, we specify that each ListBox item should consist of a Grid, divided into two columns, with a TextBlock showing the title in the first and a ProgressBar showing the completion status in the second column. To get the values out, we use some very simple data binding, which is all explained in the data binding part of this tutorial.

In the Code-behind file, we have declared a very simple TodoItem class to hold each of our TODO items. In

the constructor of the window, we initialize a list, add three TODO items to it and then assign it to the ItemsSource of the ListBox. The combination of the ItemsSource and the ItemTemplate we specified in the XAML part, this is all WPF need to render all of the items as a TODO list.

Please notice the **HorizontalContentAlignment** property that I set to **Stretch** on the ListBox. The default content alignment for a ListBox item is **Left**, which means that each item only takes up as much horizontal space as it needs. The result? Well, not quite what we want:



By using the Stretch alignment, each item is stretched to take up the full amount of available space, as you can see from the previous screenshot.

#### 1.16.2.2. Working with ListBox selection

As mentioned, a key difference between the ItemsControl and the ListBox is that the ListBox handles and displays user selection for you. Therefore, a lot of ListBox question revolves around somehow working with the selection. To help with some of these questions, I have created a bigger example, showing you some selection related tricks:

```
<Window x:Class="WpfTutorialSamples.ListBox_control.ListBoxSelectionSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListBoxSelectionSample" Height="250" Width="450">
    <DockPanel Margin="10">
        <StackPanel DockPanel.Dock="Right" Margin="10,0">
            <StackPanel.Resources>
                <Style TargetType="Button">
                    <Setter Property="Margin" Value="0,0,0,5" />
                </Style>
            </StackPanel.Resources>
            <TextBlock FontWeight="Bold" Margin="0,0,0,10">ListBox
selection</TextBlock>
```

```

                <Button Name="btnShowSelectedItem" Click
="btnShowSelectedItem_Click">Show selected</Button>
                <Button Name="btnSelectLast" Click="btnSelectLast_Click">
Select last</Button>
                <Button Name="btnSelectNext" Click="btnSelectNext_Click">
Select next</Button>
                <Button Name="btnSelectCSharp" Click
="btnSelectCSharp_Click">Select C#</Button>
                <Button Name="btnSelectAll" Click="btnSelectAll_Click">
Select all</Button>
            </StackPanel>
            <ListBox Name="lbTodoList" HorizontalContentAlignment
="Stretch" SelectionMode="Extended" SelectionChanged
="lbTodoList_SelectionChanged">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <Grid Margin="0,2">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="*" />
                                <ColumnDefinition Width="100" />
                            </Grid.ColumnDefinitions>
                            <TextBlock Text="{Binding Title}" />
                            <ProgressBar Grid.Column="1" Minimum="0"
Maximum="100" Value="{Binding Completion}" />
                        </Grid>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </DockPanel>
    </Window>

using System;
using System.Windows;
using System.Collections.Generic;

namespace WpfTutorialSamples.ListBox_control
{
    public partial class ListBoxSelectionSample : Window
    {
        public ListBoxSelectionSample()

```

```

    {
        InitializeComponent();
        List<TodoItem> items = new List<TodoItem>();
        items.Add(new TodoItem() { Title = "Complete this WPF tutorial", Completion = 45 });
        items.Add(new TodoItem() { Title = "Learn C#", Completion = 80 });
        items.Add(new TodoItem() { Title = "Wash the car", Completion = 0 });

        lbTodoList.ItemsSource = items;
    }

    private void lbTodoList_SelectionChanged(object sender,
System.Windows.Controls.SelectionChangedEventArgs e)
{
    if(lbTodoList.SelectedItem != null)
        this.Title = (lbTodoList.SelectedItem as
TodoItem).Title;
}

private void btnShowSelectedItem_Click(object sender,
RoutedEventArgs e)
{
    foreach(object o in lbTodoList.SelectedItems)
        MessageBox.Show((o as TodoItem).Title);
}

private void btnSelectLast_Click(object sender,
RoutedEventArgs e)
{
    lbTodoList.SelectedIndex = lbTodoList.Items.Count - 1;
}

private void btnSelectNext_Click(object sender,
RoutedEventArgs e)
{
    int nextIndex = 0;
    if((lbTodoList.SelectedIndex >= 0) &&
(lbTodoList.SelectedIndex < (lbTodoList.Items.Count - 1)))

```

```

        nextIndex = lbTodoList.SelectedIndex + 1;
        lbTodoList.SelectedIndex = nextIndex;
    }

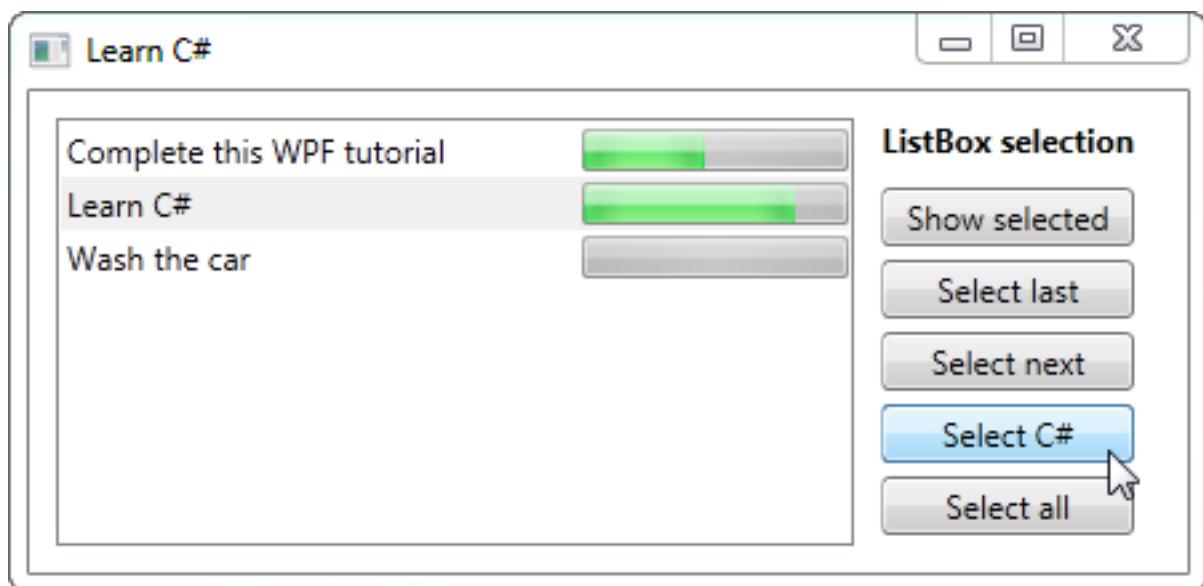
    private void btnSelectCSharp_Click(object sender,
RoutedEventArgs e)
{
    foreach(object o in lbTodoList.Items)
    {
        if((o is TodoItem) && ((o as
TodoItem).Title.Contains("C#")))
        {
            lbTodoList.SelectedItem = o;
            break;
        }
    }
}

private void btnSelectAll_Click(object sender, RoutedEventArgs
e)
{
    foreach(object o in lbTodoList.Items)
        lbTodoList.SelectedItems.Add(o);
}
}

public class TodoItem
{
    public string Title { get; set; }
    public int Completion { get; set; }
}
}

```

As you can see, I have defined a range of buttons to the right of the `ListBox`, to either get or manipulate the selection. I've also changed the **SelectionMode** to **Extended**, to allow for the selection of multiple items. This can be done either programmatically, as I do in the example, or by the end-user, by holding down **[Ctrl]** or **[Shift]** while clicking on the items.



For each of the buttons, I have defined a click handler in the Code-behind. Each action should be pretty self-explanatory and the C# code used is fairly simple, but if you're still in doubt, try running the example on your own machine and test out the various possibilities in the example.

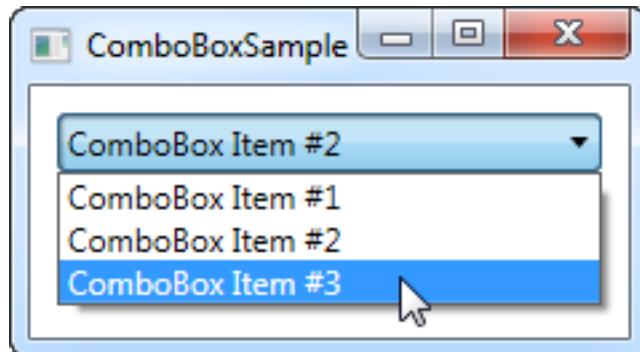
#### 1.16.2.3. Summary

The `ListBox` control is much like the `ItemsControl` and several of the same techniques can be used. The `ListBox` does offer a bit more functionality when compared to the `ItemsControl`, especially the selection handling. For even more functionality, like column headers, you should have a look at the `ListView` control, which is given a very thorough description later on in this tutorial with several articles explaining all the functionality.

### 1.16.3. The ComboBox control

The ComboBox control is in many ways like the ListBox control, but takes up a lot less space, because the list of items is hidden when not needed. The ComboBox control is used many places in Windows, but to make sure that everyone knows how it looks and works, we'll jump straight into a simple example:

```
<Window x:Class="WpfTutorialSamples.ComboBox_control.ComboBoxSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ComboBoxSample" Height="150" Width="200">
    <StackPanel Margin="10">
        <ComboBox>
            <ComboBoxItem>ComboBox Item #1</ComboBoxItem>
            <ComboBoxItem IsSelected="True">ComboBox Item #2</
ComboBoxItem>
            <ComboBoxItem>ComboBox Item #3</ComboBoxItem>
        </ComboBox>
    </StackPanel>
</Window>
```



In the screenshot, I have activated the control by clicking it, causing the list of items to be displayed. As you can see from the code, the ComboBox, in its simple form, is very easy to use. All I've done here is manually add some items, making one of them the default selected item by setting the `IsSelected` property on it.

#### 1.16.3.1. Custom content

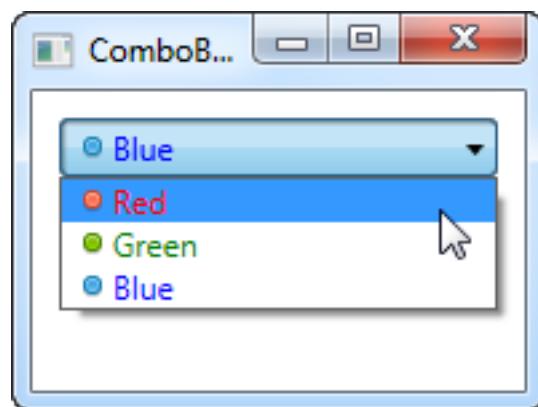
In the first example we only showed text in the items, which is pretty common for the ComboBox control, but since the ComboBoxItem is a ContentControl, we can actually use pretty much anything as content. Let's try making a slightly more sophisticated list of items:

```
<Window x:Class
        ="WpfTutorialSamples.ComboBox_control.ComboBoxCustomContentSample"
        xmlns
```

```

="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ComboBoxCustomContentSample" Height="150" Width="200">
<StackPanel Margin="10">
    <ComboBox>
        <ComboBoxItem>
            <StackPanel Orientation="Horizontal">
                <Image Source
="WpfTutorialSamples;component/Images/bullet_red.png" />
                <TextBlock Foreground="Red">Red</TextBlock>
            </StackPanel>
        </ComboBoxItem>
        <ComboBoxItem>
            <StackPanel Orientation="Horizontal">
                <Image Source
="WpfTutorialSamples;component/Images/bullet_green.png" />
                <TextBlock Foreground="Green">Green</TextBlock>
            </StackPanel>
        </ComboBoxItem>
        <ComboBoxItem>
            <StackPanel Orientation="Horizontal">
                <Image Source
="WpfTutorialSamples;component/Images/bullet_blue.png" />
                <TextBlock Foreground="Blue">Blue</TextBlock>
            </StackPanel>
        </ComboBoxItem>
    </ComboBox>
</StackPanel>
</Window>

```



For each of the ComboBoxItem's we now add a StackPanel, in which we add an Image and a TextBlock.

This gives us full control of the content as well as the text rendering, as you can see from the screenshot, where both text color and image indicates a color value.

### 1.16.3.2. Data binding the ComboBox

As you can see from the first examples, manually defining the items of a ComboBox control is easy using XAML, but you will likely soon run into a situation where you need the items to come from some kind of data source, like a database or just an in-memory list. Using WPF data binding and a custom template, we can easily render a list of colors, including a preview of the color:

```
<Window x:Class
        ="WpfTutorialSamples.ComboBox_control.ComboBoxDataBindingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ComboBoxDataBindingSample" Height="200" Width="200">
    <StackPanel Margin="10">
        <ComboBox Name="cmbColors">
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <Rectangle Fill="{Binding Name}" Width
="16" Height="16" Margin="0,2,5,2" />
                        <TextBlock Text="{Binding Name}" />
                    </StackPanel>
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>
    </StackPanel>
</Window>

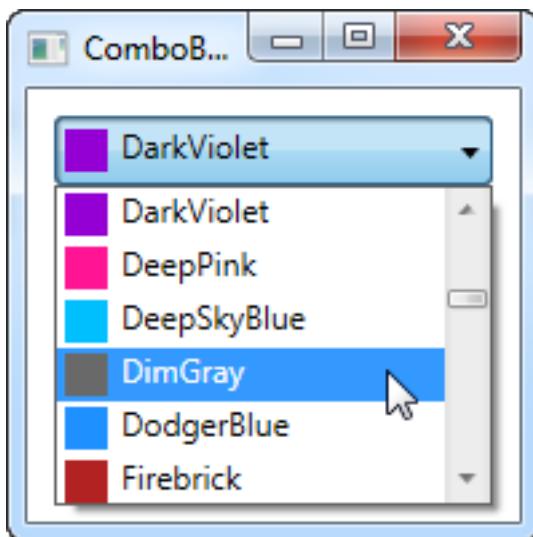
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Media;

namespace WpfTutorialSamples.ComboBox_control
{
    public partial class ComboBoxDataBindingSample : Window
    {
        public ComboBoxDataBindingSample()
        {
```

```

        InitializeComponent();
        cmbColors.ItemsSource = typeof(Colors).GetProperties();
    }
}

```



It's actually quite simple: In the Code-behind, I obtain a list of all the colors using a Reflection based approach with the `Colors` class. I assign it to the `ItemsSource` property of the `ComboBox`, which then renders each color using the template I have defined in the XAML part.

Each item, as defined by the `ItemTemplate`, consists of a `StackPanel` with a `Rectangle` and a `TextBlock`, each bound to the color value. This gives us a complete list of colors, with minimal effort - and it looks pretty good too, right?

#### 1.16.3.3. IsEditable

In the first examples, the user was only able to select from our list of items, but one of the cool things about the `ComboBox` is that it supports the possibility of letting the user both select from a list of items or enter their own value. This is extremely useful in situations where you want to help the user by giving them a pre-defined set of options, while still giving them the option to manually enter the desired value. This is all controlled by the `IsEditable` property, which changes the behavior and look of the `ComboBox` quite a bit:

```

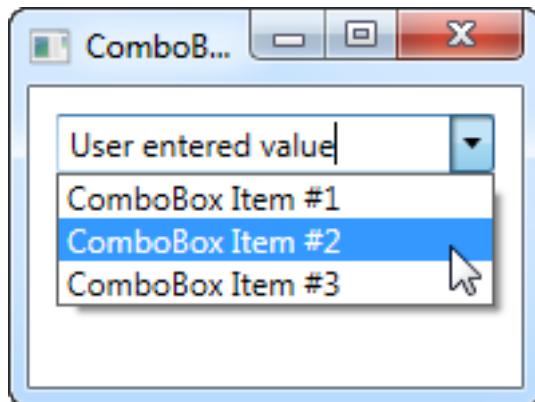
<Window x:Class
        ="WpfTutorialSamples.ComboBox_control.ComboBoxEditableSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ComboBoxEditableSample" Height="150" Width="200">
    <StackPanel Margin="10">
        <ComboBox IsEditable="True">

```

```

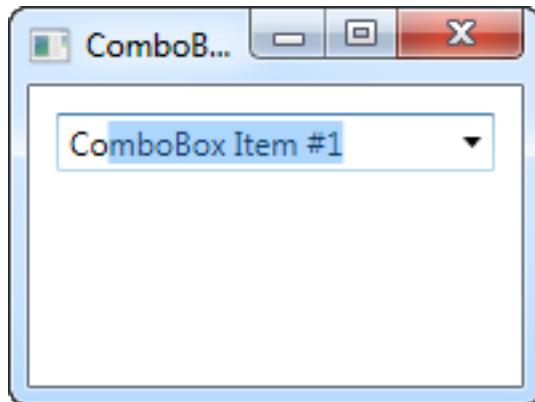
<ComboBoxItem>ComboBox Item #1</ComboBoxItem>
<ComboBoxItem>ComboBox Item #2</ComboBoxItem>
<ComboBoxItem>ComboBox Item #3</ComboBoxItem>
</ComboBox>
</StackPanel>
</Window>

```



As you can see, I can enter a completely different value or pick one from the list. If picked from the list, it simply overwrites the text of the ComboBox.

As a lovely little bonus, the ComboBox will automatically try to help the user select an existing value when the user starts typing, as you can see from the next screenshot, where I just started typing "Co":



By default, the matching is not case-sensitive but you can make it so by setting the **IsTextSearchCaseSensitive** to True. If you don't want this auto complete behavior at all, you can disable it by setting the **IsTextSearchEnabled** to False.

#### 1.16.3.4. Working with ComboBox selection

A key part of using the ComboBox control is to be able to read the user selection, and even control it with code. In the next example, I've re-used the data bound ComboBox example, but added some buttons for controlling the selection. I've also used the **SelectionChanged** event to capture when the selected item is changed, either by code or by the user, and act on it.

Here's the sample:

```
<Window x:Class
        ="WpfTutorialSamples.ComboBox_control.ComboBoxSelectionSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ComboBoxSelectionSample" Height="125" Width="250">
    <StackPanel Margin="10">
        <ComboBox Name="cmbColors" SelectionChanged
        ="cmbColors_SelectionChanged">
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <Rectangle Fill="{Binding Name}" Width
                        ="16" Height="16" Margin="0,2,5,2" />
                        <TextBlock Text="{Binding Name}" />
                    </StackPanel>
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>
        <WrapPanel Margin="15" HorizontalAlignment="Center">
            <Button Name="btnPrevious" Click="btnPrevious_Click"
Width="55">Previous</Button>
            <Button Name="btnNext" Click="btnNext_Click" Margin="5,0"
Width="55">Next</Button>
            <Button Name="btnBlue" Click="btnBlue_Click" Width="55">
Blue</Button>
        </WrapPanel>
    </StackPanel>
</Window>

using System;
using System.Collections.Generic;
using System.Reflection;
using System.Windows;
using System.Windows.Media;

namespace WpfTutorialSamples.ComboBox_control
{
    public partial class ComboBoxSelectionSample : Window
```

```

{
    public ComboBoxSelectionSample()
    {
        InitializeComponent();
        cmbColors.ItemsSource = typeof(Colors).GetProperties();
    }

    private void btnPrevious_Click(object sender, RoutedEventArgs e)
    {
        if(cmbColors.SelectedIndex > 0)
            cmbColors.SelectedIndex = cmbColors.SelectedIndex - 1;
    }

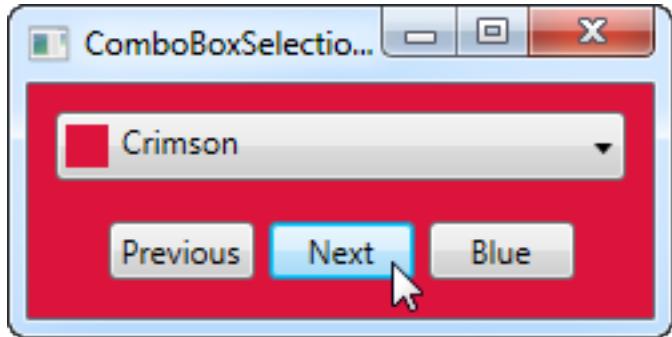
    private void btnNext_Click(object sender, RoutedEventArgs e)
    {
        if(cmbColors.SelectedIndex < cmbColors.Items.Count-1)
            cmbColors.SelectedIndex = cmbColors.SelectedIndex + 1;
    }

    private void btnBlue_Click(object sender, RoutedEventArgs e)
    {
        cmbColors.SelectedItem = typeof(Colors).GetProperty("Blue");
    }

    private void cmbColors_SelectionChanged(object sender,
System.Windows.Controls.SelectionChangedEventArgs e)
    {
        Color selectedColor = (Color)(cmbColors.SelectedItem as PropertyInfo).GetValue(null, null);
        this.Background = new SolidColorBrush(selectedColor);
    }
}

```

The interesting part of this example is the three event handlers for our three buttons, as well as the **SelectionChanged** event handler. In the first two, we select the previous or the next item by reading the



**SelectedIndex** property and then subtracting or adding one to it. Pretty simple and easy to work with.

In the third event handler, we use the **SelectedItem** to select a specific item based on the value. I do a bit of extra work here (using .NET reflection), because the ComboBox is bound to a list of properties, each being a color, instead of a simple list of colors, but basically it's all about giving the value contained by one of the items to the **SelectedItem** property.

In the fourth and last event handler, I respond to the selected item being changed. When that happens, I read the selected color (once again using Reflection, as described above) and then use the selected color to create a new background brush for the Window. The effect can be seen on the screenshot.

If you're working with an editable ComboBox (IsEditable property set to true), you can read the **Text** property to know the value the user has entered or selected.

# 1.17. The ListView control

---

## 1.17.1. Introduction to the ListView control

The ListView control is very commonly used in Windows applications, to represent lists of data. A great example of this is the file lists in Windows Explorer, where each file can be shown by its name and, if desired, with columns containing information about the size, last modification date and so on.

### 1.17.1.1. ListView in WPF vs. WinForms

If you have previously worked with WinForms, then you have a good idea about how practical the ListView is, but you should be aware that the ListView in WPF isn't used like the WinForms version. Once again the main difference is that while the WinForms ListView simply calls Windows API functions to render a common Windows ListView control, the WPF ListView is an independent control that doesn't rely on the Windows API.

The WPF ListView does use a ListViewItem class for its most basic items, but if you compare it to the WinForms version, you might start looking for properties like ImageIndex, Group and SubItems, but they're not there. The WPF ListView handles stuff like item images, groups and their sub items in a completely different way.

### 1.17.1.2. Summary

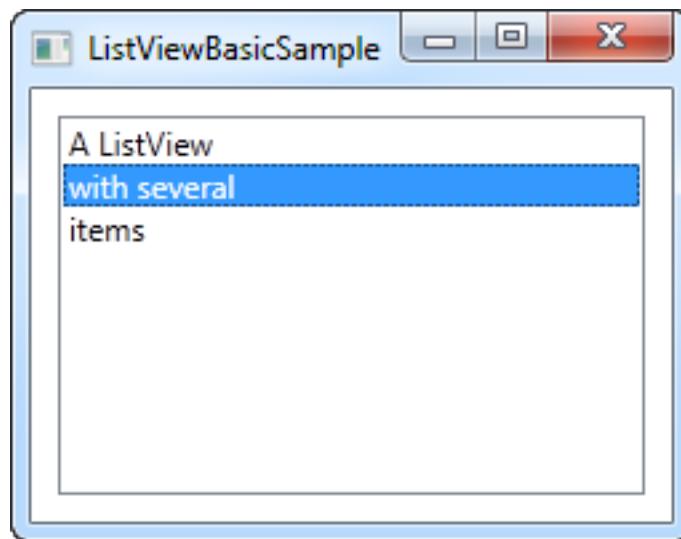
The ListView is a complex control, with lots of possibilities and especially in the WPF version, you get to customize it almost endlessly if you want to. For that reason, we have dedicated an entire category to all the ListView articles here on the site. Click on to the next article to get started.

## 1.17.2. A simple ListView example

The WPF ListView control is very bare minimum in its most simple form. In fact, it will look a whole lot like the WPF ListBox, until you start adding specialized views to it. That's not so strange, since a ListView inherits directly from the ListBox control. So, a default ListView is actually just a ListBox, with a different selection mode (more on that later).

Let's try creating a ListView in its most simple form:

```
<Window x:Class
        ="WpfTutorialSamples.ListView_control.ListViewBasicSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewBasicSample" Height="200" Width="200">
    <Grid>
        <ListView Margin="10">
            <ListViewItem>A ListView</ListViewItem>
            <ListViewItem IsSelected="True">with several</
ListviewItem>
            <ListViewItem>items</ListViewItem>
        </ListView>
    </Grid>
</Window>
```

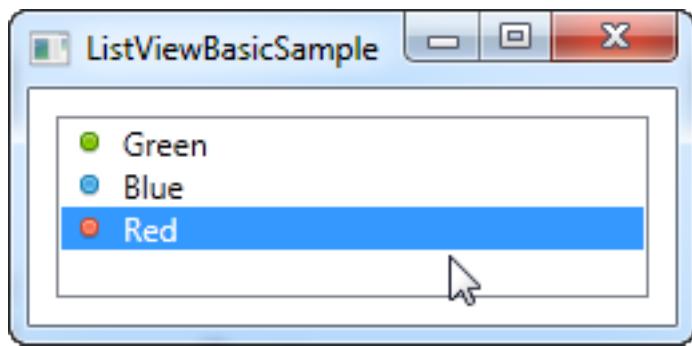


This is pretty much as simple as it gets, using manually specified ListViewItem to fill the list and with nothing but a text label representing each item - a bare minimum WPF ListView control.

### 1.17.2.1. ListViewItem with an image

Because of the look-less nature of WPF, specifying an image for a ListViewItem isn't just about assigning an image ID or key to a property. Instead, you take full control of it and specify the controls needed to render both image and text in the ListViewItem. Here's an example:

```
<Window x:Class
    ="WpfTutorialSamples.ListView_control.ListViewBasicSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewBasicSample" Height="200" Width="200">
    <Grid>
        <ListView Margin="10">
            <ListViewItem>
                <StackPanel Orientation="Horizontal">
                    <Image Source
                =" /WpfTutorialSamples;component/Images/bullet_green.png" Margin
                ="0,0,5,0" />
                    <TextBlock>Green</TextBlock>
                </StackPanel>
            </ListViewItem>
            <ListViewItem>
                <StackPanel Orientation="Horizontal">
                    <Image Source
                =" /WpfTutorialSamples;component/Images/bullet_blue.png" Margin="0,0,5,0"
                />
                    <TextBlock>Blue</TextBlock>
                </StackPanel>
            </ListViewItem>
            <ListViewItem IsSelected="True">
                <StackPanel Orientation="Horizontal">
                    <Image Source
                =" /WpfTutorialSamples;component/Images/bullet_red.png" Margin="0,0,5,0"
                />
                    <TextBlock>Red</TextBlock>
                </StackPanel>
            </ListViewItem>
        </ListView>
    </Grid>
</Window>
```



What we do here is very simple. Because the ListViewItem derives from the ContentControl class, we can specify a WPF control as its content. In this case, we use a StackPanel, which has an Image and a TextBlock as its child controls.

### 1.17.2.2. Summary

As you can see, building a ListView manually in XAML is very simple, but in most cases, your ListView data will come from some sort of data source, which should be rendered in the ListView at runtime. We will look into doing just that in the next chapter.

### 1.17.3. ListView, data binding and ItemTemplate

In the previous article, we manually populated a ListView control through XAML code, but in WPF, it's all about data binding. The concept of data binding is explained in detail in another part of this tutorial, but generally speaking it's about separating data from layout. So, let's try binding some data to a ListView:

```
<Window x:Class
        ="WpfTutorialSamples.ListView_control.ListViewDataBindingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewDataBindingSample" Height="300" Width="300">
    <Grid>
        <ListView Margin="10" Name="lvDataBinding"></ListView>
    </Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;

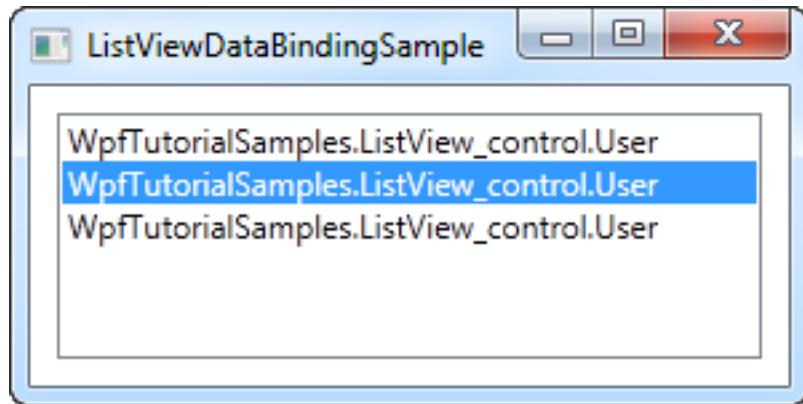
namespace WpfTutorialSamples.ListView_control
{
    public partial class ListViewDataBindingSample : Window
    {
        public ListViewDataBindingSample()
        {
            InitializeComponent();
            List<User> items = new List<User>();
            items.Add(new User() { Name = "John Doe", Age = 42 });
            items.Add(new User() { Name = "Jane Doe", Age = 39 });
            items.Add(new User() { Name = "Sammy Doe", Age = 13 });
            lvDataBinding.ItemsSource = items;
        }
    }

    public class User
    {
        public string Name { get; set; }

        public int Age { get; set; }
    }
}
```

```
    }  
}
```

We populate a list of our own User objects, each user having a name and an age. The data binding process happens automatically as soon as we assign the list to the ItemsSource property of the ListView, but the result is a bit discouraging:



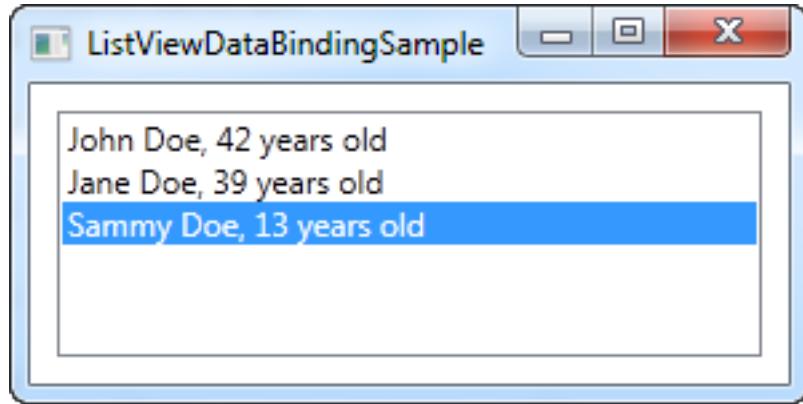
Each user is represented by their type name in the ListView. This is to be expected, because .NET doesn't have a clue about how you want your data to be displayed, so it just calls the `ToString()` method on each object and uses that to represent the item.

We can use that to our advantage and override the `ToString()` method, to get a more meaningful output. Try replacing the User class with this version:

```
public class User  
{  
    public string Name { get; set; }  
  
    public int Age { get; set; }  
  
    public override string ToString()  
    {  
        return this.Name + ", " + this.Age + " years old";  
    }  
}
```

This is a much more user friendly display and will do just fine in some cases, but relying on a simple string is not that flexible. Perhaps you want a part of the text to be bold or another color? Perhaps you want an image? Fortunately, WPF makes all of this very simple using templates.

#### 1.17.3.1. ListView with an ItemTemplate



WPF is all about templating, so specifying a data template for the ListView is very easy. In this example, we'll do a bunch of custom formatting in each item, just to show you how flexible this makes the WPF ListView.

```
<Window x:Class="WpfTutorialSamples.ListView_control.ListViewItemTemplateSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewItemTemplateSample" Height="150" Width="350">
    <Grid>
        <ListView Margin="10" Name="lvDataBinding">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <WrapPanel>
                        <TextBlock Text="Name: " />
                        <TextBlock Text="{Binding Name}" FontWeight="Bold" />
                        <TextBlock Text=" , " />
                        <TextBlock Text="Age: " />
                        <TextBlock Text="{Binding Age}" FontWeight="Bold" />
                        <TextBlock Text=" (" />
                        <TextBlock Text="{Binding Mail}" />
                    </WrapPanel>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </Grid>
</Window>
```

```

using System;
using System.Collections.Generic;
using System.Windows;

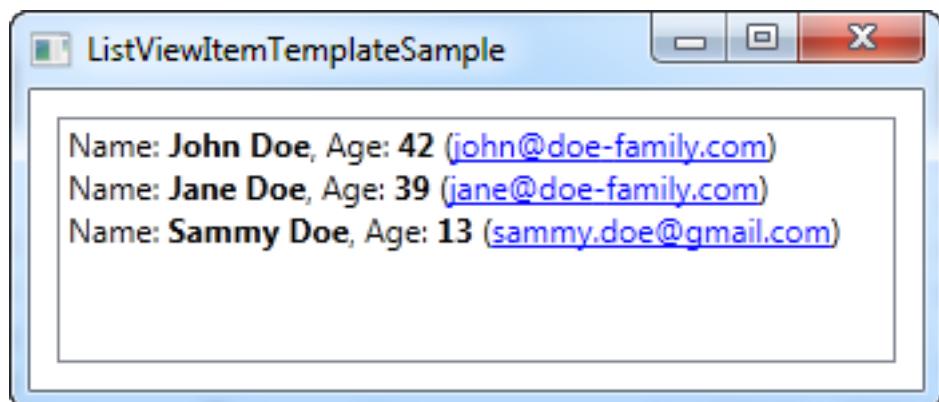
namespace WpfTutorialSamples.ListView_control
{
    public partial class ListViewItemTemplateSample : Window
    {
        public ListViewItemTemplateSample()
        {
            InitializeComponent();
            List<User> items = new List<User>();
            items.Add(new User() { Name = "John Doe", Age = 42, Mail = "john@doe-family.com" });
            items.Add(new User() { Name = "Jane Doe", Age = 39, Mail = "jane@doe-family.com" });
            items.Add(new User() { Name = "Sammy Doe", Age = 13, Mail = "sammy.doe@gmail.com" });
            lvDataBinding.ItemsSource = items;
        }
    }

    public class User
    {
        public string Name { get; set; }

        public int Age { get; set; }

        public string Mail { get; set; }
    }
}

```



We use a bunch of `TextBlock` controls to build each item, where we put part of the text in bold. For the e-mail address, which we added to this example, we underline it, give it a blue color and change the mouse cursor, to make it behave like a hyperlink.

### 1.17.3.2. Summary

Using an `ItemTemplate` and data binding, we produced a pretty cool `ListView` control. However, it still looks a lot like a `ListBox`. A very common usage scenario for a `ListView` is to have columns, sometimes (e.g. in WinForms) referred to as a details view. WPF comes with a built-in view class to handle this, which we will talk about in the next chapter.

## 1.17.4. ListView with a GridView

In the previous ListView articles, we have used the most basic version of the WPF ListView, which is the one without a custom View specified. This results in a ListView that acts very much like the WPF ListBox, with some subtle differences. The real power lies in the views though and WPF comes with one specialized view built-in: The GridView.

By using the GridView, you can get several columns of data in your ListView, much like you see it in Windows Explorer. Just to make sure that everyone can visualize it, we'll start off with a basic example:

```
<Window x:Class
    ="WpfTutorialSamples.ListView_control.ListViewGridViewSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewGridViewSample" Height="200" Width="400">
<Grid>
    <ListView Margin="10" Name="lvUsers">
        <ListView.View>
            <GridView>
                <GridViewColumn Header="Name" Width="120"
DisplayMemberBinding="{Binding Name}" />
                <GridViewColumn Header="Age" Width="50"
DisplayMemberBinding="{Binding Age}" />
                <GridViewColumn Header="Mail" Width="150"
DisplayMemberBinding="{Binding Mail}" />
            </GridView>
        </ListView.View>
    </ListView>
</Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples.ListView_control
{
    public partial class ListViewGridViewSample : Window
    {
        public ListViewGridViewSample()
        {
```

```

        InitializeComponent();
        List<User> items = new List<User>();
        items.Add(new User() { Name = "John Doe", Age = 42, Mail
= "john@doe-family.com" });
        items.Add(new User() { Name = "Jane Doe", Age = 39, Mail
= "jane@doe-family.com" });
        items.Add(new User() { Name = "Sammy Doe", Age = 7, Mail
= "sammy.doe@gmail.com" });
        lvUsers.ItemsSource = items;
    }
}

public class User
{
    public string Name { get; set; }

    public int Age { get; set; }

    public string Mail { get; set; }
}
}

```

Name	Age	Mail
John Doe	42	john@doe-family.com
Jane Doe	39	jane@doe-family.com
Sammy Doe	7	sammy.doe@gmail.com

So, we use the same User class as previously, for test data, which we then bind to the ListView. This is all the same as we saw in previous chapters, but as you can see from the screenshot, the layout is very different. This is the power of data binding - the same data, but presented in a completely different way, just by changing the markup.

In the markup (XAML), we define a View for the ListView, using the `ListView.View` property. We set it to a `GridView`, which is currently the only included view type in WPF (you can easily create your own though!).

The GridView is what gives us the column-based view that you see on the screenshot.

Inside of the GridView, we define three columns, one for each of the pieces of data that we wish to show. The **Header** property is used to specify the text that we would like to show for the column and then we use the **DisplayMemberBinding** property to bind the value to a property from our User class.

#### 1.17.4.1. Templated cell content

Using the **DisplayMemberBinding** property is pretty much limited to outputting simple strings, with no custom formatting at all, but the WPF ListView is much more flexible than that. By specifying a **CellTemplate**, we take full control of how the content is rendered within the specific column cell.

The GridViewColumn will use the DisplayMemberBinding as its first priority, if it's present. The second choice will be the CellTemplate property, which we'll use for this example:

```
<Window x:Class
        ="WpfTutorialSamples.ListView_control.ListViewGridViewCellTemplateSample"
        ">

    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ListViewGridViewCellTemplateSample" Height="200" Width
    ="400">

    <Grid>
        <ListView Margin="10" Name="lvUsers">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="Name" Width="120"
DisplayMemberBinding="{Binding Name}" />
                    <GridViewColumn Header="Age" Width="50"
DisplayMemberBinding="{Binding Age}" />
                    <GridViewColumn Header="Mail" Width="150">
                        <GridViewColumn.CellTemplate>
                            <DataTemplate>
                                <TextBlock Text="{Binding Mail}"
TextDecorations="Underline" Foreground="Blue" Cursor="Hand" />
                            </DataTemplate>
                        </GridViewColumn.CellTemplate>
                    </GridViewColumn>
                </GridView>
            </ListView.View>
        </ListView>
    </Grid>

```

```
</Grid>  
</Window>
```

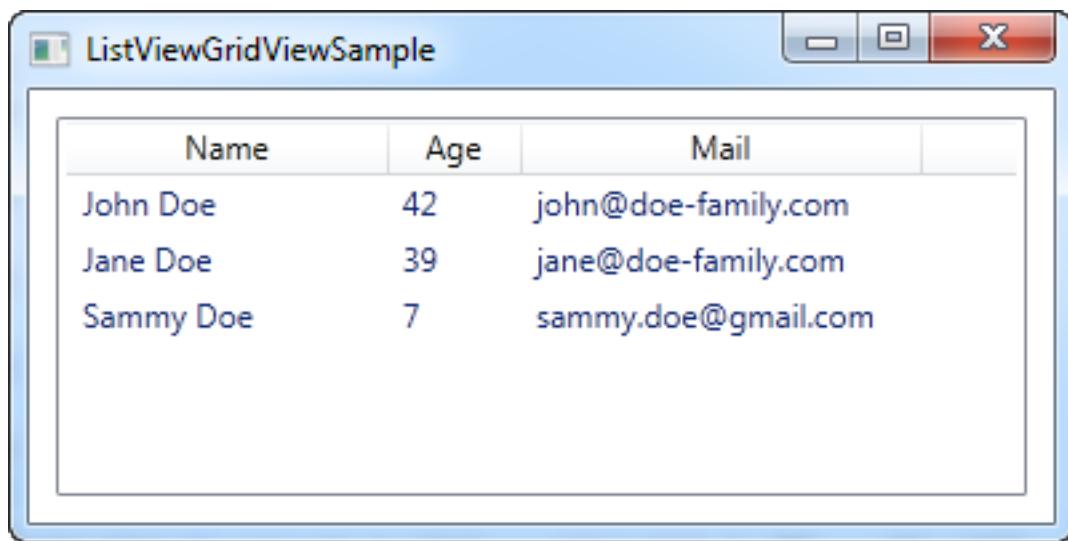
Name	Age	Mail
John Doe	42	<a href="mailto:john@doe-family.com">john@doe-family.com</a>
Jane Doe	39	<a href="mailto:jane@doe-family.com">jane@doe-family.com</a>
Sammy Doe	7	<a href="mailto:sammy.doe@gmail.com">sammy.doe@gmail.com</a>

*Please notice: The Code-behind code for this example is the same as the one used for the first example in this article.*

We specify a custom **CellTemplate** for the last column, where we would like to do some special formatting for the e-mail addresses. For the other columns, where we just want basic text output, we stick with the **DisplayMemberBinding**, simply because it requires way less markup.

## 1.17.5. How-to: ListView with left aligned column names

In a normal ListView, the column names are left aligned, but for some reason, Microsoft decided to center the names by default in the WPF ListView. In many cases this will make your application look out-of-style compared to other Windows applications. This is how the ListView will look in WPF by **default**:



Let's try changing that to left aligned column names. Unfortunately, there are no direct properties on the GridViewColumn to control this, but fortunately that doesn't mean that it can't be changed.

Using a Style, targeted at the GridViewColumnHeader, which is the element used to show the header of a GridViewColumn, we can change the HorizontalAlignment property. In this case it defaults to Center, but we can change it to Left, to accomplish what we want:

```
<Window x:Class="WpfTutorialSamples.ListView_control.ListViewGridViewSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewGridViewSample" Height="200" Width="400">
    <Grid>
        <ListView Margin="10" Name="lvUsers">
            <ListView.Resources>
                <Style TargetType="{x:Type GridViewColumnHeader}">
                    <Setter Property="HorizontalContentAlignment" Value="Left" />
                </Style>
            </ListView.Resources>
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="Name" Width="120" />
```

```

        DisplayMemberBinding="{Binding Name}" />
            <GridViewColumn Header="Age" Width="50" />
        DisplayMemberBinding="{Binding Age}" />
            <GridViewColumn Header="Mail" Width="150" />
        DisplayMemberBinding="{Binding Mail}" />
    
```

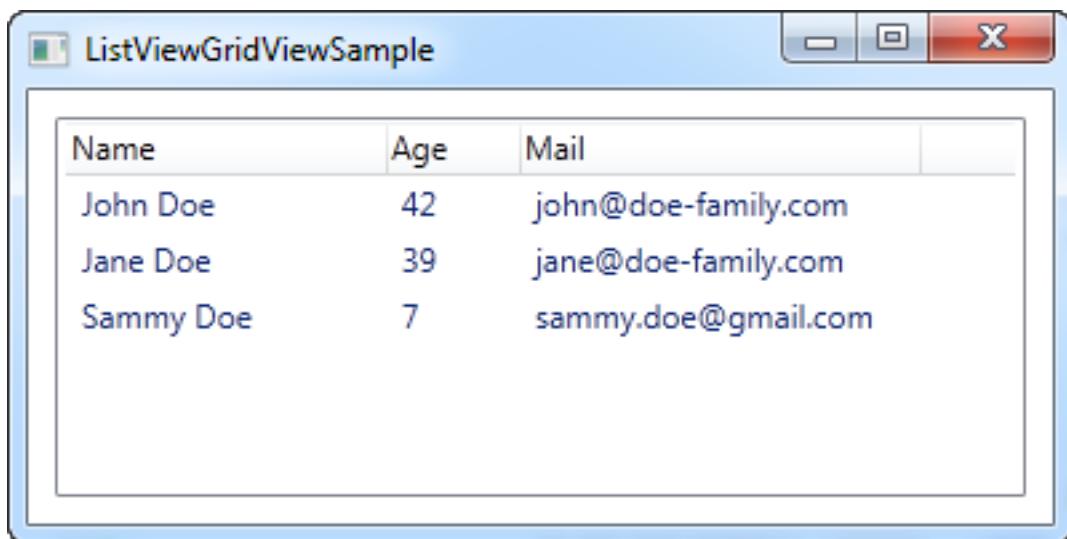
</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>



The part that does all the work for us, is the Style defined in the Resources of the ListView:

```

<Style TargetType="{x:Type GridViewColumnHeader}">
    <Setter Property="HorizontalContentAlignment"
Value="Left" />
</Style>

```

#### 1.17.5.1. Local or global style

By defining the Style within the control itself, it only applies to this particular ListView. In many cases you might like to make it apply to all the ListViews within the same Window/Page or perhaps even globally across the application. You can do this by either copying the style to the Window resources or the Application resources. Here's the same example, where we have applied the style to the entire Window instead of just the particular ListView:

```

<Window x:Class
="WpfTutorialSamples.ListView_control.ListViewGridViewSample"
    xmlns

```

```

="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ListViewGridViewSample" Height="200" Width="400">
<Window.Resources>
    <Style TargetType="{x:Type GridViewColumnHeader}">
        <Setter Property="HorizontalContentAlignment" Value
="Left" />
    </Style>
</Window.Resources>
<Grid>
    <ListView Margin="10" Name="lvUsers">
        <ListView.View>
            <GridView>
                <GridViewColumn Header="Name" Width="120"
DisplayMemberBinding="{Binding Name}" />
                <GridViewColumn Header="Age" Width="50"
DisplayMemberBinding="{Binding Age}" />
                <GridViewColumn Header="Mail" Width="150"
DisplayMemberBinding="{Binding Mail}" />
            </GridView>
        </ListView.View>
    </ListView>
</Grid>
</Window>

```

In case you want another alignment, e.g. right alignment, you just change the value of the style like this:

```
<Setter Property="HorizontalContentAlignment" Value="Right" />
```

## 1.17.6. ListView grouping

As we already talked about earlier, the WPF ListView is very flexible. Grouping is yet another thing that it supports out of the box, and it's both easy to use and extremely customizable. Let's jump straight into the first example, then I'll explain it and afterwards we can use the standard WPF tricks to customize the appearance even further.

For this article, I've borrowed the sample code from a previous article and then expanded on it to support grouping. It looks like this:

```
<Window x:Class
    ="WpfTutorialSamples.ListView_control.ListViewGroupSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewGroupSample" Height="300" Width="300">
<Grid Margin="10">
    <ListView Name="lvUsers">
        <ListView.View>
            <GridView>
                <GridViewColumn Header="Name" Width="120"
DisplayMemberBinding="{Binding Name}" />
                <GridViewColumn Header="Age" Width="50"
DisplayMemberBinding="{Binding Age}" />
            </GridView>
        </ListView.View>

        <ListView.GroupStyle>
            <GroupStyle>
                <GroupStyle.HeaderTemplate>
                    <DataTemplate>
                        <TextBlock FontWeight="Bold" FontSize
="14" Text="{Binding Name}" />
                    </DataTemplate>
                </GroupStyle.HeaderTemplate>
            </GroupStyle>
        </ListView.GroupStyle>
    </ListView>
</Grid>
</Window>

using System;
```

```

using System.Collections.Generic;
using System.Windows;
using System.Windows.Data;

namespace WpfTutorialSamples.ListView_control
{
    public partial class ListViewGroupSample : Window
    {
        public ListViewGroupSample()
        {
            InitializeComponent();
            List<User> items = new List<User>();
            items.Add(new User() { Name = "John Doe", Age = 42, Sex =
SexType.Male });
            items.Add(new User() { Name = "Jane Doe", Age = 39, Sex =
SexType.Female });
            items.Add(new User() { Name = "Sammy Doe", Age = 13, Sex =
= SexType.Male });
            lvUsers.ItemsSource = items;

            CollectionView view =
(CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource)
;
            PropertyGroupDescription groupDescription = new
PropertyGroupDescription("Sex");
            view.GroupDescriptions.Add(groupDescription);
        }
    }

    public enum SexType { Male, Female };

    public class User
    {
        public string Name { get; set; }

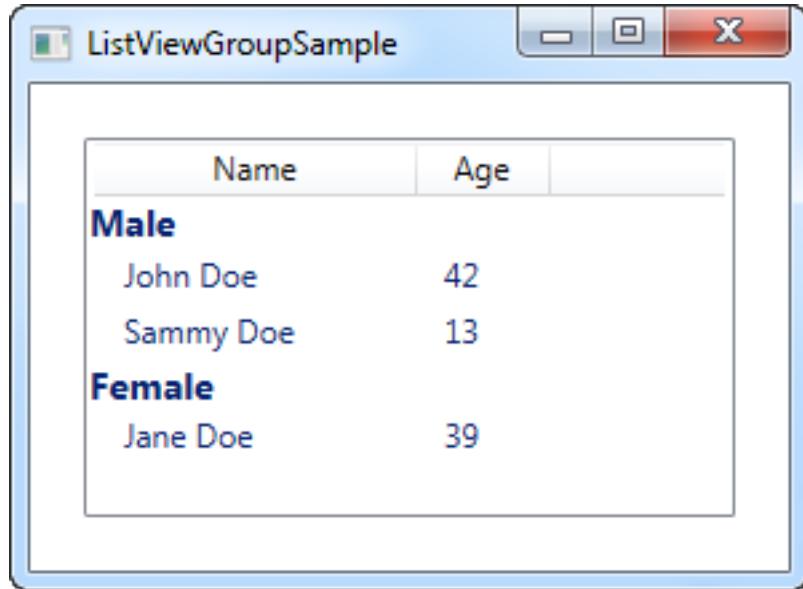
        public int Age { get; set; }

        public string Mail { get; set; }

        public SexType Sex { get; set; }
    }
}

```

```
}
```



In XAML, I have added a GroupStyle to the ListView, in which I define a template for the header of each group. It consists of a TextBlock control, where I've used a slightly larger and bold text to show that it's a group - as we'll see later on, this can of course be customized a lot more. The TextBlock Text property is bound to a Name property, **but please be aware that this is not the Name property on the data object (in this case the User class)**. Instead, it is the name of the group, as assigned by WPF, based on the property we use to divide the objects into groups.

In Code-behind, we do the same as we did before: We create a list and add some User objects to it and then we bind the list to the ListView - nothing new there, except for the new Sex property that I've added, which tells whether the user is male or female.

After assigning an ItemsSource, we use this to get a CollectionView that the ListView creates for us. This specialized View instance contains a lot of possibilities, including the ability to group the items. We use this by adding a so-called PropertyGroupDescription to the GroupDescriptions of the view. This basically tells WPF to group by a specific property on the data objects, in this case the Sex property.

#### 1.17.6.1. Customizing the group header

The above example was great for showing the basics of ListView grouping, but the look was a tad boring, so let's exploit the fact that WPF lets us define our own templates and spice things up. A common request is to be able to collapse and expand the group, and while WPF doesn't provide this behavior by default, it's somewhat easy to implement yourself. We'll do it by completely re-templating the group container.

It might look a bit cumbersome, but the principles used are somewhat simple and you will see them in other situations when you customize the WPF controls. Here's the code:

```
<Window x:Class
```

```

="WpfTutorialSamples.ListView_control.ListViewCollapseExpandGroupSample"
    xmlns
="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ListViewCollapseExpandGroupSample" Height="300" Width
="300">
    <Grid Margin="10">
        <ListView Name="lvUsers">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="Name" Width="120"
DisplayMemberBinding="{Binding Name}" />
                    <GridViewColumn Header="Age" Width="50"
DisplayMemberBinding="{Binding Age}" />
                </GridView>
            </ListView.View>

            <ListView.GroupStyle>
                <GroupStyle>
                    <GroupStyle.ContainerStyle>
                        <Style TargetType="x:Type GroupItem">
                            <Setter Property="Template">
                                <Setter.Value>
                                    <ControlTemplate>
                                        <Expander IsExpanded
="True">
                                            <Expander.Header>
                                                <StackPanel
Orientation="Horizontal">
                                                    <
TextBlock Text="{Binding Name}" FontWeight="Bold" Foreground="Gray"
FontSize="22" VerticalAlignment="Bottom" />
                                                    <
TextBlock Text="{Binding ItemCount}" FontSize="22" Foreground="Green"
FontWeight="Bold" FontStyle="Italic" Margin="10,0,0,0" VerticalAlignment
="Bottom" />
                                                    <
TextBlock Text=" item(s)" FontSize="22" Foreground="Silver" FontStyle
="Italic" VerticalAlignment="Bottom" />
                                            </StackPanel>
                                        </Expander>
                                    </ControlTemplate>
                                </Setter.Value>
                            </Setter>
                        </Style>
                    </GroupStyle.ContainerStyle>
                </GroupStyle>
            </ListView.GroupStyle>
        </ListView>
    </Grid>

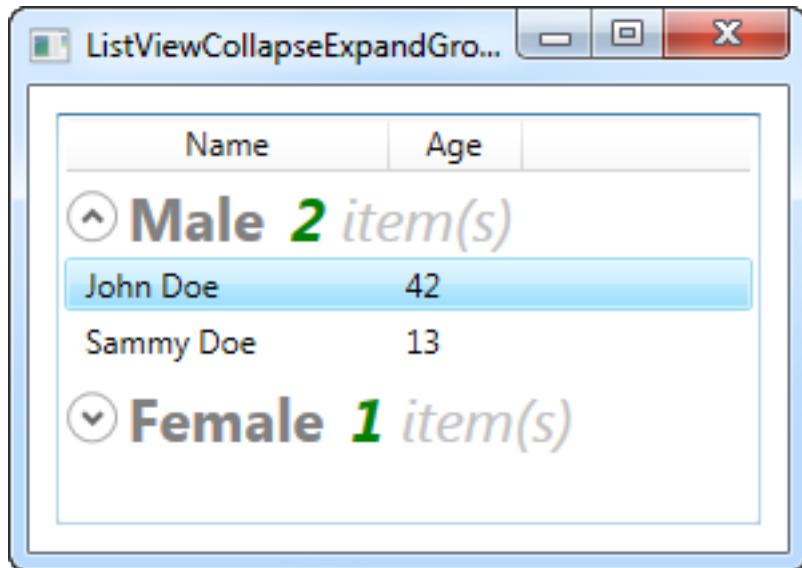
```

```

>                                         </Expander.Header>
>                                         <ItemsPresenter
/>                                         </Expander>
                                         </ControlTemplate>
                                         </Setter.Value>
                                         </Setter>
                                         </Style>
                                         </GroupStyle.ContainerStyle>
                                         </GroupStyle>
                                         </ListView.GroupStyle>
                                         </ListView>
                                         </Grid>
</Window>

```

The Code-behind is exactly the same as used in the first example - feel free to scroll up and grab it.



Now our groups look a bit more exciting, and they even include an expander button, that will toggle the visibility of the group items when you click it (that's why the single female user is not visible on the screenshot - I collapsed that particular group). By using the ItemCount property that the group exposes, we can even show how many items each group currently consists of.

As you can see, it requires a bit more markup than we're used to, but this example also goes a bit beyond what we usually do, so that seems fair. When you read through the code, you will quickly realize that many of the lines are just common elements like style and template.

### 1.17.6.2. Summary

Adding grouping to the WPF ListView is very simple - all you need is a GroupStyle with a HeaderTemplate, to tell the ListView how to render a group, and a few lines of Code-behind code to tell WPF which property to group by. As you can see from the last example, the group is even very customizable, allowing you to create some really cool views, without too much work.

## 1.17.7. ListView sorting

In the last chapter we saw how we could group items in the WPF ListView by accessing the View instance of the ListView and then adding a group description. Applying sorting to a ListView is just as easy, and most of the process is exactly the same. Let's try a simple example where we sort the user objects by their age:

```
<Window x:Class
        ="WpfTutorialSamples.ListView_control.ListViewSortingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewSortingSample" Height="200" Width="300">
    <Grid Margin="10">
        <ListView Name="lvUsers">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="Name" Width="120"
DisplayMemberBinding="{Binding Name}" />
                    <GridViewColumn Header="Age" Width="50"
DisplayMemberBinding="{Binding Age}" />
                </GridView>
            </ListView.View>
        </ListView>
    </Grid>
</Window>

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows;
using System.Windows.Data;

namespace WpfTutorialSamples.ListView_control
{
    public partial class ListViewSortingSample : Window
    {
        public ListViewSortingSample()
        {
            InitializeComponent();
            List<User> items = new List<User>();
            items.Add(new User() { Name = "John Doe", Age = 42 });
        }
    }
}
```

```

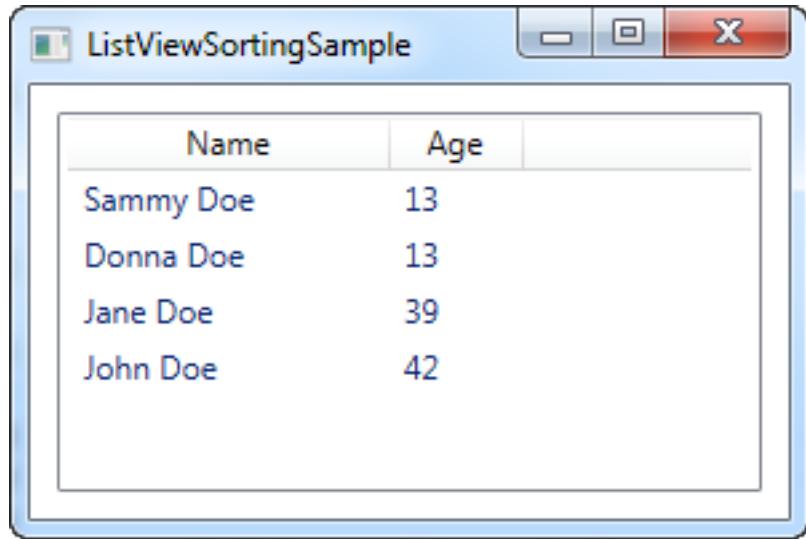
        items.Add(new User() { Name = "Jane Doe", Age = 39 });
        items.Add(new User() { Name = "Sammy Doe", Age = 13 });
        items.Add(new User() { Name = "Donna Doe", Age = 13 });
        lvUsers.ItemsSource = items;

        CollectionView view =
(CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource)
;
        view.SortDescriptions.Add(new SortDescription("Age",
ListSortDirection.Ascending));
    }
}

public class User
{
    public string Name { get; set; }

    public int Age { get; set; }
}
}

```



The XAML looks just like a previous example, where we simply have a couple of columns for displaying information about the user - nothing new here.

In the Code-behind, we once again create a list of User objects, which we then assign as the ItemsSource of the ListView. Once we've done that, we use the ItemsSource property to get the CollectionView instance that the ListView automatically creates for us and which we can use to manipulate how the ListView shows our objects.

With the view object in our hand, we add a new SortDescription to it, specifying that we want our list sorted by the Age property, in ascending order. As you can see from the screenshot, this works perfectly well - the list is sorted by age, instead of being in the same order as the items were added.

#### 1.17.7.1. Multiple sort criteria

As shown in the first example, sorting is very easy, but on the screenshot you'll see that Sammy comes before Donna. They have the same age, so in this case, WPF will just use the order in which they were added. Fortunately, WPF lets us specify as many sort criteria as we want. In the example above, try changing the view-related code into something like this:

```
CollectionView view =  
(CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource)  
;  
view.SortDescriptions.Add(new SortDescription("Age",  
ListSortDirection.Ascending));  
view.SortDescriptions.Add(new SortDescription("Name",  
ListSortDirection.Ascending));
```

Name	Age
Donna Doe	13
Sammy Doe	13
Jane Doe	39
John Doe	42

Now the view will be sorted using age first, and when two identical values are found, the name will be used as a secondary sorting parameter.

#### 1.17.7.2. Summary

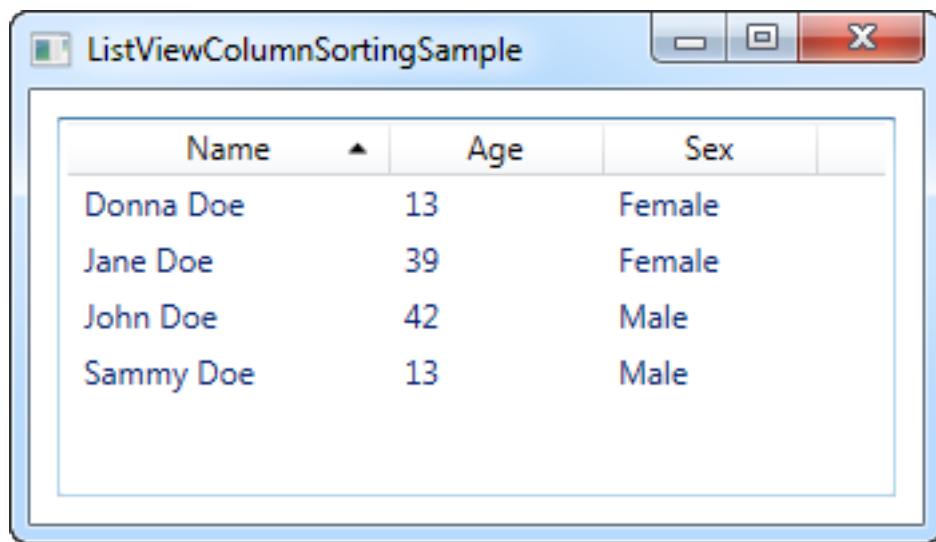
It's very easy to sort the contents of a ListView, as seen in the above examples, but so far, all the sorting is decided by the programmer and not the end-user. In the next article I'll give you a how-to article showing you how to let the user decide the sorting by clicking on the columns, as seen in Windows.

## 1.17.8. How-to: ListView with column sorting

In the last chapter we saw how we could easily sort a ListView from Code-behind, and while this will suffice for some cases, it doesn't allow the end-user to decide on the sorting. Besides that, there was no indication on which column the ListView was sorted by. In Windows, and in many user interfaces in general, it's common to illustrate sort directions in a list by drawing a triangle next to the column name currently used to sort by.

In this how-to article, I'll give you a practical solution that gives us all of the above, but please bear in mind that some of the code here goes a bit beyond what we have learned so far - that's why it has the "how-to" label.

This article builds upon the previous one, but I'll still explain each part as we go along. Here's our goal - a ListView with column sorting, including visual indication of sort field and direction. The user simply clicks a column to sort by and if the same column is clicked again, the sort direction is reversed. Here's how it looks:



### 1.17.8.1. The XAML

The first thing we need is some XAML to define our user interface. It currently looks like this:

```
<Window x:Class="WpfTutorialSamples.ListView_control.ListViewColumnSortingSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ListViewColumnSortingSample" Height="200" Width="350">
    <Grid Margin="10">
        <ListView Name="lvUsers">
            <ListView.View>
                <GridView>
```

```

        <GridViewColumn Width="120"
DisplayMemberBinding="{Binding Name}">
            <GridViewColumn.Header>
                <GridViewColumnHeader Tag="Name"
Click="lvUsersColumnHeader_Click">Name</GridViewColumnHeader>
            </GridViewColumn.Header>
        </GridViewColumn>
        <GridViewColumn Width="80" DisplayMemberBinding
=" {Binding Age}">
            <GridViewColumn.Header>
                <GridViewColumnHeader Tag="Age" Click
="lvUsersColumnHeader_Click">Age</GridViewColumnHeader>
            </GridViewColumn.Header>
        </GridViewColumn>
        <GridViewColumn Width="80" DisplayMemberBinding
=" {Binding Sex}">
            <GridViewColumn.Header>
                <GridViewColumnHeader Tag="Sex" Click
="lvUsersColumnHeader_Click">Sex</GridViewColumnHeader>
            </GridViewColumn.Header>
        </GridViewColumn>
    </GridView>
</ListView.View>
</ListView>
</Grid>
</Window>

```

Notice how I have specified headers for each of the columns using an actual `GridViewColumnHeader` element instead of just specifying a string. This is done so that I may set additional properties, in this case the `Tag` property as well as the `Click` event.

The `Tag` property is used to hold the field name that will be used to sort by, if this particular column is clicked. This is done in the `lvUsersColumnHeader_Click` event that each of the columns subscribes to.

That was the key concepts of the XAML. Besides that, we bind to our Code-behind properties `Name`, `Age` and `Sex`, which we'll discuss now.

#### 1.17.8.2. The Code-behind

In Code-behind, there are quite a few things happening. I use a total of three classes, which you would normally divide up into individual files, but for convenience, I have kept them in the same file, giving us a total of ~100 lines. First the code and then I'll explain how it works:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Media;

namespace WpfTutorialSamples.ListView_control
{
    public partial class ListViewColumnSortingSample : Window
    {
        private GridViewColumnHeader listViewSortCol = null;
        private SortAdorner listViewSortAdorner = null;

        public ListViewColumnSortingSample()
        {
            InitializeComponent();
            List<User> items = new List<User>();
            items.Add(new User() { Name = "John Doe", Age = 42, Sex =
SexType.Male });
            items.Add(new User() { Name = "Jane Doe", Age = 39, Sex =
SexType.Female });
            items.Add(new User() { Name = "Sammy Doe", Age = 13, Sex =
= SexType.Male });
            items.Add(new User() { Name = "Donna Doe", Age = 13, Sex =
= SexType.Female });
            lvUsers.ItemsSource = items;
        }

        private void lvUsers.ColumnHeader_Click(object sender,
RoutedEventArgs e)
        {
            GridViewColumnHeader column = (sender as
GridViewColumnHeader);
            string sortBy = column.Tag.ToString();
            if(listViewSortCol != null)
            {

```

```

AdornerLayer.GetAdornerLayer(listViewSortCol).Remove(listViewSortAdorner
);
    lvUsers.Items.SortDescriptions.Clear();
}

ListSortDirection newDir = ListSortDirection.Ascending;
if(listViewSortCol == column &&
listViewSortAdorner.Direction == newDir)
    newDir = ListSortDirection.Descending;

listViewSortCol = column;
listViewSortAdorner = new SortAdorner(listViewSortCol,
newDir);

AdornerLayer.GetAdornerLayer(listViewSortCol).Add(listViewSortAdorner);
    lvUsers.Items.SortDescriptions.Add(new
SortDescription(sortBy, newDir));
}
}

public enum SexType { Male, Female };

public class User
{
    public string Name { get; set; }

    public int Age { get; set; }

    public string Mail { get; set; }

    public SexType Sex { get; set; }
}

public class SortAdorner : Adorner
{
    private static Geometry ascGeometry =
        Geometry.Parse("M 0 4 L 3.5 0 L 7 4 Z");

    private static Geometry descGeometry =
        Geometry.Parse("M 0 0 L 3.5 4 L 7 0 Z");
}

```

```

public ListSortDirection Direction { get; private set; }

public SortAdorner(UIElement element, ListSortDirection dir)
    : base(element)
{
    this.Direction = dir;
}

protected override void OnRender(DrawingContext
drawingContext)
{
    base.OnRender(drawingContext);

    if(AdornedElement.RenderSize.Width < 20)
        return;

    TranslateTransform transform = new TranslateTransform
    (
        AdornedElement.RenderSize.Width - 15,
        (AdornedElement.RenderSize.Height - 5) / 2
    );
    drawingContext.PushTransform(transform);

    Geometry geometry = ascGeometry;
    if(this.Direction == ListSortDirection.Descending)
        geometry = descGeometry;
    drawingContext.DrawGeometry(Brushes.Black, null,
geometry);

    drawingContext.Pop();
}
}
}

```

Allow me to start from the bottom and then work my way up while explaining what happens. The last class in the file is an Adorner class called **SortAdorner**. All this little class does is to draw a triangle, either pointing up or down, depending on the sort direction. WPF uses the concept of adorners to allow you to paint stuff over other controls, and this is exactly what we want here: The ability to draw a sorting triangle on top of our ListView column header.

The **SortAdorner** works by defining two **Geometry** objects, which are basically used to describe 2D shapes - in this case a triangle with the tip pointing up and one with the tip pointing down. The `Geometry.Parse()` method uses the list of points to draw the triangles, which will be explained more thoroughly in a later article.

The **SortAdorner** is aware of the sort direction, because it needs to draw the proper triangle, but is not aware of the field that we order by - this is handled in the UI layer.

The **User** class is just a basic information class, used to contain information about a user. Some of this information is used in the UI layer, where we bind to the Name, Age and Sex properties.

In the Window class, we have two methods: The constructor where we build a list of users and assign it to the `ItemsSource` of our `ListView`, and then the more interesting click event handler that will be hit when the user clicks a column. In the top of the class, we have defined two private variables: `listViewSortCol` and `listViewSortAdorner`. These will help us keep track of which column we're currently sorting by and the adorner we placed to indicate it.

In the `IvUsersColumnHeader_Click` event handler, we start off by getting a reference to the column that the user clicked. With this, we can decide which property on the `User` class to sort by, simply by looking at the `Tag` property that we defined in XAML. We then check if we're already sorting by a column - if that is the case, we remove the adorner and clear the current sort descriptions.

After that, we're ready to decide the direction. The default is ascending, but we do a check to see if we're already sorting by the column that the user clicked - if that is the case, we change the direction to descending.

In the end, we create a new `SortAdorner`, passing in the column that it should be rendered on, as well as the direction. We add this to the `AdornerLayer` of the column header, and at the very end, we add a `SortDescription` to the `ListView`, to let it know which property to sort by and in which direction.

#### 1.17.8.3. Summary

Congratulations, you now have a fully sortable `ListView` with visual indication of sort column and direction. In case you want to know more about some of the concepts used in this article, like data binding, geometry or `ListView`s in general, then please check out some of the other articles, where each of the subjects are covered in depth.

## 1.17.9. ListView filtering

We've already done several different things with the ListView, like grouping and sorting, but another very useful ability is filtering. Obviously, you could just limit the items you add to the ListView in the first place, but often you would need to filter the ListView dynamically, in runtime, usually based on a user entered filter string. Luckily for us, the view mechanisms of the ListView also make it easy to do just that, like we saw it with sorting and grouping.

Filtering is actually quite easy to do, so let's jump straight into an example, and then we'll discuss it afterwards:

```
<Window x:Class="WpfTutorialSamples.ListView_control.FilteringSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="FilteringSample" Height="200" Width="300">
    <DockPanel Margin="10">
        <TextBox DockPanel.Dock="Top" Margin="0,0,0,10" Name
        ="txtFilter" TextChanged="txtFilter_TextChanged" />
        <ListView Name="lvUsers">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="Name" Width="120"
DisplayMemberBinding="{Binding Name}" />
                    <GridViewColumn Header="Age" Width="50"
DisplayMemberBinding="{Binding Age}" />
                </GridView>
            </ListView.View>
        </ListView>
    </DockPanel>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Data;

namespace WpfTutorialSamples.ListView_control
{
    public partial class FilteringSample : Window
    {
        public FilteringSample()
        {
```

```

    {

        InitializeComponent();

        List<User> items = new List<User>();
        items.Add(new User() { Name = "John Doe", Age = 42 });
        items.Add(new User() { Name = "Jane Doe", Age = 39 });
        items.Add(new User() { Name = "Sammy Doe", Age = 13 });
        items.Add(new User() { Name = "Donna Doe", Age = 13 });
        lvUsers.ItemsSource = items;

        CollectionView view =
(CollectionView)CollectionViewSource.GetDefaultView(lvUsers.ItemsSource)
;

        view.Filter = UserFilter;
    }

    private bool UserFilter(object item)
{
    if(String.IsNullOrEmpty(txtFilter.Text))
        return true;
    else
        return ((item as User).Name.IndexOf(txtFilter.Text,
 StringComparison.OrdinalIgnoreCase) >= 0);
}

    private void txtFilter_TextChanged(object sender,
System.Windows.Controls.TextChangedEventArgs e)
{
    CollectionViewSource.GetDefaultView(lvUsers.ItemsSource).Refresh();
}
}

public enum SexType { Male, Female };

public class User
{
    public string Name { get; set; }

    public int Age { get; set; }
}

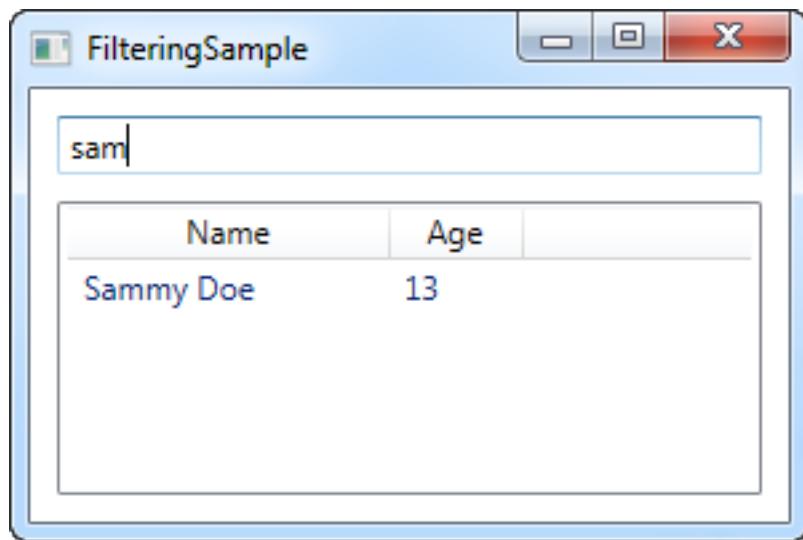
```

```

        public string Mail { get; set; }

        public SexType Sex { get; set; }
    }
}

```



The XAML part is pretty simple: We have a TextBox, where the user can enter a search string, and then a ListView to show the result in.

In Code-behind, we start off by adding some User objects to the ListView, just like we did in previous examples. The interesting part happens in the last two lines of the constructor, where we obtain a reference to the **CollectionView** instance for the ListView and then assign a delegate to the **Filter** property. This delegate points to the function called **UserFilter**, which we have implemented just below. It takes each item as the first (and only) parameter and then returns a boolean value that indicates whether or not the given item should be visible on the list.

In the **UserFilter()** method, we take a look at the TextBox control (**txtFilter**), to see if it contains any text - if it does, we use it to check whether or not the name of the User (which is the property we have decided to filter on) contains the entered string, and then return true or false depending on that. If the TextBox is empty, we return true, because in that case we want all the items to be visible.

The **txtFilter\_TextChanged** event is also important. Each time the text changes, we get a reference to the View object of the ListView and then call the **Refresh()** method on it. This ensures that the Filter delegate is called each time the user changes the value of the search/filter string text box.

#### 1.17.9.1. Summary

This was a pretty simple implementation, but since you get access to each item, in this case of the User class, you can do any sort of custom filtering that you like, since you have access to all of the data about each of the items in the list. For instance, the above example could easily be changed to filter on age, by looking at the Age property instead of the Name property, or you could modify it to look at more than one

property, e.g. to filter out users with an age below X AND a name that doesn't contain "Y".

# 1.18. The TreeView control

---

## 1.18.1. TreeView introduction

The TreeView control enabled you to display hierarchical data, with each piece of data represented by a node in the tree. Each node can then have child nodes, and the child nodes can have child nodes and so on. If you have ever used the Windows Explorer, you also know how a TreeView looks - it's the control that shows the current folder structure on your machine, in the left part of the Windows Explorer window.

### 1.18.1.1. TreeView in WPF vs. WinForms

If you have previously worked with the TreeView control in WinForms, you might think of the TreeView control as one that's easy to use but hard to customize. In WPF it's a little bit the other way around, at least for newbies: It feels a bit complicated to get started with, but it's a LOT easier to customize. Just like most other WPF controls, the TreeView is almost lookless once you start, but it can be styled almost endlessly without much effort.

Just like with the ListView control, the TreeView control does have its own item type, the TreeViewItem, which you can use to populate the TreeView. If you come from the WinForms world, you will likely start by generating TreeViewItem's and adding them to the Items property, and this is indeed possible. But since this is WPF, the preferred way is to bind the TreeView to a hierarchical data structure and then use an appropriate template to render the content.

We'll show you how to do it both ways, and while the good, old WinForms inspired way might seem like the easy choice at first, you should definitely give the WPF way a try - in the long run, it offers more flexibility and will fit in better with the rest of the WPF code you write.

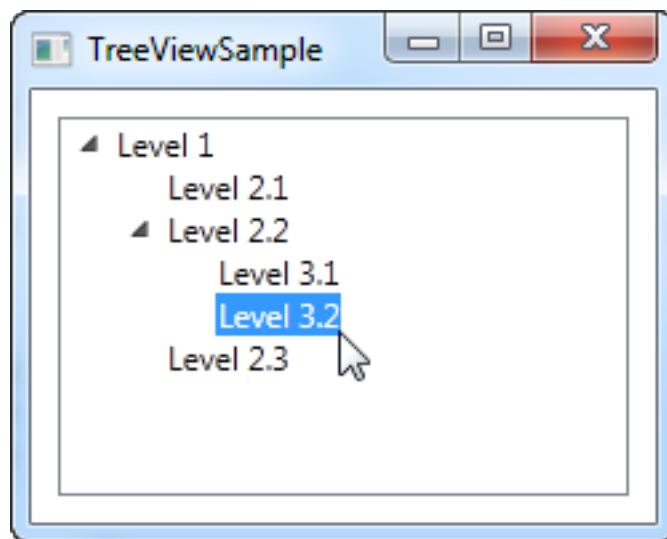
### 1.18.1.2. Summary

The WPF TreeView is indeed a complex control. In the first example, which we'll get into already in the next chapter, it might seem simple, but once you dig deeper, you'll see the complexity. Fortunately, the WPF TreeView control rewards you with great usability and flexibility. To show you all of them, we have dedicated an entire category to all the TreeView articles. Click on to the next one to get started.

## 1.18.2. A simple TreeView example

As we talked about in the previous article, the WPF TreeView can be used in a very simple manner, by adding TreeViewItem objects to it, either from Code-behind or simply by declaring them directly in your XAML. This is indeed very easy to get started with, as you can see from the example here:

```
<Window x:Class="WpfTutorialSamples.TreeView_control.TreeViewSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TreeViewSample" Height="200" Width="250">
    <Grid Margin="10">
        <TreeView>
            <TreeViewItem Header="Level 1" IsExpanded="True">
                <TreeViewItem Header="Level 2.1" />
                <TreeViewItem Header="Level 2.2" IsExpanded="True">
                    <TreeViewItem Header="Level 3.1" />
                    <TreeViewItem Header="Level 3.2" />
                </TreeViewItem>
                <TreeViewItem Header="Level 2.3" />
            </TreeViewItem>
        </TreeView>
    </Grid>
</Window>
```



We simply declare the TreeViewItem objects directly in the XAML, in the same structure that we want to display them in, where the first tag is a child of the TreeView control and its child objects are also child tags to its parent object. To specify the text we want displayed for each node, we use the **Header** property. By default, a TreeViewItem is not expanded, but to show you the structure of the example, I have used the **IsExpanded** property to expand the two parent items.

### 1.18.2.1. TreeViewItem's with images and other controls

The **Header** is an interesting property, though. As you can see, I can just specify a text string and then have it rendered directly without doing anything else, but this is WPF being nice to us - internally, it wraps the text inside of a **TextBlock** control, instead of forcing you to do it. This shows us that we can stuff pretty much whatever we want to into the Header property instead of just a string and then have the **TreeView** render it - a great example of why it's so easy to customize the look of WPF controls.

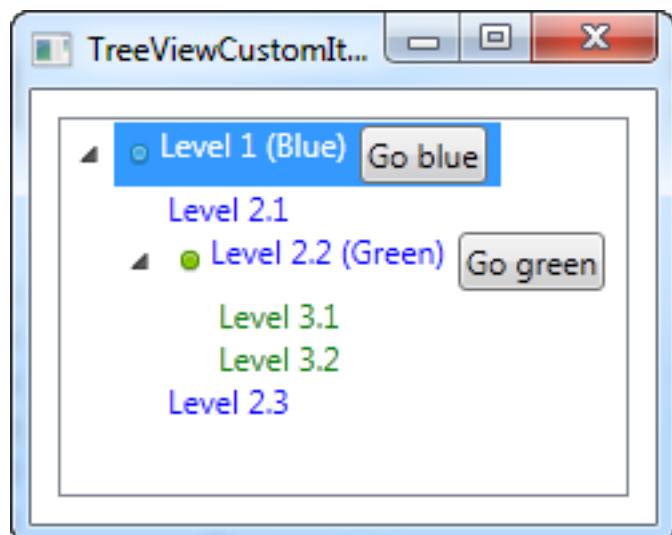
One of the common requests from people coming from WinForms or even other UI libraries is the ability to show an image next to the text label of a **TreeView** item. This is very easy to do with WinForms, because the **TreeView** is built exactly for this scenario. With the WPF **TreeView**, it's a bit more complex, but you're rewarded with a lot more flexibility than you could ever get from the WinForms **TreeView**. Here's an example of it:

```
<Window x:Class
    ="WpfTutorialSamples.TreeView_control.TreeViewCustomItemsSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TreeViewCustomItemsSample" Height="200" Width="250">
    <Grid Margin="10">
        <TreeView>
            <TreeViewItem IsExpanded="True">
                <TreeViewItem.Header>
                    <StackPanel Orientation="Horizontal">
                        <Image Source
                    =" /WpfTutorialSamples;component/Images/bullet_blue.png" />
                        <TextBlock Text="Level 1 (Blue)" />
                    </StackPanel>
                </TreeViewItem.Header>
                <TreeViewItem>
                    <TreeViewItem.Header>
                        <StackPanel Orientation="Horizontal">
                            <TextBlock Text="Level 2.1"
Foreground="Blue" />
                        </StackPanel>
                    </TreeViewItem.Header>
                </TreeViewItem>
            <TreeViewItem IsExpanded="True">
                <TreeViewItem.Header>
                    <StackPanel Orientation="Horizontal">
                        <Image Source
```

```

        ="/WpfTutorialSamples;component/Images/bullet_green.png" />
        <TextBlock Text="Level 2.2 (Green)" Foreground="Blue" />
    </StackPanel>
</TreeViewItem.Header>
<TreeViewItem>
    <TreeViewItem.Header>
        <TextBlock Text="Level 3.1" Foreground="Green" />
    </TreeViewItem.Header>
</TreeViewItem>
<TreeViewItem>
    <TreeViewItem.Header>
        <TextBlock Text="Level 3.2" Foreground="Green" />
    </TreeViewItem.Header>
</TreeViewItem>
<TreeViewItem>
    <TreeViewItem.Header>
        <TextBlock Text="Level 2.3" Foreground="Blue" />
    </TreeViewItem.Header>
</TreeViewItem>
</TreeViewItem>
</TreeView>
</Grid>
</Window>

```



I did a whole bunch of things here, just to show you the kind of flexibility you get: I colored the child items and I added images and even buttons to the parent items. Because we're defining the entire thing with simple markup, you can do almost anything, but as you can see from the example code, it does come with a price: Huge amounts of XAML code, for a tree with just six nodes in total!

### 1.18.2.2. Summary

While it is entirely possible to define an entire TreeView just using markup, as we did in the above examples, it's not the best approach in most situations, and while you could do it from Code-behind instead, this would have resulted in even more lines of code. Once again the solution is **data binding**, which we'll look into in the next chapters.

## 1.18.3. TreeView, data binding and multiple templates

The WPF TreeView supports data binding, like pretty much all other WPF controls does, but because the TreeView is hierarchical in nature, a normal DataTemplate often won't suffice. Instead, we use the HierarchicalDataTemplate, which allows us to template both the tree node itself, while controlling which property to use as a source for child items of the node.

### 1.18.3.1. A basic data bound TreeView

In the following example, I'll show you just how easy it is to get started with the HierarchicalDataTemplate:

```
<Window x:Class
        ="WpfTutorialSamples.TreeView_control.TreeViewDataBindingSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:self="clr-namespace:WpfTutorialSamples.TreeView_control"
        Title="TreeViewDataBindingSample" Height="150" Width="200">
    <Grid Margin="10">
        <TreeView Name="trvMenu">
            <TreeView.ItemTemplate>
                <HierarchicalDataTemplate DataType="{x:Type
self:MenuItem}" ItemsSource="{Binding Items}">
                    <TextBlock Text="{Binding Title}" />
                </HierarchicalDataTemplate>
            </TreeView.ItemTemplate>
        </TreeView>
    </Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.IO;
using System.Collections.ObjectModel;

namespace WpfTutorialSamples.TreeView_control
{
    public partial class TreeViewDataBindingSample : Window
    {
        public TreeViewDataBindingSample()
        {
```

```

        InitializeComponent();
        MenuItem root = new MenuItem() { Title = "Menu" };
        MenuItem childItem1 = new MenuItem() { Title = "Child item #1" };
        childItem1.Items.Add(new MenuItem() { Title = "Child item #1.1" });
        childItem1.Items.Add(new MenuItem() { Title = "Child item #1.2" });
        root.Items.Add(childItem1);
        root.Items.Add(new MenuItem() { Title = "Child item #2" });
    });
    trvMenu.Items.Add(root);
}
}

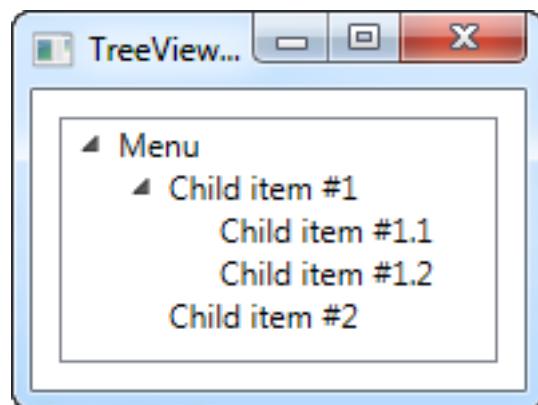
public class MenuItem
{
    public MenuItem()
    {
        this.Items = new ObservableCollection<MenuItem>();
    }

    public string Title { get; set; }

    public ObservableCollection<MenuItem> Items { get; set; }
}

}

```



In the XAML markup, I have specified a **HierarchicalDataTemplate** for the **ItemTemplate** of the TreeView. I instruct it to use the **Items** property for finding child items, by setting the **ItemsSource** property of the

template, and inside of it I define the actual template, which for now just consists of a TextBlock bound to the **Title** property.

This first example was very simple, in fact so simple that we might as well have just added the TreeView items manually, instead of generating a set of objects and then binding to them. However, as soon as things get a bit more complicated, the advantages of using data bindings gets more obvious.

### 1.18.3.2. Multiple templates for different types

In the next example, I've taken a slightly more complex case, where I want to show a tree of families and their members. A family should be represented in one way, while each of its members should be shown in another way. I achieve this by creating two different templates and specifying them as resources of the tree (or the Window or the Application - that's really up to you), and then allowing the TreeView to pick the correct template based on the underlying type of data.

Here's the code - the explanation of it will follow right after:

```
<Window x:Class
        ="WpfTutorialSamples.TreeView_control.TreeViewMultipleTemplatesSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:self="clr-namespace:WpfTutorialSamples.TreeView_control"
        Title="TreeViewMultipleTemplatesSample" Height="200" Width
        ="250">
    <Grid Margin="10">
        <TreeView Name="trvFamilies">
            <TreeView.Resources>
                <HierarchicalDataTemplate DataType="{x:Type
self:Family}" ItemsSource="{Binding Members}">
                    <StackPanel Orientation="Horizontal">
                        <Image Source
                            ="/WpfTutorialSamples;component/Images/group.png" Margin="0,0,5,0" />
                        <TextBlock Text="{Binding Name}" />
                        <TextBlock Text=" [" Foreground="Blue" />
                        <TextBlock Text="{Binding Members.Count}"
                            Foreground="Blue" />
                        <TextBlock Text=" ]" Foreground="Blue" />
                    </StackPanel>
                </HierarchicalDataTemplate>
                <DataTemplate DataType="{x:Type self:FamilyMember}">
                    <StackPanel Orientation="Horizontal">
```

```

                <Image Source
        ="/WpfTutorialSamples;component/Images/user.png" Margin="0,0,5,0" />
                    <TextBlock Text="{Binding Name}" />
                    <TextBlock Text=" (" Foreground="Green" />
                    <TextBlock Text="{Binding Age}" Foreground
        ="Green" />
                    <TextBlock Text=" years)" Foreground
        ="Green" />
                </StackPanel>
            </DataTemplate>
        </TreeView.Resources>
    </TreeView>
</Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.Collections.ObjectModel;

namespace WpfTutorialSamples.TreeView_control
{
    public partial class TreeViewMultipleTemplatesSample : Window
    {
        public TreeViewMultipleTemplatesSample()
        {
            InitializeComponent();

            List<Family> families = new List<Family>();

            Family family1 = new Family() { Name = "The Doe's" };
            family1.Members.Add(new FamilyMember() { Name = "John
Doe", Age = 42 });
            family1.Members.Add(new FamilyMember() { Name = "Jane
Doe", Age = 39 });
            family1.Members.Add(new FamilyMember() { Name = "Sammy
Doe", Age = 13 });
            families.Add(family1);

            Family family2 = new Family() { Name = "The Moe's" };

```

```

        family2.Members.Add(new FamilyMember() { Name = "Mark
Moe", Age = 31 });
        family2.Members.Add(new FamilyMember() { Name = "Norma
Moe", Age = 28 });
        families.Add(family2);

        trvFamilies.ItemsSource = families;
    }
}

public class Family
{
    public Family()
    {
        this.Members = new ObservableCollection<FamilyMember>();
    }

    public string Name { get; set; }

    public ObservableCollection<FamilyMember> Members { get; set;
}
}

public class FamilyMember
{
    public string Name { get; set; }

    public int Age { get; set; }
}

```

As mentioned, the two templates are declared as a part of the TreeView resources, allowing the TreeView to select the appropriate template based on the data type that it's about to show. The template defined for the **Family** type is a hierarchical template, using the **Members** property to show its family members.

The template defined for the **FamilyMember** type is a regular DataTemplate, since this type doesn't have any child members. However, if we had wanted each FamilyMember to keep a collection of their children and perhaps their children's children, then we would have used a hierarchical template instead.

In both templates, we use an image representing either a family or a family member, and then we show some interesting data about it as well, like the amount of family members or the person's age.



In the code-behind, we simply create two Family instances, fill each of them with a set of members, and then add each of the families to a list, which is then used as the items source for the TreeView.

#### 1.18.3.3. Summary

Using data binding, the TreeView is very customizable and with the ability to specify multiple templates for rendering different data types, the possibilities are almost endless.

## 1.18.4. TreeView - Selection/Expansion state

In the previous couple of TreeView articles, we used data binding to display custom objects in a WPF TreeView. This works really well, but it does leave you with one problem: Because each tree node is now represented by your custom class, for instance FamilyMember as we saw in the previous article, you no longer have direct control over TreeView node specific functionality like selection and expansion state. In praxis this means that you can't select or expand/collapse a given node from code-behind.

Lots of solutions exists to handle this, ranging from "hacks" where you use the item generators of the TreeView to get the underlying TreeViewItem, where you can control the IsExpanded and IsSelected properties, to much more advanced MVVM-inspired implementations. In this article I would like to show you a solution that lies somewhere in the middle, making it easy to implement and use, while still not being a complete hack.

### 1.18.4.1. A TreeView selection/expansion solution

The basic principle is to implement two extra properties on your data class: IsExpanded and IsSelected. These two properties are then hooked up to the TreeView, using a couple of styles targeting the TreeViewItem, inside of the **ItemContainerStyle** for the TreeView.

You could easily implement these two properties on all of your objects, but it's much easier to inherit them from a base object. If this is not feasible for your solution, you could create an interface for it and then implement this instead, to establish a common ground. For this example, I've chosen the base class method, because it allows me to very easily get the same functionality for my other objects. Here's the code:

```
<Window x:Class
        ="WpfTutorialSamples.TreeView_control.TreeViewSelectionExpansionSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TreeViewSelectionExpansionSample" Height="200" Width
        ="300">
    <DockPanel Margin="10">
        <WrapPanel Margin="0,10,0,0" DockPanel.Dock="Bottom"
        HorizontalAlignment="Center">
            <Button Name="btnSelectNext" Click="btnSelectNext_Click"
Width="120">Select next</Button>
            <Button Name="btnToggleExpansion" Click
            ="btnToggleExpansion_Click" Width="120" Margin="10,0,0,0">Toggle
            expansion</Button>
        </WrapPanel>
        <TreeView Name="trvPersons">
```

```

<TreeView.ItemTemplate>
    <HierarchicalDataTemplate ItemsSource="{Binding
Children}">
        <StackPanel Orientation="Horizontal">
            <Image Source
="/WpfTutorialSamples/component/Images/user.png" Margin="0,0,5,0" />
            <TextBlock Text="{Binding Name}" Margin
="0,0,4,0" />
        </StackPanel>
    </HierarchicalDataTemplate>
</TreeView.ItemTemplate>
<TreeView.ItemContainerStyle>
    <Style TargetType="TreeViewItem">
        <Setter Property="IsSelected" Value="{Binding
IsSelected}" />
        <Setter Property="IsExpanded" Value="{Binding
IsExpanded}" />
    </Style>
</TreeView.ItemContainerStyle>
</TreeView>
</DockPanel>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Windows.Controls;

namespace WpfTutorialSamples.TreeView_control
{
    public partial class TreeViewSelectionExpansionSample : Window
    {
        public TreeViewSelectionExpansionSample()
        {
            InitializeComponent();

            List<Person> persons = new List<Person>();
            Person person1 = new Person() { Name = "John Doe", Age =

```

```

42 } ;

        Person person2 = new Person() { Name = "Jane Doe", Age =
39 } ;

        Person child1 = new Person() { Name = "Sammy Doe", Age =
13 } ;
        person1.Children.Add(child1);
        person2.Children.Add(child1);

        person2.Children.Add(new Person() { Name = "Jenny Moe",
Age = 17 });

        Person person3 = new Person() { Name = "Becky Toe", Age =
25 } ;

        persons.Add(person1);
        persons.Add(person2);
        persons.Add(person3);

        person2.IsExpanded = true;
        person2.isSelected = true;

        trvPersons.ItemsSource = persons;
    }

    private void btnSelectNext_Click(object sender,
RoutedEventArgs e)
{
    if(trvPersons.SelectedItem != null)
    {
        var list = (trvPersons.ItemsSource as List<Person>);
        int curIndex = list.IndexOf(trvPersons.SelectedItem
as Person);
        if(curIndex >= 0)
            curIndex++;
        if(curIndex >= list.Count)
            curIndex = 0;
        if(curIndex >= 0)
            list[curIndex].isSelected = true;
    }
}

```

```

        }

    }

    private void btnToggleExpansion_Click(object sender,
RoutedEventArgs e)
{
    if(trvPersons.SelectedItem != null)
        (trvPersons.SelectedItem as Person).IsExpanded =
!(trvPersons.SelectedItem as Person).IsExpanded;
}

}

public class Person : TreeViewItemBase
{
    public Person()
    {
        this.Children = new ObservableCollection<Person>();
    }

    public string Name { get; set; }

    public int Age { get; set; }

    public ObservableCollection<Person> Children { get; set; }
}

public class TreeViewItemBase : INotifyPropertyChanged
{
    private bool isSelected;
    public bool IsSelected
    {
        get { return this.isSelected; }
        set
        {
            if(value != this.isSelected)
            {
                this.isSelected = value;
        }
    }
}

```

```

        NotifyPropertyChanged( "IsSelected" );
    }
}

private bool isExpanded;
public bool IsExpanded
{
    get { return this.isExpanded; }
    set
    {
        if(value != this.isExpanded)
        {
            this.isExpanded = value;
            NotifyPropertyChanged( "IsExpanded" );
        }
    }
}

public event PropertyChangedEventHandler PropertyChanged;

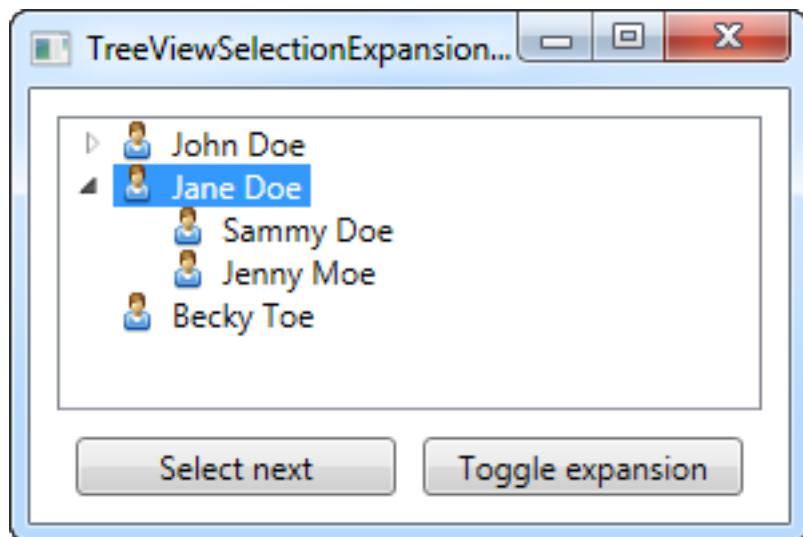
public void NotifyPropertyChanged(string propName)
{
    if(this.PropertyChanged != null)
        this.PropertyChanged(this, new
PropertyChangedEventArgs(propName));
}
}

```

I'm sorry for the rather large amount of code in one place. In a real world solution, it would obviously be spread out over multiple files instead and the data for the tree would likely come from an actual data source, instead of being generated on the fly. Allow me to explain what happens in the example.

#### 1.18.4.2. XAML part

I have defined a couple of buttons to be placed in the bottom of the dialog, to use the two new properties. Then we have the TreeView, for which I have defined an ItemTemplate (as demonstrated in a previous chapter) as well as an ItemContainerStyle. If you haven't read the chapters on styling yet, you might not completely understand that part, but it's simply a matter of tying together the properties on our own custom



class with the **IsSelected** and **IsExpanded** properties on the **TreeViewItems**, which is done with Style setters. You can learn more about them elsewhere in this tutorial.

#### 1.18.4.3. Code-behind part

In the code-behind, I have defined a **Person** class, with a couple of properties, which inherits our extra properties from the **TreeViewItemBase** class. You should be aware that the **TreeViewItemBase** class implements the **INotifyPropertyChanged** interface and uses it to notify of changes to these two essential properties - without this, selection/expansion changes won't be reflected in the UI. The concept of notification changes are explained in the Data binding chapters.

In the main Window class I simply create a range of persons, while adding children to some of them. I add the persons to a list, which I assign as the **ItemsSource** of the **TreeView**, which, with a bit of help from the defined template, renders them the way they are shown on the screenshot.

The most interesting part happens when I set the **IsExpanded** and **IsSelected** properties on the *person2* object. This is what causes the second person (Jane Doe) to be initially selected and expanded, as shown on the screenshot. We also use these two properties on the Person objects (inherited from the **TreeViewItemBase** class) in the event handlers for the two test buttons (please bear in mind that, to keep the code as small and simple as possible, the selection button only works for the top level items).

#### 1.18.4.4. Summary

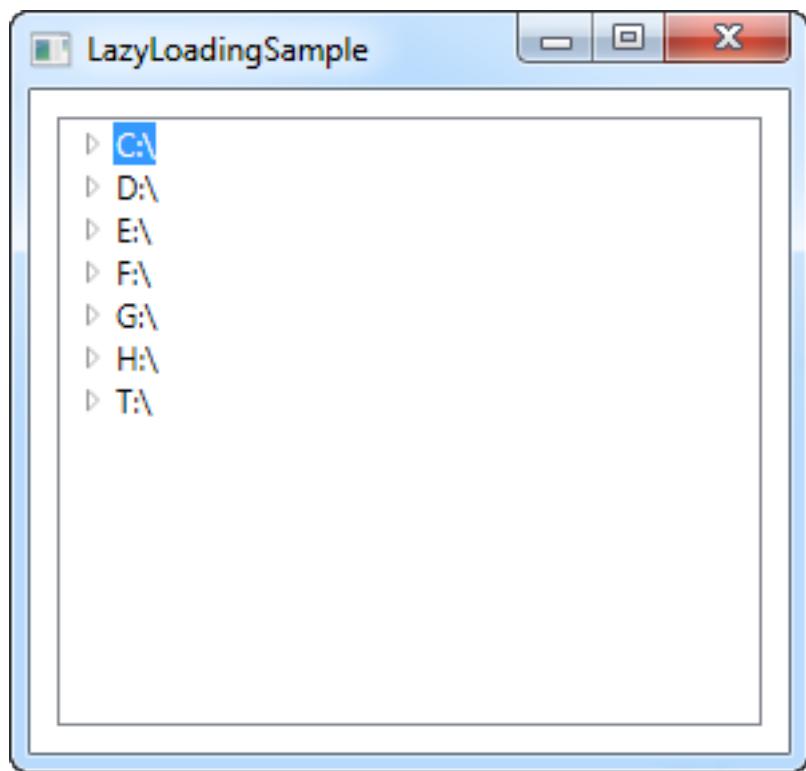
By creating and implementing a base class for the objects that you wish to use and manipulate within a **TreeView**, and using the gained properties in the **ItemContainerStyle**, you make it a lot easier to work with selections and expansion states. There are many solutions to tackle this problem with, and while this should do the trick, you might be able to find a solution that fits your needs better. As always with programming, it's all about using the right tool for the job at hand.

## 1.18.5. Lazy loading TreeView items

The usual process when using the TreeView is to bind to a collection of items or to manually add each level at the same time. However, in some situations, you want to delay the loading of a nodes child items until they are actually needed. This is especially useful if you have a very deep tree, with lots of levels and child nodes and a great example of this, is the folder structure of your Windows computer.

Each drive on your Windows computer has a range of child folders, and each of those child folders have child folders beneath them and so on. Looping through each drive and each drives child folders could become extremely time consuming and your TreeView would soon consist of a lot of nodes, with a high percentage of them never being needed. This is the perfect task for a lazy-loaded TreeView, where child folders are only loaded on demand.

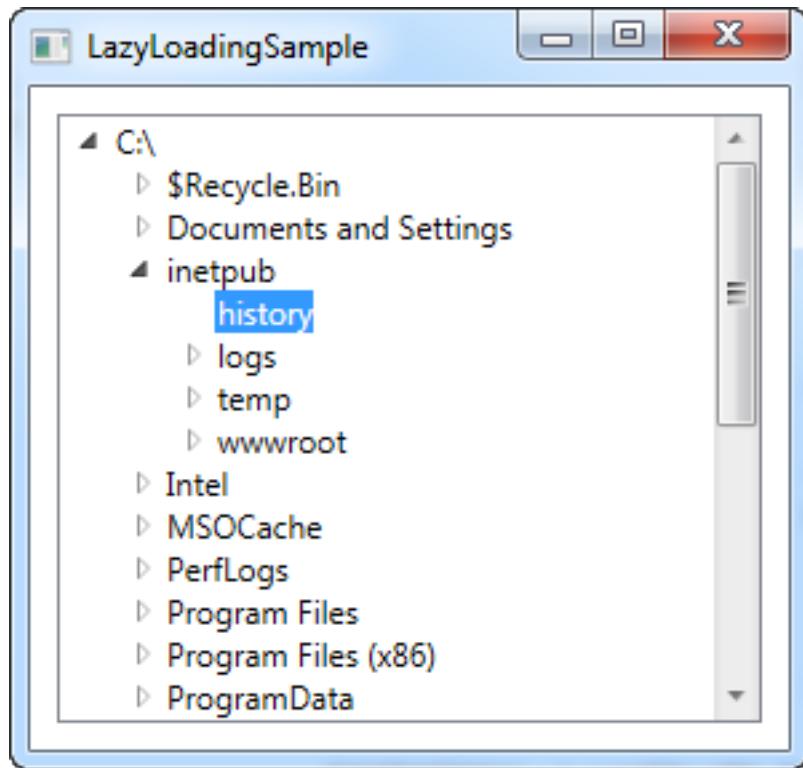
To achieve this, we simply add a dummy folder to each drive or child folder, and then when the user expands it, we remove the dummy folder and replace it with the actual values. This is how our application looks when it starts - by that time, we have only obtained a list of available drives on the computer:



You can now start expanding the nodes, and the application will automatically load the sub folders. If a folder is empty, it will be shown as empty once you try to expand it, as it can be seen on the next screenshot:

So how is it accomplished? Let's have a look at the code:

```
<Window x:Class="WpfTutorialSamples.TreeView_control.LazyLoadingSample"
        xmlns="
```



```
= "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="LazyLoadingSample" Height="300" Width="300">
<Grid>
    <TreeView Name="trvStructure" TreeViewItem.Expanded
="TreeViewItem_Expanded" Margin="10" />
</Grid>
</Window>

using System;
using System.IO;
using System.Windows;
using System.Windows.Controls;

namespace WpfTutorialSamples.TreeView_control
{
    public partial class LazyLoadingSample : Window
    {
        public LazyLoadingSample()
        {
            InitializeComponent();
            DriveInfo[] drives = DriveInfo.GetDrives();
            foreach(DriveInfo driveInfo in drives)
                trvStructure.Items.Add(CreateTreeItem(driveInfo));
        }
    }
}
```

```

        }

    public void TreeViewItem_Expanded(object sender,
RoutedEventArgs e)
{
    TreeViewItem item = e.Source as TreeViewItem;
    if((item.Items.Count == 1) && (item.Items[0] is string))
    {
        item.Items.Clear();

        DirectoryInfo expandedDir = null;
        if(item.Tag is DriveInfo)
            expandedDir = (item.Tag as
DriveInfo).RootDirectory;
        if(item.Tag is DirectoryInfo)
            expandedDir = (item.Tag as DirectoryInfo);
        try
        {
            foreach(DirectoryInfo subDir in
expandedDir.GetDirectories())
                item.Items.Add(CreateTreeItem(subDir));
        }
        catch { }
    }
}

private TreeViewItem CreateTreeItem(object o)
{
    TreeViewItem item = new TreeViewItem();
    item.Header = o.ToString();
    item.Tag = o;
    item.Items.Add("Loading...");
    return item;
}
}
}

```

The **XAML** is very simple and only one interesting detail is present: The way we subscribe to the Expanded event of TreeViewItem's. Notice that this is indeed the TreeViewItem and not the TreeView itself, but because the event bubbles up, we are able to just capture it in one place for the entire TreeView, instead of having to subscribe to it for each item we add to the tree. This event gets called each time an item is

expanded, which we need to be aware of to load its child items on demand.

In **Code-behind**, we start by adding each drive found on the computer to the TreeView control. We assign the **DriveInfo** instance to the Tag property, so that we can later retrieve it. Notice that we use a custom method to create the TreeViewItem, called **CreateTreeItem()**, since we can use the exact same method when we want to dynamically add a child folder later on. Notice in this method how we add a child item to the Items collection, in the form of a string with the text "Loading...".

Next up is the TreeViewItem\_Expanded event. As already mentioned, this event is raised each time a TreeView item is expanded, so the first thing we do is to check whether this item has already been loaded, by checking if the child items currently consists of only one item, which is a string - if so, we have found the "Loading..." child item, which means that we should now load the actual contents and replace the placeholder item with it.

We now use the items Tag property to get a reference to the **DriveInfo** or **DirectoryInfo** instance that the current item represents, and then we get a list of child directories, which we add to the clicked item, once again using the **CreateTreeItem()** method. Notice that the loop where we add each child folder is in a try..catch block - this is important, because some paths might not be accessible, usually for security reasons. You could grab the exception and use it to reflect this in the interface in one way or another.

#### 1.18.5.1. Summary

By subscribing to the Expanded event, we can easily create a lazy-loaded TreeView, which can be a much better solution than a statically created one in several situations.

# 1.19. The DataGrid control

---

## 1.19.1. The DataGrid control

The DataGrid control looks a lot like the ListView, when using a GridView, but it offers a lot of additional functionality. For instance, the DataGrid can automatically generate columns, depending on the data you feed it with. The DataGrid is also editable by default, allowing the end-user to change the values of the underlying data source.

The most common usage for the DataGrid is in combination with a database, but like most WPF controls, it works just as well with an in-memory source, like a list of objects. Since it's a lot easier to demonstrate, we'll mostly be using the latter approach in this tutorial.

### 1.19.1.1. A simple DataGrid

You can start using the DataGrid without setting any properties, because it supports so much out of the box. In this first example, we'll do just that, and then assign a list of our own User objects as the items source:

```
<Window x:Class
        ="WpfTutorialSamples.DataGrid_control.SimpleDataGridSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SimpleDataGridSample" Height="180" Width="300">
    <Grid Margin="10">
        <DataGrid Name="dgSimple"></DataGrid>
    </Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples.DataGrid_control
{
    public partial class SimpleDataGridSample : Window
    {
        public SimpleDataGridSample()
        {
            InitializeComponent();
        }
    }
}
```

```

        List<User> users = new List<User>();
        users.Add(new User() { Id = 1, Name =
"John Doe", Birthday = new DateTime(1971, 7, 23) });
        users.Add(new User() { Id = 2, Name =
"Jane Doe", Birthday = new DateTime(1974, 1, 17) });
        users.Add(new User() { Id = 3, Name =
"Sammy Doe", Birthday = new DateTime(1991, 9, 2) });

        dgSimple.ItemsSource = users;
    }
}

public class User
{
    public int Id { get; set; }

    public string Name { get; set; }

    public DateTime Birthday { get; set; }
}
}

```

Id	Name	Birthday
1	John Doe	7/23/1971 12:00:00 AM
2	Jane Doe	1/17/1974 12:00:00 AM
3	Sammy Doe	9/2/1991 12:00:00 AM

That's really all you need to start using the DataGrid. The source could just as easily have been a database table/view or even an XML file - the DataGrid is not picky about where it gets its data from.

If you click inside one of the cells, you can see that you're allowed to edit each of the properties by default. As a nice little bonus, you can try clicking one of the column headers - you will see that the DataGrid supports sorting right out of the box!

The last and empty row will let you add to the data source, simply by filling out the cells.

### 1.19.1.2. Summary

As you can see, it's extremely easy to get started with the DataGrid, but it's also a highly customizable control. In the next chapters, we'll look into all the cool stuff you can do with the DataGrid, so read on.

## 1.19.2. DataGrid columns

In the previous chapter, we had a look at just how easy you could get a WPF DataGrid up and running. One of the reasons why it was so easy is the fact that the DataGrid will automatically generate appropriate columns for you, based on the data source you use.

However, in some situations you might want to manually define the columns shown, either because you don't want all the properties/columns of the data source, or because you want to be in control of which inline editors are used.

### 1.19.2.1. Manually defined columns

Let's try an example that looks a lot like the one in the previous chapter, but where we define all the columns manually, for maximum control. You can select the column type based on the data that you wish to display/edit. As of writing, the following column types are available:

- `DataGridTextColumn`
- `DataGridCheckBoxColumn`
- `DataGridComboBoxColumn`
- `DataGridHyperlinkColumn`
- `DataGridTemplateColumn`

Especially the last one, the `DataGridTemplateColumn`, is interesting. It allows you to define any kind of content, which opens up the opportunity to use custom controls, either from the WPF library or even your own or 3rd party controls. Here's an example:

```
<Window x:Class
        ="WpfTutorialSamples.DataGrid_control.DataGridColumnssSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataGridColumnssSample" Height="200" Width="300">
    <Grid Margin="10">
        <DataGrid Name="dgUsers" AutoGenerateColumns="False">
            <DataGrid.Columns>

                <DataGridTextColumn Header="Name" Binding="{Binding
Name}" />

                <DataGridTemplateColumn Header="Birthday">
                    <DataGridTemplateColumn.CellTemplate>
                        <DataTemplate>
                            <DatePicker SelectedDate="{Binding
Birthday}" BorderThickness="0" />
                        </DataTemplate>
                    </DataGridTemplateColumn.CellTemplate>
                </DataGridTemplateColumn>
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

```

        </DataTemplate>
    </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>

</DataGrid.Columns>
</DataGrid>
</Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples.DataGrid_control
{
    public partial class DataGridViewSample : Window
    {
        public DataGridViewSample()
        {
            InitializeComponent();

            List<User> users = new List<User>();
            users.Add(new User() { Id = 1, Name = "John Doe",
                Birthday = new DateTime(1971, 7, 23) });
            users.Add(new User() { Id = 2, Name = "Jane Doe",
                Birthday = new DateTime(1974, 1, 17) });
            users.Add(new User() { Id = 3, Name = "Sammy Doe",
                Birthday = new DateTime(1991, 9, 2) });

            dgUsers.ItemsSource = users;
        }
    }

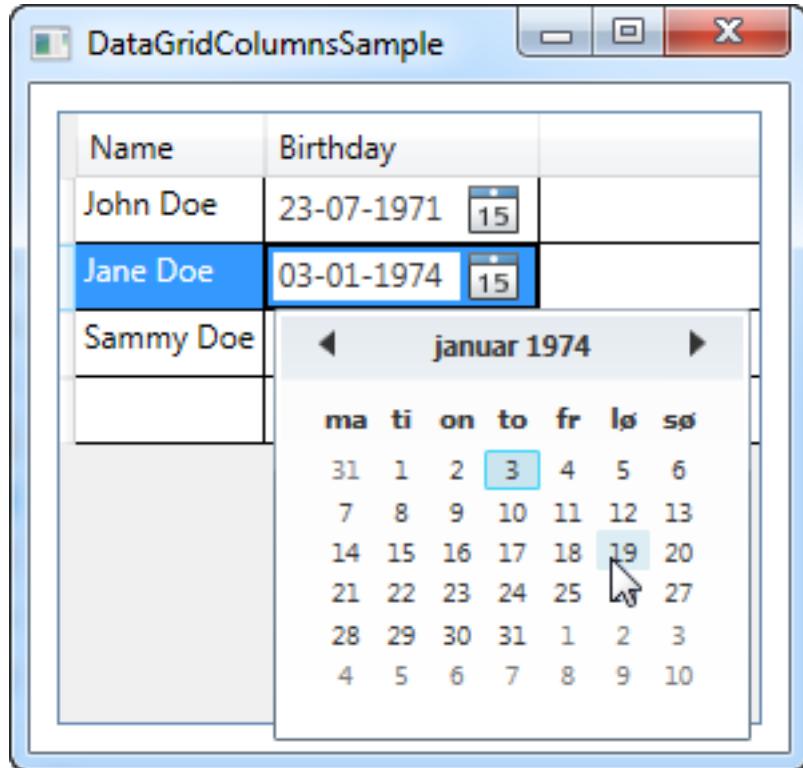
    public class User
    {
        public int Id { get; set; }

        public string Name { get; set; }

        public DateTime Birthday { get; set; }
    }
}

```

```
}
```



In the markup, I have added the `AutoGenerateColumns` property on the `DataGrid`, which I have set to false, to get control of the columns used. As you can see, I have left out the ID column, as I decided that I didn't care for it for this example. For the Name property, I've used a simple text based column, so the most interesting part of this example comes with the Birthday column, where I've used a `DataGridViewTemplateColumn` with a `DatePicker` control inside of it. This allows the end-user to pick the date from a calendar, instead of having to manually enter it, as you can see on the screenshot.

#### 1.19.2.2. Summary

By turning off automatically generated columns using the `AutoGenerateColumns` property, you get full control of which columns are shown and how their data should be viewed and edited. As seen by the example of this article, this opens up for some pretty interesting possibilities, where you can completely customize the editor and thereby enhance the end-user experience.

### 1.19.3. DataGridView with row details

A very common usage scenario when using a DataGridView control is the ability to show details about each row, typically right below the row itself. The WPF DataGridView control supports this very well, and fortunately it's also very easy to use. Let's start off with an example and then we'll discuss how it works and the options it gives you afterwards:

```
<Window x:Class
        ="WpfTutorialSamples.DataGridView_control.DataGridViewDetailsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataGridViewDetailsSample" Height="200" Width="400">
    <Grid Margin="10">
        <DataGrid Name="dgUsers" AutoGenerateColumns="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="Name" Binding="{Binding
Name}" />
                <DataGridTextColumn Header="Birthday" Binding="
{Binding Birthday}" />
            </DataGrid.Columns>
            <DataGrid.RowDetailsTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding Details}" Margin="10"
/>
                </DataTemplate>
            </DataGrid.RowDetailsTemplate>
        </DataGrid>
    </Grid>
</Window>

using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples.DataGridView_control
{
    public partial class DataGridViewDetailsSample : Window
    {
        public DataGridViewDetailsSample()
        {
            InitializeComponent();
        }
    }
}
```

```

        List<User> users = new List<User>();
        users.Add(new User() { Id = 1, Name = "John Doe",
Birthday = new DateTime(1971, 7, 23) });
        users.Add(new User() { Id = 2, Name = "Jane Doe",
Birthday = new DateTime(1974, 1, 17) });
        users.Add(new User() { Id = 3, Name = "Sammy Doe",
Birthday = new DateTime(1991, 9, 2) });

        dgUsers.ItemsSource = users;
    }
}

public class User
{
    public int Id { get; set; }

    public string Name { get; set; }

    public DateTime Birthday { get; set; }

    public string Details
    {
        get
        {
            return String.Format("{0} was born on {1} and this
is a long description of the person.", this.Name, this
.Birthday.ToString("yyyy-MM-dd"));
        }
    }
}

```

As you can see, I have expanded the example from previous chapters with a new property on the User class: The Description property. It simply returns a bit of information about the user in question, for our details row.

In the markup, I have defined a couple of columns and then I use the **RowDetailsTemplate** to specify a template for the row details. As you can see, it works much like any other WPF template, where I use a DataTemplate with one or several controls inside of it, along with a standard binding against a property on the data source, in this case the Description property.

As you can see from the resulting screenshot, or if you run the sample yourself, the details are now shown below the selected row. As soon as you select another row, the details for that row will be shown and the details for the previously selected row will be hidden.

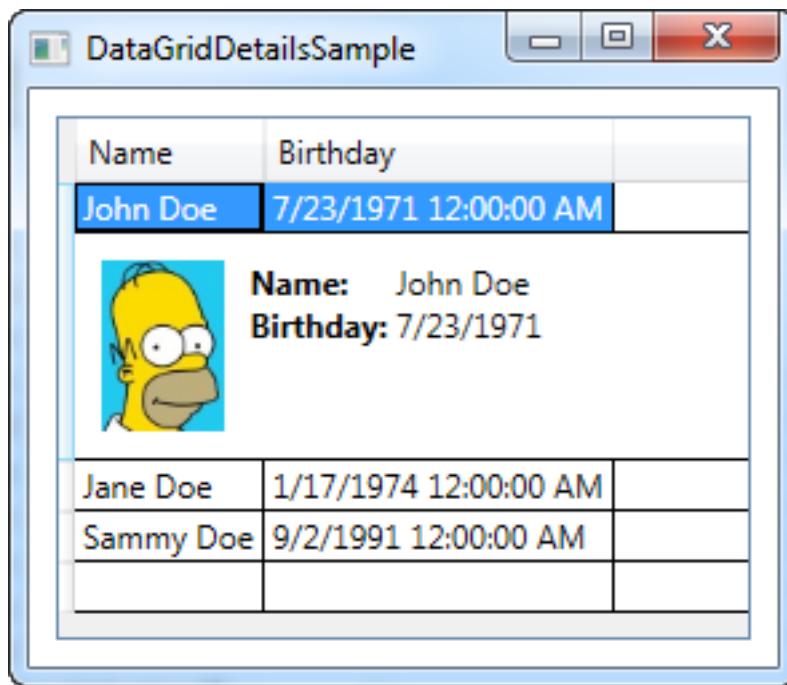
#### 1.19.3.1. Controlling row details visibility

Using the **RowDetailsVisibilityMode** property, you can change the above mentioned behavior though. It defaults to **VisibleWhenSelected**, where details are only visible when its parent row is selected, but you can change it to **Visible** or **Collapsed**. If you set it to Visible, all details rows will be visible all the time, like this:

If you set it to Collapsed, all details will be invisible all the time.

### 1.19.3.2. More details

The first example of this article might have been a tad boring, using just a single, plain `TextBlock` control. Of course, with this being a `DataTemplate`, you can do pretty much whatever you want, so I decided to extend the example a bit, to give a better idea of the possibilities. Here's how it looks now:



As you can see from the code listing, it's mostly about expanding the details template into using a panel, which in turn can host more panels and/or controls. Using a Grid panel, we can get the tabular look of the user data, and an `Image` control allows us to show a picture of the user (which you should preferably load from a locale resource and not a remote one, like I do in the example - and sorry for being too lazy to find a matching image of Jane and Sammy Doe).

```
<Window x:Class="WpfTutorialSamples.DataGrid_control.DataGridDetailsSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataGridDetailsSample" Height="300" Width="300">
    <Grid Margin="10">
        <DataGrid Name="dgUsers" AutoGenerateColumns="False">
            <DataGrid.Columns>
                <DataGridTextColumn Header="Name" Binding="{Binding Name}" />
                <DataGridTextColumn Header="Birthday" Binding="{Binding Birthday}" />
            </DataGrid.Columns>
        </DataGrid>
    </Grid>

```

```

        </DataGrid.Columns>
        <DataGrid.RowDetailsTemplate>
            <DataTemplate>
                <DockPanel Background="GhostWhite">
                    <Image DockPanel.Dock="Left" Source="{Binding ImageUrl}" Height="64" Margin="10" />
                    <Grid Margin="0,10">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="Auto" />
                            <ColumnDefinition Width="*" />
                        </Grid.ColumnDefinitions>
                        <Grid.RowDefinitions>
                            <RowDefinition Height="Auto" />
                            <RowDefinition Height="Auto" />
                            <RowDefinition Height="Auto" />
                        </Grid.RowDefinitions>

                        <TextBlock Text="ID: " FontWeight="Bold" />
                        <TextBlock Text="{Binding Id}" Grid.Column="1" />
                        <TextBlock Text="Name: " FontWeight="Bold" Grid.Row="1" />
                        <TextBlock Text="{Binding Name}" Grid.Column="1" Grid.Row="1" />
                        <TextBlock Text="Birthday: " FontWeight="Bold" Grid.Row="2" />
                        <TextBlock Text="{Binding Birthday, StringFormat=d}" Grid.Column="1" Grid.Row="2" />

                    </Grid>
                </DockPanel>
            </DataTemplate>
        </DataGrid.RowDetailsTemplate>
    </DataGrid>
</Grid>
</Window>

using System;

```

```

using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples.DataGrid_control
{
    public partial class DataGridDetailsSample : Window
    {
        public DataGridDetailsSample()
        {
            InitializeComponent();
            List<User> users = new List<User>();
            users.Add(new User() { Id = 1, Name = "John Doe",
                Birthday = new DateTime(1971, 7, 23), ImageUrl = "http://www.wpf-
                tutorial.com/images/misc/john_doe.jpg" });
            users.Add(new User() { Id = 2, Name = "Jane Doe",
                Birthday = new DateTime(1974, 1, 17) });
            users.Add(new User() { Id = 3, Name = "Sammy Doe",
                Birthday = new DateTime(1991, 9, 2) });

            dgUsers.ItemsSource = users;
        }
    }

    public class User
    {
        public int Id { get; set; }

        public string Name { get; set; }

        public DateTime Birthday { get; set; }

        public string ImageUrl { get; set; }
    }
}

```

### 1.19.3.3. Summary

Being able to show details for a DataGrid row is extremely useful, and with the WPF DataGrid it's both easy and highly customizable, as you can see from the examples provided in this tutorial.

# 1.20. Styles

---

## 1.20.1. Introduction to WPF styles

If you come from the world of developing for the web, using [HTML](#) and [CSS](#), you'll quickly realize that XAML is much like HTML: Using tags, you define a structural layout of your application. You can even make your elements look a certain way, using inline properties like Foreground, FontSize and so on, just like you can locally style your HTML tags.

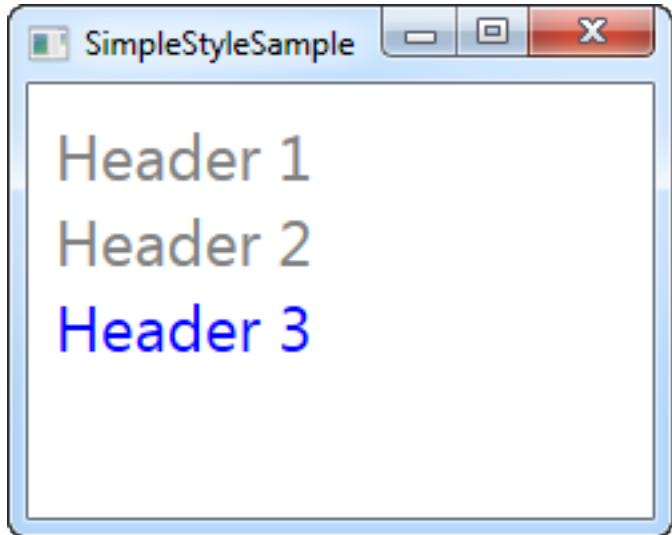
But what happens when you want to use the exact same font size and color on three different TextBlock controls? You can copy/paste the desired properties to each of them, but what happens when three controls becomes 50 controls, spread out over several windows? And what happens when you realize that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

WPF introduces styling, which is to XAML what CSS is to HTML. Using styles, you can group a set of properties and assign them to specific controls or all controls of a specific type, and just like in CSS, a style can inherit from another style.

### 1.20.1.1. Basic style example

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

```
<Window x:Class="WpfTutorialSamples.Styles.SimpleStyleSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SimpleStyleSample" Height="200" Width="250">
    <StackPanel Margin="10">
        <StackPanel.Resources>
            <Style TargetType="TextBlock">
                <Setter Property="Foreground" Value="Gray" />
                <Setter Property="FontSize" Value="24" />
            </Style>
        </StackPanel.Resources>
        <TextBlock>Header 1</TextBlock>
        <TextBlock>Header 2</TextBlock>
        <TextBlock Foreground="Blue">Header 3</TextBlock>
    </StackPanel>
</Window>
```



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

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

#### 1.20.1.2. Summary

WPF styles make it very easy to create a specific look and then use it for several controls, and while this first example was very local, I will show you how to create global styles in the next chapters.

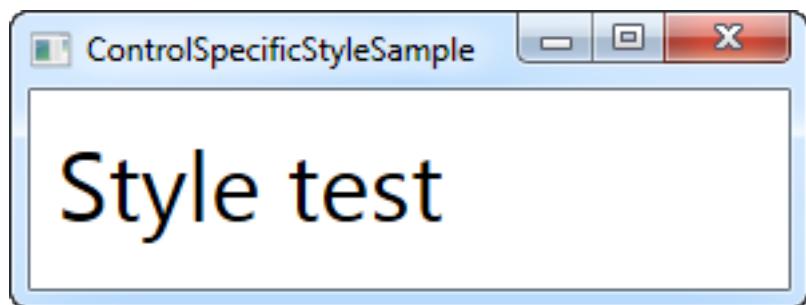
## 1.20.2. Using WPF styles

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

### 1.20.2.1. Local control specific style

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

```
<Window x:Class="WpfTutorialSamples.Styles.ControlSpecificStyleSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ControlSpecificStyleSample" Height="100" Width="300">
    <Grid Margin="10">
        <TextBlock Text="Style test">
            <TextBlock.Style>
                <Style>
                    <Setter Property="TextBlock.FontSize" Value
="36" />
                </Style>
            </TextBlock.Style>
        </TextBlock>
    </Grid>
</Window>
```

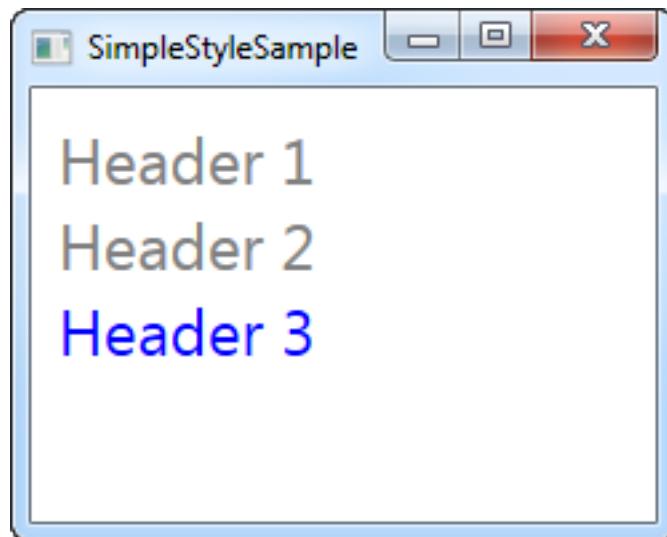


In this example, the style only affects this specific TextBlock control, so why bother? Well, in this case, it makes no sense at all. I could have replaced all that extra markup with a single FontSize property on the TextBlock control, but as we'll see later, styles can do a bit more than just set properties,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

### 1.20.2.2. Local child control style

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

```
<Window x:Class="WpfTutorialSamples.Styles.SimpleStyleSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SimpleStyleSample" Height="200" Width="250">
    <StackPanel Margin="10">
        <StackPanel.Resources>
            <Style TargetType="TextBlock">
                <Setter Property="Foreground" Value="Gray" />
                <Setter Property="FontSize" Value="24" />
            </Style>
        </StackPanel.Resources>
        <TextBlock>Header 1</TextBlock>
        <TextBlock>Header 2</TextBlock>
        <TextBlock Foreground="Blue">Header 3</TextBlock>
    </StackPanel>
</Window>
```

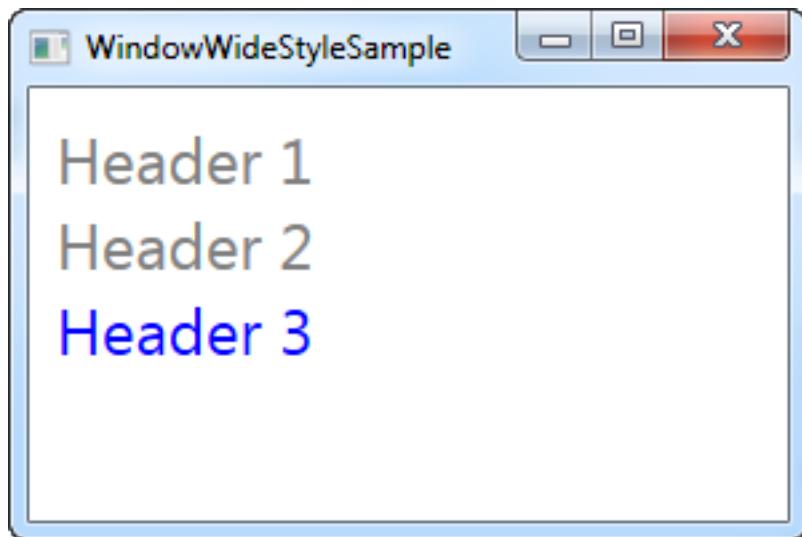


This is great for the more local styling needs. For instance, it would make perfect sense to do this in a dialog where you simply needed a set of controls to look the same, instead of setting the individual properties on each of them.

### 1.20.2.3. Window-wide styles

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

```
<Window x:Class="WpfTutorialSamples.Styles.WindowWideStyleSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WindowWideStyleSample" Height="200" Width="300">
    <Window.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Foreground" Value="Gray" />
            <Setter Property="FontSize" Value="24" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock>Header 1</TextBlock>
        <TextBlock>Header 2</TextBlock>
        <TextBlock Foreground="Blue">Header 3</TextBlock>
    </StackPanel>
</Window>
```



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

#### 1.20.2.4. Application-wide styles

If you want your styles to be used all over the application, across different windows, you can define it for the

entire application. This is done in the App.xaml file that Visual Studio has likely created for you, and it's done just like in the window-wide example:

## App.xaml

```
<Application x:Class="WpfTutorialSamples.App"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Styles/WindowWideStyleSample.xaml">
    <Application.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Foreground" Value="Gray" />
            <Setter Property="FontSize" Value="24" />
        </Style>
    </Application.Resources>
</Application>
```

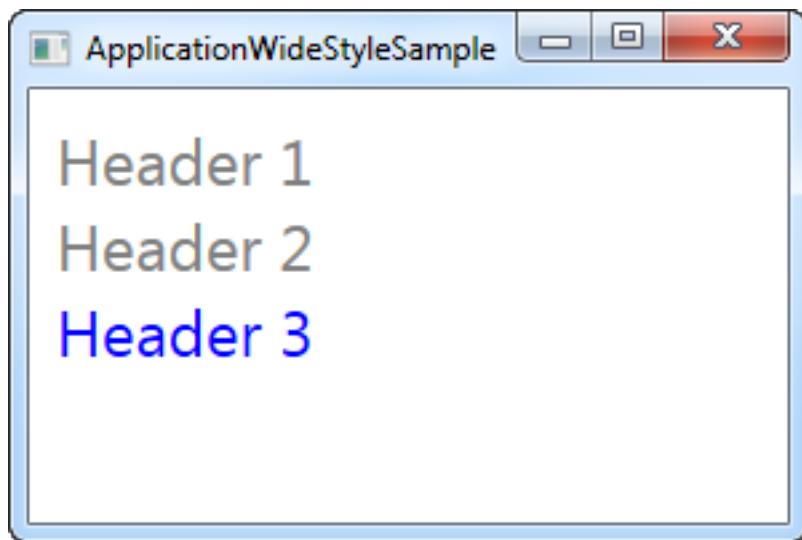
## Window

```
<Window x:Class="WpfTutorialSamples.Styles.WindowWideStyleSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ApplicationWideStyleSample" Height="200" Width="300">
    <StackPanel Margin="10">
        <TextBlock>Header 1</TextBlock>
        <TextBlock>Header 2</TextBlock>
        <TextBlock Foreground="Blue">Header 3</TextBlock>
    </StackPanel>
</Window>
```

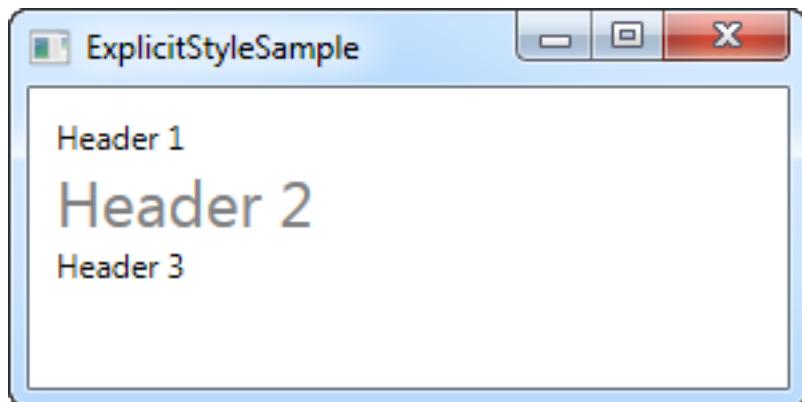
### 1.20.2.5. Explicitly using styles

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

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



```
<Window x:Class="WpfTutorialSamples.Styles.ExplicitStyleSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ExplicitStyleSample" Height="150" Width="300">
    <Window.Resources>
        <Style x:Key="HeaderStyle" TargetType="TextBlock">
            <Setter Property="Foreground" Value="Gray" />
            <Setter Property="FontSize" Value="24" />
        </Style>
    </Window.Resources>
    <StackPanel Margin="10">
        <TextBlock>Header 1</TextBlock>
        <TextBlock Style="{StaticResource HeaderStyle}">Header 2</TextBlock>
        <TextBlock>Header 3</TextBlock>
    </StackPanel>
</Window>
```



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

#### 1.20.2.6. Summary

WPF styling allows you to easily re-use a certain look for your controls all over the application. Using the x:Key property, you can decide whether a style should be explicitly referenced to take effect, or if it should target all controls no matter what.

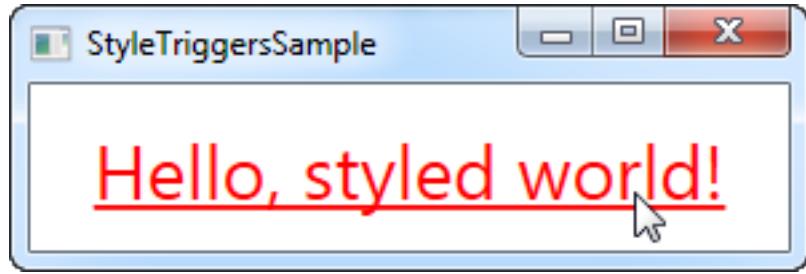
## 1.20.3. Trigger, DataTrigger & EventTrigger

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

### 1.20.3.1. Property trigger

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

```
<Window x:Class="WpfTutorialSamples.Styles.StyleTriggersSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StyleTriggersSample" Height="100" Width="300">
    <Grid>
        <TextBlock Text="Hello, styled world!" FontSize="28"
HorizontalAlignment="Center" VerticalAlignment="Center">
            <TextBlock.Style>
                <Style TargetType="TextBlock">
                    <Setter Property="Foreground" Value="Blue"></
Setter>
                    <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value
="True">
                            <Setter Property="Foreground" Value
="Red" />
                            <Setter Property="TextDecorations"
Value="Underline" />
                        </Trigger>
                    </Style.Triggers>
                </Style>
            </TextBlock.Style>
        </TextBlock>
    </Grid>
</Window>
```



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

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

### 1.20.3.2. Data triggers

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

```
<Window x:Class="WpfTutorialSamples.Styles.StyleDataTriggerSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StyleDataTriggerSample" Height="200" Width="200">
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" >
        <CheckBox Name="cbSample" Content="Hello, world?" />
        <TextBlock HorizontalAlignment="Center" Margin="0,20,0,0" >
            <TextBlock.Style>
                <Style TargetType="TextBlock">
                    <Setter Property="Text" Value="No" />
                    <Setter Property="Foreground" Value="Red" />
                    <Style.Triggers>
                        <DataTrigger Binding="{Binding ElementName=cbSample, Path=.IsChecked}" Value="True">
                            <Setter Property="Text" Value="Yes!" />
                            <Setter Property="Foreground" Value="Green" />
                        
```

```

        </DataTrigger>
    </Style.Triggers>
</Style>
</TextBlock.Style>
</TextBlock>
</StackPanel>
</Window>

```



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

#### 1.20.3.3. Event triggers

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

```

<Window x:Class="WpfTutorialSamples.Styles.StyleEventTriggerSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyleEventTriggerSample" Height="100" Width="300">
    <Grid>
        <TextBlock Name="lblStyled" Text="Hello, styled world!"
        FontSize="18" HorizontalAlignment="Center" VerticalAlignment="Center">
            <TextBlock.Style>

```

```

<Style TargetType="TextBlock">
    <Style.Triggers>
        <EventTrigger RoutedEvent="MouseEnter">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
                            Duration="0:0:0.300" Storyboard.TargetProperty="FontSize" To="28" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
        <EventTrigger RoutedEvent="MouseLeave">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
                            Duration="0:0:0.800" Storyboard.TargetProperty="FontSize" To="18" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </Style.Triggers>
</Style>
</TextBlock.Style>
</TextBlock>
</Grid>
</Window>

```



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

cool.

#### 1.20.3.4. Summary

WPF styles make it easy to get a consistent look, and with triggers, this look becomes dynamic. Styles are great in your application, but they're even better when used in control templates etc. You can read more about that elsewhere in this tutorial.

In the next article, we'll look at multi triggers, which allow us to apply styles based on multiple properties.

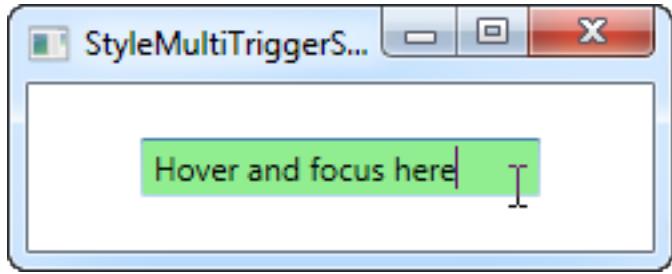
## 1.20.4. WPF MultiTrigger and MultiDataTrigger

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

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

### 1.20.4.1. MultiTrigger

```
<Window x:Class="WpfTutorialSamples.Styles.StyleMultiTriggerSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StyleMultiTriggerSample" Height="100" Width="250">
    <Grid>
        <TextBox VerticalAlignment="Center" HorizontalAlignment
        ="Center" Text="Hover and focus here" Width="150">
            <TextBox.Style>
                <Style TargetType="TextBox">
                    <Style.Triggers>
                        <MultiTrigger>
                            <MultiTrigger.Conditions>
                                <Condition Property
                                ="IsKeyboardFocused" Value="True" />
                                <Condition Property
                                ="IsMouseOver" Value="True" />
                            </MultiTrigger.Conditions>
                            <MultiTrigger.Setters>
                                <Setter Property="Background"
Value="LightGreen" />
                            </MultiTrigger.Setters>
                        </MultiTrigger>
                    </Style.Triggers>
                </Style>
            </TextBox.Style>
        </TextBox>
    </Grid>
</Window>
```



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

#### 1.20.4.2. MultiDataTrigger

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

```
<Window x:Class="WpfTutorialSamples.Styles.StyleMultiDataTriggerSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StyleMultiDataTriggerSample" Height="150" Width="200">
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <CheckBox Name="cbSampleYes" Content="Yes" />
        <CheckBox Name="cbSampleSure" Content="I'm sure" />
        <TextBlock HorizontalAlignment="Center" Margin="0,20,0,0"
FontSize="28">
            <TextBlock.Style>
                <Style TargetType="TextBlock">
                    <Setter Property="Text" Value="Unverified" />
                    <Setter Property="Foreground" Value="Red" />
                    <Style.Triggers>
                        <MultiDataTrigger>
                            <MultiDataTrigger.Conditions>
                                <Condition Binding="{Binding ElementName=cbSampleYes, Path=IsChecked}" Value="True" />
                                <Condition Binding="{Binding ElementName=cbSampleSure, Path=IsChecked}" Value="True" />
                            </MultiDataTrigger.Conditions>
                            <Setter Property="Text" Value="Verfied" />
                            <Setter Property="Foreground" Value="Green" />
                        </MultiDataTrigger>
                    <Style.Triggers>
                <Style>
            </TextBlock>
    </StackPanel>
</Window>
```

```

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

</MultiDataTrigger>

</Style.Triggers>

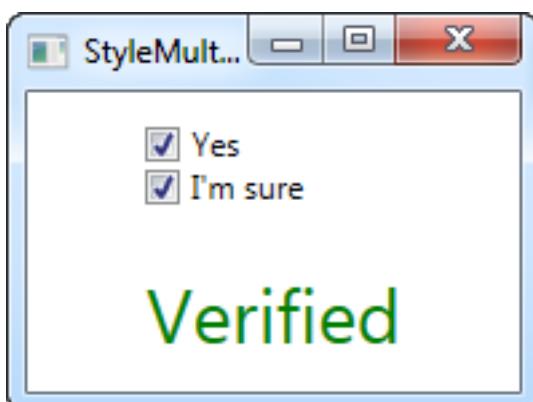
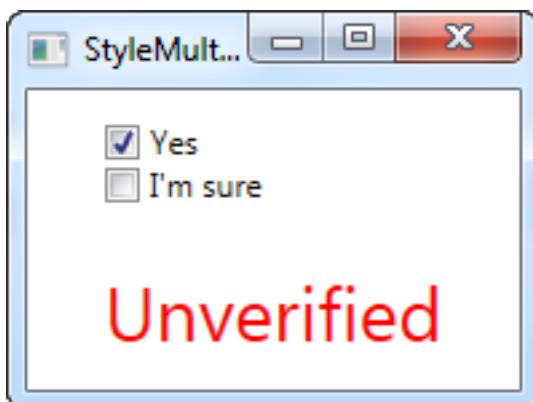
</Style>

</TextBlock.Style>

</TextBlock>

</StackPanel>

</Window>



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

#### 1.20.4.3. Summary

As you can see, multi triggers are pretty much just as easy to use as regular triggers and they can be extremely useful, especially when developing your own controls.

## 1.20.5. Trigger animations

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

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

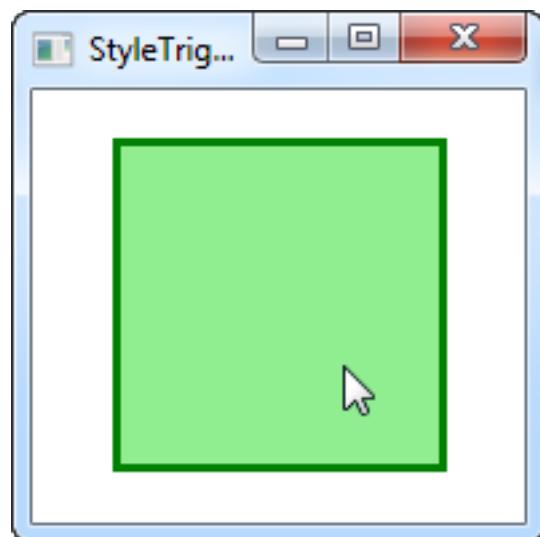
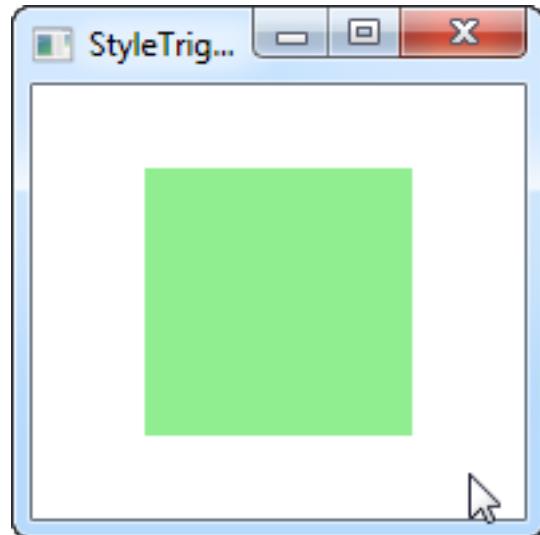
```
<Window x:Class="WpfTutorialSamples.Styles.StyleTriggerEnterExitActions"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StyleTriggerEnterExitActions" Height="200" Width="200"
        UseLayoutRounding="True">
    <Grid>
        <Border Background="LightGreen" Width="100" Height="100"
        BorderBrush="Green">
            <Border.Style>
                <Style TargetType="Border">
                    <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value
="True">
                            <Trigger.EnterActions>
                                <BeginStoryboard>
                                    <Storyboard>
                                        <ThicknessAnimation
Duration="0:0:0.400" To="3" Storyboard.TargetProperty="BorderThickness"
/>
                                        <DoubleAnimation
Duration="0:0:0.300" To="125" Storyboard.TargetProperty="Height" />
                                        <DoubleAnimation
Duration="0:0:0.300" To="125" Storyboard.TargetProperty="Width" />
                                    </Storyboard>
                                </BeginStoryboard>
                            </Trigger.EnterActions>
                            <Trigger.ExitActions>
                                <BeginStoryboard>
                                    <Storyboard>
                                        <ThicknessAnimation
Duration="0:0:0.250" To="0" Storyboard.TargetProperty="BorderThickness"
/>

```

```

        <DoubleAnimation
Duration="0:0:0.150" To="100" Storyboard.TargetProperty="Height" />
        <DoubleAnimation
Duration="0:0:0.150" To="100" Storyboard.TargetProperty="Width" />
    </Storyboard>
</BeginStoryboard>
</Trigger.ExitActions>
</Trigger>
</Style.Triggers>
</Style>
</Border.Style>
</Border>
</Grid>
</Window>

```



In this example, we have a green square. It has a trigger that fires once the mouse is over, in which case it

fires of several animations, all defined in the **EnterActions** part of the trigger. In there, we animate the thickness of the border from its default 0 to a thickness of 3, and then we animate the width and height from 100 to 125. This all happens simultaneously, because they are a part of the same **Storyboard**, and even at slightly different speeds, since we have full control of how long each animation should run.

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

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

#### 1.20.5.1. Summary

Using animations with style triggers is very easy, and while we haven't fully explored all you can do with WPF animations yet, the example used above should give you an idea on just how flexible both animations and styles are.

# 1.21. Audio & Video

---

## 1.21.1. Playing audio

WPF comes with excellent built-in audio and video support, as you'll see in the next couple of chapters of this tutorial. In this particular article, we'll be discussing the ability to play audio, coming from actual audio files, e.g. in the MP3 format, but first, let's have a look at couple of simpler approaches.

### 1.21.1.1. System sounds and the SoundPlayer

WPF has a class called **SoundPlayer**, which will play audio content based on the WAV format for you. WAV is not a very widely used format today, mainly because it's uncompressed and therefore takes up a LOT of space.

So while the SoundPlayer class is simple to use, it's not terribly useful. Instead, we'll be focusing on the **MediaPlayer** and **MediaElement** classes, which allows the playback of MP3 files, but first, let's have a look at the simplest way of playing a sound in your WPF application - the SystemSounds class.

The SystemSounds class offers several different sounds, which corresponds to the sound defined for this event by the user in Windows, like Exclamation and Question. You can piggyback on these sounds and settings and play them with a single line of code:

```
SystemSounds.Beep.Play();
```

Here's a complete example, where we use all of the currently available sounds:

```
<Window x:Class="WpfTutorialSamples.Audio_and_Video.SystemSoundsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SystemSoundsSample" Height="200" Width="150">
    <StackPanel Margin="10" HorizontalAlignment="Center"
    VerticalAlignment="Center">
        <Button Name="btnAsterisk" Click="btnAsterisk_Click">Asterisk
    </Button>
        <Button Name="btnBeep" Margin="0,5" Click="btnBeep_Click">Beep
    </Button>
        <Button Name="btnExclamation" Click="btnExclamation_Click">Exclamation</Button>
        <Button Name="btnHand" Margin="0,5" Click="btnHand_Click">Hand
    </Button>
        <Button Name="btnQuestion" Click="btnQuestion_Click">Question
```

```
</Button>
</StackPanel>
</Window>

using System;
using System.Media;
using System.Windows;

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class SystemSoundsSample : Window
    {
        public SystemSoundsSample()
        {
            InitializeComponent();
        }

        private void btnAsterisk_Click(object sender, RoutedEventArgs e)
        {
            SystemSounds.Asterisk.Play();
        }

        private void btnBeep_Click(object sender, RoutedEventArgs e)
        {
            SystemSounds.Beep.Play();
        }

        private void btnExclamation_Click(object sender,
RoutedEventArgs e)
        {
            SystemSounds.Exclamation.Play();
        }

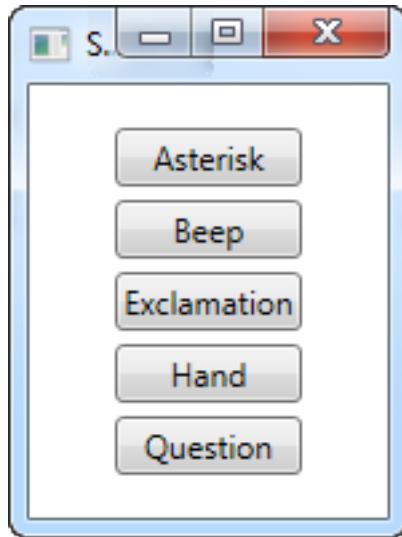
        private void btnHand_Click(object sender, RoutedEventArgs e)
        {
            SystemSounds.Hand.Play();
        }

        private void btnQuestion_Click(object sender, RoutedEventArgs
```

```

        e)
    {
        SystemSounds.Question.Play();
    }
}

```



There are of course several limitations to using this approach. First of all, you only get access to these five sounds, and second of all, the user may have disabled them in Windows, in which case the expected sound will be replaced with silence. On the other hand, if you only want to use these sounds the same way that Windows does, it makes it extremely easy to produce a sound for warnings, questions etc. In that case, it's a good thing that your application will respect the user's choice of silence.

#### 1.21.1.2. The MediaPlayer class

The MediaPlayer class uses Windows Media Player technology to play both audio and video in several modern formats, e.g. MP3 and MPEG. In this article, we'll be using it for playing just audio, and then focus on video in the next article.

Playing an MP3 file with the MediaPlayer class is very simple, as we'll see in this next example:

```

<Window x:Class
    ="WpfTutorialSamples.Audio_and_Video.MediaPlayerAudioSample"
    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MediaPlayerAudioSample" Height="100" Width="200">
    <Grid VerticalAlignment="Center" HorizontalAlignment="Center">
        <Button Name="btnOpenAudioFile" Click="btnOpenAudioFile_Click">
    Open Audio file</Button>

```

```

    </Grid>
</Window>

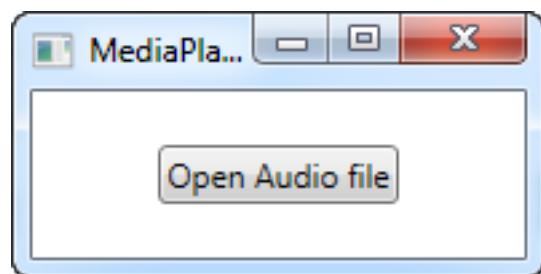
using System;
using System.Windows;
using System.Windows.Media;
using Microsoft.Win32;

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class MediaPlayerAudioSample : Window
    {
        private MediaPlayer mediaPlayer = new MediaPlayer();

        public MediaPlayerAudioSample()
        {
            InitializeComponent();
        }

        private void btnOpenAudioFile_Click(object sender,
RoutedEventArgs e)
        {
            OpenFileDialog openFileDialog = new OpenFileDialog();
            openFileDialog.Filter = "MP3 files (*.mp3)|*.mp3|All
files (*.*)|*.*";
            if(openFileDialog.ShowDialog() == true)
            {
                mediaPlayer.Open(new Uri(openFileDialog.FileName));
                mediaPlayer.Play();
            }
        }
    }
}

```



In this example, we just have a single button, which will show an OpenFileDialog and let you select an MP3 file. Once that is done, it will use the already created MediaPlayer instance to open and play this file. Notice that the MediaPlayer object is created outside of the event handler. This makes sure that the object is not prematurely garbage collected because it goes out of scope once the event handler is done, which would result in the playback stopping.

Please also notice that no exception handling is done for this example, as usual to keep the example as compact as possible, but in this case also because the Open() and Play() methods actually doesn't throw any exceptions. Instead, you can use the MediaOpened and MediaFailed events to act when things go right or wrong.

#### 1.21.1.3. Controlling the MediaPlayer

In our first MediaPlayer example, we just opened and automatically started playing a file, without giving the user a chance to control the playback process, but obviously, the MediaPlayer control offers you full control of playback. Here's an example showing you the most important functions:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.MediaPlayerAudioControlSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MediaPlayerAudioControlSample" Height="120" Width="300"
>
    <StackPanel Margin="10">
        <Label Name="lblStatus" Content="Not playing..." HorizontalContentAlignment="Center" Margin="5" />
        <WrapPanel HorizontalAlignment="Center">
            <Button Name="btnPlay" Click="btnPlay_Click">Play</Button>
            <Button Name="btnPause" Margin="5,0" Click="btnPause_Click">Pause</Button>
            <Button Name="btnStop" Click="btnStop_Click">Stop</Button>
        </WrapPanel>
    </StackPanel>
</Window>

using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Threading;
```

```

using Microsoft.Win32;

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class MediaPlayerAudioControlSample : Window
    {
        private MediaPlayer mediaPlayer = new MediaPlayer();

        public MediaPlayerAudioControlSample()
        {
            InitializeComponent();

            OpenFileDialog openFileDialog = new OpenFileDialog();
            openFileDialog.Filter = "MP3 files (*.mp3)|*.mp3|All files (*.*)|*.*";
            if(openFileDialog.ShowDialog() == true)
                mediaPlayer.Open(new Uri(openFileDialog.FileName));

            DispatcherTimer timer = new DispatcherTimer();
            timer.Interval = TimeSpan.FromSeconds(1);
            timer.Tick += timer_Tick;
            timer.Start();
        }

        void timer_Tick(object sender, EventArgs e)
        {
            if(mediaPlayer.Source != null)
                lblStatus.Content = String.Format("{0} / {1}",
mediaPlayer.Position.ToString(@"mm\:ss"),
mediaPlayer.NaturalDuration.TimeSpan.ToString(@"mm\:ss"));
            else
                lblStatus.Content = "No file selected...";
        }

        private void btnPlay_Click(object sender, RoutedEventArgs e)
        {
            mediaPlayer.Play();
        }

        private void btnPause_Click(object sender, RoutedEventArgs e)

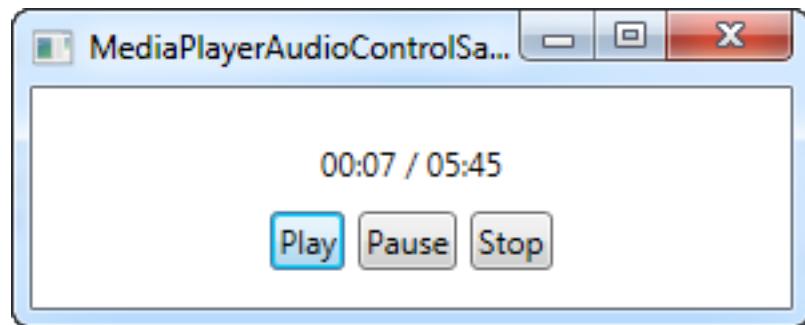
```

```

    {
        mediaPlayer.Pause();
    }

    private void btnStop_Click(object sender, RoutedEventArgs e)
    {
        mediaPlayer.Stop();
    }
}

```



In this example, we have expanded our player a bit, so that it now contains a Play, Pause and Stop button, as well as a label for showing the current playback status. The MP3 file to be played is loaded the same way, but we do it as soon as the application starts, to keep the example simple.

Right after the MP3 is loaded, we start a timer, which ticks every second. We use this event to update the status label, which will show the current progress as well as the entire length of the loaded file.

The three buttons each simply call a corresponding method on the MediaPlayer object - Play, Pause and Stop.

#### 1.21.1.4. Summary

There are several more options that you can let your user control, but I want to save that for when we have talked about the video aspects of the MediaPlayer class - at that point, I'll do a more complete example of a media player capable of playing both audio and video files, with more options.

## 1.21.2. Playing video

In the previous article, we used the **MediaPlayer** class to play an MP3 file, but the cool part about the **MediaPlayer** class is that it can work with video files as well. However, since a video actually needs to be displayed somewhere in the interface, as opposed to an audio file, we need a wrapper element to visually represent the **MediaPlayer** instance. This is where the **MediaElement** comes into play.

### 1.21.2.1. The MediaElement

The **MediaElement** acts as a wrapper around **MediaPlayer**, so that you can display video content at a given place in your application, and because of that, it can play both audio and video files, although the visual representation doesn't really matter in dealing with audio files.

I want to show you just how easy you can show video content in your WPF application, so here's a bare minimum example:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.MediaPlayerVideoSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MediaPlayerVideoSample" Height="300" Width="300">
    <Grid>
        <MediaElement Source
        ="http://hubblesource.stsci.edu/sources/video/clips/details/images/hst_1
        .mpg" />
    </Grid>
</Window>
```

And that's it - a single line of XAML inside your window and you're displaying video (this specific video is about the Hubble Space Telescope - more information can be found at [this website](#)) in your WPF application.

### 1.21.2.2. Dealing with video size

Our examples in this article so far has just used the same size for the **MediaElement**, not taking the dimensions of the video into consideration. This is possible because the **MediaElement** can stretch/shrink the content to fit the available width/height and will do so by default. This is caused by the **Stretch** property, which is set to **Uniform** by default, meaning that the video will be stretched, while respecting the aspect ratio.

If your window is larger than your video, this might work just fine, but perhaps you don't want any stretching to occur? Or perhaps you want the window to adjust to fit your video's dimensions, instead of the other way around?



The first thing you need to do is to turn off stretching by setting the **Stretch** property to **None**. This will ensure that the video is rendered in its natural size. Now if you want the window to adjust to that, it's actually quite simple - just use the **ResizeToContent** property on the Window to accomplish this. Here's a full example:

```
<Window x:Class
    ="WpfTutorialSamples.Audio_and_Video.MediaPlayerVideoSizeSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MediaPlayerVideoSizeSample" Height="500" Width="500"
SizeToContent="WidthAndHeight">
    <Grid>
        <MediaElement Source
    ="http://hubblesource.stsci.edu/sources/video/clips/details/images/hst_1
.mpg" Name="mePlayer" Stretch="None" />
    </Grid>
</Window>
```

As you can see, despite the initial values of 500 for the Width and Height properties on the Window, the size is adjusted (down, in this case) to match the resolution of the video.

Please notice that this might cause the window to have a size of zero (only the title bar and borders will be



visible) during startup, while the video is loaded. To prevent this, you can set the **MinWidth** and **MinHeight** properties on the **Window** to something that suits your needs.

#### 1.21.2.3. Controlling the MediaElement/MediaPlayer

As you can see if you run our previous examples, the video starts playing as soon as the player has buffered enough data, but you can change this behavior by using the **LoadedBehavior** property. We'll do that in the next example, where we'll also add a couple of buttons to control the playback:

```
<Window x:Class
    ="WpfTutorialSamples.Audio_and_Video.MediaPlayerVideoControlSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MediaPlayerVideoControlSample" Height="300" Width="300"
>
    <Grid Margin="10">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <MediaElement Source
    ="http://hubblesource.stsci.edu/sources/video/clips/details/images/hst_1
.mpvg" LoadedBehavior="Manual" Name="mePlayer" />
        <StackPanel Grid.Row="1">
            <Label Name="lblStatus" Content="Not playing...">
```

```

HorizontalContentAlignment="Center" Margin="5" />
    <WrapPanel HorizontalAlignment="Center">
        <Button Name="btnPlay" Click="btnPlay_Click">Play</
Button>
        <Button Name="btnPause" Margin="5,0" Click
="btnPause_Click">Pause</Button>
        <Button Name="btnStop" Click="btnStop_Click">Stop</
Button>
    </WrapPanel>
</StackPanel>
</Grid>
</Window>

using System;
using System.Windows;
using System.Windows.Threading;

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class MediaPlayerVideoControlSample : Window
    {
        public MediaPlayerVideoControlSample()
        {
            InitializeComponent();

            DispatcherTimer timer = new DispatcherTimer();
            timer.Interval = TimeSpan.FromSeconds(1);
            timer.Tick += timer_Tick;
            timer.Start();
        }

        void timer_Tick(object sender, EventArgs e)
        {
            if(mePlayer.Source != null)
            {
                if(mePlayer.NaturalDuration.HasTimeSpan)
                    lblStatus.Content = String.Format("{0} / {1}",
mePlayer.Position.ToString(@"mm\:ss"),
mePlayer.NaturalDuration.TimeSpan.ToString(@"mm\:ss"));
            }
        }
    }
}

```

```

        else
            lblStatus.Content = "No file selected...";

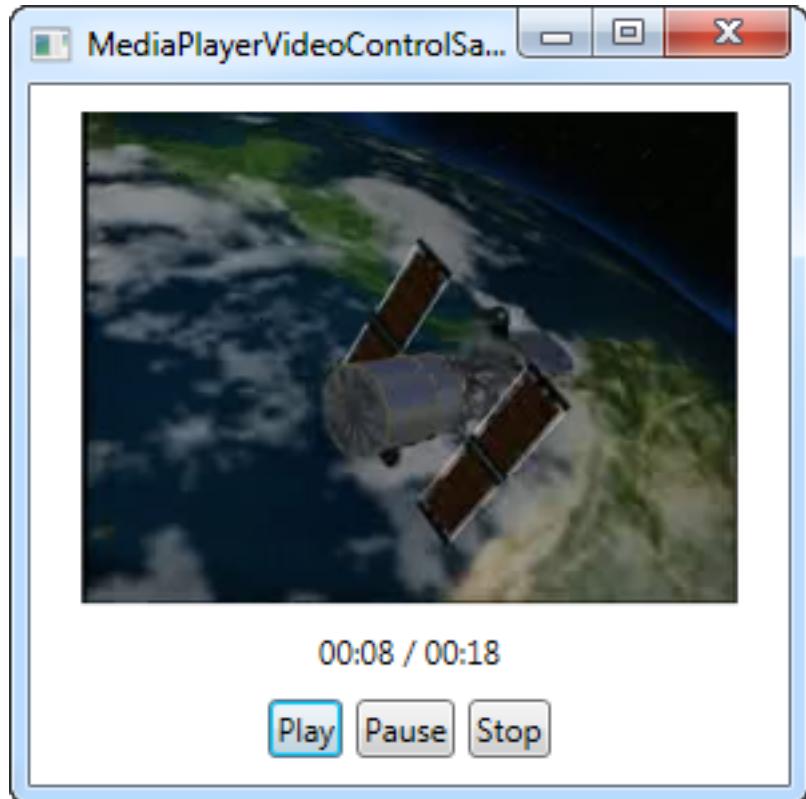
    }

private void btnPlay_Click(object sender, RoutedEventArgs e)
{
    mePlayer.Play();
}

private void btnPause_Click(object sender, RoutedEventArgs e)
{
    mePlayer.Pause();
}

private void btnStop_Click(object sender, RoutedEventArgs e)
{
    mePlayer.Stop();
}
}

```



This example is much like the one we did in the previous article for audio, just for video in this case. We have a bottom area with a set of buttons for controlling the playback, a label for showing the status, and

then a **MediaElement** control in the top area to show the actual video.

Upon application start, we create and start a timer, which ticks every second. We use this event to update the status label, which will show the current progress as well as the entire length of the loaded file, as seen on the screenshot.

The three buttons each simply call a corresponding method on the MediaElement control - Play(), Pause() and Stop().

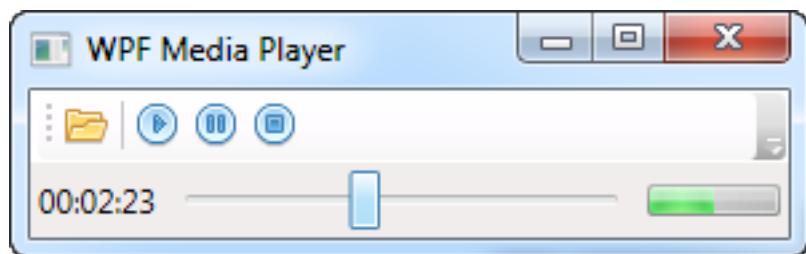
#### 1.21.2.4. Summary

Once again it's clear how easy WPF makes even advanced things like playing a video. So far, we've worked with some basic examples, but in the next chapter, I'm going to combine all the stuff we've learned about audio and video playback into a single, media player with a lot more functionality than we've seen so far. Read on!

### 1.21.3. How-to: Creating a complete Audio/Video player

As a conclusion to the last chapters on playing audio and video, I decided to create a more complete sample, where we take advantage of the fact that the MediaPlayer/MediaElement classes can handle both audio and video.

I will take the concepts used in the articles about playing audio and video and combine them with several controls which we have already discussed previously in this article, and turn it all into a WPF Media Player. The result will look something like this:



But that's just when it plays audio/MP3 files. Once a video is loaded, the interface automatically expands to show the video content inside the window:



Let me tell you a bit about how this thing was built. In the end, you can of course see the entire source code, ready for you to play with.

### 1.21.3.1. The interface

The interface has been split into three vertical areas: The top, where the toolbar is located, the middle, where the video (if a video file is loaded) is shown, and the bottom, where we find a status bar, complete with a time status, a **Slider** for seeing and controlling progress and a **ProgressBar** for showing the volume. All of the controls used here have been explained previously in the tutorial, so we won't focus too much on that.

Notice the use of WPF commands, instead of click events for the buttons. This allows us to easily re-use the functionality in case we want to add e.g. a main menu or a context menu with some of the same functionality. It also makes it easier for us to toggle the functionality on and off, depending on the current state of the player.

Also notice that we have set the `MediaElement Stretch` property to **None**, and the `Window SizeToContentMode` to **WidthAndHeight**. This is what keeps the window to the minimum size needed to show the interface as well as the video, if one is playing.

For showing the Volume, we've used a `ProgressBar` control in the lower, right corner. This doesn't currently let the user control the volume, but merely reflects the **Volume** property on the `MediaElement` control, through a classic data binding. We've implemented a small but neat trick for letting the user control the volume anyway though - more on that below.

### 1.21.3.2. The code

In Code-behind, we re-use several techniques already used in our previous examples. For instance, we initiate a `DispatcherTimer` and let it tick every second, to show the current playback progress in the interface. In the `Tick` event of the timer, we update **Slider** control, by setting **Minimum**, **Maximum** and current **Value** according to the file being played, and by hooking up to the **ValueChanged** event on the slider, we use that to update the label showing the current playback progress in hours, minutes and seconds.

The `Slider` control also allows the user to skip to another part of the file, simply by dragging the "thumb" to another location. We handle this by implementing events for **DragStarted** and **DragCompleted** - the first one to set a variable (`userIsDraggingSlider`) that tells the timer not to update the `Slider` while we drag, and the second one to skip to the designated part when the user releases the mouse button.

There are **CanExecute** and **Executed** handlers for the four commands we use and especially the ones for Pause and Stop are interesting. Since we can't get a current state from the `MediaElement` control, we have to keep track of the current state ourselves. This is done with a local variable called **mediaPlayer.isPlaying**, which we regularly check to see if the Pause and Stop buttons should be enabled.

The last little detail you should notice is the **Grid\_MouseWheel** event. The main Grid covers the entire window, so by subscribing to this event, we get notified when the user scrolls the wheel. When that happens, as a little gimmick, we turn the volume up or down, depending on the direction (we get that by looking at the `Delta` property, which is negative when scrolling down and positive when scrolling up). This is immediately reflected in the user interface, where a **ProgressBar** control is bound to the `Volume` property

of the MediaElement.

### 1.21.3.3. The complete source code

With all the theory behind the example described, here's the complete source code:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.AudioVideoPlayerCompleteSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF Media Player" Height="300" Width="300"
        MinWidth="300" SizeToContent="WidthAndHeight">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Open" CanExecute
        ="Open_CanExecute" Executed="Open_Executed" />
        <CommandBinding Command="MediaCommands.Play" CanExecute
        ="Play_CanExecute" Executed="Play_Executed" />
        <CommandBinding Command="MediaCommands.Pause" CanExecute
        ="Pause_CanExecute" Executed="Pause_Executed" />
        <CommandBinding Command="MediaCommands.Stop" CanExecute
        ="Stop_CanExecute" Executed="Stop_Executed" />
    </Window.CommandBindings>
    <Grid MouseWheel="Grid_MouseWheel">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <ToolBar>
            <Button Command="ApplicationCommands.Open">
                <Image Source
                    =" /WpfTutorialSamples;component/Images/folder.png" />
            </Button>
            <Separator />
            <Button Command="MediaCommands.Play">
                <Image Source
                    =" /WpfTutorialSamples;component/Images/control_play_blue.png" />
            </Button>
            <Button Command="MediaCommands.Pause">
                <Image Source
```

```

        =" /WpfTutorialSamples;component/Images/control_pause_blue.png" />
    </Button>
    <Button Command="MediaCommands.Stop">
        <Image Source
        =" /WpfTutorialSamples;component/Images/control_stop_blue.png" />
    </Button>
</ToolBar>

<MediaElement Name="mePlayer" Grid.Row="1" LoadedBehavior
="Manual" Stretch="None" />

<StatusBar Grid.Row="2">
    <StatusBar.ItemsPanel>
        <ItemsPanelTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="Auto" />
                </Grid.ColumnDefinitions>
            </Grid>
        </ItemsPanelTemplate>
    </StatusBar.ItemsPanel>
    <StatusBarItem>
        <TextBlock Name="lblProgressStatus">00:00:00</
TextBlock>
    </StatusBarItem>
    <StatusBarItem Grid.Column="1" HorizontalContentAlignment
="Stretch">
        <Slider Name="sliProgress" Thumb.DragStarted
="sliProgress_DragStarted" Thumb.DragCompleted
="sliProgress_DragCompleted" ValueChanged="sliProgress_ValueChanged" />
    </StatusBarItem>
    <StatusBarItem Grid.Column="2">
        <ProgressBar Name="pbVolume" Width="50" Height="12"
Maximum="1" Value="{Binding ElementName=mePlayer, Path=Volume}" />
    </StatusBarItem>
</StatusBar>
</Grid>
</Window>

```

```

using System;
using System.Windows;
using System.Windows.Controls.Primitives;
using System.Windows.Input;
using System.Windows.Threading;
using Microsoft.Win32;

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class AudioVideoPlayerCompleteSample : Window
    {
        private bool mediaPlayerIsPlaying = false;
        private bool userIsDraggingSlider = false;

        public AudioVideoPlayerCompleteSample()
        {
            InitializeComponent();

            DispatcherTimer timer = new DispatcherTimer();
            timer.Interval = TimeSpan.FromSeconds(1);
            timer.Tick += timer_Tick;
            timer.Start();
        }

        private void timer_Tick(object sender, EventArgs e)
        {
            if((mePlayer.Source != null) &&
(mePlayer.NaturalDuration.HasTimeSpan) && (!userIsDraggingSlider))
            {
                sliProgress.Minimum = 0;
                sliProgress.Maximum =
mePlayer.NaturalDuration.TimeSpan.TotalSeconds;
                sliProgress.Value = mePlayer.Position.TotalSeconds;
            }
        }

        private void Open_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
    }
}

```

```

        }

    private void Open_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Media files
(*.mp3;*.mpg;*.mpeg)|*.mp3;*.mpg;*.mpeg|All files (*.*)|*.*";
    if(openFileDialog.ShowDialog() == true)
        mePlayer.Source = new Uri(openFileDialog.FileName);
}

private void Play_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
    e.CanExecute = (mePlayer != null) && (mePlayer.Source !=
null);
}

private void Play_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    mePlayer.Play();
    mediaPlayerIsPlaying = true;
}

private void Pause_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
    e.CanExecute = mediaPlayerIsPlaying;
}

private void Pause_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    mePlayer.Pause();
}

private void Stop_CanExecute(object sender,
CanExecuteRoutedEventArgs e)

```

```

    {
        e.CanExecute = mediaPlayerIsPlaying;
    }

    private void Stop_Executed(object sender,
ExecutedRoutedEventArgs e)
{
    mePlayer.Stop();
    mediaPlayerIsPlaying = false;
}

private void sliProgress_DragStarted(object sender,
DragStartedEventArgs e)
{
    userIsDraggingSlider = true;
}

private void sliProgress_DragCompleted(object sender,
DragCompletedEventArgs e)
{
    userIsDraggingSlider = false;
    mePlayer.Position =
TimeSpan.FromSeconds(sliProgress.Value);
}

private void sliProgress_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
{
    lblProgressStatus.Text =
TimeSpan.FromSeconds(sliProgress.Value).ToString(@"hh\:mm\:ss");
}

private void Grid_MouseWheel(object sender,
MouseWheelEventArgs e)
{
    mePlayer.Volume += (e.Delta > 0) ? 0.1 : -0.1;
}

}
}

```

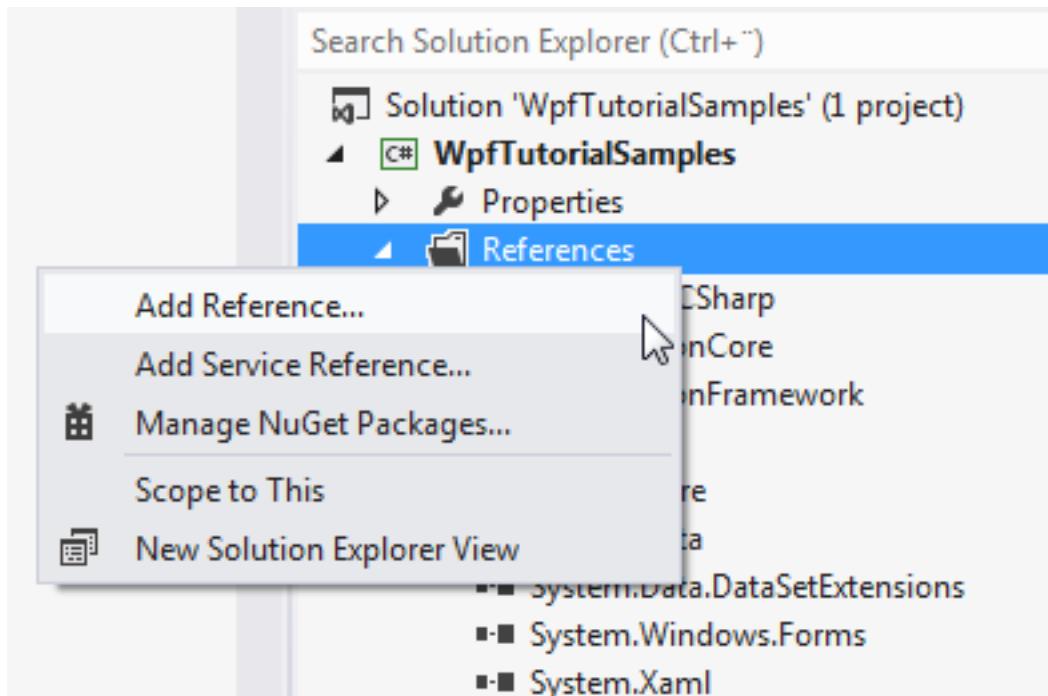
#### 1.21.3.4. Summary

The code listing might look a bit overwhelming, but as you can see, there's a lot of repetition in it. If you take that out of the picture, you will soon realize that creating a pretty capable media player in WPF is really not that hard! Feel free to expand on this example for your own projects - how about implementing a playlist feature?

## 1.21.4. Speech synthesis (making WPF talk)

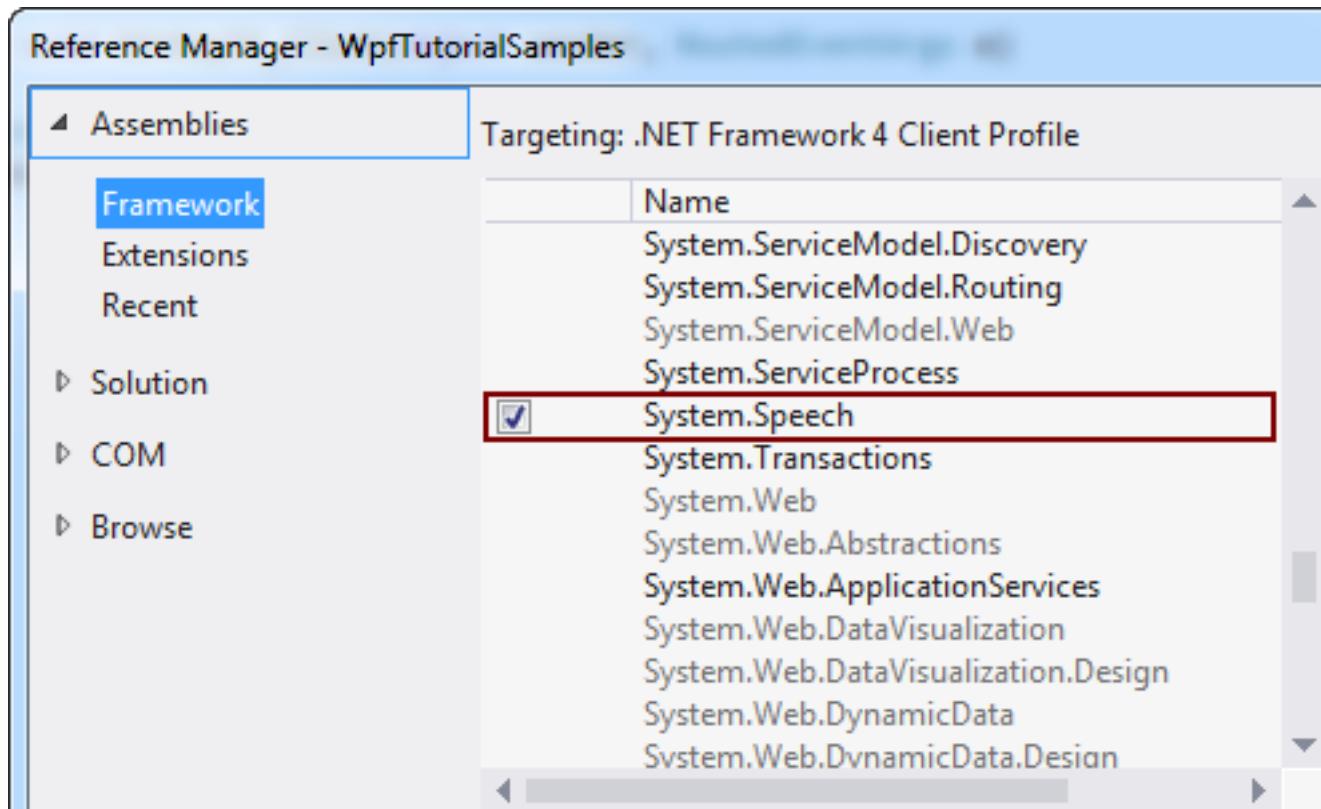
In the System.Speech assembly, Microsoft has added something really cool: Speech Synthesis, the ability to transform text into spoken words, and Speech Recognition, the ability to translate spoken words into text. We'll be focusing on the speech synthesis in this article, and then get into speech recognition in the next one.

To transform text into spoken words, we'll be using the **SpeechSynthesizer** class. This class resides in the System.Speech assembly, which we'll need to add to use it in our application. Depending on which version of Visual Studio you use, the process looks something like this:



With the appropriate assembly added, we can now use the **SpeechSynthesizer** class from the **System.Speech.Synthesis** namespace. With that in place, we'll kick off with yet another very simple "Hello, world!" inspired example, this time in spoken words:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.SpeechSynthesisSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SpeechSynthesisSample" Height="150" Width="150">
    <Grid>
        <Button Name="btnSayIt" Click="btnSayHello_Click"
VerticalAlignment="Center" HorizontalAlignment="Center">Say hello!</
        Button>
    </Grid>
```



```
</Window>

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

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class SpeechSynthesisSample : Window
    {
        public SpeechSynthesisSample()
        {
            InitializeComponent();
        }

        private void btnSayHello_Click(object sender, RoutedEventArgs e)
        {
            SpeechSynthesizer speechSynthesizer = new
SpeechSynthesizer();
            speechSynthesizer.Speak("Hello, world!");
        }
    }
}
```

}



This is pretty much as simple as it gets, and since the screenshot really doesn't help a lot in demonstrating speech synthesis, I suggest that you try building the example yourself, to experience it.

#### 1.21.4.1. Controlling pronunciation

The SpeechSynthesizer can do more than that though. Through the use of the PromptBuilder class, we can get much more control of how a sentence is spoken. This next example, which is an extension of the first example, will illustrate that:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.SpeechSynthesisPromptBuilderSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SpeechSynthesisPromptBuilderSample" Height="150" Width
        ="150">
    <Grid>
        <Button Name="btnSayIt" Click="btnSayHello_Click"
VerticalAlignment="Center" HorizontalAlignment="Center">Say hello!</
Button>
    </Grid>
</Window>

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

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class SpeechSynthesisPromptBuilderSample : Window
```

```

    {

        public SpeechSynthesisPromptBuilderSample()
        {
            InitializeComponent();
        }

        private void btnSayHello_Click(object sender, RoutedEventArgs
e)
        {
            PromptBuilder promptBuilder = new PromptBuilder();
            promptBuilder.AppendText("Hello world");

            PromptStyle promptStyle = new PromptStyle();
            promptStyle.Volume = PromptVolume.Soft;
            promptStyle.Rate = PromptRate.Slow;
            promptBuilder.StartStyle(promptStyle);
            promptBuilder.AppendText("and hello to the universe too."
);

            promptBuilder.EndStyle();

            promptBuilder.AppendText("On this day, ");

            promptBuilder.AppendTextWithHint(DateTime.Now.ToShortDateString(),
SayAs.Date);

            promptBuilder.AppendText(", we're gathered here to learn"
);

            promptBuilder.AppendText("all", PromptEmphasis.Strong);
            promptBuilder.AppendText("about");
            promptBuilder.AppendTextWithHint("WPF", SayAs.SpellOut);

            SpeechSynthesizer speechSynthesizer = new
SpeechSynthesizer();
            speechSynthesizer.Speak(promptBuilder);
        }
    }
}

```

This is where it gets interesting. Try running the example and see how nicely this works. By supplying the SpeechSynthesizer with something more than just a text string, we can get a lot of control of how the



various parts of the sentence are spoken. In this case, the application will say the following:

*Hello world and hello to the universe too. On this day, &lt;today's date&gt;, we're gathered here to learn all about WPF.*

Now try sending that directly to the SpeechSynthesizer and you'll probably giggle a bit of the result. What we do instead is guide the Speak() method into how the various parts of the sentence should be used. First of all, we ask WPF to speak the "and hello to the universe too"-part in a lower volume and a slower rate, as if it was whispered.

The next part that doesn't just use default pronunciation is the date. We use the special SayAs enumeration to specify that the date should be read out as an actual date and not just a set of numbers, spaces and special characters.

We also ask that the word "all" is spoken with a stronger emphasis, to make the sentence more dynamic, and in the end, we ask that the word "WPF" is spelled out (W-P-F) instead of being pronounced as an actual word.

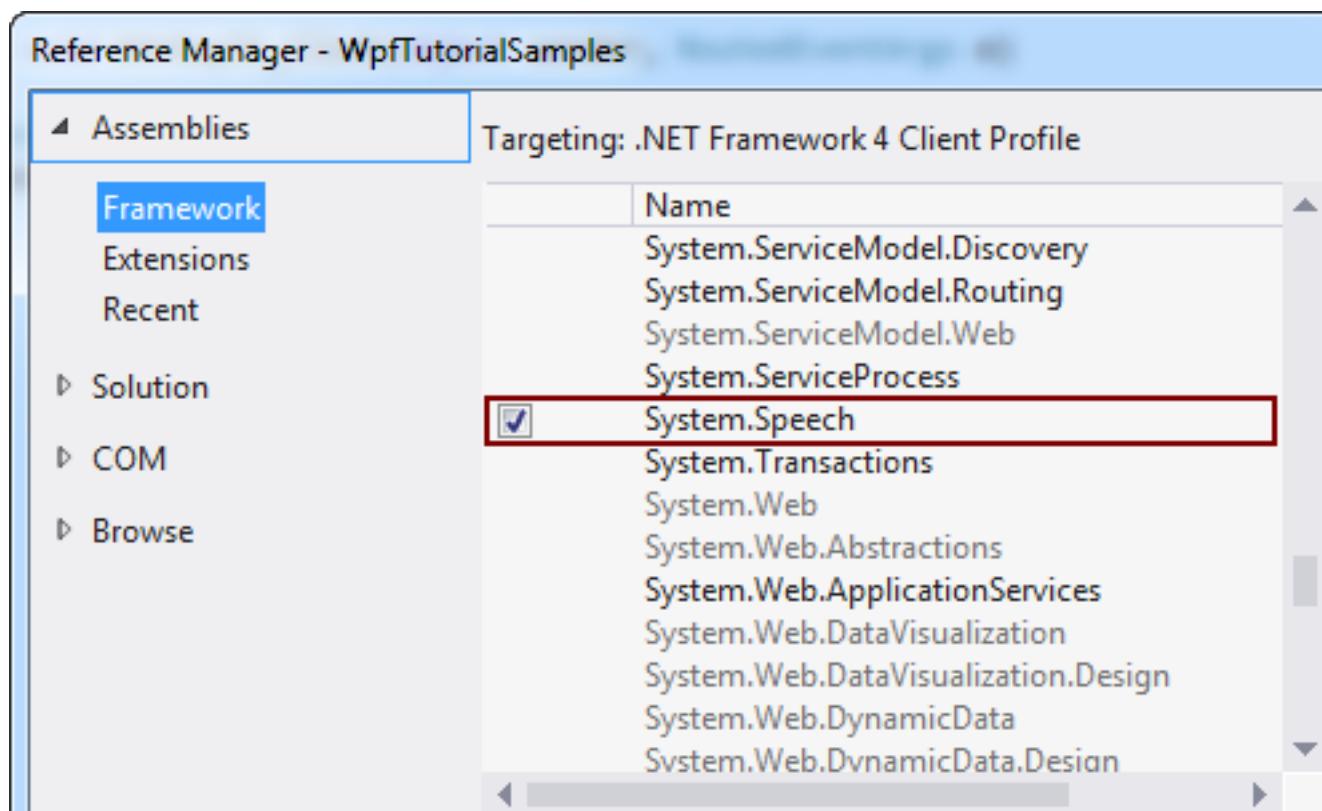
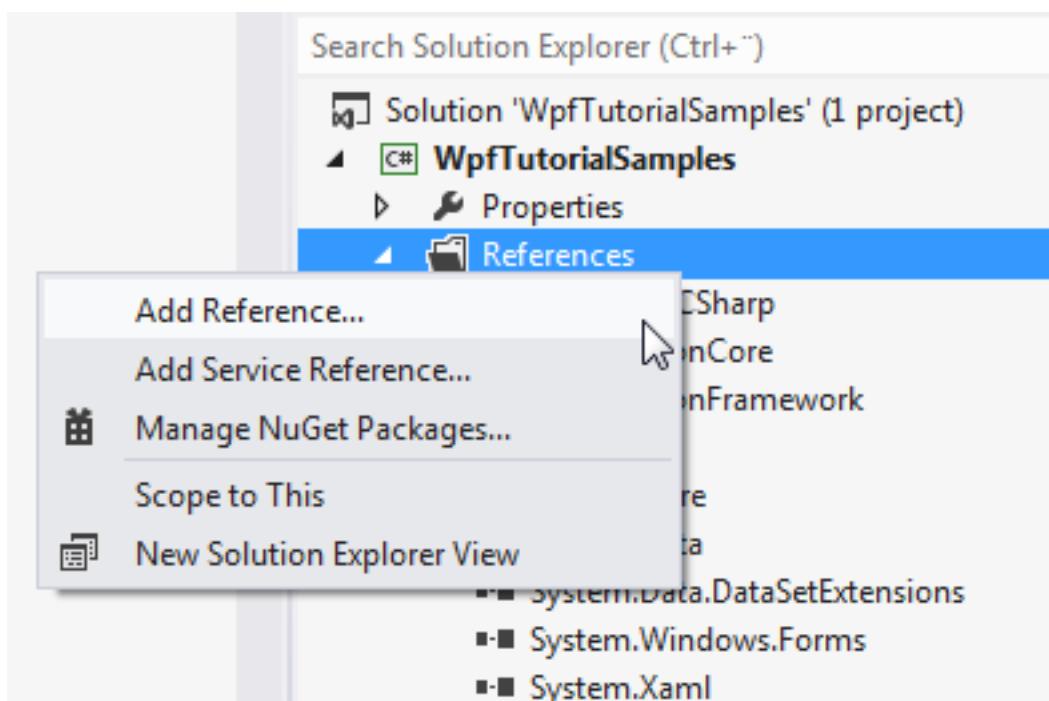
All in all, this allows us to make the SpeechSynthesizer a lot easier to understand!

#### 1.21.4.2. Summary

Making your WPF application speak is very easy, and by using the PromptBuilder class, you can even get a lot of control of how your words are spoken. This is a very powerful feature, but it might not be relevant to a lot of today's applications. It's still very cool though!

## 1.21.5. Speech recognition (making WPF listen)

In the previous article we discussed how we could transform text into spoken words, using the `SpeechSynthesizer` class. In this article we'll go the other way around, by turning spoken words into text. To do that, we'll be using the `SpeechRecognition` class, which resides in the `System.Speech` assembly. This assembly is not a part of your solutions by default, but we can easily add it. Depending on which version of Visual Studio you use, the process looks something like this:

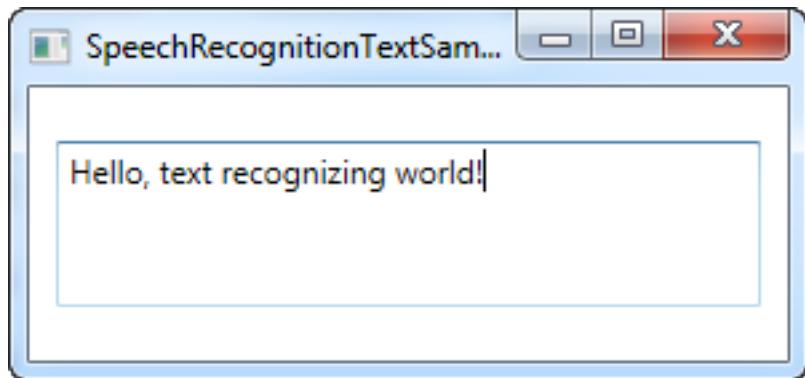


With that taken care of, let's start out with an extremely simple speech recognition example:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.SpeechRecognitionTextSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SpeechRecognitionTextSample" Height="200" Width="300">
    <DockPanel Margin="10">
        <TextBox Margin="0,10" Name="txtSpeech" AcceptsReturn="True"
    />
    </DockPanel>
</Window>

using System;
using System.Speech.Recognition;
using System.Windows;

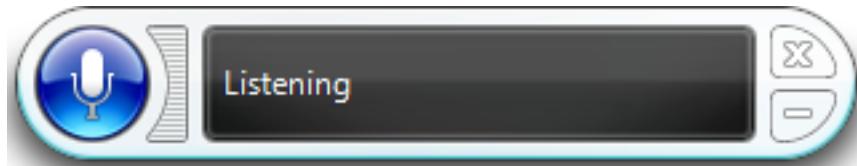
namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class SpeechRecognitionTextSample : Window
    {
        public SpeechRecognitionTextSample()
        {
            InitializeComponent();
            SpeechRecognizer speechRecognizer = new
SpeechRecognizer();
        }
    }
}
```



This is actually all you need - the text in the screenshot above was dictated through my headset and then

inserted into the TextBox control as text, through the use of speech recognition.

As soon as you initialize a **SpeechRecognizer** object, Windows starts up its speech recognition application, which will do all the hard work and then send the result to the active application, in this case ours. It looks like this:



*If you haven't used speech recognition on your computer before, then Windows will take you through a guide which will help you get started and make some necessary adjustments.*

This first example will allow you to dictate text to your application, which is great, but what about commands? Windows and WPF will actually work together here and turn your buttons into commands, reachable through speech, without any extra work. Here's an example:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.SpeechRecognitionTextCommandsSample"
        ">

    xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SpeechRecognitionTextCommandsSample" Height="200" Width
        ="300">

    <DockPanel Margin="10">
        <WrapPanel DockPanel.Dock="Top">
            <Button Name="btnNew" Click="btnNew_Click">New</Button>
            <Button Name="btnOpen" Click="btnOpen_Click">Open</Button>
        </WrapPanel>
        <TextBox Margin="0,10" Name="txtSpeech" AcceptsReturn="True"
        TextWrapping="Wrap" />
    </DockPanel>
</Window>
```

```

using System;
using System.Speech.Recognition;
using System.Windows;

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class SpeechRecognitionTextCommandsSample : Window
    {
        public SpeechRecognitionTextCommandsSample()
        {
            InitializeComponent();
            SpeechRecognizer recognizer = new SpeechRecognizer();
        }

        private void btnNew_Click(object sender, RoutedEventArgs e)
        {
            txtSpeech.Text = "";
        }

        private void btnOpen_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Command invoked: Open");
        }

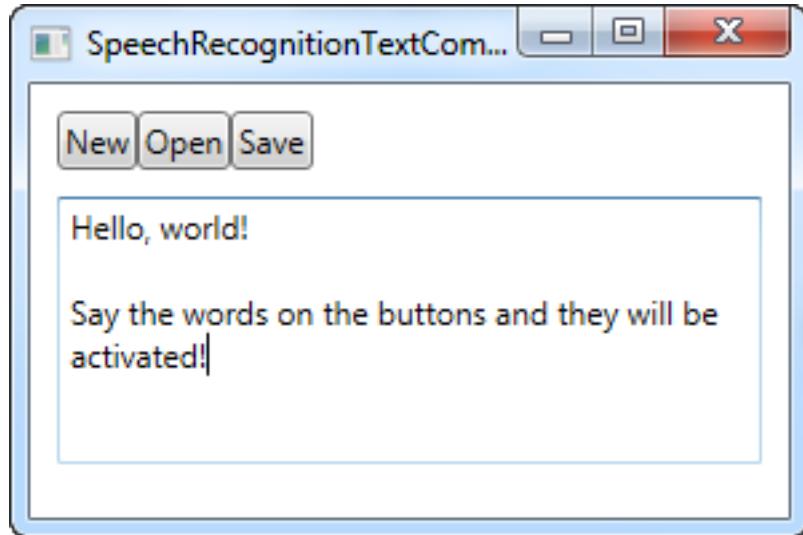
        private void btnSave_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Command invoked: Save");
        }
    }
}

```

You can try running the example and then speaking out one of the commands, e.g. "New" or "Open". This actually allows you to dictate text to the TextBox, while at the same time invoking commands from the user interface - pretty cool indeed!

#### 1.21.5.1. Specific commands

In the above example, Windows will automatically go into dictation mode as soon as focus is given to a text box. Windows will then try to distinguish between dictation and commands, but this can of course be difficult in certain situations.



So while the above examples have been focusing on dictation and interaction with UI elements, this next example will focus on the ability to listen for and interpret specific commands only. This also means that dictation will be ignored completely, even though text input fields have focus.

For this purpose, we will use the **SpeechRecognitionEngine** class instead of the **SpeechRecognizer** class. A huge difference between the two is that the **SpeechRecognitionEngine** class doesn't require the Windows speech recognition to be running and won't take you through the voice recognition guide. Instead, it will use basic voice recognition and listen only for grammar which you feed into the class.

In the next example, we'll feed a set of commands into the recognition engine. The idea is that it should listen for two words: A command/property and a value, which in this case will be used to change the color, size and weight of the text in a Label control, solely based on your voice commands. Before I show you the entire code sample, I want to focus on the way we add the commands to the engine. Here's the code:

```
GrammarBuilder grammarBuilder = new GrammarBuilder();
Choices commandChoices = new Choices("weight", "color", "size");
grammarBuilder.Append(commandChoices);

Choices valueChoices = new Choices();
valueChoices.Add("normal", "bold");
valueChoices.Add("red", "green", "blue");
valueChoices.Add("small", "medium", "large");
grammarBuilder.Append(valueChoices);

speechRecognizer.LoadGrammar(new Grammar(grammarBuilder));
```

We use a **GrammarBuilder** to build a set of grammar rules which we can load into the **SpeechRecognitionEngine**. It has several append methods, with the simplest one being **Append()**. This method takes a list of choices. We create a **Choices** instance, with the first part of the instruction - the command/property which we want to access. These choices are added to the builder with the **Append()** method.

Now, each time you call an append method on the GrammarBuilder, you instruct it to listen for a word. In our case, we want it to listen for two words, so we create a secondary set of choices, which will hold the value for the designated command/property. We add a range of values for each of the possible commands - one set of values for the weight command, one set of values for the color command and one set of values for the size command. They're all added to the same **Choices** instance and then appended to the builder.

In the end, we load it into the SpeechRecognitionEngine instance by calling the **LoadGrammar()** method, which takes a Grammar instance as parameter - in this case based on our GrammarBuilder instance.

So, with that explained, let's take a look at the entire example:

```
<Window x:Class
        ="WpfTutorialSamples.Audio_and_Video.SpeechRecognitionCommandsSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SpeechRecognitionCommandsSample" Height="200" Width
        ="325"
        Closing="Window_Closing">
    <DockPanel>
        <WrapPanel DockPanel.Dock="Bottom" HorizontalAlignment
        ="Center" Margin="0,10">
            <ToggleButton Name="btnToggleListening" Click
            ="btnToggleListening_Click">Listen</ToggleButton>
        </WrapPanel>
        <Label Name="lblDemo" HorizontalAlignment="Center"
        VerticalAlignment="Center" FontSize="48">Hello, world!</Label>
    </DockPanel>
</Window>

using System;
using System.Globalization;
using System.Speech.Recognition;
using System.Windows;
using System.Windows.Media;

namespace WpfTutorialSamples.Audio_and_Video
{
    public partial class SpeechRecognitionCommandsSample : Window
    {
        private SpeechRecognitionEngine speechRecognizer = new
        SpeechRecognitionEngine();
```

```

public SpeechRecognitionCommandsSample()
{
    InitializeComponent();
    speechRecognizer.SpeechRecognized += 
speechRecognizer_SpeechRecognized;

    GrammarBuilder grammarBuilder = new GrammarBuilder();
    Choices commandChoices = new Choices("weight", "color",
"size");
    grammarBuilder.Append(commandChoices);

    Choices valueChoices = new Choices();
    valueChoices.Add("normal", "bold");
    valueChoices.Add("red", "green", "blue");
    valueChoices.Add("small", "medium", "large");
    grammarBuilder.Append(valueChoices);

    speechRecognizer.LoadGrammar(new
Grammar(grammarBuilder));
    speechRecognizer.SetInputToDefaultAudioDevice();
}

private void btnToggleListening_Click(object sender,
RoutedEventArgs e)
{
    if(btnToggleListening.IsChecked == true)

speechRecognizer.RecognizeAsync(RecognizeMode.Multiple);
    else
        speechRecognizer.RecognizeAsyncStop();
}

private void speechRecognizer_SpeechRecognized(object sender,
SpeechRecognizedEventArgs e)
{
    lblDemo.Content = e.Result.Text;
    if(e.Result.Words.Count == 2)
    {
        string command = e.Result.Words[0].Text.ToLower();

```

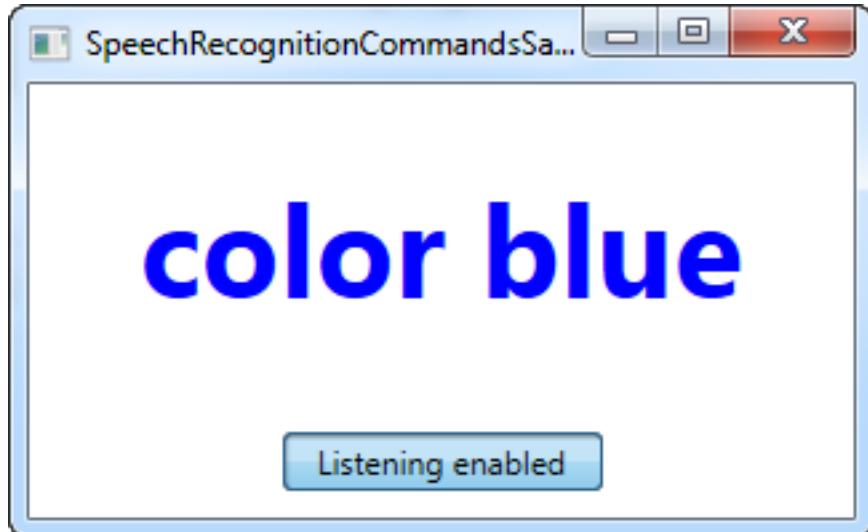
```

        string value = e.Result.Words[1].Text.ToLower();
        switch(command)
        {
            case "weight":
                FontWeightConverter weightConverter = new
FontWeightConverter();
                lblDemo.FontWeight =
(FontWeight)weightConverter.ConvertFromString(value);
                break;
            case "color":
                lblDemo.Foreground = new
SolidColorBrush((Color)ColorConverter.ConvertFromString(value));
                break;
            case "size":
                switch(value)
                {
                    case "small":
                        lblDemo.FontSize = 12;
                        break;
                    case "medium":
                        lblDemo.FontSize = 24;
                        break;
                    case "large":
                        lblDemo.FontSize = 48;
                        break;
                }
                break;
        }
    }

    private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    speechRecognizer.Dispose();
}
}
}

```

On the screenshot, you see the resulting application, after I've used the voice commands "weight bold" and



"color blue" - pretty cool, right?

The grammar aspects of the example has already been explained and the interface is very simple, so let's focus on the rest of the Code-behind.

We use a **ToggleButton** to enable or disable listening, using the `RecognizeAsync()` and `RecognizeAsyncStop()` methods. The `RecognizeAsync()` takes a parameter which informs the recognition engine if it should do a single recognition or multiple recognitions. For our example, we want to give several commands, so `Multiple` is used. So, to enable listening, just click the button, and to disable it, just click it again. The state is visually represented by the button, which will be "down" when enabled and normal when disabled.

Now, besides building the Grammar, the most interesting part is where we interpret the command. This is done in the **SpeechRecognized** event, which we hook up to in the constructor. We use the fully recognized text to update the demo label, to show the latest command, and then we use the `Words` property to dig deeper into the actual command.

First off, we check that it has exactly two words - a command/property and a value. If that is the case, we check the command part first, and for each possible command, we handle the value accordingly.

For the weight and color commands, we can convert the value into something the label can understand automatically, using a converter, but for the sizes, we interpret the given values manually, since the values I've chosen for this example can't be converted automatically. Please be aware that you should handle exceptions in all cases, since a command like "weight blue" will try to assign the value blue to the `FontWeight`, which will naturally result in an exception.

#### 1.21.5.2. Summary

As you can hopefully see, speech recognition with WPF is both easy and very powerful - especially the last example should give you a good idea just how powerful! With the ability to use dictation and/or specific voice commands, you can really provide excellent means for alternative input in your applications.

## 1.22. Misc.

---

### 1.22.1. The DispatcherTimer

In WinForms, there's a control called the Timer, which can perform an action repeatedly within a given interval. WPF has this possibility as well, but instead of an invisible control, we have the **DispatcherTimer** control. It does pretty much the same thing, but instead of dropping it on your form, you create and use it exclusively from your Code-behind code.

The DispatcherTimer class works by specifying an interval and then subscribing to the **Tick** event that will occur each time this interval is met. The DispatcherTimer is not started before you call the **Start()** method or set the **.IsEnabled** property to true.

Let's try a simple example where we use a DispatcherTimer to create a digital clock:

```
<Window x:Class="WpfTutorialSamples.Misc.DispatcherTimerSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DispatcherTimerSample" Height="150" Width="250">
    <Grid>
        <Label Name="lblTime" FontSize="48" HorizontalAlignment
        ="Center" VerticalAlignment="Center" />
    </Grid>
</Window>

using System;
using System.Windows;
using System.Windows.Threading;

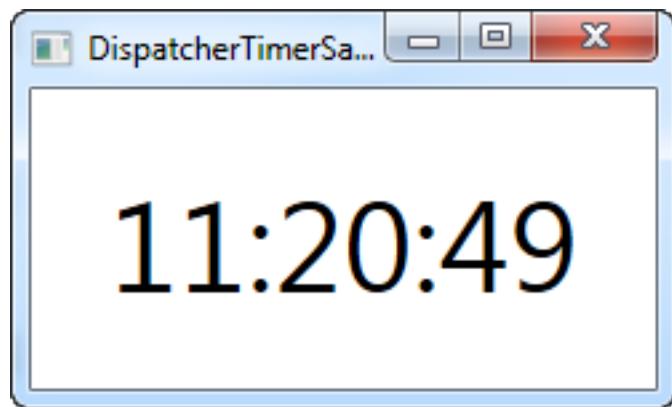
namespace WpfTutorialSamples.Misc
{
    public partial class DispatcherTimerSample : Window
    {
        public DispatcherTimerSample()
        {
            InitializeComponent();
            DispatcherTimer timer = new DispatcherTimer();
            timer.Interval = TimeSpan.FromSeconds(1);
            timer.Tick += timer_Tick;
            timer.Start();
        }
    }
}
```

```

        }

        void timer_Tick(object sender, EventArgs e)
        {
            lblTime.Content = DateTime.Now.ToString();
        }
    }
}

```



The XAML part is extremely simple - it's merely a centered label with a large font size, used to display the current time.

Code-behind is where the magic happens in this example. In the constructor of the window, we create a DispatcherTimer instance. We set the **Interval** property to one second, subscribe to the Tick event and then we start the timer. In the Tick event, we simply update the label to show the current time.

Of course, the DispatcherTimer can work at smaller or much bigger intervals. For instance, you might only want something to happen every 30 seconds or 5 minutes - just use the `TimeSpan.From*` methods, like `FromSeconds` or `FromMinutes`, or create a new `TimeSpan` instance that completely fits your needs.

To show what the DispatcherTimer is capable of, let's try updating more frequently... A lot more frequently!

```

using System;
using System.Windows;
using System.Windows.Threading;

namespace WpfTutorialSamples.Misc
{
    public partial class DispatcherTimerSample : Window
    {
        public DispatcherTimerSample()
        {

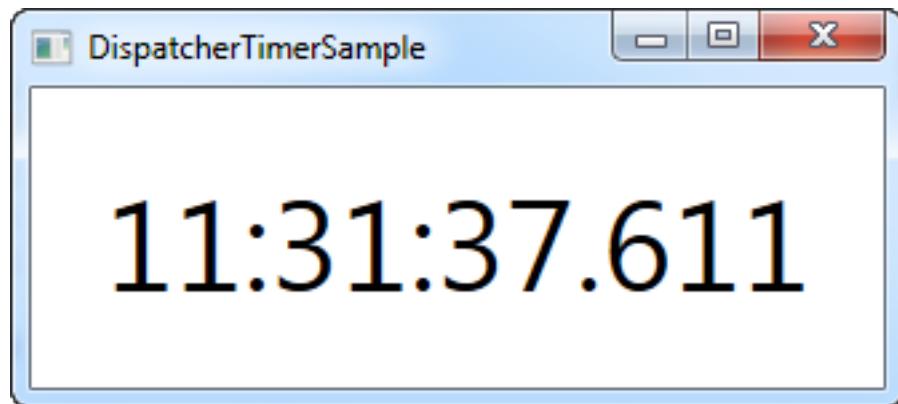
```

```

        InitializeComponent();
        DispatcherTimer timer = new DispatcherTimer();
        timer.Interval = TimeSpan.FromMilliseconds(1);
        timer.Tick += timer_Tick;
        timer.Start();
    }

    void timer_Tick(object sender, EventArgs e)
    {
        lblTime.Content = DateTime.Now.ToString("HH:mm:ss.ffff");
    }
}

```



As you can see, we now ask the DispatcherTimer to fire every millisecond! In the Tick event, we use a custom time format string to show the milliseconds in the label as well. Now you have something that could easily be used as a stopwatch - just add a couple of buttons to the Window and then have them call the **Stop()**, **Start()** and **Restart()** methods on the timer.

#### 1.22.1.1. Summary

There are many situations where you would need something in your application to occur at a given interval, and using the DispatcherTimer, it's quite easy to accomplish. Just be aware that if you do something complicated in your Tick event, it shouldn't run too often, like in the last example where the timer ticks each millisecond - that will put a heavy strain on the computer running your application.

Also be aware that the DispatcherTimer is not 100% precise in all situations. The tick operations are placed on the Dispatcher queue, so if the computer is under a lot of pressure, your operation might be delayed. The .NET framework promises that the Tick event will never occur too early, but can't promise that it won't be slightly delayed. However, for most use cases, the DispatcherTimer is more than precise enough.

If you need your timer to have a higher priority in the queue, you can set the DispatcherPriority by sending one of the values along on the DispatcherTimer priority. More information about it can be found on this

[MSDN article.](#)

## 1.22.2. Multi-threading with the BackgroundWorker

By default, each time your application executes a piece of code, this code is run on the same thread as the application itself. This means that while this code is running, nothing else happens inside of your application, including the updating of your UI.

This is quite a surprise to people who are new to Windows programming, when they first do something that takes more than a second and realize that their application actually hangs while doing so. The result is a lot of frustrated forum posts from people who are trying to run a lengthy process while updating a progress bar, only to realize that the progress bar is not updated until the process is done running.

The solution to all of this is the use of multiple threads, and while C# makes this quite easy to do, multi-threading comes with a LOT of pitfalls, and for a lot of people, they are just not that comprehensible. This is where the **BackgroundWorker** comes into play - it makes it simple, easy and fast to work with an extra thread in your application.

### 1.22.2.1. How the BackgroundWorker works

The most difficult concept about multi-threading in a Windows application is the fact that you are not allowed to make changes to the UI from another thread - if you do, the application will immediately crash. Instead, you have to invoke a method on the UI (main) thread which then makes the desired changes. This is all a bit cumbersome, but not when you use the BackgroundWorker.

When performing a task on a different thread, you usually have to communicate with the rest of the application in two situations: When you want to update it to show how far you are in the process, and then of course when the task is done and you want to show the result. The BackgroundWorker is built around this idea, and therefore comes with the two events **ProgressChanged** and **RunWorkerCompleted**.

The third event is called **DoWork** and the general rule is that you can't touch anything in the UI from this event. Instead, you call the **ReportProgress()** method, which in turn raises the **ProgressChanged** event, from where you can update the UI. Once you're done, you assign a result to the worker and then the **RunWorkerCompleted** event is raised.

So, to sum up, the **DoWork** event takes care of all the hard work. All of the code in there is executed on a different thread and for that reason you are not allowed to touch the UI from it. Instead, you bring data (from the UI or elsewhere) into the event by using the argument on the **RunWorkerAsync()** method, and the resulting data out of it by assigning it to the **e.Result** property.

The **ProgressChanged** and **RunWorkerCompleted** events, on the other hand, are executed on the same thread as the BackgroundWorker is created on, which will usually be the main/UI thread and therefore you are allowed to update the UI from them. Therefore, the only communication that can be performed between your running background task and the UI is through the **ReportProgress()** method.

That was a lot of theory, but even though the BackgroundWorker is easy to use, it's important to understand how and what it does, so you don't accidentally do something wrong - as already stated, errors in multi-threading can lead to some nasty problems.

## 1.22.2.2. A BackgroundWorker example

Enough with the theory - let's see what it's all about. In this first example, we want a pretty simply but time consuming job performed. Each number between 0 and 10.000 gets tested to see if it's divisible with the number 7. This is actually a piece of cake for today's fast computers, so to make it more time consuming and thereby easier to prove our point, I've added a one millisecond delay in each of the iterations.

Our sample application has two buttons: One that will perform the task synchronously (on the same thread) and one that will perform the task with a BackgroundWorker and thereby on a different thread. This should make it very easy to see the need for an extra thread when doing time consuming tasks. The code looks like this:

```
<Window x:Class="WpfTutorialSamples.Misc.BackgroundWorkerSample"
        xmlns
        ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BackgroundWorkerSample" Height="300" Width="375">
    <DockPanel Margin="10">
        <DockPanel DockPanel.Dock="Top">
            <Button Name="btnDoSynchronousCalculation" Click
                    ="btnDoSynchronousCalculation_Click" DockPanel.Dock="Left"
                    HorizontalAlignment="Left">Synchronous (same thread)</Button>
            <Button Name="btnDoAsynchronousCalculation" Click
                    ="btnDoAsynchronousCalculation_Click" DockPanel.Dock="Right"
                    HorizontalAlignment="Right">Asynchronous (worker thread)</Button>
        </DockPanel>
        <ProgressBar DockPanel.Dock="Bottom" Height="18" Name
                    ="pbCalculationProgress" />
        <ListBox Name="lbResults" Margin="0,10" />
    </DockPanel>
</Window>

using System;
using System.ComponentModel;
using System.Windows;

namespace WpfTutorialSamples.Misc
{
    public partial class BackgroundWorkerSample : Window
    {
```

```

public BackgroundWorkerSample()
{
    InitializeComponent();
}

private void btnDoSynchronousCalculation_Click(object sender,
RoutedEventArgs e)
{
    int max = 10000;
    pbCalculationProgress.Value = 0;
    lbResults.Items.Clear();

    int result = 0;
    for(int i = 0; i < max; i++)
    {
        if(i % 42 == 0)
        {
            lbResults.Items.Add(i);
            result++;
        }
        System.Threading.Thread.Sleep(1);
        pbCalculationProgress.Value = Convert.ToInt32(((double)i / max) * 100);
    }
    MessageBox.Show("Numbers between 0 and 10000 divisible by
7: " + result);
}

private void btnDoAsynchronousCalculation_Click(object sender,
RoutedEventArgs e)
{
    pbCalculationProgress.Value = 0;
    lbResults.Items.Clear();

    BackgroundWorker worker = new BackgroundWorker();
    worker.WorkerReportsProgress = true;
    worker.DoWork += worker_DoWork;
    worker.ProgressChanged += worker_ProgressChanged;
    worker.RunWorkerCompleted += worker_RunWorkerCompleted;
    worker.RunWorkerAsync(10000);
}

```

```

    }

    void worker_DoWork(object sender, DoWorkEventArgs e)
    {
        int max = (int)e.Argument;
        int result = 0;
        for(int i = 0; i < max; i++)
        {
            int progressPercentage = Convert.ToInt32(((double)i
/ max) * 100);

            if(i % 42 == 0)
            {
                result++;
                (sender as
BackgroundWorker).ReportProgress(progressPercentage, i);
            }
            else
                (sender as
BackgroundWorker).ReportProgress(progressPercentage);
                System.Threading.Thread.Sleep(1);
        }

        e.Result = result;
    }

    void worker_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
    pbCalculationProgress.Value = e.ProgressPercentage;
    if(e.UserState != null)
        lbResults.Items.Add(e.UserState);
}

void worker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    MessageBox.Show("Numbers between 0 and 10000 divisible by
7: " + e.Result);
}

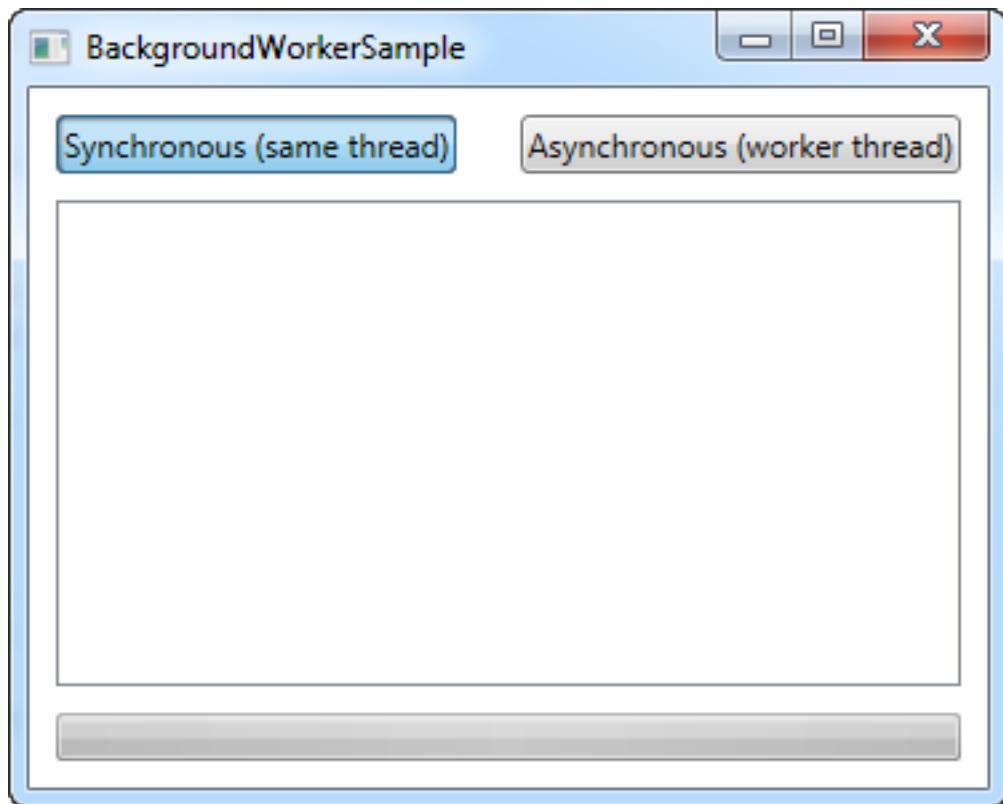
```

```
}
```

The XAML part consists of a couple of buttons, one for running the process synchronously (on the UI thread) and one for running it asynchronously (on a background thread), a ListBox control for showing all the calculated numbers and then a ProgressBar control in the bottom of the window to show... well, the progress!

In Code-behind, we start off with the synchronous event handler. As mentioned, it loops from 0 to 10.000 with a small delay in each iteration, and if the number is divisible with the number 7, then we add it to the list. In each iteration we also update the ProgressBar, and once we're all done, we show a message to the user about how many numbers were found.

If you run the application and press the first button, it will look like this, no matter how far you are in the process:



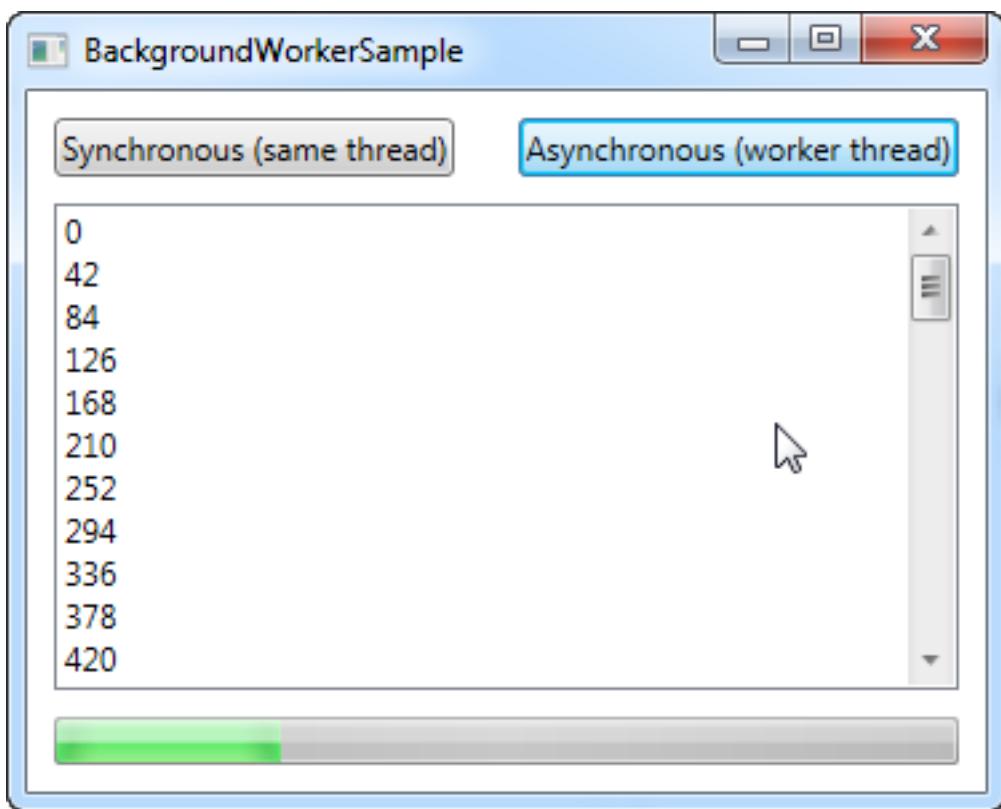
No items on the list and, no progress on the ProgressBar, and the button hasn't even been released, which proves that there hasn't been a single update to the UI ever since the mouse was pressed down over the button.

Pressing the second button instead will use the BackgroundWorker approach. As you can see from the code, we do pretty much the same, but in a slightly different way. All the hard work is now placed in the **DoWork** event, which the worker calls after you run the **RunWorkerAsync()** method. This method takes input from your application which can be used by the worker, as we'll talk about later.

As already mentioned, we're not allowed to update the UI from the `DoWork` event. Instead, we call the `ReportProgress` method on the worker. If the current number is divisible with 7, we include it to be added on the list - otherwise we only report the current progress percentage, so that the `ProgressBar` may be updated.

Once all the numbers have been tested, we assign the result to the `e.Result` property. This will then be carried to the `RunWorkerCompleted` event, where we show it to the user. This might seem a bit cumbersome, instead of just showing it to the user as soon as the work is done, but once again, it ensures that we don't communicate with the UI from the `DoWork` event, which is not allowed.

The result is, as you can see, much more user friendly:



The window no longer hangs, the button is clicked but not suppressed, the list of possible numbers is updated on the fly and the `ProgressBar` is going steadily up - the interface just got a whole lot more responsive.

#### 1.22.2.3. Input and output

Notice that both the input, in form of the argument passed to the `RunWorkerAsync()` method, as well as the output, in form of the value assigned to the `e.Result` property of the `DoWork` event, are of the object type. This means that you can assign any kind of value to them. Our example was basic, since both input and output could be contained in a single integer value, but it's normal to have more complex input and output.

This is accomplished by using a more complex type, in many cases a struct or even a class which you

create yourself and pass along. By doing this, the possibilities are pretty much endless and you can transport as much and complex data as you want between your BackgroundWorker and your application/UI.

The same is actually true for the ReportProgress method. Its secondary argument is called userState and is an object type, meaning that you can pass whatever you want to the ProgressChanged method.

#### 1.22.2.4. Summary

The BackgroundWorker is an excellent tool when you want multi-threading in your application, mainly because it's so easy to use. In this chapter we looked at one of the things made very easy by the BackgroundWorker, which is progress reporting, but support for cancelling the running task is very handy as well. We'll look into that in the next chapter.

### 1.22.3. Cancelling the BackgroundWorker

As we saw in the previous article, the multi-threading has the added advantage of being able to show progress and not having your application hang while performing time-consuming operation.

Another problem you'll face if you perform all the work on the UI thread is the fact that there's no way for the user to cancel a running task - and why is that? Because if the UI thread is busy performing your lengthy task, no input will be processed, meaning that no matter how hard your user hits a Cancel button or the Esc key, nothing happens.

Fortunately for us, the `BackgroundWorker` is built to make it easy for you to support progress and cancellation, and while we looked at the whole progress part in the previous chapter, this one will be all about how to use the cancellation support. Let's jump straight to an example:

```
<Window x:Class
    ="WpfTutorialSamples.Misc.BackgroundWorkerCancellationSample"
        xmlns
    ="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BackgroundWorkerCancellationSample" Height="120" Width
    ="200">
    <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center"
>
    <TextBlock Name="lblStatus" HorizontalAlignment="Center"
Margin="0,10" FontWeight="Bold">Not running...</TextBlock>
    <WrapPanel>
        <Button Name="btnStart" Width="60" Margin="10,0" Click
    ="btnStart_Click">Start</Button>
        <Button Name="btnCancel" Width="60" Click
    ="btnCancel_Click">Cancel</Button>
    </WrapPanel>
</StackPanel>
</Window>

using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Media;

namespace WpfTutorialSamples.Misc
{
    public partial class BackgroundWorkerCancellationSample : Window
```

```

{
    private BackgroundWorker worker = null;

    public BackgroundWorkerCancellationSample()
    {
        InitializeComponent();
        worker = new BackgroundWorker();
        worker.WorkerSupportsCancellation = true;
        worker.WorkerReportsProgress = true;
        worker.DoWork += worker_DoWork;
        worker.ProgressChanged += worker_ProgressChanged;
        worker.RunWorkerCompleted += worker_RunWorkerCompleted;
    }

    private void btnStart_Click(object sender, RoutedEventArgs e)
    {
        worker.RunWorkerAsync();
    }

    private void btnCancel_Click(object sender, RoutedEventArgs e)
    {
        worker.CancelAsync();
    }

    void worker_DoWork(object sender, DoWorkEventArgs e)
    {
        for(int i = 0; i <= 100; i++)
        {
            if(worker.CancellationPending == true)
            {
                e.Cancel = true;
                return;
            }
            worker.ReportProgress(i);
            System.Threading.Thread.Sleep(250);
        }
        e.Result = 42;
    }

    void worker_ProgressChanged(object sender,

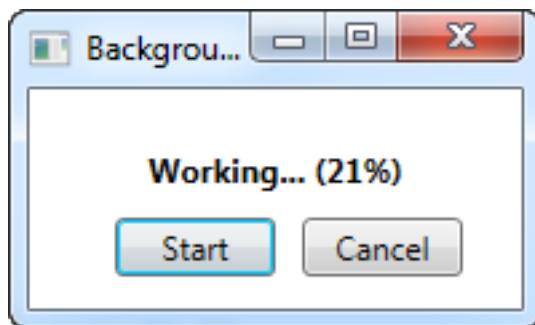
```

```

ProgressChangedEventArgs e)
{
    lblStatus.Text = "Working... (" + e.ProgressPercentage +
"%)";
}

void worker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    if(e.Cancelled)
    {
        lblStatus.Foreground = Brushes.Red;
        lblStatus.Text = "Cancelled by user...";
    }
    else
    {
        lblStatus.Foreground = Brushes.Green;
        lblStatus.Text = "Done... Calc result: " + e.Result;
    }
}
}

```



So, the XAML is very fundamental - just a label for showing the current status and then a couple of buttons for starting and cancelling the worker.

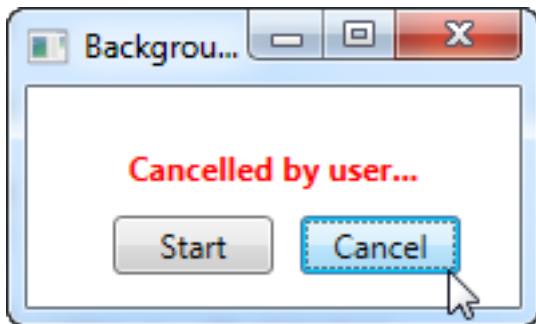
In Code-behind, we start off by creating the `BackgroundWorker` instance. Pay special attention to the **WorkerSupportsCancellation** and **WorkerReportsProgress** properties which we set to true - without that, an exception will be thrown if we try to use these features.

The cancel button simply calls the **CancelAsync()** method - this will signal to the worker that the user would like to cancel the running process by setting the **CancellationPending** property to true, but that is all you can do from the UI thread - the rest will have to be done from inside the **DoWork** event.

In the **DoWork** event, we count to 100 with a 250 millisecond delay on each iteration. This gives us a nice and lengthy task, for which we can report progress and still have time to hit that Cancel button.

Notice how I check the **CancellationPending** property on each iteration - if the worker is cancelled, this property will be true and I will have the opportunity to jump out of the loop. I set the **Cancel** property on the event arguments, to signal that the process was cancelled - this value is used in the **RunWorkerCompleted** event to see if the worker actually completed or if it was cancelled.

In the **RunWorkerCompleted** event, I simply check if the worker was cancelled or not and then update the status label accordingly.



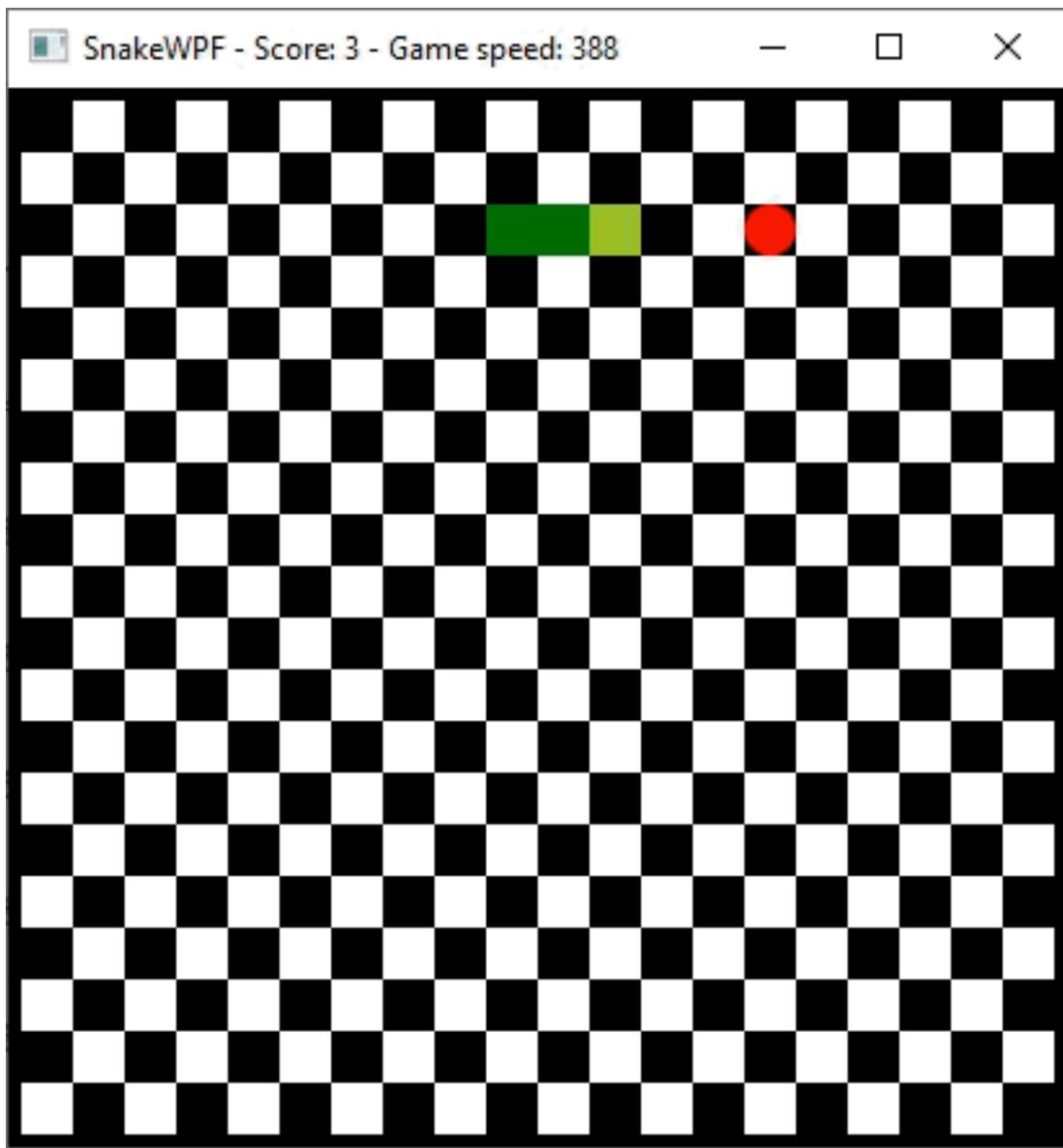
#### 1.22.3.1. Summary

So to sum up, adding cancellation support to your `BackgroundWorker` is easy - just make sure that you set **WorkerSupportsCancellation** property to true and check the **CancellationPending** property while performing the work. Then, when you want to cancel the task, you just have to call the **CancelAsync()** method on the worker and you're done.

## 1.23. Creating a Game: SnakeWPF

### 1.23.1. Introduction

A quick summary of this introductory article: We'll be implementing the classic Snake game in WPF, and the final result will look something like this:



Creating games is often the reason that a lot of, especially young, people are drawn towards learning programming. But the question usually is: How do I get started and what do I need? Well, you need a programming language, like C#, C++ or any of the other popular languages, and if you're fluent in your preferred language, you don't really need anything else: Just start at the bottom by adding pixels to the screen and at some point, you might have a working game.

However, most people would prefer a bit of help with the low-level stuff. Why add pixels to the screen manually if there's a library or framework that can do it for you, so you can focus on building an entertaining

game? There are many frameworks out there which will help you do this, and actually, **one of them is the WPF framework.**

Now granted, WPF is not the most obvious choice when you want to create games - it's definitely a framework that focuses mostly on creating user interfaces for business-oriented applications. But still, there are many elements in the WPF framework that you can use to create a game, and perhaps equally important: You get all the mechanisms to paint and control a Window in Windows.

So, if you're looking to create a simple game, WPF might actually be a fine choice. At least it will be a great help for all the most basic aspects, like creating a Window, drawing a simple area for the game etc. If you want to add stuff like advanced 3D graphics and fast moving objects, might need more help from another library/framework, but it will do just fine for a simple game - for instance, a classic [Snake game!](#)

#### 1.23.1.1. SnakeWPF

As a proof of concept, I have decided to create a WPF-based version of the extremely classical Snake game. It will use a regular WPF Window as its game area, as well as regular WPF controls/shapes to create the actual gameplay. The reason why I chose Snake is because it's fairly easy to implement (there's not that much logic to code) and because it can be implemented using simple geometric figures like squares and circles, which can be used very easily with the WPF framework. But also because it's still a really funny game, despite it's simplistic nature!

If you don't know the Snake game, I can only assume that you never owned a Nokia cellphone during the late 90's/early 2000's. The first version of Snake was written and demonstrated many years before that, but it became a major hit when Nokia decided to include their own version of it on all their cellphones.

**The gameplay is as simple as it is entertaining:** You move a virtual snake in one direction (left, right, up or down) in the hunt for food (sometimes an apple). When your snake hits the apple, it's consumed, your snake grows and a new apple appears on the screen. If you hit the walls or your own snake tail, the game ends and you have to start all over. The more apples you eat, the higher score you get but the more difficult it will get not to hit your own tail.

There are MANY variations to the gameplay - for instance, the speed with which your snake moves will often increase each time you eat an apple, making it harder and harder, but not all Snake implementations will do this. Another variation is the walls - some implementations will allow you to go through the wall and out on the opposite side, while other implementations will have the game end as soon as you hit the wall.

In our SnakeWPF, the walls are hard (the snake dies if it hits them), and the speed will increase exponentially for each apple you eat, up to a certain point.

#### 1.23.1.2. Summary

Over the next several articles, we'll be implementing a nice version of the classic Snake game using the WPF framework. We'll start with the background in the next article, and in the end, we'll have our first, fully functional WPF-based game.

Please notice that while this IS a WPF tutorial, we will need a bit more C# code than normal, to implement the game logic etc. I will try to explain most of it as we move along, but in case you need a bit more knowledge about C#, don't forget that we have a nice, [complete C# tutorial](#) in our network!

## 1.23.2. Creating the game area

To create our SnakeWPF game, we'll start by creating the map. It will be a confined area, where the snake has to move within - a snake pit, if you will. I have decided that my snake pit should look like a chess board, made up equally sized squares, which will have the same dimensions as the body of the snake. We'll create the map in two iterations: Some of it will be laid out in XAML, because it's easy, while we'll draw the background squares in Code-behind, because it's repetitive and dynamic.

### 1.23.2.1. Game area XAML

So, let's start with the XAML - a simple Window with Canvas panel inside of a Border control, to create the confined area:

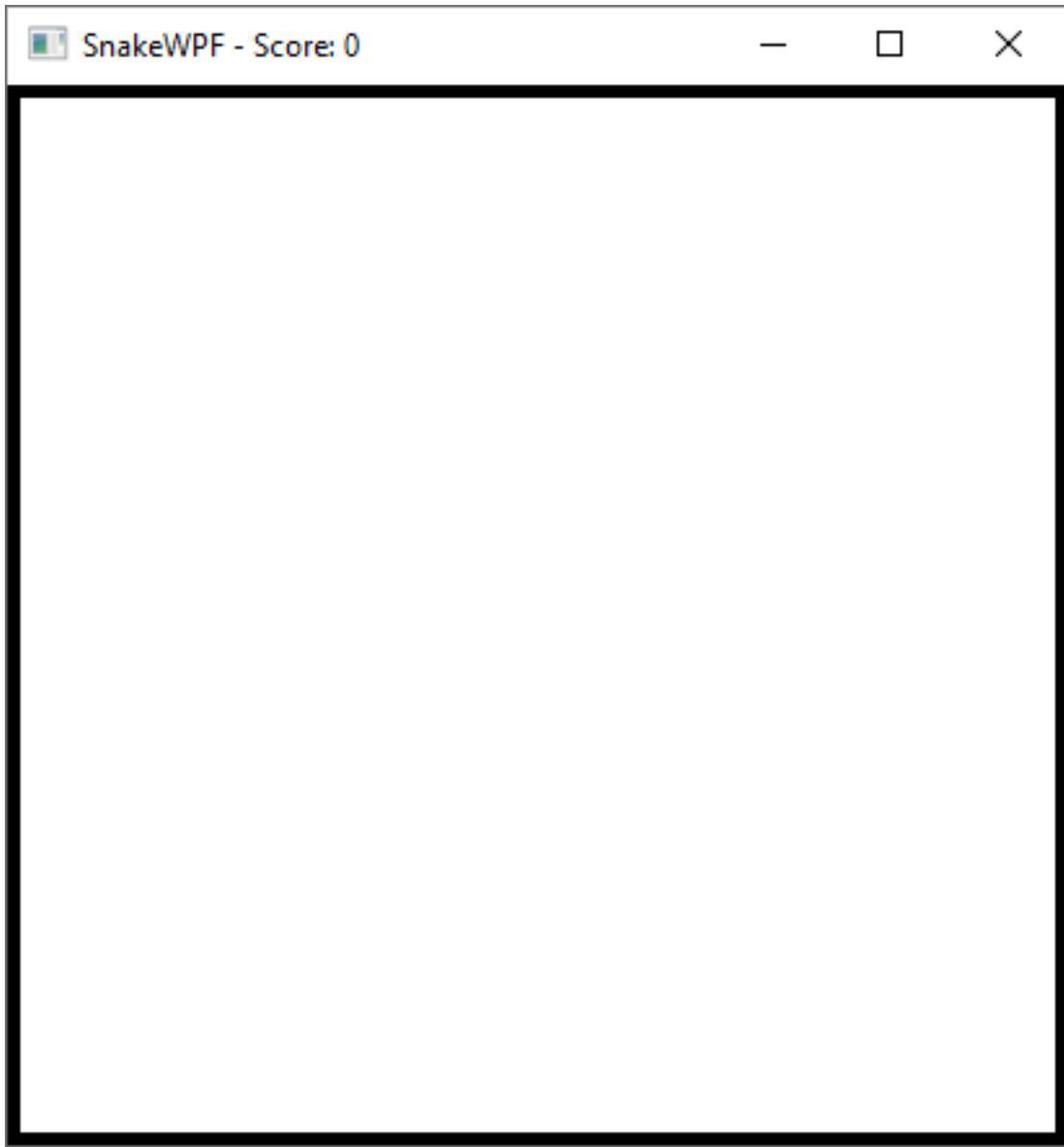
```
<Window x:Class="WpfTutorialSamples.Games.SnakeWPFSample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expressionblend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:local="clr-namespace:WpfTutorialSamples.Games"
    mc:Ignorable="d"
    Title="SnakeWPF - Score: 0" SizeToContent="WidthAndHeight">
<Border BorderBrush="Black" BorderThickness="5">
    <Canvas Name="GameArea" ClipToBounds="True" Width="400" Height
    ="400">

        </Canvas>
</Border>
</Window>
```

Our game now looks like this:

We use a Canvas as the actual game area, because it will allow us to add controls to it where we get full control of the positions. We'll use that later, but for now, pay attention to the following things:

- No width/height is defined for the Window - instead, we defined it for the Canvas, because that's the part we need to fully control. We then make sure that the Window will adjust its size accordingly by setting the **SizeToContent** property to **WidthAndHeight**. Had we instead defined the width/height for the Window, the available space within it would depend on how much border the Operating System used for Windows, which could depend on themes etc.
- We set the **ClipToBounds** property to True for the Canvas - this is



important, because otherwise the controls we add would be able to expand beyond the boundaries of the Canvas panel

#### 1.23.2.2. Drawing the background from Code-behind

As mentioned, I want a checkerboard background for the game area. It consists of a lot of squares, so it's easier to add it from Code-behind (or to use an image, but that's not as dynamic!). We need to do this as soon as all controls inside the Window has been initialized/rendered, and fortunately for us, the Window has an event for that: The **ContentRendered** event. We'll subscribe to that in the Window declaration:

```
Title="SnakeWPF - Score: 0" SizeToContent="WidthAndHeight"  
ContentRendered="Window_ContentRendered"
```

Now move to Code-behind and let's get started. First of all, we need to define a size to use when drawing the snake, the background squares etc. It can be done in the top of your Window class:

```
public partial class SnakeWPFSample : Window  
{
```

```
const int SnakeSquareSize = 20;  
.....
```

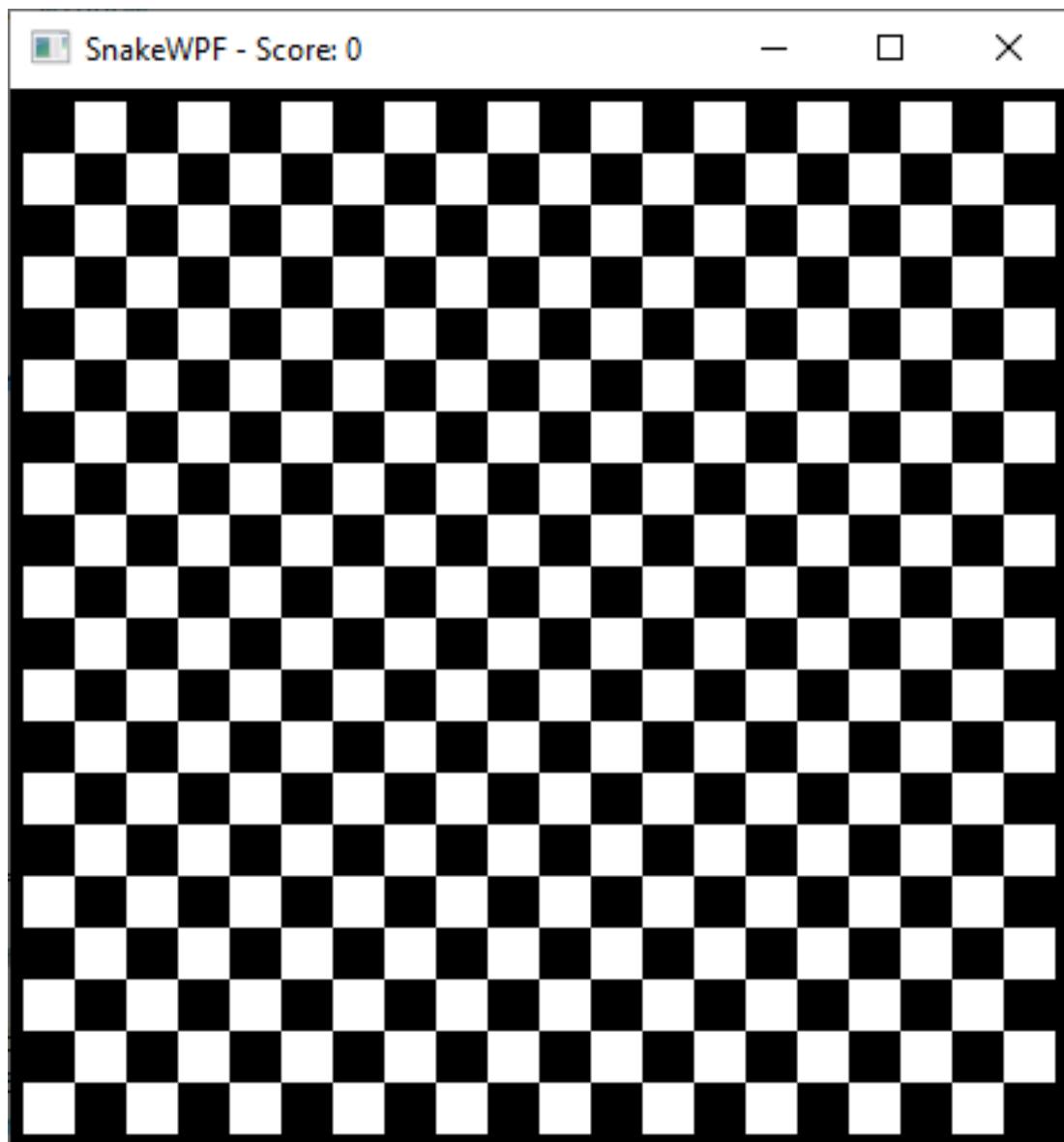
Now, in our **ContentRendered** event we'll call the **DrawGameArea()** method, which will do all the hard work. It looks like this:

```
private void Window_ContentRendered(object sender, EventArgs e)  
{  
    DrawGameArea();  
}  
  
private void DrawGameArea()  
{  
    bool doneDrawingBackground = false;  
    int nextX = 0, nextY = 0;  
    int rowCounter = 0;  
    bool nextIsOdd = false;  
  
    while(doneDrawingBackground == false)  
    {  
        Rectangle rect = new Rectangle  
        {  
            Width = SnakeSquareSize,  
            Height = SnakeSquareSize,  
            Fill = nextIsOdd ? Brushes.White : Brushes.Black  
        };  
        GameArea.Children.Add(rect);  
        Canvas.SetTop(rect, nextY);  
        Canvas.SetLeft(rect, nextX);  
  
        nextIsOdd = !nextIsOdd;  
        nextX += SnakeSquareSize;  
        if(nextX >= GameArea.ActualWidth)  
        {  
            nextX = 0;  
            nextY += SnakeSquareSize;  
            rowCounter++;  
            nextIsOdd = (rowCounter % 2 != 0);  
        }  
  
        if(nextY >= GameArea.ActualHeight)
```

```
        doneDrawingBackground = true;  
    }  
}
```

As previously mentioned, these articles requires a bit more C# knowledge than the rest of the articles in this tutorial, so I won't go over each line, but here's a general description of what we do: Inside the while loop, we continuously create instances of the Rectangle control and add it to the Canvas (*GameArea*). We fill it with either a White or a Black Brush, and it uses our **SnakeSquareSize** constant as both Width and Height, since we want it to be a square. On each iteration, we use the nextX and nextY to control when to move to the next line (when we reach the right border) and when to stop (when we reach the bottom AND the right border at the same time).

Here's the result:



### 1.23.2.3. Summary

In this article, we have defined the XAML used to host all the game content, and we have "painted" a checkerboard pattern on the game area, by adding WPF Rectangle controls in black and white to it. The next step will be to start adding the actual snake, as well as the food it will be eating.

### 1.23.3. Creating & moving the Snake

In the last article, we made a nice area for the snake in our SnakeWPF implementation to move around. With that in place, it's now time to create the actual snake and then make it move around the area. Once again, we'll use the WPF **Rectangle** class to form a snake of a certain length, with each element being the same width and height as the background squares, or as we call it: The `SnakeSquareSize` constant!

#### 1.23.3.1. Creating the Snake

We'll draw the snake with a method called `DrawSnake()` - the method is actually quite simple, but it does require quite a bit of extra stuff, including a new class called `SnakePart`, as well as some extra fields on the `Window` class. Let's start with the `SnakePart` class, which you would normally define in a new file (`SnakePart.cs`, for instance):

```
using System.Windows;

namespace WpfTutorialSamples.Games
{
    public class SnakePart
    {
        public UIElement UIElement { get; set; }

        public Point Position { get; set; }

        public bool IsHead { get; set; }
    }
}
```

This simple class will contain information about each part of the snake: Where is the element positioned in our game area, which `UIElement` (a `Rectangle`, in our case) represents the part, and is this the head-part of the snake or not? We'll use all of it later, but first, inside our `Window` class, we need to define a couple of fields to be used in our `DrawSnake()` method (and later in other methods as well):

```
public partial class SnakeWPFSample : Window
{
    const int SnakeSquareSize = 20;

    private SolidColorBrush snakeBodyBrush = Brushes.Green;
    private SolidColorBrush snakeHeadBrush = Brushes.YellowGreen;
    private List<SnakePart> snakeParts = new List<SnakePart>();
    .....
```

We define two **SolidColorBrush**'es, one for the body and one for the head. We also define a

List<SnakePart>, which will keep a reference to all parts of the snake. With that in place, we can now implement our DrawSnake() method:

```
private void DrawSnake()
{
    foreach(SnakePart snakePart in snakeParts)
    {
        if(snakePart.UiElement == null)
        {
            snakePart.UiElement = new Rectangle()
            {
                Width = SnakeSquareSize,
                Height = SnakeSquareSize,
                Fill = (snakePart.IsHead ? snakeHeadBrush :
snakeBodyBrush)
            };
            GameArea.Children.Add(snakePart.UiElement);
            Canvas.SetTop(snakePart.UiElement, snakePart.Position.Y);
            Canvas.SetLeft(snakePart.UiElement,
snakePart.Position.X);
        }
    }
}
```

As you can see, this method is not particularly complicated: We loop over the `snakeParts` List, and for each part, we check if a **UIElement** has been specified for this part - if not, we create it (represented by a Rectangle) and add it to the game area, while saving a reference to it on the **UiElement** property of the SnakePart instance. Notice how we use the Position property of the SnakePart instance to position the actual element inside the GameArea Canvas.

The trick here is of course that the actual parts of the snake will be defined elsewhere, allowing us to add one or several parts to the snake, give them the desired position, and then have the `DrawSnake()` method do the actual work for us. We'll do that as a part of the same process used to move the snake.

### 1.23.3.2. Moving the Snake

To feed something to the `DrawSnake()` method, we need to populate the `snakeParts` list. This list constantly serves as the basis of where to draw each element of the snake, so we'll also be using it to create movement for the snake. The process of moving the snake basically consists of adding a new element to it, in the direction where the snake is currently moving, and then delete the last part of the snake. This will make it look like we're actually moving each element, but in fact, we're just adding new ones while deleting the old ones.

So, we'll need a `MoveSnake()` method, which I'll show you in just a minute, but first, we need to add some more to the top of our `Window` class definition:

```
public partial class SnakeWPFSSample : Window
{
    const int SnakeSquareSize = 20;

    private SolidColorBrush snakeBodyBrush = Brushes.Green;
    private SolidColorBrush snakeHeadBrush = Brushes.YellowGreen;
    private List<SnakePart> snakeParts = new List<SnakePart>();

    public enum SnakeDirection { Left, Right, Up, Down };
    private SnakeDirection snakeDirection = SnakeDirection.Right;
    private int snakeLength;
    .....
```

We have added a new enumeration, called **SnakeDirection**, which should be pretty self-explanatory. For that, we have a private field to hold the actual, current direction (`snakeDirection`), and then we have an integer variable to hold the desired length of the snake (`snakeLength`). With that in place, we're ready to implement the **MoveSnake()** method. It's a bit long, so I've added inline-comments to each of the important parts of it:

```
private void MoveSnake()
{
    // Remove the last part of the snake, in preparation of the new part
    // added below
    while(snakeParts.Count >= snakeLength)
    {
        GameArea.Children.Remove(snakeParts[0].UiElement);
        snakeParts.RemoveAt(0);
    }
    // Next up, we'll add a new element to the snake, which will be the
    // (new) head
    // Therefore, we mark all existing parts as non-head (body) elements and
    // then
    // we make sure that they use the body brush
    foreach(SnakePart snakePart in snakeParts)
    {
        (snakePart.UiElement as Rectangle).Fill = snakeBodyBrush;
        snakePart.IsHead = false;
    }
}
```

```

// Determine in which direction to expand the snake, based on the
current direction
SnakePart snakeHead = snakeParts[snakeParts.Count - 1];
double nextX = snakeHead.Position.X;
double nextY = snakeHead.Position.Y;
switch(snakeDirection)
{
    case SnakeDirection.Left:
        nextX -= SnakeSquareSize;
        break;
    case SnakeDirection.Right:
        nextX += SnakeSquareSize;
        break;
    case SnakeDirection.Up:
        nextY -= SnakeSquareSize;
        break;
    case SnakeDirection.Down:
        nextY += SnakeSquareSize;
        break;
}
// Now add the new head part to our list of snake parts...
snakeParts.Add(new SnakePart())
{
    Position = new Point(nextX, nextY),
    IsHead = true
});
//... and then have it drawn!
DrawSnake();
// We'll get to this later...
//DoCollisionCheck();
}

```

With that in place, we now have all the logic needed to create movement for the snake. Notice how we constantly use the `SnakeSquareSize` constant in all aspects of the game, from drawing the background checkerboard pattern to creating and adding to the snake.

### 1.23.3.3. Summary

From the first article, we now have a background and from this article, we have the code to draw and move

the snake. But even with this logic in place, there's still no actual movement or even an actual snake on the game area, because we haven't called any of these methods yet.

The call-to-action for the movement of the snake must come from a repeating source, because the snake should be constantly moving as long as the game is running - in WPF, we have the **DispatcherTimer** class which will help us with just that. The continuous movement of the snake, using a timer, will be the subject for the next article.

## 1.23.4. Continuous movement with DispatcherTimer

In the previous articles, we created a nice looking game area for our Snake and then we added code to perform the actual creation and movement of the snake. However, as already mentioned, the code for movement is not something that should be called just once - instead, it should be called again and again, to keep the snake moving as long as the game is running. In other words, we'll need a Timer.

Generally speaking, in programming, a Timer is usually a mechanism that will allow for a task to be repeated again and again, based on an interval. In other words, each time the timer "ticks", a piece of code is executed, and the timer ticks based on the defined interval. This is exactly what we need to keep our snake moving, so we'll add a **DispatcherTimer** to our Window:

```
public partial class SnakeWPFSample : Window
{
    private System.Windows.Threading.DispatcherTimer gameTickTimer =
new System.Windows.Threading.DispatcherTimer();
    ...
}
```

With that in place, we now need to subscribe to its one and only event: The **Tick** event. We'll do it in the constructor of the Window:

```
public SnakeWPFSample()
{
    InitializeComponent();
    gameTickTimer.Tick += GameTickTimer_Tick;
}
```

And here's the implementation of the event:

```
private void GameTickTimer_Tick(object sender, EventArgs e)
{
    MoveSnake();
}
```

So, each time the timer ticks, the Tick event is called, which in return calls the MoveSnake() method that we implemented previously. To finally see the result of all our hard labor and have a visual, moving snake, we basically just have to create the initial snake parts and then start the timer. We'll create a method called **StartNewGame()**, which we'll use for starting both the first game as well as any number of additional new games when the player dies. We'll start with a very basic version of it though, and then I will expand it with more functionality as we move along - for now, let's just get this snake moving!

First step is to add yet another set of constants, which we'll use to start the new game:

```
public partial class SnakeWPFSample : Window
```

```
{
const int SnakeSquareSize = 20;
const int SnakeStartLength = 3;
const int SnakeStartSpeed = 400;
const int SnakeSpeedThreshold = 100;
.....
```

Only the first three constants are used at this point, to control size, length and start speed of the Snake. We'll use the `SnakeSpeedThreshold` later, but for now, let's add a simple implementation of the `StartNewGame()` method as promised:

```
private void StartNewGame( )
{
    snakeLength = SnakeStartLength;
    snakeDirection = SnakeDirection.Right;
    snakeParts.Add(new SnakePart() { Position = new Point(SnakeSquareSize *
5, SnakeSquareSize * 5) });
    gameTickTimer.Interval = TimeSpan.FromMilliseconds(SnakeStartSpeed);

    // Draw the snake
    DrawSnake();

    // Go!
    gameTickTimer.IsEnabled = true;
}
```

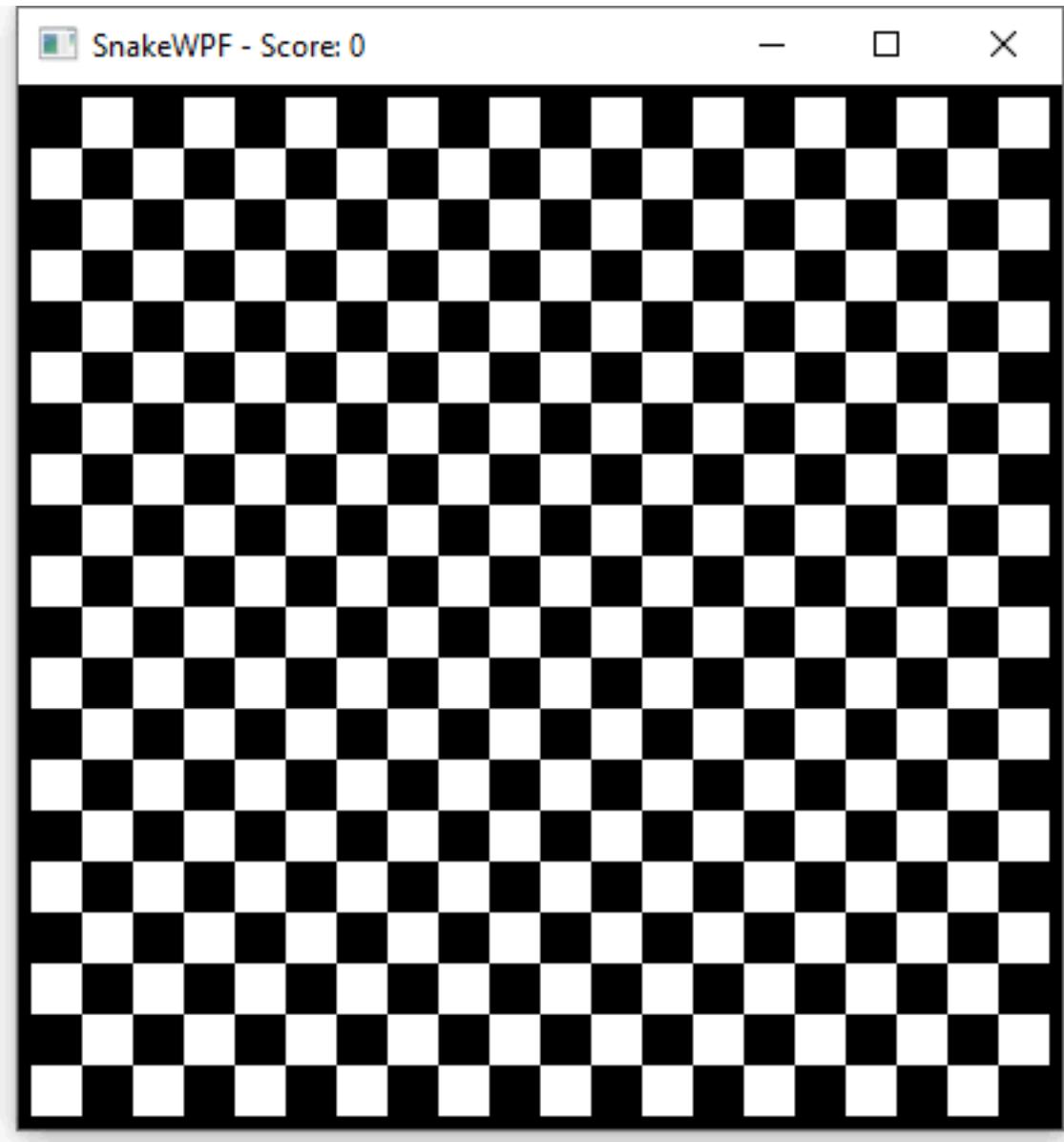
We start off by setting the `snakeLength` and `snakeDirection` based on initial values. Then we add a single part to the `snakeParts` List (more on that later), giving it a nice start position for moving right - we'll once again use the `SnakeSquareSize` constant to help calculate the proper position. With that in place, we can draw the snake by calling the `DrawSnake()` method and then enable the timer, which will basically start the movement of the snake.

We're now finally at the point where we can almost sit back and enjoy the very first version of something that actually looks like a game - in fact, all we have to do now is to call the `StartNewGame()` method. This should of course be done when the user is ready, but for now, to check that everything works, we'll simply do it as soon as everything else is initialized - we'll once again rely on the `ContentRendered` event of the `Window`, which we added in one of the first articles. Simply add a call to our `StartNewGame()` method and we're finally ready to compile and run:

```
private void Window_ContentRendered(object sender, EventArgs e)
{
    DrawGameArea();
}
```

```
StartNewGame( );  
}
```

If you did everything as described, you should now be able to start the game and see the snake being created and immediately start moving:



Notice how the snake appears from out of nothing, as a single square, and then grows to a length of three squares. That happens because we only add one part to the **snakeParts** list, but each time the **MoveSnake()** method is called by the timer, a new part is added (to make it grow), while only removing tail-parts if the current length is about to exceed the desired length of the snake, which starts at 3 (decided by the **SnakeStartLength** constant).

#### 1.23.4.1. Summary

We now have a moving snake, which is really awesome! But as you can see from the animated image above, there are still things to be done - there's no food for the snake to eat, and when the snake hits the

wall, nothing happens. We'll work on these aspects in the next articles.

## 1.23.5. Adding food for the Snake

At this stage of the SnakeWPF article series, we now have a checkerboard background as the game area, as well as a lovely looking green snake moving around it. However, as mentioned in the introduction, the purpose of the game is for the snake to eat some food - in our version it will be red apples!

So now it's time to start adding some food to the game area. We'll do it by randomly adding a red circle somewhere within the boundaries of the GameArea Canvas, but we need to make sure that we are not placing it in one of the squares already occupied by the constantly growing snake. In other words, one of the most important aspects of placing an apple on the game area is the code that decides the next position. Here's the code we'll use for doing just that:

```
private Point GetNextFoodPosition()
{
    int maxX = (int)(GameArea.ActualWidth / SnakeSquareSize);
    int maxY = (int)(GameArea.ActualHeight / SnakeSquareSize);
    int foodX = rnd.Next(0, maxX) * SnakeSquareSize;
    int foodY = rnd.Next(0, maxY) * SnakeSquareSize;

    foreach(SnakePart snakePart in snakeParts)
    {
        if((snakePart.Position.X == foodX) && (snakePart.Position.Y ==
foodY))
            return GetNextFoodPosition();
    }

    return new Point(foodX, foodY);
}
```

Be sure to also add this line in the top of the Window class declaration, along with the rest of the fields/constants:

```
public partial class SnakeWPFSample : Window
{
    private Random rnd = new Random();
    ....
```

So, to quickly explain the code: We once again use the **SnakeSquareSize** constant to help us calculate the next position for our food, in combination with the Random class, which will give us a random X and Y position. Once we have it, we run through all the current snake parts and check if their position matches the X and Y coordinates we just created - if they do, it means that we have hit an area currently occupied by the snake and then we ask for a new position by simply calling the method again (making this a recursive method).

This also means that this method can call itself an unlimited amount of times and, in theory, result in an endless loop. We could do some checking for that, but it shouldn't be necessary, since it would require that the snake was so long that no empty space was left - my bet is that the game will have ended before this happens.

With that in place, we're ready to add the code which will add the food at the newly calculated position - we'll be doing it from a method called **DrawSnakeFood()**. Thanks to all the work already handled by **GetNextFoodPosition()**, it's pretty simple, but first, be sure to declare the field used to save a reference to the food, as well as the SolidColorBrush used to draw the apple, along with the other field/constant declarations:

```
public partial class SnakeWPFSSample : Window
{
    private UIElement snakeFood = null;
    private SolidColorBrush foodBrush = Brushes.Red;
    ....
```

Here's the implementation of the method:

```
private void DrawSnakeFood()
{
    Point foodPosition = GetNextFoodPosition();
    snakeFood = new Ellipse()
    {
        Width = SnakeSquareSize,
        Height = SnakeSquareSize,
        Fill = foodBrush
    };
    GameArea.Children.Add(snakeFood);
    Canvas.SetTop(snakeFood, foodPosition.Y);
    Canvas.SetLeft(snakeFood, foodPosition.X);
}
```

As promised, it's very simple - as soon as we have the position, we simply create a new Ellipse instance and once again we use the **SnakeSquareSize** constant to make sure that it has the same size as the background tiles as well as each snake part. We save a reference to the Ellipse instance in the **snakeFood** field, because we need it later on.

With that in place, we really just need to call the **DrawSnakeFood()** method to see the result of our work. This will be done in two situations: At the beginning of the game and when the snake "eats" the food (more on that later). For now, let's add a call to it in our **StartNewGame()** method:

```
private void StartNewGame()
```

```

{
    snakeLength = SnakeStartLength;
    snakeDirection = SnakeDirection.Right;
    snakeParts.Add(new SnakePart() { Position = new
Point(SnakeSquareSize * 5, SnakeSquareSize * 5) });
    gameTickTimer.Interval =
TimeSpan.FromMilliseconds(SnakeStartSpeed);

    // Draw the snake and the snake food
    DrawSnake();
    DrawSnakeFood();

    // Go!
    gameTickTimer.IsEnabled = true;
}

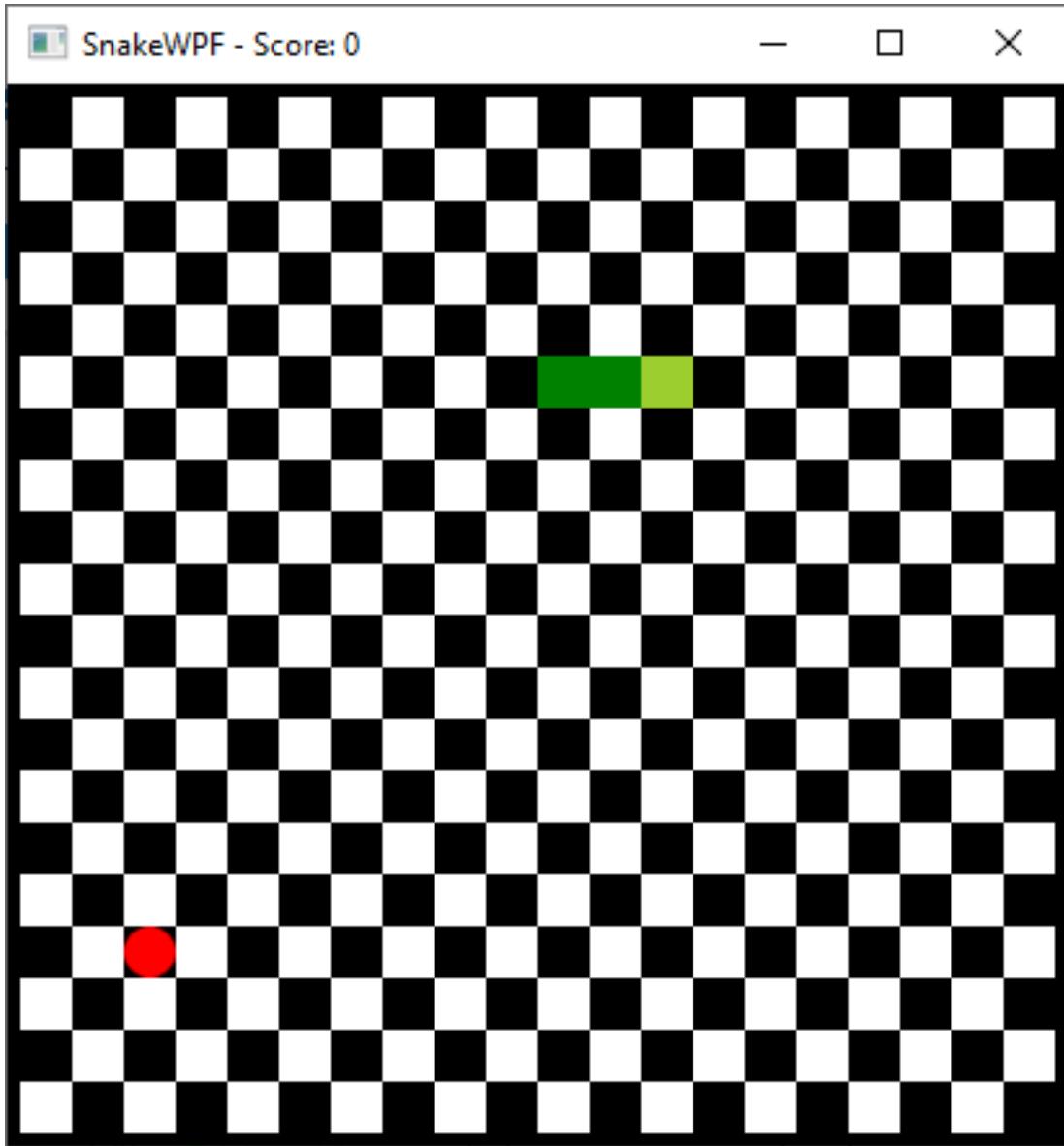
```

That's it! If you run the game now, you should see that the snake finally has some food to chase:

Quite a bit of work to place a red dot on the screen, right?

#### 1.23.5.1. Summary

In this article, we have finally added some food to the table for the snake to catch, but there's still work to be done: We need to be able to control the snake, and we need to know when it hits something (a wall, its own tail or the food). More on that in the next article.



### 1.23.6. Controlling the Snake

At this point in our article series on SnakeWPF, we now have a nice looking background and a moving snake. Unfortunately, the snake just moves in one direction until it leaves the area. We obviously need to add some code so that we can control the snake with the keyboard.

Most WPF controls have events for receiving input from the mouse and keyboard. So, depending on where you want to check the input, you can subscribe to these events for one or several controls and then perform the magic there. However, since this is a game, we want to catch keyboard input no matter where the focus might be, so we'll simply subscribe to the event directly on the Window.

For what we want to accomplish, the **KeyUp** event is a great match. So, find your XAML file for the Window and modify the Window tag so that it includes the KeyUp event like this:

```
<Window x:Class="WpfTutorialSamples.Games.SnakeWPFSample"
    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.Games"
mc:Ignorable="d"
Title="SnakeWPF - Score: 0" SizeToContent="WidthAndHeight"
ContentRendered="Window_ContentRendered" KeyUp="Window_KeyUp">>

```

In your Code-behind, add the **Window\_KeyUp** event handler, like this:

```

private void Window_KeyUp(object sender, KeyEventArgs e)
{
    SnakeDirection originalSnakeDirection = snakeDirection;
    switch(e.Key)
    {
        case Key.Up:
            if(snakeDirection != SnakeDirection.Down)
                snakeDirection = SnakeDirection.Up;
            break;
        case Key.Down:
            if(snakeDirection != SnakeDirection.Up)
                snakeDirection = SnakeDirection.Down;
            break;
        case Key.Left:
            if(snakeDirection != SnakeDirection.Right)
                snakeDirection = SnakeDirection.Left;
            break;
        case Key.Right:
            if(snakeDirection != SnakeDirection.Left)
                snakeDirection = SnakeDirection.Right;
            break;
        case Key.Space:
            StartNewGame();
            break;
    }
    if(snakeDirection != originalSnakeDirection)
        MoveSnake();
}

```

The first thing we do is to save a reference to the current direction that the snake is going in - we need this to make sure that the player is not trying to do something we won't allow, like reversing back over the snake

body (e.g. from right to left). In other words, if the snake is moving vertically and the player wants to change the direction, it has to go horizontally first - you can't go directly from up to down or from left to right.

Next up is a **switch** statement, where we check which key was pressed. Here we check if one of the arrow keys (**Up**, **Down**, **Left**, **Right**) were pressed - if so, they are allowed to change the direction of the snake, unless the change is physically impossible, as described above. Also notice that I have added a check for the **Space** key: It will call the `StartNewGame()` method, to allow the player to choose when the game is started, instead of just starting it automatically. It will also allow the player to start a new game when the previous game has ended.

At the end of the method, we check if the direction has changed in comparison to the original direction - if it has, we call the **MoveSnake()** method, so that the change is reflected immediately.

We previously added a call to the `StartNewGame()` method in the `Window_ContentRendered` event - you can now remove this and instead start the game by pressing the Space key. Now lo and behold, the snake can be controlled - it's now close to being an actual game and not just an animated snake!

#### 1.23.6.1. Summary

In this article, we added some pretty important functionality to our SnakeWPF implementation: Control of the snake! However, if you try to play it, you will notice that we still need one very important aspect, because the snake is currently disappearing out of the wall boundaries and it refuses to eat the food even when you hit it. In other words, we need to add some **collision detection**! More on that in the next article.

## 1.23.7. Collision Detection

Now that we have implemented the game area, the food and the snake, as well as continuous movement of the snake, we only need one final thing to make this look and act like an actual game: **Collision detection**. The concept evolves around constantly checking whether our Snake just hit something and we currently need it for two purposes: To see if the Snake just ate some food or if it hit an obstacle (the wall or its own tail).

### 1.23.7.1. The DoCollisionCheck() method

The collision detection will be performed in a method called **DoCollisionCheck()**, so we need to implement that. Here's how it currently should look:

```
private void DoCollisionCheck()
{
    SnakePart snakeHead = snakeParts[snakeParts.Count - 1];

    if((snakeHead.Position.X == Canvas.GetLeft(snakeFood)) &&
(snakeHead.Position.Y == Canvas.GetTop(snakeFood)))
    {
        EatSnakeFood();
        return;
    }

    if((snakeHead.Position.Y < 0) || (snakeHead.Position.Y >=
GameArea.ActualHeight) ||
(snakeHead.Position.X < 0) || (snakeHead.Position.X >=
GameArea.ActualWidth))
    {
        EndGame();
    }

    foreach(SnakePart snakeBodyPart in snakeParts.Take(snakeParts.Count
- 1))
    {
        if((snakeHead.Position.X == snakeBodyPart.Position.X) &&
(snakeHead.Position.Y == snakeBodyPart.Position.Y))
            EndGame();
    }
}
```

As promised, we do two checks: First we see if the current position of the snake's head matches the position of the current piece of food. If it does, we call the **EatSnakeFood()** method (more on that later).

We then check if the position of the snake's head exceeds the boundaries of the GameArea, to see if the snake is on its way out of one of the borders. If it is, we call the **EndGame()** method. Finally, we check if the snake's head matches one of the body part positions - if it does, the snake just collided with its own tail, which will end the game as well, with a call to **EndGame()**.

#### 1.23.7.2. The EatSnakeFood() method

The EatSnakeFood() method is responsible for doing a couple of things, because as soon as the snake eats the current piece of food, we need to add a new one, in a new location, as well as adjust the score, the length of the snake and the current game speed. For the score, we need to declare a new local variable called **currentScore**:

```
public partial class SnakeWPFSample : Window
{
    ...
    private int snakeLength;
    private int currentScore = 0;
    ...
}
```

With that in place, add the EatSnakeFood() method:

```
private void EatSnakeFood()
{
    snakeLength++;
    currentScore++;
    int timerInterval = Math.Max(SnakeSpeedThreshold, (int
)gameTickTimer.Interval.TotalMilliseconds - (currentScore * 2));
    gameTickTimer.Interval = TimeSpan.FromMilliseconds(timerInterval);

    GameArea.Children.Remove(snakeFood);
    DrawSnakeFood();
    UpdateGameStatus();
}
```

As mentioned, several things happens here:

- We increment the **snakeLength** and the **currentScore** variables by one to reflect the fact that the snake just caught a piece of food.
- We adjust the **Interval** of **gameTickTimer**, using the following rule: The **currentScore** is multiplied by 2 and then subtracted from the current interval (speed). This will make the speed grow exponentially along with the length of the snake, making the game increasingly difficult. We have previously defined a lower boundary for the speed, with the **SnakeSpeedThreshold** constant, meaning that the game speed

never drops below a 100 ms interval.

- We then remove the piece of food just consumed by the snake and then we call the **DrawSnakeFood()** method which will add a new piece of food in a new location.
- Finally, we call the **UpdateGameStatus()** method, which looks like this:

```
private void UpdateGameStatus()
{
    this.Title = "SnakeWPF - Score: " + currentScore + " - Game speed:
" + gameTickTimer.Interval.TotalMilliseconds;
}
```

This method will simply update the **Title** property of the **Window** to reflect the current score and game speed. This is an easy way of showing the current status, which can easily be extended later on if desired.

#### 1.23.7.3. The **EndGame()** method

We also need a little bit of code to execute when the game should end. We'll do this from the **EndGame()** method, which is currently called from the **DoCollisionCheck()** method. As you can see, it's currently very simple:

```
private void EndGame()
{
    gameTickTimer.IsEnabled = false;
    MessageBox.Show("Oooops, you died!\n\nTo start a new game, just
press the Space bar...", "SnakeWPF");
}
```

Besides showing a message to the user about the unfortunate passing of our beloved snake, we simply stop the **gameTickTimer**. Since this timer is what causes all things to happen in the game, as soon as it's stopped, all movement and drawing also stops.

#### 1.23.7.4. Final adjustments

We're now almost ready with the first draft of a fully-functional Snake game - in fact, we just need to make two minor adjustments. First, we need to make sure that the **DoCollisionCheck()** is called - this should happen as the last action performed in the **MoveSnake()** method, which we implemented previously:

```
private void MoveSnake()
{
    . . .
    //... and then have it drawn!
    DrawSnake();
}
```

```

    // Finally: Check if it just hit something!
    DoCollisionCheck();
}

```

Now collision detection is performed as soon as the snake has moved! Now remember how I told you that we implemented a simple variant of the **StartNewGame()** method? We need to expand it a bit, to make sure that we reset the score each time the game is (re)started, as we as a couple of other things. So, replace the **StartNewGame()** method with this slightly extended version:

```

private void StartNewGame()
{
    // Remove potential dead snake parts and leftover food...
    foreach(SnakePart snakeBodyPart in snakeParts)
    {
        if(snakeBodyPart.UiElement != null)
            GameArea.Children.Remove(snakeBodyPart.UiElement);
    }
    snakeParts.Clear();
    if(snakeFood != null)
        GameArea.Children.Remove(snakeFood);

    // Reset stuff
    currentScore = 0;
    snakeLength = SnakeStartLength;
    snakeDirection = SnakeDirection.Right;
    snakeParts.Add(new SnakePart() { Position = new
Point(SnakeSquareSize * 5, SnakeSquareSize * 5) });
    gameTickTimer.Interval =
TimeSpan.FromMilliseconds(SnakeStartSpeed);

    // Draw the snake again and some new food...
    DrawSnake();
    DrawSnakeFood();

    // Update status
    UpdateGameStatus();

    // Go!
    gameTickTimer.IsEnabled = true;
}

```

When a new game starts, the following things now happens:

- Since this might not be the first game, we need to make sure that any potential leftovers from a previous game is removed: This includes all existing parts of the snake, as well as leftover food.
- We also need to reset some of the variables to their initial settings, like the `<strong>score</strong>`, the `<strong>length</strong>`, the `<strong>direction</strong>` and the `<strong>speed</strong>` of the timer. We also add the initial snake head (which will be automatically expanded by the `MoveSnake()` method).
- We then call the `<strong>DrawSnake()</strong>` and `<strong>DrawSnakeFood()</strong>` methods to visually reflect that a new game was started.
- Then we call the `UpdateGameStatus()` method.
- And finally, we're ready to start the `<strong>gameTickTimer</strong>` - it will immediately start ticking, basically setting the game in motion.

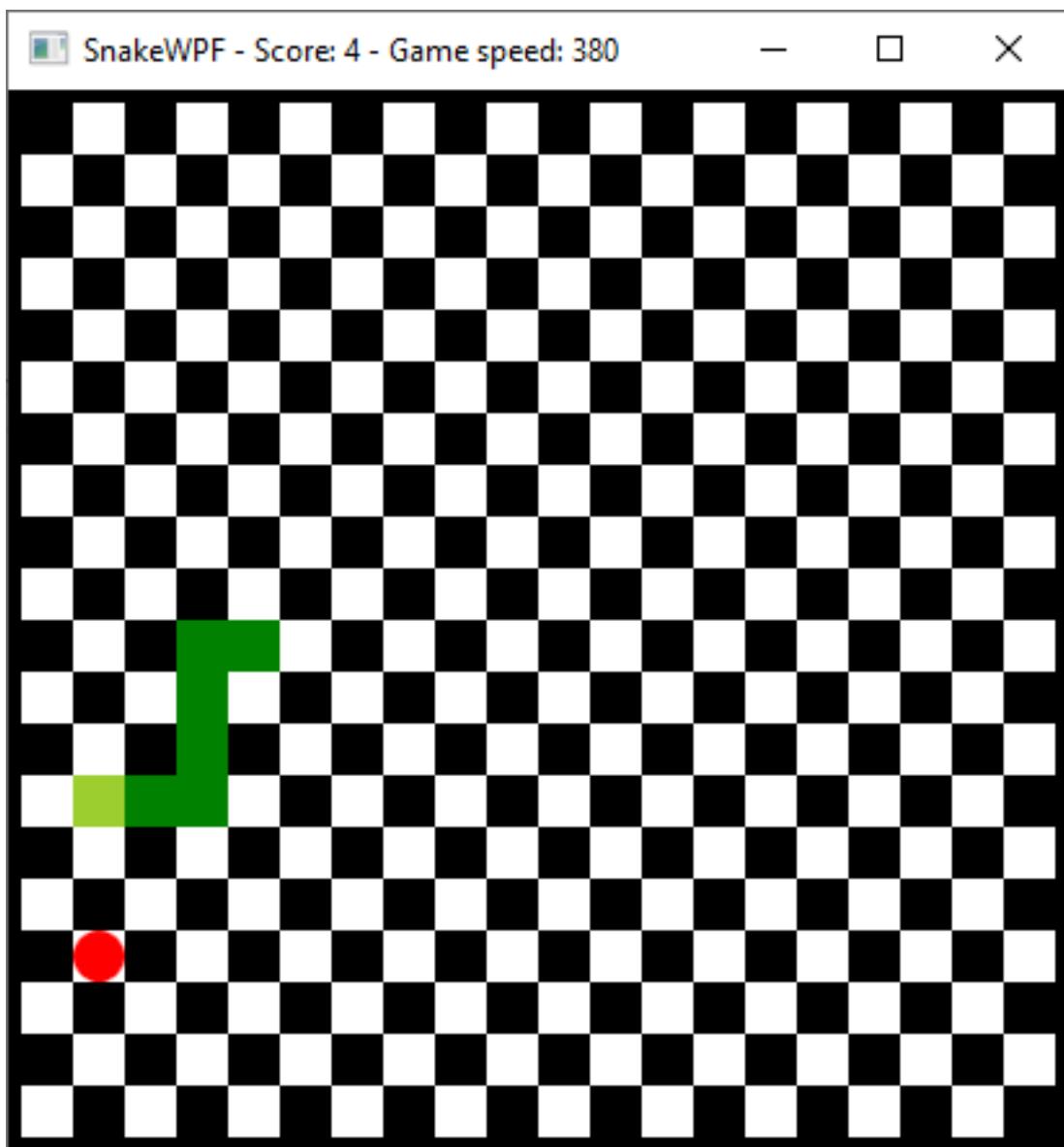
#### 1.23.7.5. Summary

If you made it all the way through this article series: **congratulations - you just built your first WPF game!** Enjoy all your hard labor by running your project, pressing the **Space** key and start playing - even in this very simple implementation, Snake is a fun and addictive game!

## 1.23.8. Improving SnakeWPF: Making it look more like a game

During the last several articles, we've built a cool Snake game in WPF. We have implemented all the game-mechanics and the result is a fully functional game. However, there are definitely a lot of improvements that could be made, because the current implementation is very bare-minimum. So, in the next articles, I will be making several improvements to our SnakeWPF game - in this article, I will be focusing on making our game look more like an actual game!

As it looks now, with its default Windows-style border/title bar, our implementation doesn't look much like a game. However, we have previously needed the title bar to display score/speed information, and as a nice bonus, we automatically got the default Windows buttons for minimizing/maximizing/closing the Window:



At this point, I would like to completely remove the default Windows title bar and instead implement our own top status bar, which should show the current score and speed, as well as a customized close button. All of it should match the current look of the game. Fortunately for us, this is quite easy to accomplish with WPF.

### 1.23.8.1. Adding a custom title bar

The first step is to add a couple of properties and a new event to the Window declaration. It should now look like this:

```
<Window x:Class="WpfTutorialSamples.Games.SnakeWPFSample"
    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.Games"
            mc:Ignorable="d"
                Title="SnakeWPF - Score: 0" SizeToContent="WidthAndHeight"
ContentRendered="Window_ContentRendered" KeyUp="Window_KeyUp"
                    ResizeMode="NoResize" WindowStyle="None" Background="Black"
MouseDown="Window_MouseDown">
```

The changes are all in the last line. We set the **ResizeMode** to **NoResize** and the **WindowStyle** to **None**. This will completely remove the title bar as well as any default borders around the Window - that's no problem for us though, because the main area of our game already has a 5 px black border.

You will also notice that I have subscribed to a new event - the **MouseDown** event. The reason is that since we lose the default title bar, there's no longer any way for the user to drag the game from one point of the screen to another. Fortunately for us, it's easy to re-create this behavior, e.g. on our own, custom title bar. However, since it doesn't look like the regular title bar, the user might be confused about where to drag, so I decided simply to make the entire window surface draggable. So, in your Code-behind, define the **Window\_MouseDown** event handler like this:

```
private void Window_MouseDown(object sender, MouseButtonEventArgs e)
{
    this.DragMove();
}
```

With that in place, your window can be dragged around no matter where you use the mouse. The next step is to add our custom title bar, which should display the score and speed, as well as a close button. The inner part of the Window XAML should now look like this:

```
<DockPanel Background="Black">
<Grid DockPanel.Dock="Top" Name="pnlTitleBar">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
```

```

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

    <Grid.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontFamily" Value="Consolas" />
            <Setter Property="Foreground" Value="White" />
            <Setter Property="FontSize" Value="24" />
            <Setter Property="FontWeight" Value="Bold" />
        </Style>
    </Grid.Resources>

    <WrapPanel Margin="10,0,0,0">
        <TextBlock>Score:</TextBlock>
        <TextBlock Name="tbStatusScore">0</TextBlock>
    </WrapPanel>
    <WrapPanel Grid.Column="1">
        <TextBlock>Speed:</TextBlock>
        <TextBlock Name="tbStatusSpeed">0</TextBlock>
    </WrapPanel>
    <Button Grid.Column="2" DockPanel.Dock="Right" Background
="Transparent" Foreground="White" FontWeight="Bold" FontSize="20"
BorderThickness="0" Name="btnClose" Click="BtnClose_Click" Padding
="10,0">X</Button>
</Grid>
<Border BorderBrush="Black" BorderThickness="5">
    <Canvas Name="GameArea" ClipToBounds="True" Width="400" Height
="400">

        </Canvas>
    </Border>
</DockPanel>
```

And don't forget to define the **BtnClose\_Click** event handler:

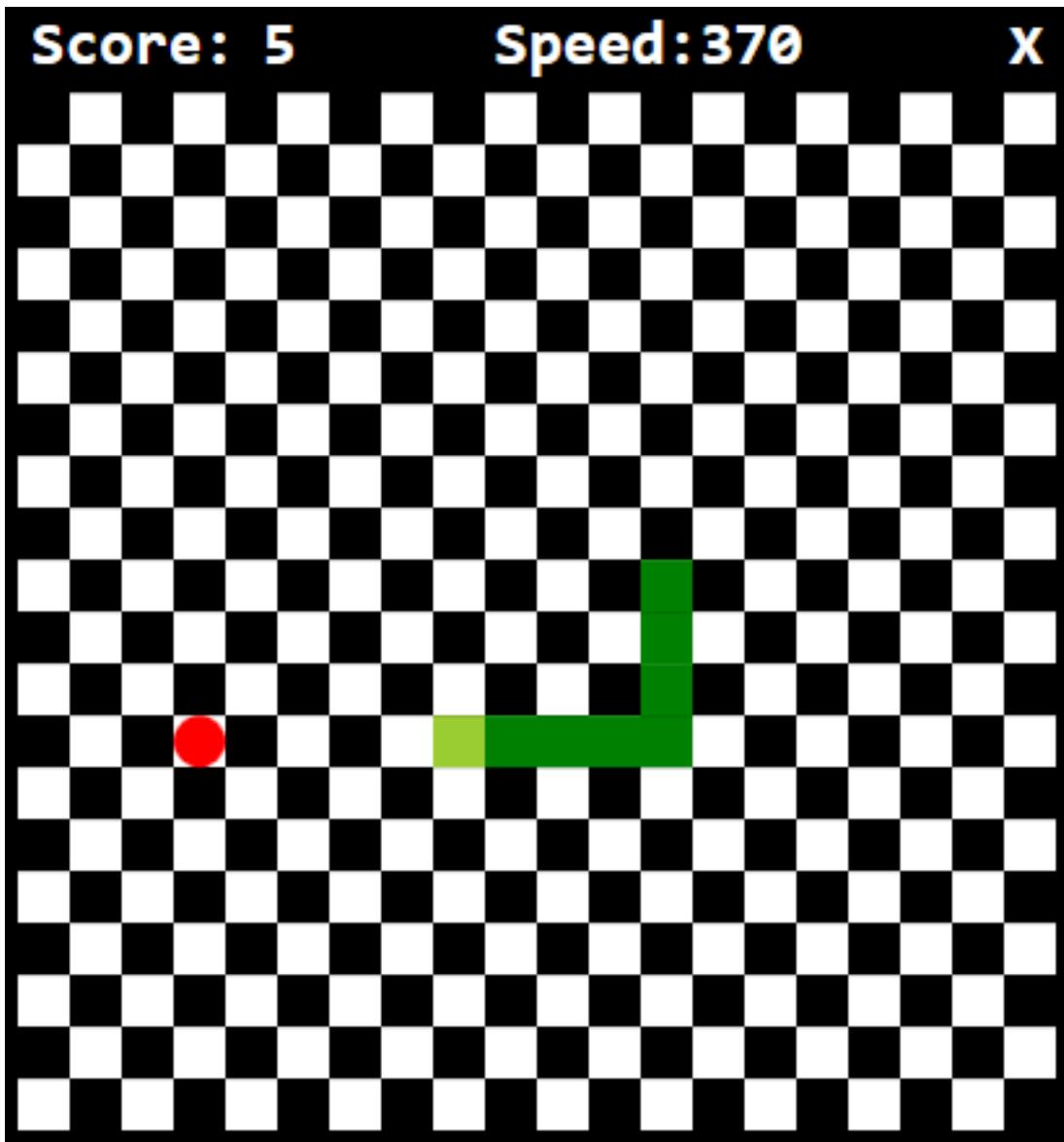
```

private void BtnClose_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

We previously implemented a method called **UpdateGameStatus()**, which updated the Title property of the Window - this method should be changed to use the new TextBlock's:

```
private void UpdateGameStatus()
{
    this.tbStatusScore.Text = currentScore.ToString();
    this.tbStatusSpeed.Text =
gameTickTimer.Interval.TotalMilliseconds.ToString();
}
```

I will tell you all about what we just did, but first, let's check out how the game looks now:



It looks quite a bit cooler, right? But let's discuss what we just did: As you can see, the original **Border** control with the **GameAreaCanvas** inside of it has now been surrounded by a **DockPanel**. This makes it easy for us to attach our new title bar, in the form of a **Grid** panel, to the top of the Window.

The Grid uses several cool WPF techniques which have been discussed elsewhere in this tutorial: We use **ColumnDefinition**'s to divide the area into two equally sized areas (for the score and speed), plus an auto-sized third column for the close button. You will also notice that we use WPF **Style**'s to apply the same visual look to all the **TextBlock** controls - the same custom font, font size, color and weight are applied to all of them, thanks to a Style defined in the Grid, targeting **TextBlock** controls.

Also notice how easy it is to customize the **Button** control used for closing the window, to completely match the rest of the look and feel of the game, simply by using the standard properties - WPF is so flexible!

### 1.23.8.2. Summary

In this article, we have made our SnakeWPF implementation look a lot more like a game, by removing the standard Windows-look and applying our own custom title bar. In upcoming articles, we'll make even more improvements!

## 1.23.9. Improving SnakeWPF: Adding a high score list

In the previous article, we made a lot of visual improvements to our SnakeWPF implementation. In this article, I would like to add a very cool feature: **A high score list!** On top of that, I would like to make the game a bit more user-friendly, by adding a welcome screen. I'll also be replacing the very non-gamey "You died"-messagebox with an in-game screen.

We need quite a bit of extra code and markup for this, but let's start with the easy part - the XAML!

### 1.23.9.1. XAML

The first thing I would like to do, is to add a bunch of XAML to the Snake window. This will mainly consists of 4 new containers (in this case Border controls), which will host a range of child controls to support various situations:

- One container for displaying a welcome message when the game starts, informing about the controls to be used etc.
- One container for displaying the high score list
- One container to display when the user has beaten one of the high scores, including a TextBox for entering a name
- One container to display when the user dies, but hasn't made it into the high score list (replacing the boring MessageBox we previously used)

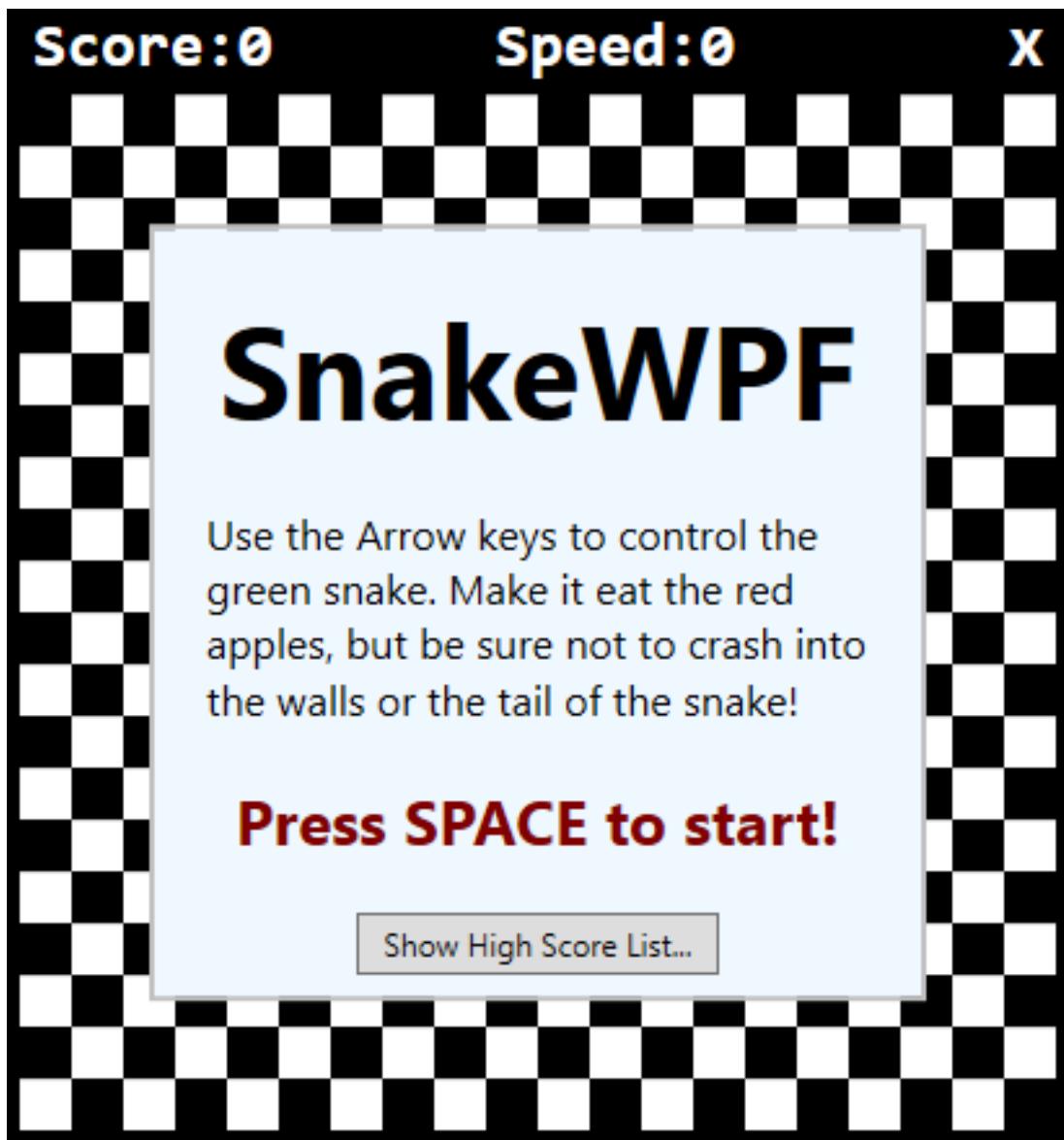
We will add these containers to the **GameArea** Canvas and then simply hide them when we don't need them. As mentioned, each container will serve a different purpose and include quite a bit of markup, but we only use WPF controls which have already been discussed in this tutorial.

#### 1.23.9.2. Welcome message

Add this piece of XAML inside the GameArea Canvas control:

```
<Border BorderBrush="Silver" BorderThickness="2" Width="300" Height="300" Canvas.Left="50" Canvas.Top="50" Name="bdrWelcomeMessage" Panel.ZIndex="1">
    <StackPanel Orientation="Vertical" Background="AliceBlue">
        <TextBlock FontWeight="Bold" FontSize="50" HorizontalAlignment="Center" Margin="0,20,0,0">SnakeWPF</TextBlock>
        <TextBlock TextWrapping="Wrap" Margin="20" FontSize="16">Use the Arrow keys to control the green snake. Make it eat the red apples, but be sure not to crash into the walls or the tail of the snake!</TextBlock>
        <TextBlock FontWeight="Bold" HorizontalAlignment="Center" FontSize="24" Foreground="Maroon">Press SPACE to start!</TextBlock>
        <Button Margin="20" Name="btnShowHighscoreList" Click="BtnShowHighscoreList_Click" HorizontalAlignment="Center" Padding="5,5,5,5">View Highscore List</Button>
    </StackPanel>
</Border>
```

```
= "10,3">Show High Score List...</Button>
</StackPanel>
</Border>
```



It briefly tells the user what the game is all about, how the Snake is controlled and how to start the game. The Border, which holds all the content, is initially visible, so this will be the first thing the user meets when the game starts. In the bottom of the screen, I've added a Button for displaying the high score list (which we'll add in just a minute). The Click event handler will be implemented in the Code-behind later.

#### 1.23.9.3. High score list

Now it gets a bit more complex, because I want to do this the WPF-way and use data-binding to display the high score list instead of e.g. manually building and updating the list. But don't worry, I'll explain it all as we move along. First, add this piece of XAML inside the GameArea Canvas, just like we did before - the Canvas will, as mentioned, hold all our Border controls, each of them offering their own piece of functionality for our game:

```

<Border BorderBrush="Silver" BorderThickness="2" Width="300" Height="300" Canvas.Left="50" Canvas.Top="50" Name="bdrHighscoreList" Panel.ZIndex="1" Visibility="Collapsed">
    <StackPanel Orientation="Vertical" Background="AliceBlue">
        <Border BorderThickness="0,0,0,2" BorderBrush="Silver" Margin="0,10">
            <TextBlock HorizontalAlignment="Center" FontSize="34" FontWeight="Bold">High Score List</TextBlock>
        </Border>
        <ItemsControl ItemsSource="{Binding Source={StaticResource HighScoreListViewSource}}">
            <ItemsControl.ItemTemplate>
                <DataTemplate>
                    <DockPanel Margin="7">
                        <TextBlock Text="{Binding PlayerName}" DockPanel.Dock="Left" FontSize="22"></TextBlock>
                        <TextBlock Text="{Binding Score}" DockPanel.Dock="Right" FontSize="22" HorizontalAlignment="Right"></TextBlock>
                    </DockPanel>
                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>
    </StackPanel>
</Border>

```

Notice how this Border is initially not displayed (Visibility = Collapsed). We use an ItemsControl (we talked about this previously in this tutorial), with a custom ItemsSource called **HighScoreListViewSource**. We will be using a CollectionViewSource to make sure that the collection we bind to is always sorted properly. We need to define this resource in the Window XAML, so as a child to the Window tag, you should add this piece of markup, making your Window declaration look like this:

```

<Window x:Class="WpfTutorialSamples.Games.SnakeWPFSample"
        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.Games"

```



```
mc:Ignorable="d"
x:Name="window"
xmlns:scm="clr-
namespace:System.ComponentModel;assembly=WindowsBase"
Title="SnakeWPF - Score: 0" SizeToContent="WidthAndHeight"
ContentRendered="Window_ContentRendered" KeyUp="Window_KeyUp"
ResizeMode="NoResize" WindowStyle="None" Background="Black"
MouseDown="Window_MouseDown">

<Window.Resources>
    <CollectionViewSource Source="{Binding ElementName=window,
Path=HighscoreList}" x:Key="HighScoreListViewSource">
        <CollectionViewSource.SortDescriptions>
            <scm:SortDescription Direction="Descending"
PropertyName="Score" />
```

```

    </CollectionViewSource.SortDescriptions>
</CollectionViewSource>
</Window.Resources>
.....

```

Notice that I sneaked in a new reference: The **xmlns:scm**, used to access the SortDescription type. I also added the x:Name property and set it to **window**, so that we can reference members defined on the MainWindow class in Code-behind.

In the Window.Resources, I have added a new **CollectionViewSource**. It uses binding to attach to a property called **HighscoreList**, which we will define in Code-behind. Notice also that I add a SortDescription to it, specifying that the list should be sorted descendent by a property called Score, basically meaning that the highest score will be displayed first and so on.

In Code-behind, we need to define the property called **HighscoreList**, which the ItemsSource relies on, but we'll get to that after we're done adding the last XAML.

#### 1.23.9.4. New high score

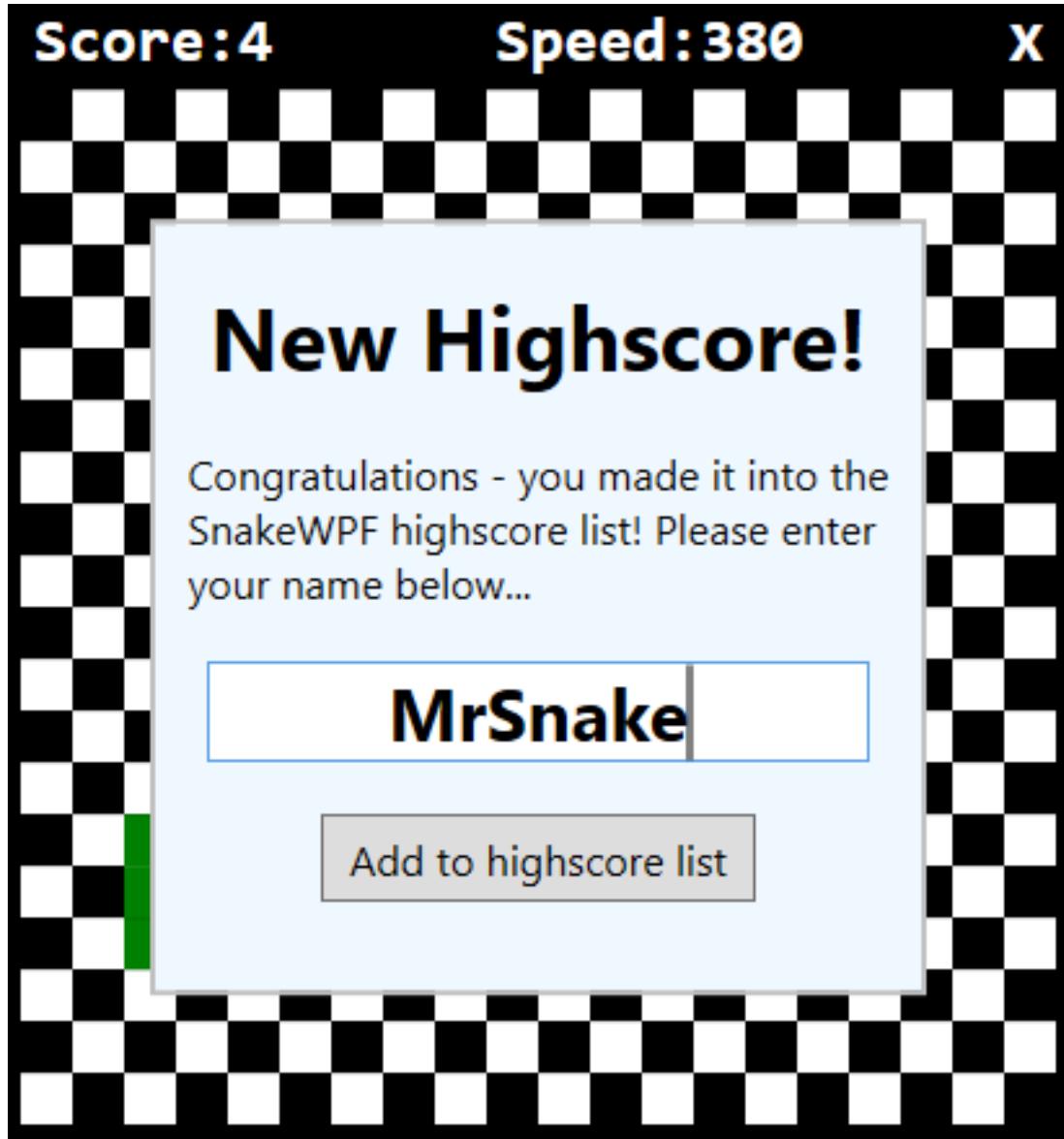
When the user beats an existing high score, we will display a nicely looking message about it. The XAML looks like this, and once again, it should be added inside the GameArea Canvas:

```

<Border BorderBrush="Silver" BorderThickness="2" Width="300" Height
="300" Canvas.Left="50" Canvas.Top="50" Name="bdrNewHighscore"
Panel.ZIndex="1" Visibility="Collapsed">
<StackPanel Orientation="Vertical" Background="AliceBlue">
    <TextBlock HorizontalAlignment="Center" FontSize="34" FontWeight
="Bold" Margin="20">New Highscore!</TextBlock>
    <TextBlock HorizontalAlignment="Center" TextWrapping="Wrap"
FontSize="16">
        Congratulations - you made it into the SnakeWPF highscore
list! Please enter your name below...
    </TextBlock>
    <TextBox Name="txtPlayerName" FontSize="28" FontWeight="Bold"
MaxLength="8" Margin="20" HorizontalContentAlignment="Center"></TextBox>

    <Button Name="btnAddToHighscoreList" FontSize="16"
HorizontalAlignment="Center" Click="BtnAddToHighscoreList_Click" Padding
="10,5">Add to highscore list</Button>
</StackPanel>
</Border>

```



All very simple with some text, a TextBox for entering the name and a Button to click to add to the list - we'll define the **BtnAddToHighscoreList\_Click** event handler later.

#### 1.23.9.5. "Oh no - you died!"

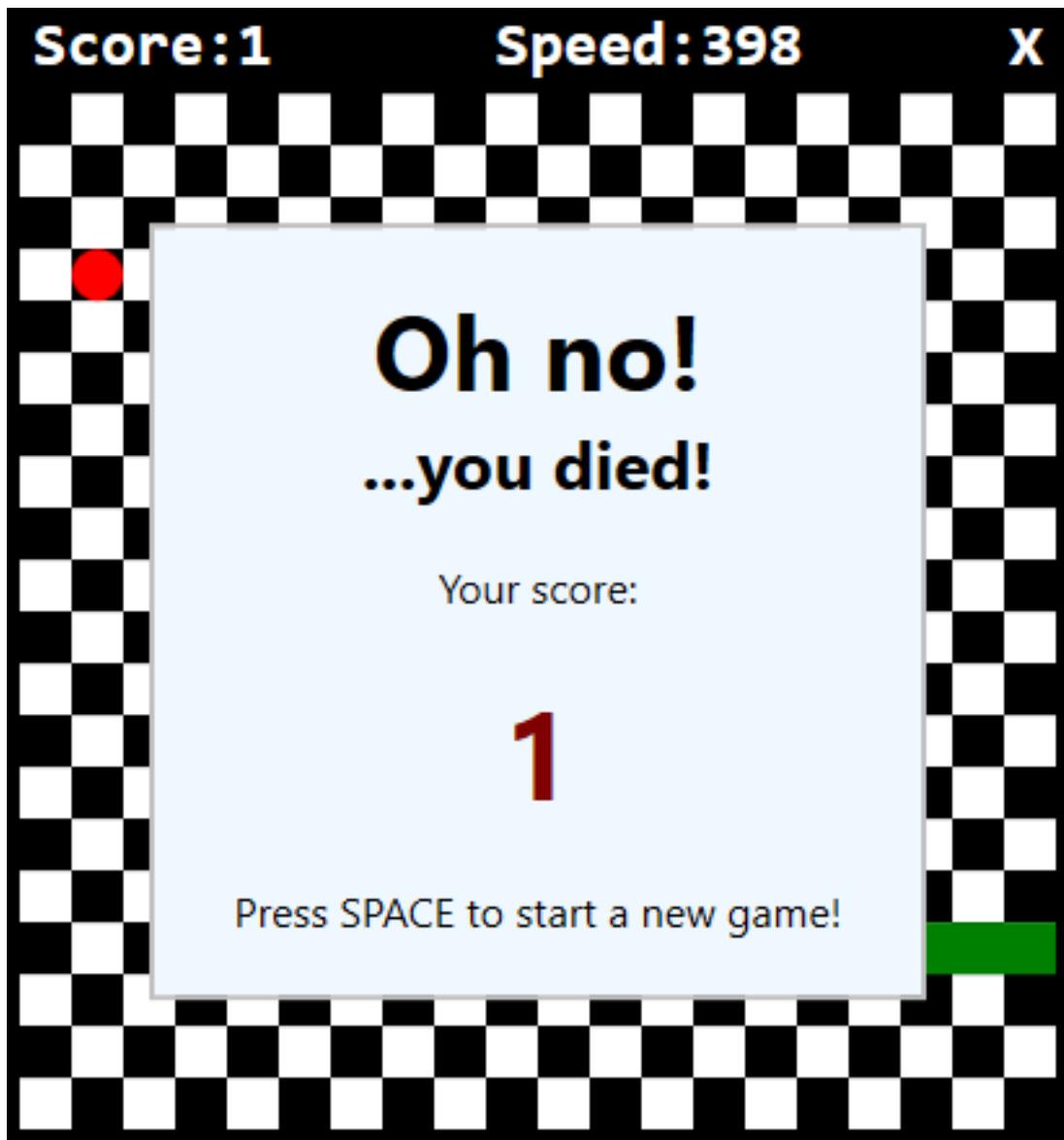
The last part is the "Oh no, you died and you didn't make it into the high score list" screen, which we'll use to replace the boring MessageBox which did the same thing previously. The XAML looks like this:

```
<Border BorderBrush="Silver" BorderThickness="2" Width="300" Height="300" Canvas.Left="50" Canvas.Top="50" Name="bdrEndOfGame" Panel.ZIndex="1" Visibility="Collapsed">
    <StackPanel Orientation="Vertical" Background="AliceBlue">
        <TextBlock HorizontalAlignment="Center" FontSize="40" FontWeight="Bold" Margin="0,20,0,0">Oh no!</TextBlock>
        <TextBlock HorizontalAlignment="Center" FontSize="26" FontWeight="Bold">...you died!</TextBlock>
        <TextBlock Margin="20" TextAlignment="Center" FontSize="16">
```

```

Your score: </TextBlock>
    <TextBlock Name="tbFinalScore" TextAlignment="Center" FontSize="48" FontWeight="Bold" Foreground="Maroon">0</TextBlock>
        <TextBlock TextAlignment="Center" FontSize="16" Margin="20">
Press SPACE to start a new game!</TextBlock>
    </StackPanel>
</Border>

```



It informs the user about the unfortunate turn of events, displays the final score and tells the user how to start a new game - pretty simple!

#### 1.23.9.6. Code-behind

With all the XAML in place, we're finally ready to implement the Code-behind stuff! First, we need to implement the event handlers we defined in the XAML. Here's the one for the "Show high score list" button:

```

private void BtnShowHighscoreList_Click(object sender, RoutedEventArgs
e)
{
    bdrWelcomeMessage.Visibility = Visibility.Collapsed;
    bdrHighscoreList.Visibility = Visibility.Visible;
}

```

Quite simple, as you can see - when the button is clicked, we hide the welcome message and then we display the high score list - we'll add that now.

#### 1.23.9.7. Implementing the high score list

The other event handler we have relates to adding a new entry to the high score list, but for that, we need a couple of other additions - first of all, the actual property for holding the high score entries:

```

public ObservableCollection<SnakeHighscore> HighscoreList
{
    get; set;
} = new ObservableCollection<SnakeHighscore>();

```

As you can see, this is an **ObservableCollection**, holding the type **SnakeHighscore**. First, be sure to include the namespace holding the ObservableCollection type:

```
using System.Collections.ObjectModel;
```

Then implement the **SnakeHighscore** class:

```

public class SnakeHighscore
{
    public string PlayerName { get; set; }

    public int Score { get; set; }
}

```

Quite a simple class, as you can see - it just serves as a container for the name and score of the player who made it into the high score list.

#### 1.23.9.8. Loading/saving the high score list

We also need some code to load and save the list - the **Save** method will be called when a new entry is added to the list, while the **Load** method is called when our game starts. I'll be using a simple XML file to hold the list, which will allow us to use the built-in **XmlSerializer** class to automatically load and save the list.

There are MANY ways of loading/saving data, and several other relevant formats like JSON or even a plain text file, but I wanted to keep this part in as few lines of code as possible, since it's not that relevant for a WPF tutorial. Also, the XmlSerializer approach makes the code pretty flexible - you can easily add new properties to the SnakeHighscore class and they will be automatically persisted. Here's the **LoadHighscoreList()** method:

```
private void LoadHighscoreList( )
{
    if(File.Exists("snake_highscorelist.xml"))
    {
        XmlSerializer serializer = new XmlSerializer(typeof(List<
SnakeHighscore>));
        using(Stream reader = new
FileStream("snake_highscorelist.xml", FileMode.Open))
        {
            List<SnakeHighscore> tempList = (List<SnakeHighscore>
)serializer.Deserialize(reader);
            this.HighscoreList.Clear();
            foreach(var item in tempList.OrderByDescending(x =>
x.Score))
                this.HighscoreList.Add(item);
        }
    }
}
```

You need to include a couple of additional namespaces for this:

```
using System.IO;
using System.Xml.Serialization;
```

Be sure to call the **LoadHighscoreList()** method, e.g. in the constructor of the Window:

```
public SnakeWPFsample()
{
    InitializeComponent();
    gameTickTimer.Tick += GameTickTimer_Tick;
    LoadHighscoreList();
}
```

Next, we implement the **SaveHighscoreList()** method:

```
private void SaveHighscoreList( )
```

```

{
    XmlSerializer serializer = new
    XmlSerializer(typeof(ObservableCollection<SnakeHighscore>));
    using(Stream writer = new FileStream("snake_highscorelist.xml",
    FileMode.Create))
    {
        serializer.Serialize(writer, this.HighscoreList);
    }
}

```

The Save method is most relevant to call when we add a new entry - this happens in the **BtnAddToHighscoreList\_Click()** event handler, which should look like this:

```

private void BtnAddToHighscoreList_Click(object sender, RoutedEventArgs
e)
{
    int newIndex = 0;
    // Where should the new entry be inserted?
    if((this.HighscoreList.Count > 0) && (currentScore <
this.HighscoreList.Max(x => x.Score)))
    {
        SnakeHighscore justAbove =
this.HighscoreList.OrderByDescending(x => x.Score).First(x => x.Score >=
currentScore);
        if(justAbove != null)
            newIndex = this.HighscoreList.IndexOf(justAbove) + 1;
    }
    // Create & insert the new entry
    this.HighscoreList.Insert(newIndex, new SnakeHighscore()
    {
        PlayerName = txtPlayerName.Text,
        Score = currentScore
    });
    // Make sure that the amount of entries does not exceed the maximum
    while(this.HighscoreList.Count > MaxHighscoreListEntryCount)
        this.HighscoreList.RemoveAt(MaxHighscoreListEntryCount);

    SaveHighscoreList();

    bdrNewHighscore.Visibility = Visibility.Collapsed;
    bdrHighscoreList.Visibility = Visibility.Visible;
}

```

```
}
```

It's quite simple: We try to decide if the new entry should be added at the top of the list (a new best!) or if it belongs further down the list. Once we have the new index, we insert a new instance of the **SnakeHighscore** class, using the current score and the name entered by the player. We then remove any unwanted entries from the bottom of the list, if the list suddenly has more items than we want (**MaxHighscoreListEntryCount**). Then we save the list (**SaveHighscoreList()**) and hide the **bdrNewHighscore** container, switching the view to the **bdrHighscoreList** container.

But there are still a couple of things to do. First of all, these new screens (dead message, high score list etc.) needs to be hidden each time a new game is stared. So, the top of the **StartNewGame()** method, which we implemented in a previous article, should now look like this:

```
private void StartNewGame()
{
    bdrWelcomeMessage.Visibility = Visibility.Collapsed;
    bdrHighscoreList.Visibility = Visibility.Collapsed;
    bdrEndOfGame.Visibility = Visibility.Collapsed;
    .....
}
```

The next thing we need to do is to modify the **EndGame()** method. Instead of just displaying the MessageBox, we need to check if the user just made it into the high score list or not and then display the proper message container:

```
private void EndGame()
{
    bool isNewHighscore = false;
    if(currentScore > 0)
    {
        int lowestHighscore = (this.HighscoreList.Count > 0 ?
this.HighscoreList.Min(x => x.Score) : 0);
        if((currentScore > lowestHighscore) ||
(this.HighscoreList.Count < MaxHighscoreListEntryCount))
        {
            bdrNewHighscore.Visibility = Visibility.Visible;
            txtPlayerName.Focus();
            isNewHighscore = true;
        }
    }
    if(!isNewHighscore)
    {
        tbFinalScore.Text = currentScore.ToString();
    }
}
```

```

        bdrEndOfGame.Visibility = Visibility.Visible;
    }
    gameTickTimer.IsEnabled = false;
}

```

The method basically checks if there's still available spots in the high score list (we define a maximum of 5 entries) or if the user just beat one of the existing scores - if so, we allow the user to add their name by displaying the **bdrNewHighscore** container. If no new high score was accomplished, we display the **bdrEndOfGame** container instead. Be sure to define the **MaxHighscoreListEntryCount** constant:

```
const int MaxHighscoreListEntryCount = 5;
```

With all that in place, start the game and do your best - as soon as the game ends, you should hopefully have made it into your brand new SnakeWPF high score list!

#### 1.23.9.9. Summary

In this article, we made a LOT of improvements to our SnakeWPF implementation. The most obvious one is of course the high score list, which did require quite a bit of extra markup/code, but it was totally worth it! On top of that, we made some nice usability improvements, while once again making our project look even more like a real game.

## 1.23.10. Improving SnakeWPF: Adding sound

Most games will have sound effects to enhance the experience, but so far, our little snake implementation is completely quiet. We have previously talked about both audio and video in this tutorial, so if you have read these articles, you know that playing a sound with WPF is quite easy. In fact, if you can live with the system sounds, it can be done with a single line of code:

```
SystemSounds.Beep.Play();
```

If you need a bit more than that, you can use the **MediaPlayer** class to play your own audio files (e.g. MP3). You can read all about it in this article: [Playing audio with WPF](#). A fun little project could be to record the sound of you taking a bite of an apple and then playing it when the snake eats an apple - it's quite easy to accomplish!

### 1.23.10.1. Making the Snake talk

For this tutorial, I decided to go another way than just playing regular sound bites - I want the Snake to talk! This might sound difficult, but only if you haven't read all the articles in this tutorial, because if you have, you know that WPF has great support for [Speech synthesis](#). With this, we can make the snake talk very easily!

First of all, you need to add a reference to the **System.Speech** assembly to your project. For specific instructions on adding exactly this assembly to your project, please see this previous article: [Speech synthesis \(making WPF talk\)](#). In the top, you'll find a detailed walk-through of how to accomplish this.

I've decided to make the Snake talk in several situations, so I will create a common **SpeechSynthesizer** instance which I will re-use each time, but start by adding a reference to the **System.Speech.Synthesis** namespace in the top:

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

Then declare and initialize the instance at the top of your Window:

```
public partial class MainWindow : Window
{
    private SpeechSynthesizer speechSynthesizer = new
    SpeechSynthesizer();
    . . . . .
}
```

### 1.23.10.2. "Yummy!"

That's all we need to make the Snake talk - but when should it talk and what should it say? I decided to make it say "yummy" each time it eats an apple. This happens in the **EatSnakeFood()** method which we implemented previously, so just add a single line in the top of it:

```

private void EatSnakeFood()
{
    speechSynthesizer.SpeakAsync( "yummy" );
    .....
}

```

That's it - run the game and enjoy the "yummy" each time the snake eats an apple. Of course the text string can be change to another word or even a sentence, if you feel like it.

#### 1.23.10.3. "Oh no - you died!"

Saying "yummy" is super simple, as you can see, but the **SpeechSynthesizer** can do a lot more than that! To demonstrate it, I have decided to make the Snake talk about its own death and the final score of the game - in other words, we need to add some talk to the **EndGame()** method. There will be quite a few lines of extra code, because I want several sentences with different speech settings, so I've decided to encapsulate the end-of-game-talk-code in its own method called **SpeakEndOfGameInfo()**. It's called at the bottom of the **EndGame()** method, implemented previously in this tutorial, so just add the call at the bottom of the method:

```

private void EndGame( )
{
    .....
    SpeakEndOfGameInfo(isNewHighscore);
}

```

Here's how our implementation of it looks:

```

private void SpeakEndOfGameInfo(bool isNewHighscore)
{
    PromptBuilder promptBuilder = new PromptBuilder();

    promptBuilder.StartStyle(new PromptStyle())
    {
        Emphasis = PromptEmphasis.Reduced,
        Rate = PromptRate.Slow,
        Volume = PromptVolume.ExtraLoud
    });
    promptBuilder.AppendText( "oh no" );
    promptBuilder.AppendBreak(TimeSpan.FromMilliseconds( 200 ) );
    promptBuilder.AppendText( "you died" );
    promptBuilder.EndStyle();

    if(isNewHighscore)

```

```

{
    promptBuilder.AppendBreak(TimeSpan.FromMilliseconds(500));
    promptBuilder.StartStyle(new PromptStyle()
    {
        Emphasis = PromptEmphasis.Moderate,
        Rate = PromptRate.Medium,
        Volume = PromptVolume.Medium
    });
    promptBuilder.AppendText("new high score:");
    promptBuilder.AppendBreak(TimeSpan.FromMilliseconds(200));
    promptBuilder.AppendTextWithHint(currentScore.ToString(),
        SayAs.NumberCardinal);
    promptBuilder.EndStyle();
}
speechSynthesizer.SpeakAsync(promptBuilder);
}

```

So that's quite a bit of code, but if you look closer, you'll see that there's a lot of the same stuff - we basically use a **PromptBuilder** instance to create a longer sentence, with various pronunciation settings. At the end, this will make the Snake say "Oh no - you died" each time you die. If you made it into the high score list, added in one of the previous articles, it will add "New high score:", followed by the actual score.

Each part of this is spoken with appropriate settings, using **PromptStyle** instances - for instance, "oh no - you died" is spoken slowly, with an increased volume. We also use the **AppendBreak()** method to add natural breaks between the different parts. You can read more about these techniques in our [previous article on the speech synthesis](#).

#### 1.23.10.4. Summary

Thanks to the *System.Speech* assembly and the **SpeechSynthesizer** class, our Snake has just learned to talk! This makes the game a lot more fun, while demonstrating how cool, powerful and flexible the .NET framework is.