Introduction to Windows Presentation Foundation

Overview

The Windows Presentation Foundation is Microsofts next generation UI framework to create applications with a rich user experience. It is part of the .NET framework 3.0 and higher.

WPF combines application UIs, 2D graphics, 3D graphics, documents and multimedia into one single framework. Its vector based rendering engine uses hardware acceleration of modern graphic cards. This makes the UI faster, scalable and resolution independent.

The following illustration gives you an overview of the main new features of WPF

Separation of Appearance and Behavior

WPF separates the appearance of an user interface from its behavior. The appearance is generally specified in the Extensible Application Markup Language (XAML), the behavior is implemented in a managed programming language like C# or Visual Basic. The two parts are tied together by databinding, events and commands. The separation of appearance and behavior brings the following benefits:

- Appearance and behaviour are loosely coupled
- Designers and developers can work on separate models.
- Graphical design tools can work on simple XML documents instead of parsing code.

Rich composition

Controls in WPF are extremely composable. You can define almost any type of controls as content of another. Although these flexibility sounds horrible to designers, its a very powerful feature if you use it appropriate. Put an image into a button to create an image button, or put a list of videos into a combobox to choose a video file.


```
<Button>
    <StackPanel Orientation="Horizontal">
        <Image Source="speaker.png" Stretch="Uniform"/>
        <TextBlock Text="Play Sound" />
    </StackPanel>
</Button>
```

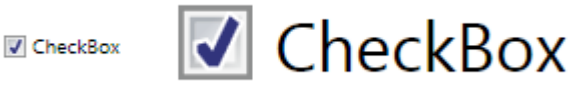
Highly customizable

Because of the strict separation of appearance and behavior you can easily change the look of a control. The concept of styles let you skin controls almost like CSS in HTML. Templates let you replace the entire appearance of a control.

The following example shows an default WPF button and a customized button.

Resolution independence

All measures in WPF are logical units - not pixels. A logical unit is a 1/96 of an inch. If you increase the resolution of your screen, the user interface stays the same size - if just gets crispier. Since WPF builds on a vector based rendering engine its incredibly easy to build scaleable user interfaces.

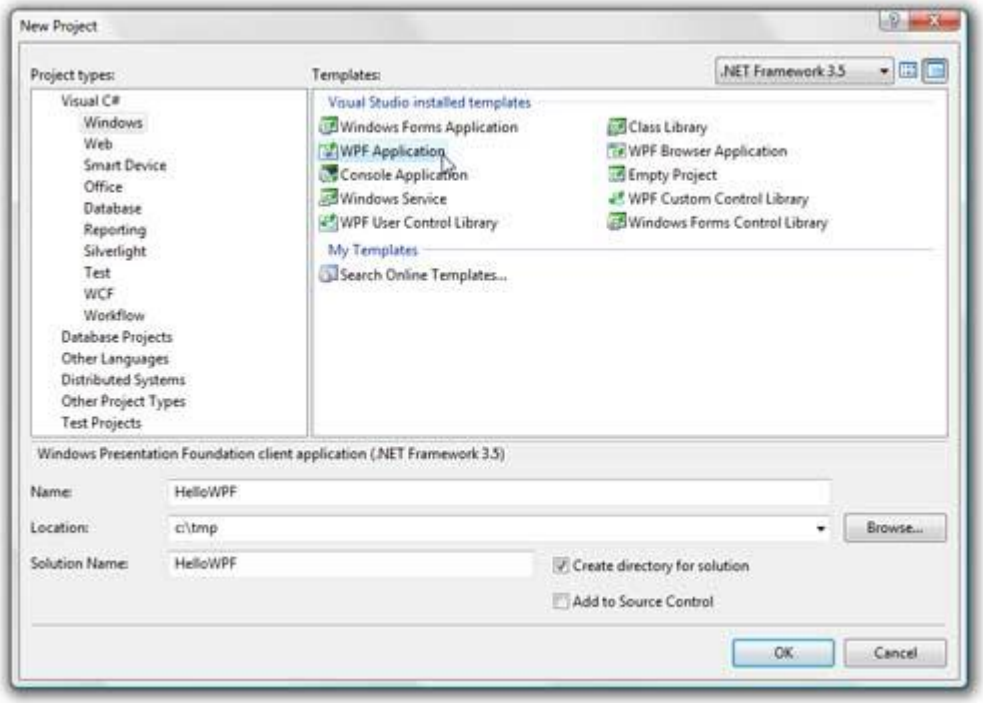


How to create a simple WPF application

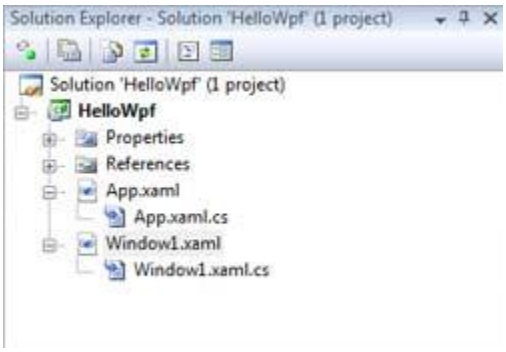
In Visual Studio 2008

Open Visual Studio 2008 and choose "File", "New", "Project..." in the main menu. Choose "WPF Application" as project type.

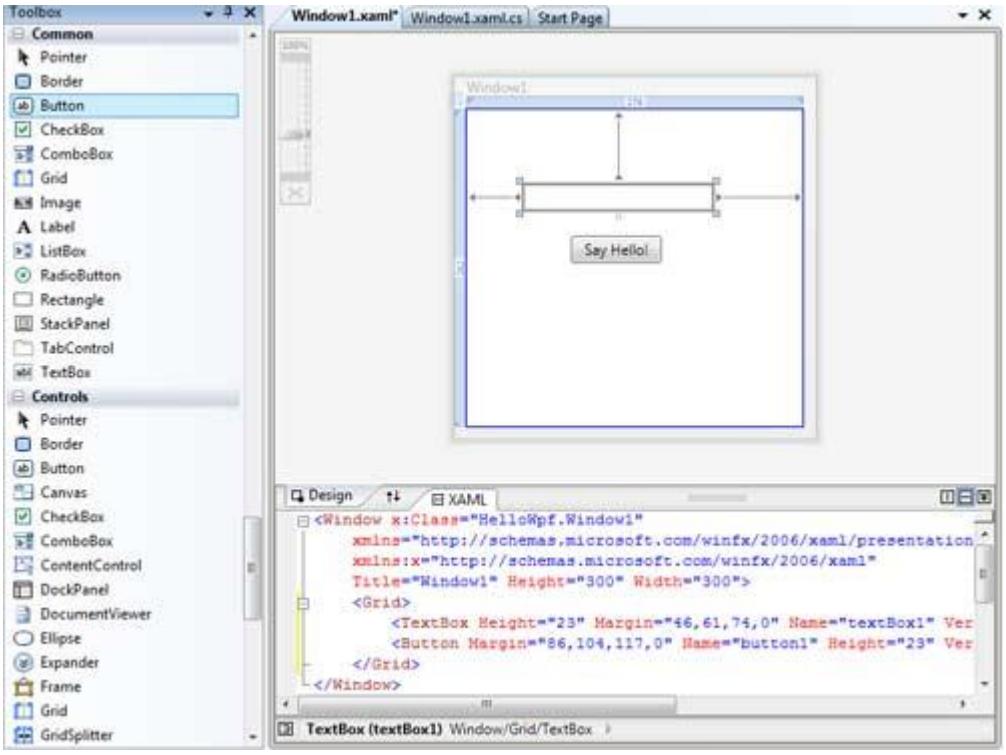
Choose a folder for your project and give it a name. Then press "OK"



Visual Studio creates the project and automatically adds some files to the solution. A Window1.xaml and an App.xaml. The structure looks quite similar to WinForms, except that the `Window1.designer.cs` file is no longer code but it's now declared in XAML as `Window1.xaml`



Open the `Window1.xaml` file in the WPF designer and drag a Button and a TextBox from the toolbox to the Window



Select the Button and switch to the event view in the properties window (click on the little yellow lightning icon). Doubleclick on the "Click" event to create a method in the codebehind that is called, when the user clicks on the button.

Note: If you do not find a yellow lightning icon, you need to install the Service Pack 1 for VisualStudio on your machine. Alternatively you can doubleclick on the button in the designer to achieve the same result.

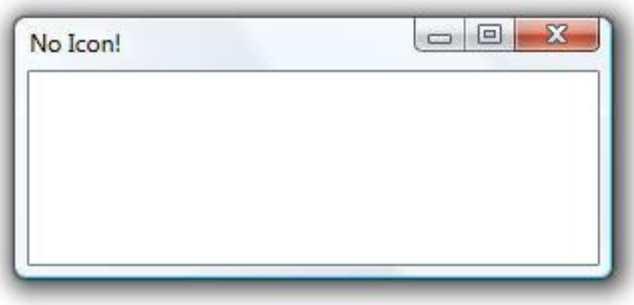
Visual Studio automatically creates a method in the code-behind file that gets called when the button is clicked.

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    textBox1.Text = "Hello WPF!";
}
```

The textbox has automatically become assigned the name `textBox1` by the WPF designer. Set text Text to "Hello WPF!" when the button gets clicked and we are done!. Start the application by hit [F5] on your keyboard.

Isn't this cool!

How to remove the icon of a WPF window



Unfortumately WPF does not provide any function to remove the icon of a window. One solution could be setting the icon to a transparent icon. But this way the extra space between the window border and title remains.

The better approach is to use a function provided by the Win32 API to remove the icon.

```
public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        protected override void OnSourceInitialized(EventArgs e)
        {
            IconHelper.RemoveIcon(this);
        }
    }



public static class IconHelper
    {
        [DllImport("user32.dll")]
        static extern int GetWindowLong(IntPtr hwnd, int index);

        [DllImport("user32.dll")]
```

```
        static extern int SetWindowLong(IntPtr hwnd, int index, int
newStyle);

        [DllImport("user32.dll")]
        static extern bool SetWindowPos(IntPtr hwnd, IntPtr hwndInsertAfter,
                    int x, int y, int width, int height, uint flags);

        [DllImport("user32.dll")]
        static extern IntPtr SendMessage(IntPtr hwnd, uint msg,
                    IntPtr wParam, IntPtr lParam);

        const int GWL_EXSTYLE = -20;
        const int WS_EX_DLGMODALFRAME = 0x0001;
        const int SWP_NOSIZE = 0x0001;
        const int SWP_NOMOVE = 0x0002;
        const int SWP_NOZORDER = 0x0004;
        const int SWP_FRAMECHANGED = 0x0020;
        const uint WM_SETICON = 0x0080;

        public static void RemoveIcon(Window window)
        {
            // Get this window's handle
            IntPtr hwnd = new WindowInteropHelper(window).Handle;

            // Change the extended window style to not show a window icon
            int extendedStyle = GetWindowLong(hwnd, GWL_EXSTYLE);
            SetWindowLong(hwnd, GWL_EXSTYLE, extendedStyle |
WS_EX_DLGMODALFRAME);

            // Update the window's non-client area to reflect the changes
            SetWindowPos(hwnd, IntPtr.Zero, 0, 0, 0, 0, SWP_NOMOVE |
                  SWP_NOSIZE | SWP_NOZORDER | SWP_FRAMECHANGED);
        }

    }
```

WPF ListView Control

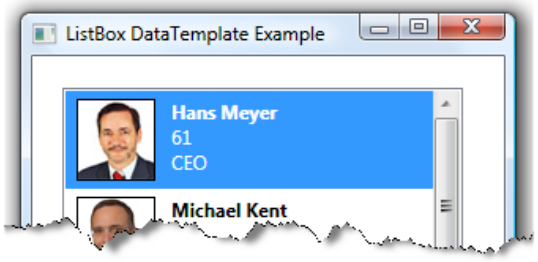
How to Hide the Header of a ListView

To hide the header of a ListView you can modify the `Visibility` property of the ColumnHeaderContainer by overriding the style locally.

```
<ListView>
    <ListView.View>
        <GridView>
            <GridView.ColumnHeaderContainerStyle>
               <Style>
                   <Setter Property="FrameworkElement.Visibility"
Value="Collapsed"/>
               </Style>
            </GridView.ColumnHeaderContainerStyle>
            <GridView.Columns>
                ...
            </GridView.Columns>
        </GridView>
    </ListView.View>
</ListView>
```

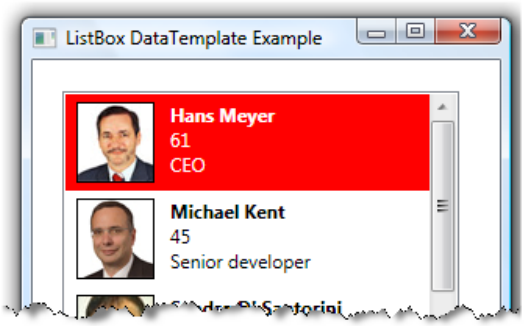
Change the Background of a selected ListBox Item

If you select an item in a listbox it gets the default selection color (usually blue) as background. Even if you specify a custom data template. The reason is that the blue background (or gray if the control is not focussed) is drawn outside of the data template. So you have no chance to override it from within the data template.



The color used to draw the blue and gray background are system colors. So the easiest way to get rid of these backgrounds is to locally override the highlight and control brushes of the system colors.

The best way to do this is to create a style for the listbox. Place the style in the resources of a parent element. For e.g. `Window.Resources`




```
<Style x:Key="myListboxStyle">
    <Style.Resources>
        <!-- Background of selected item when focussed -->
        <SolidColorBrush x:Key="{x:Static SystemColors.HighlightBrushKey}"
Color="Red" />
        <!-- Background of selected item when not focussed -->
        <SolidColorBrush x:Key="{x:Static SystemColors.ControlBrushKey}"
Color="Green" />
    </Style.Resources>
</Style>
```

The following XAML snippet shows how to apply the style to the listbox.

```
<Grid x:Name="LayoutRoot">
  <ListBox Style="{StaticResource myListboxStyle}" />
</Grid>
```

Change the arrangement of items in a listbox

All WPF controls deriving from `ItemsControl` provide an `ItemsPanel` property that allows you to replace the internal layout panel that arranges the items.

Horizontal

We override the `ItemsPanel` property and set it to a `StackPanel` layout manager with an orientation set to `Horizontal`. We use an `VirtualizingStackPanel` that works just like a normal StackPanel, except that is does only create the items that are visible. If you scroll it automatically creates the new items that become visible and recycles the hidden.

```
<ListBox>
    <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
</ListBox>
```

Wrapped

Using a `WrapPanel` to layout the items row by row is a bit more complex. Since the layout panel is wrapped by a `ScrollContentPresenter` (as seen by the scrolbar of the exmple above) the available width is infinite. So the WrapPanel does not see a reason to wrap to a new line.

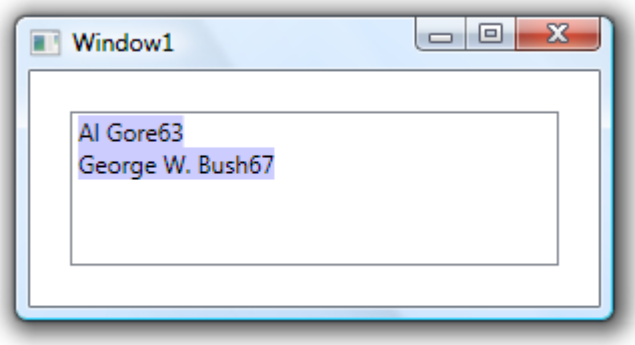
What we need to do is to set the width of the warp panel to the ActualWidth if the internal ScrollContentPresenter. The ScrollContentPresenter can be found by using the FindAncestor mode of the relative source extension.

```
<ListBox>
```

```
    <ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapPanel IsItemsHost="True"
                       Width="{Binding ActualWidth,
                RelativeSource={RelativeSource Mode=FindAncestor,
                AncestorType={x:Type ScrollContentPresenter}}}" />
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
</ListBox>
```

How to strech an WPF ListBox Item to span the whole width

If you create a data template with a right-aligned element (like the age in our exmple), it only spans over the needed width, because the content of a listbox is left-aligned by default.

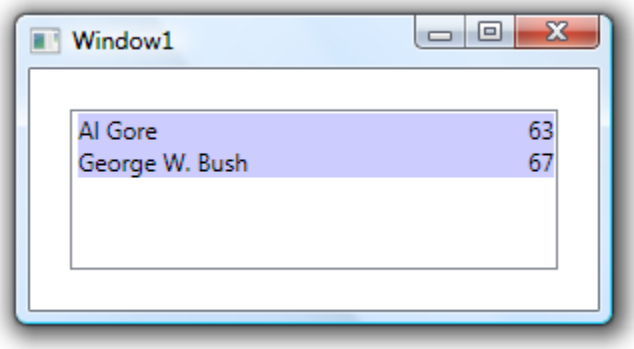


This is the DataTemplate I created for this example.

```
<Style TargetType="ListBox" x:Key="strechedItemStyle">
    <Setter Property="ItemTemplate">
        <Setter.Value>
            <DataTemplate>
                <Grid Background="#330000FF">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto" />
```

```
                        <ColumnDefinition Width="*" />
                    </Grid.ColumnDefinitions>
                    <TextBlock Text="{Binding Name}"
HorizontalAlignment="Left"  Grid.Column="0"/>
                    <TextBlock Text="{Binding Age}"
HorizontalAlignment="Right" Grid.Column="1"/>
                </Grid>
            </DataTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

To make the listbox item span the whole width of the listbox you need to set the `HorizontalContentAlignment` property to `Stretch`.



This is the way to set it directly on the listbox element:

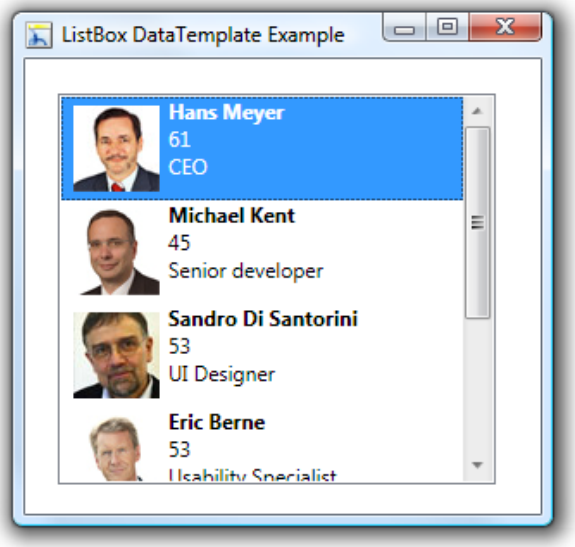
```
```

How to apply a DataTemplate to a ListBox

Introducing DataTemplates

All controls deriving from `ItemsControl` have a `DataTemplate` that specifies how an object bound to an item is presented to the user. The default template renders a single line of text per item - as we know a listbox from HTML or WinForms.

But WPF allows us to put whatever we like into a DataTemplate. If we want to display a list of customers we can provide a data template that shows an image for each customer next to its name and age. In my opinion this is one of the most powerful features in WPF.

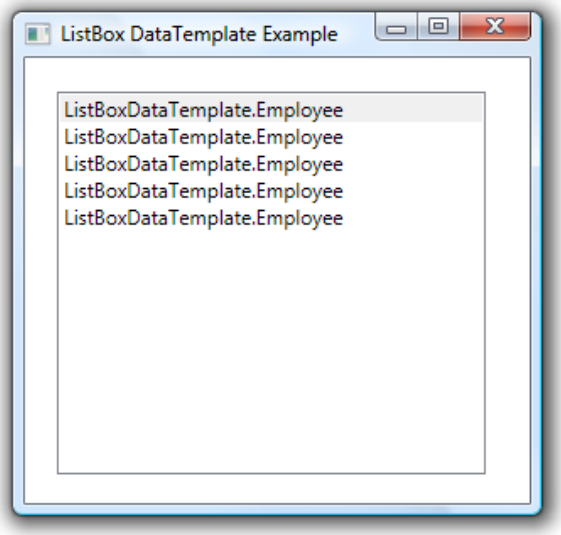


In the following steps I will show you how to create such an DataTemplate

Bind the data to the ListBox

We bind a list of employees to the listbox. Because the listbox does not know anything about our employee class, it calls the `ToString()` method. The result is not yet very pleasing.

```
<Grid x:Name="LayoutRoot">
   <ListBox Margin="10" ItemsSource="{Binding}"/>
</Grid>
```



Override the ToString() method to improve the display

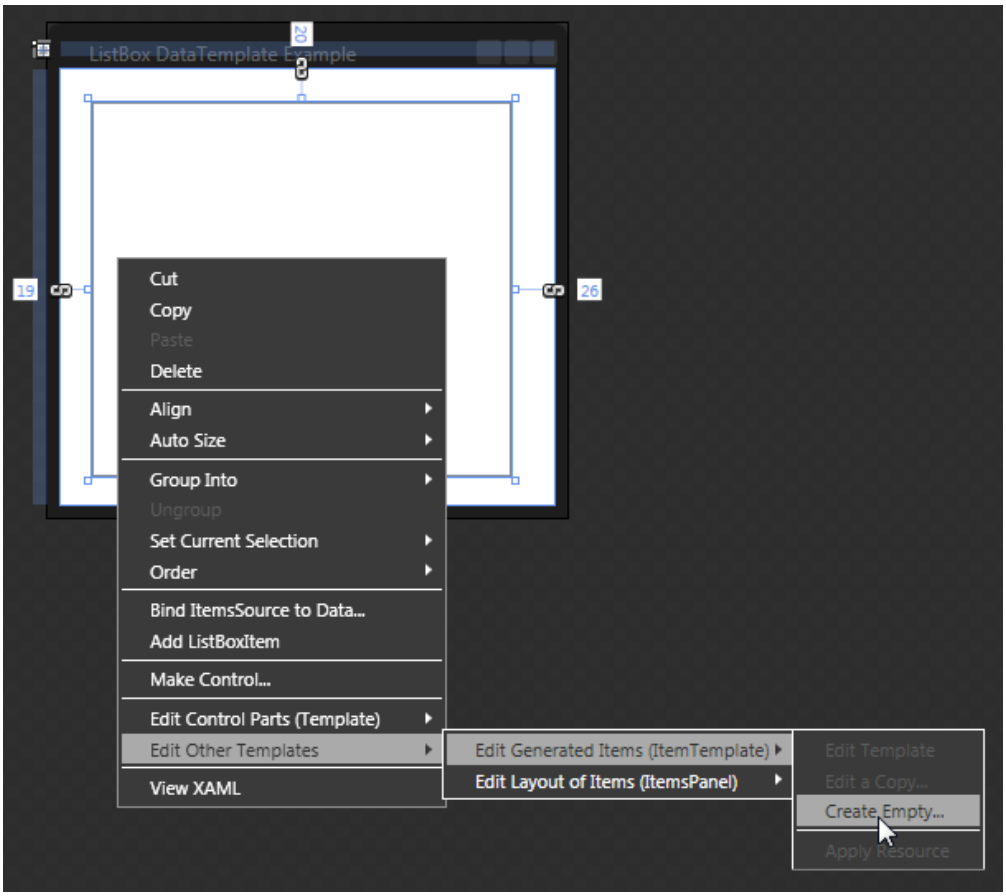
What we do next is to overriding the ToString() method to provide some useful information instead of just the type name. But a string is not really what we call a cool user experience.

```
public override string ToString()
{
    return string.Format("{0} {1} ({2}), {3})",
                         Firstname, Lastname, Age, Role);
}
```

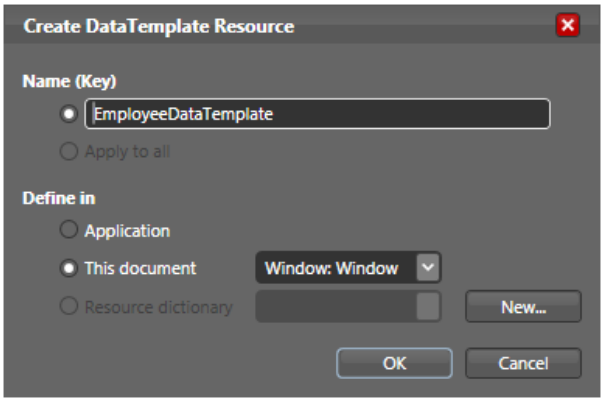
Create a DataTemplate

The ultimate flexibility is to create your own `DataTemplate`. A data template can consist of multiple controls like images, textblocks, etc. It han have a flexible width and height, background color and shape. There are no limits for your creativity.

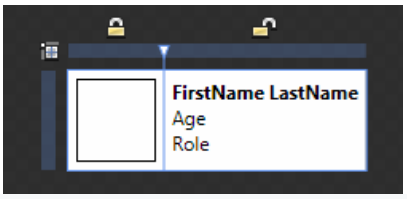
An easy way to create a data template is to use Expression Blend. Open the context menu of your list box and choose "Edit Other Templates" -> "Edit Generated Items" -> "Create Empty".



Blend will ask you to specify a name for your DataTemplate because it will define it in the resources of either the Application, the current document of an external resource dictionary. The name you specify is the key for the resource dictionary.



After you press OK, you will be in the editor to design the data template. The item data (in our sample an employee) is set to the DataContext, so you can easily access all properties of the employee by using databinding.




```
<DataTemplate x:Key="EmployeeDataTemplate">
   <Grid>
      <Grid.ColumnDefinitions>
         <ColumnDefinition Width="60"/>
         <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
   <Border Margin="5" BorderBrush="Black" BorderThickness="1">
      <Image Source="{Binding Path=Image}" Stretch="Fill" Width="50"
Height="50" />
   </Border>
   <StackPanel Grid.Column="1" Margin="5">
      <StackPanel Orientation="Horizontal" TextBlock.FontWeight="Bold" >
         <TextBlock Text="{Binding Path=Firstname, FallbackValue=FirstName}"
/>
         <TextBlock Text="{Binding Path=Lastname, FallbackValue=LastName}"
Padding="3,0,0,0"/>
      </StackPanel>
      <TextBlock Text="{Binding Path=Age, FallbackValue=Age}" />
      <TextBlock Text="{Binding Path=Role, FallbackValue=Role}" />
   </StackPanel>
  </Grid>
</DataTemplate>
```

WPF Slider Control

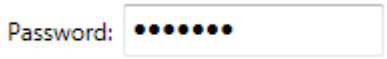
How to Make a Slider Snap to Integer Values

If you just set the Minimum and Maximum of a slider and choose a value the result is determined by the pixel position of the thumb. The value is typically a high-precision value with many decimal places. To allow only integer values you have to set the `IsSnapToTickEnabled` property to True.

```
<Slider Minimum="0"
        Maximum="20"
        IsSnapToTickEnabled="True"
        TickFrequency="2"
```

WPF PasswordBox Control

The password box control is a special type of TextBox designed to enter passwords. The typed in characters are replaced by asterisks. Since the password box contains a sensible password it does not allow cut, copy, undo and redo commands.




```
<StackPanel>
      <Label Content="Password:" />
      <PasswordBox x:Name="passwordBox" Width="130" />
</StackPanel>
```

Change the password character

To replace the asteriks character by another character, set the PasswordChar property to the character you desire.

```
<PasswordBox x:Name="passwordBox" PasswordChar="*" />
```

Limit the length of the password

To limit the length of the password a user can enter set the `MaxLength` property to the amount of characters you allow.

```
<PasswordBox x:Name="passwordBox" MaxLength="8" />
```

Databind the Password Property of a WPF PasswordBox

When you try to databind the password property of a PasswordBox you will recognize that you cannot do data binding on it. The reason for this is, that the password property is not backed by a DependencyProperty.

The reason is databinding passwords is not a good design for security reasons and should be avoided. But sometimes this security is not necessary, then it's only cumbersome that you cannot bind to the password property. In this special cases you can take advantage of the following PasswortBoxHelper.

```
<StackPanel>
    <PasswordBox w:PasswordHelper.Attach="True"
         w:PasswordHelper.Password="{Binding Text, ElementName=plain,
Mode=TwoWay}"
                 Width="130"/>
    <TextBlock Padding="10,0" x:Name="plain" />
</StackPanel>
```

The PasswordHelper is attached to the password box by calling the `PasswordHelper.Attach` property. The attached property `PasswordHelper.Password` provides a bindable copy of the original password property of the PasswordBox control.

```
public static class PasswordHelper
{
    public static readonly DependencyProperty PasswordProperty =
        DependencyProperty.RegisterAttached("Password",
        typeof(string), typeof(PasswordHelper),
        new FrameworkPropertyMetadata(string.Empty,
OnPasswordPropertyChanged));

    public static readonly DependencyProperty AttachProperty =
        DependencyProperty.RegisterAttached("Attach",
```

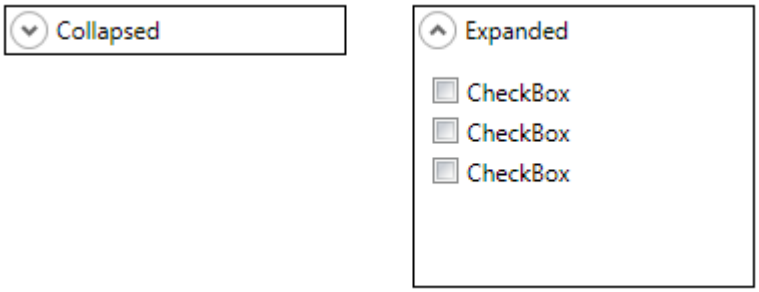
```
        typeof(bool), typeof(PasswordHelper), new PropertyMetadata(false,
Attach));

    private static readonly DependencyProperty IsUpdatingProperty =
       DependencyProperty.RegisterAttached("IsUpdating", typeof(bool),
       typeof(PasswordHelper));


    public static void SetAttach(DependencyObject dp, bool value)
    {
        dp.SetValue(AttachProperty, value);
    }

    public static bool GetAttach(DependencyObject dp)
    {
        return (bool)dp.GetValue(AttachProperty);
    }

    public static string GetPassword(DependencyObject dp)
    {
        return (string)dp.GetValue(PasswordProperty);
    }

    public static void SetPassword(DependencyObject dp, string value)
    {
        dp.SetValue(PasswordProperty, value);
    }

    private static bool GetIsUpdating(DependencyObject dp)
    {
        return (bool)dp.GetValue(IsUpdatingProperty);
    }

    private static void SetIsUpdating(DependencyObject dp, bool value)
    {
        dp.SetValue(IsUpdatingProperty, value);
    }
```

```
private static void OnPasswordPropertyChanged(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
{
    PasswordBox passwordBox = sender as PasswordBox;
    passwordBox.PasswordChanged -= PasswordChanged;

    if (!(bool)GetIsUpdating(passwordBox))
    {
        passwordBox.Password = (string)e.NewValue;
    }
    passwordBox.PasswordChanged += PasswordChanged;
}

private static void Attach(DependencyObject sender,
    DependencyPropertyChangedEventArgs e)
{
    PasswordBox passwordBox = sender as PasswordBox;

    if (passwordBox == null)
        return;

    if ((bool)e.OldValue)
    {
        passwordBox.PasswordChanged -= PasswordChanged;
    }

    if ((bool)e.NewValue)
    {
        passwordBox.PasswordChanged += PasswordChanged;
    }
}

private static void PasswordChanged(object sender, RoutedEventArgs e)
{
    PasswordBox passwordBox = sender as PasswordBox;
    SetIsUpdating(passwordBox, true);
    SetPassword(passwordBox, passwordBox.Password);
    SetIsUpdating(passwordBox, false);
```

```
    }
}
```

WPF Expander Control

Introduction



The `Expander` control is like a GroupBox but with the additional feature to collapse and expand its content. It derives from `HeaderedContentControl` so it has a `Header` property to set the header content, and a `Content` property for the expandable content.

It has a `IsExpanded` property to get and set if the expander is in expanded or collapsed state.

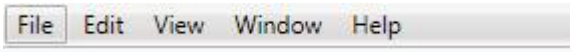
In collapsed state the expander takes only the space needed by the header. In expanded state it takes the size of header and content together.

```
<Expander Header="More Options">
    <StackPanel Margin="10,4,0,0">
        <CheckBox Margin="4" Content="Option 1" />
        <CheckBox Margin="4" Content="Option 2" />
        <CheckBox Margin="4" Content="Option 3" />
    </StackPanel>
</Expander>
```

Menus in WPF

Menu

The `Menu` control derives from `HeaderedItemsControl`. It stacks it items horizontally and draws the typical gray background. The only property that the Menu adds to `ItemsControl` is the `IsMainMenu` property. This controls if the menu grabs the focus if the user presses F10 or the ALT key.

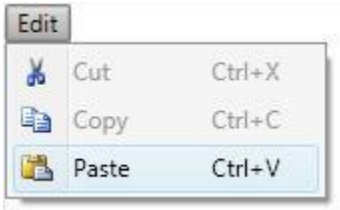



```
<Menu IsMainMenu="True">
    <MenuItem Header="_File" />
    <MenuItem Header="_Edit" />
    <MenuItem Header="_View" />
    <MenuItem Header="_Window" />
    <MenuItem Header="_Help" />
</Menu>
```

MenuItem

The `MenuItem` is a `HeaderedItemsControl`. The content of the `Header` property is the caption of the menu. The `Items` of a MenuItems are its sub menus. The `Icon` property renders a second content on the left of the caption. This is typically used to draw a little image. But it can be used for type of content.

You can define a keyboard shortcut by adding an underscore in front of a character.




```
<MenuItem Header="_Edit">
    <MenuItem Header="_Cut" Command="Cut">
        <MenuItem.Icon>
            <Image Source="Images/cut.png" />
        </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="_Copy" Command="Copy">
        <MenuItem.Icon>
            <Image Source="Images/copy.png" />
        </MenuItem.Icon>
    </MenuItem>
    <MenuItem Header="_Paste" Command="Paste">
```

```
        <MenuItem.Icon>
            <Image Source="Images/paste.png" />
        </MenuItem.Icon>
    </MenuItem>
</MenuItem>
```

Checkable MenuItems

You can make a menu item checkable by setting the `IsCheckable` property to true. The check state can be queried by the `IsChecked` property. To get notified when the check state changes you can add a handler to the `Checked` and `Unchecked` property.


```
<MenuItem Header="_Debug">
    <MenuItem Header="Enable Debugging" IsCheckable="True" />
</MenuItem>
```

Separators

Separator is a simple control to group menu items. It's rendered as a horizontal line. It can also be used in `ToolBar` and `StatusBar`.


```
<Menu>
    <MenuItem Header="_File">
        <MenuItem Header="_New..." />
        <Separator />
```

```
        <MenuItem Header="_Open..." />
        <Separator />
        <MenuItem Header="_Save" />
        <MenuItem Header="_Save As..." />
        <Separator />
        <MenuItem Header="_Exit" />
    </MenuItem>
</Menu>
```

Callbacks

You can register a callback to any menu item by adding a callback to the `Click` event.

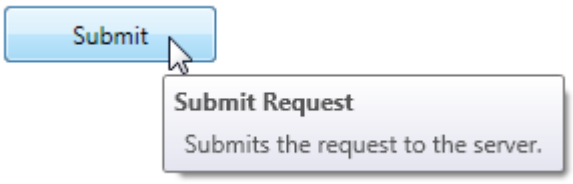
```
<Menu>
    <MenuItem Header="_File">
        <MenuItem Header="_New..."  Click="New_Click"/>
    </MenuItem>
</Menu>
```

```
private void New_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("You clicked 'New...'");
}
```

Keyboard Shortcuts

To add a keyboard shortcut to a menu item, add a underscode "_" in front of the caracter you want to use as your hot key. This automatically sets the `InputGestureText` to an appropriate value. But you can also override the proposed text by setting this property to a text of your choice.

ToolTips in WPF




```
<Button Content="Submit">
    <Button.ToolTip>
        <ToolTip>
            <StackPanel>
                <TextBlock FontWeight="Bold">Submit Request</TextBlock>
                <TextBlock>Submits the request to the server.</TextBlock>
            </StackPanel>
        </ToolTip>
    </Button.ToolTip>
</Button>
```

How to show ToolTips on disabled controls

When you disable a control with `IsEnabled=False` the tooltip does not show anymore. If you want to have the tooltip appear anyway you have to set the attaached property `ToolTipService.ShowOnDisabled` to `True`.

```
<Button IsEnabled="False"
        ToolTip="Saves the current document"
        ToolTipService.ShowOnDisabled="True"
        Content="Save">
</Button>
```

How to change the show duration of a ToolTip

The static class `ToolTipService` allows you to modify the show duration of the tooltip

```
<Button ToolTip="Saves the current document"
        ToolTipService.ShowDuration="20"
        Content="Save">
</Button>
```

Radio Button

Introduction

The `RadioButton` control has its name from old analog radios which had a number of programmable station buttons. When you pushed one in, the previosly selected poped out. So only one station can be selected at a time.

The `RadioButton` control has the same behavior. It lets the user choose **one option out of a few**. It the list of options gets longer, you should prefer a combo or list box instead.

To define which RadioButtons belong togehter, you have to **set the `GroupName`** to the same name.

To **preselect** one option set the **`IsChecked` property to `True`.**

```
<StackPanel>
    <RadioButton GroupName="Os" Content="Windows XP" IsChecked="True"/>
```

```
    <RadioButton GroupName="Os" Content="Windows Vista" />
    <RadioButton GroupName="Os" Content="Windows 7" />
    <RadioButton GroupName="Office" Content="Microsoft Office 2007"
IsChecked="True"/>
    <RadioButton GroupName="Office" Content="Microsoft Office 2003"/>
    <RadioButton GroupName="Office" Content="Open Office"/>
</StackPanel>
```

How to DataBind Radio Buttons in WPF

The radio button control has a known issue with data binding. If you bind the `IsChecked` property to a boolean and check the RadioButton, the value gets `True`. But when you check another RadioButton, the databound value still remains true.

The reason for this is, that the Binding gets lost during the unchecking, because the controls internally calls `ClearValue()` on the dependency property.

```
<Window.Resources>
   <EnumMatchToBooleanConverter x:Key="enumConverter" />
</Window.Resources>


<RadioButton Content="Option 1" GroupName="Options1"
             IsChecked="{Binding Path=CurrentOption, Mode=TwoWay,
                                 Converter={StaticResource enumConverter},
                                 ConverterParameter=Option1}"  />
<RadioButton Content="Option 2" GroupName="Options2"
             IsChecked="{Binding Path=CurrentOption, Mode=TwoWay,
                                 Converter={StaticResource enumConverter},
                                 ConverterParameter=Option2}"  />
<RadioButton Content="Option 3" GroupName="Options3"
             IsChecked="{Binding Path=CurrentOption, Mode=TwoWay,
                                 Converter={StaticResource enumConverter},
                                 ConverterParameter=Option3}"  />
```

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

            string checkValue = value.ToString();
            string targetValue = parameter.ToString();
            return checkValue.Equals(targetValue,
                     StringComparison.InvariantCultureIgnoreCase);
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            if (value == null || parameter == null)
                return null;

            bool useValue = (bool)value;
            string targetValue = parameter.ToString();
            if (useValue)
                return Enum.Parse(targetType, targetValue);

            return null;
        }
    }
```

Popup Control

Introduction follows...

How to make the popup close, when it loses focus

Just set the `StaysOpen` property to `False`. Unfortunately this is not the default behavior

```
<Popup StaysOpen="False" />
```

Context Menus in WPF

Context Menus can be defined on any WPF controls by setting the `ContextMenu` property to an instance of a `ContextMenu`. The items of a context menu are normal `MenuItems`.


```
<RichTextBox>
    <RichTextBox.ContextMenu>
        <ContextMenu>
            <MenuItem Command="Cut">
                <MenuItem.Icon>
                    <Image Source="Images/cut.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Command="Copy">
                <MenuItem.Icon>
                    <Image Source="Images/copy.png" />
                </MenuItem.Icon>
            </MenuItem>
            <MenuItem Command="Paste">
                <MenuItem.Icon>
                    <Image Source="Images/paste.png" />
                </MenuItem.Icon>
            </MenuItem>
        </ContextMenu>
    </RichTextBox.ContextMenu>
</RichTextBox>
```

Show ContextMenus on a disabled controls

If you rightclick on a disabled control, no context menu is shown by default. To enable the context menu for disabled controls you can set the `ShowOnDisabled` attached property of the `ContextMenuService` to `True`.

```
<RichTextBox IsEnabled="False" ContextMenuService.ShowOnDisabled="True">
    <RichTextBox.ContextMenu>
        <ContextMenu>
            ...
        </ContextMenu>
    </RichTextBox.ContextMenu>
</RichTextBox>
```

Merge ContextMenus

If you want to fill a menu with items coming from multiple sources, you can use the `CompositeCollection` to merge multiple collection into one.

```
<Window 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">
  <Grid Background="Transparent">
    <Grid.Resources>
        <x:Array Type="{x:Type sys:Object}" x:Key="extensions">
            <Separator />
            <MenuItem Header="Extension MenuItem 1" />
            <MenuItem Header="Extension MenuItem 2" />
            <MenuItem Header="Extension MenuItem 3" />
        </x:Array>
    </Grid.Resources>
    <Grid.ContextMenu>
        <ContextMenu>
            <ContextMenu.ItemsSource>
                <CompositeCollection>
                    <MenuItem Header="Standard MenuItem 1" />
                    <MenuItem Header="Standard MenuItem 2" />
                    <MenuItem Header="Standard MenuItem 3" />
```

```
                    <CollectionContainer Collection="{StaticResource
extensions}" />
                </CompositeCollection>
            </ContextMenu.ItemsSource>
        </ContextMenu>
    </Grid.ContextMenu>
  </Grid>
</Window>
```

Introduction to WPF Layout

Why layout is so important

Layout of controls is critical to an applications usability. Arranging controls based on fixed pixel coordinates may work for an limited enviroment, but as soon as you want to use it on different screen resolutions or with different font sizes it will fail. WPF provides a rich set built-in layout panels that help you to avoid the common pitfalls.

These are the five most popular layout panels of WPF:

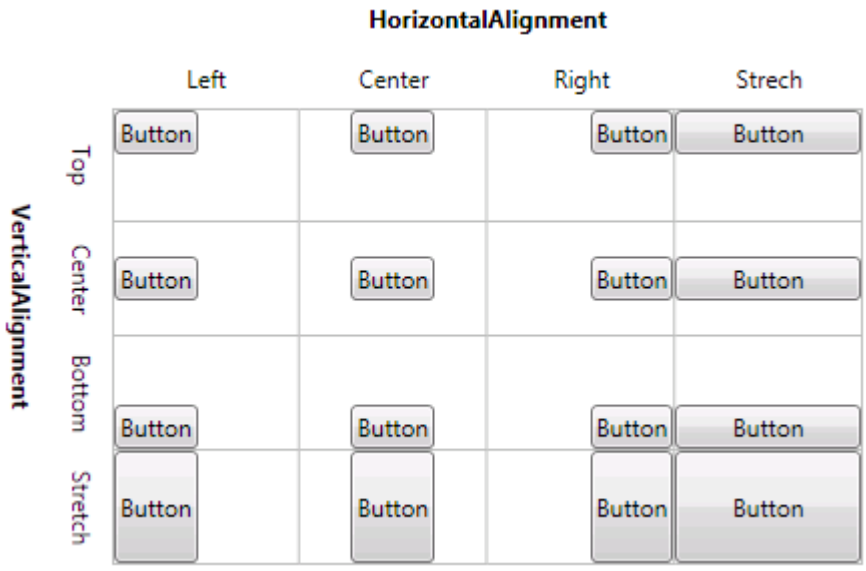
- Grid Panel
- Stack Panel
- Dock Panel
- Wrap Panel
- Canvas Panel

Best Practices

- Avoid fixed positions - use the `Alignment` properties in combination with `Margin` to position elements in a panel
- Avoid fixed sizes - set the `Width` and `Height` of elements to `Auto` whenever possible.
- Don't abuse the canvas panel to layout elements. Use it only for vector graphics.
- Use a StackPanel to layout buttons of a dialog
- Use a GridPanel to layout a static data entry form. Create a Auto sized column for the labels and a Star sized column for the TextBoxes.
- Use an ItemControl with a grid panel in a DataTemplate to layout dynamic key value lists. Use the SharedSize feature to synchronize the label widths.

Vertical and Horizontal Alignment

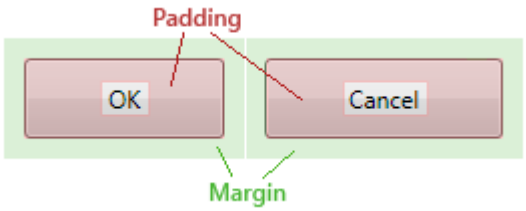
Use the `VerticalAlignment` and `HorizontalAlignmant` properties to dock the controls to one or multiple sides of the panel. The following illustrations show how the sizing behaves with the different combinations.



Margin and Padding

The `Margin` and `Padding` properties can be used to reserve some space around of within the control.

- The `Margin` is the extra space **around** the control.
- The `Padding` is extra space **inside** the control.
- The `Padding` of an outer control is the `Margin` of an inner control.



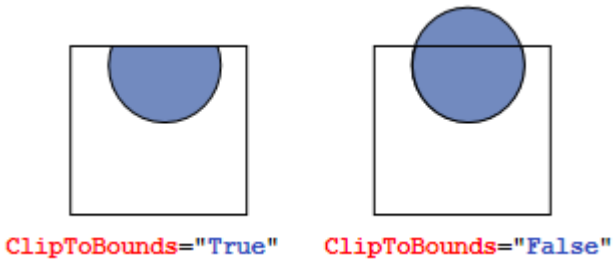
Height and Width

Alltough its not a recommended way, all controls provide a `Height` and `Width` property to give an element a fixed size. A better way is to use the `MinHeight`, `MaxHeight`, `MinWidth` and `MaxWidth` properties to define a acceptable range.
If you set the width or height to `Auto` the control sizes itself to the size of the content.

Overflow Handling

Clipping

Layout panels typically clip those parts of child elements that overlap the border of the panel. This behavior can be controlled by setting the `ClipToBounds` property to true or false.



Scrolling

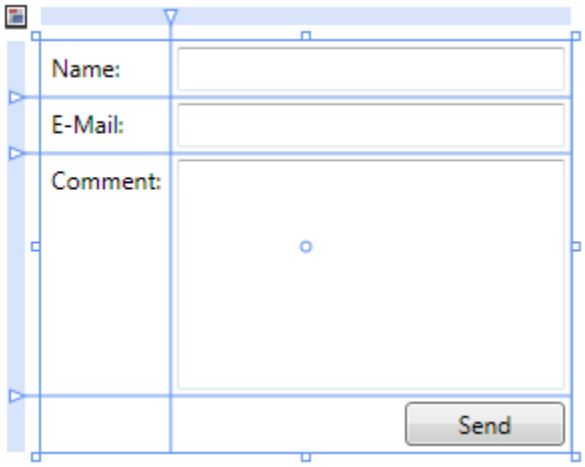
When the content is too big to fit the available size, you can wrap it into a `ScrollViewer`. The ScrollViewer uses two scroll bars to choose the visible area.

The visibility of the scrollbars can be controlled by the vertical and horizontal `ScrollbarVisibility` properties.

```
<ScrollViewer>
    <StackPanel>
        <Button Content="First Item" />
        <Button Content="Second Item" />
        <Button Content="Third Item" />
    </StackPanel>
</ScrollViewer>
```

Grid Panel

Introduction



The grid is a layout panel that arranges its child controls in a tabular structure of rows and columns. Its functionality is similar to the HTML table but more flexible. A cell can contain multiple controls, they can span over multiple cells and even overlap themselves.

The resize behaviour of the controls is defined by the `HorizontalAlignment` and `VerticalAlignment` properties who define the anchors. The distance between the anchor and the grid line is specified by the margin of the control

Define Rows and Columns

The grid has one row and column by default. To create additional rows and columns, you have to add `RowDefinition` items to the `RowDefinitions` collection and `ColumnDefinition` items to the `ColumnDefinitions` collection. The following example shows a grid with three rows and two columns.

The size can be specified as an absolute amount of logical units, as a percentage value or automatically.

Fixed Fixed size of logical units (1/96 inch)

Auto Takes as much space as needed by the contained control

Star (*) Takes as much space as available, percentally divided over all star-sized columns. Star-sizes are like percentages, except that the sum of all star columns does not have to be 100%. Remember that star-sizing does not work if the grid size is calculated based on its content.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="28" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="200" />
    </Grid.ColumnDefinitions>
</Grid>
```

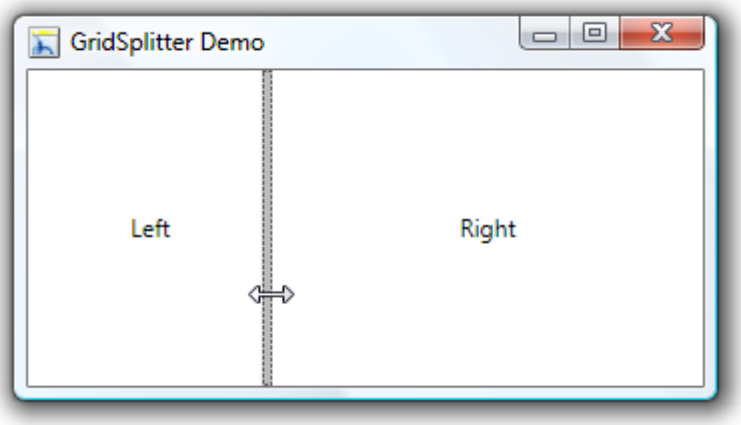
How to add controls to the grid

To add controls to the grid layout panel just put the declaration between the opening and closing tags of the `Grid`. Keep in mind that the row- and columndefinitions must precced any definition of child controls.

The grid layout panel provides the two attached properties `Grid.Column` and `Grid.Row` to define the location of the control.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="28" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="200" />
    </Grid.ColumnDefinitions>
    <Label Grid.Row="0" Grid.Column="0" Content="Name:"/>
    <Label Grid.Row="1" Grid.Column="0" Content="E-Mail:"/>
    <Label Grid.Row="2" Grid.Column="0" Content="Comment:"/>
    <TextBox Grid.Column="1" Grid.Row="0" Margin="3" />
    <TextBox Grid.Column="1" Grid.Row="1" Margin="3" />
    <TextBox Grid.Column="1" Grid.Row="2" Margin="3" />
    <Button Grid.Column="1" Grid.Row="3" HorizontalAlignment="Right"
            MinWidth="80" Margin="3" Content="Send"  />
</Grid>
```

Resizable columns or rows



WPF provides a control called the `GridSplitter`. This control is added like any other control to a cell of the grid. The special thing is that is grabs itself the nearest gridline to change its width or height when you drag this control around.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Label Content="Left" Grid.Column="0" />
    <GridSplitter HorizontalAlignment="Right"
                  VerticalAlignment="Stretch"
                  Grid.Column="1" ResizeBehavior="PreviousAndNext"
                  Width="5" Background="#FFBCBCBC"/>
    <Label Content="Right" Grid.Column="2" />
</Grid>
```

The best way to align a grid splitter is to place it in its own auto-sized column. Doing it this way prevents overlapping to adjacent cells. To ensure that the grid splitter changes the size of the previous and next cell you have to set the `ResizeBehavior` to `PreviousAndNext`.

The splitter normally recognizes the resize direction according to the ratio between its height and width. But if you like you can also manually set the `ResizeDirection` to `Columns` or `Rows`.

```
<GridSplitter ResizeDirection="Columns"/>
```

How to share the width of a column over multiple grids

The shared size feature of the grid layout allows it to synchronize the width of columns over multiple grids. The feature is very useful if you want to realize a multi-column listview by using a grid as layout panel within the data template. Because each item contains its own grid, the columns will not have the same width.

By setting the attached property `Grid.IsSharedSizeScope` to `true` on a parent element you define a scope within the column-widths are shared.

To synchronize the width of two columndefinitions, set the `SharedSizeGroup` to the same name.

```
<ItemsControl Grid.IsSharedSizeScope="True" >
  <ItemsControl.ItemTemplate>
    <DataTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition SharedSizeGroup="FirstColumn" Width="Auto"/>
          <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <TextBlock Text="{Binding Path=Key}" TextWrapping="Wrap"/>
        <TextBlock Text="{Binding Path=Value}" Grid.Column="1"
TextWrapping="Wrap"/>
      </Grid>
    </DataTemplate>
  </ItemsControl.ItemTemplate>
</ItemsControl>
```

Useful Hints

Columns and rows that participate in size-sharing do not respect Star sizing. In the size-sharing scenario, Star sizing is treated as Auto. Since TextWrapping on TextBlocks within an SharedSize column does not work you can exclude your last column from the shared size. This often helps to resolve the problem.

Using GridLenghts from code

If you want to add columns or rows by code, you can use the `GridLength` class to define the differenz types of sizes.

Auto sized	`GridLength.Auto`
Star sized	`new GridLength(1,GridUnitType.Star)`
Fixed size	`new GridLength(100,GridUnitType.Pixel)`

```
Grid grid = new Grid();

ColumnDefinition col1 = new ColumnDefinition();
```

```
col1.Width = GridLength.Auto;
ColumnDefinition col2 = new ColumnDefinition();
col2.Width = new GridLength(1,GridUnitType.Star);

grid.ColumnDefinitions.Add(col1);
grid.ColumnDefinitions.Add(col2);
```

WPF StackPanel

Introduction

The `StackPanel` in WPF is a simple and useful layout panel. It stacks its child elements below or beside each other, dependening on its orientation. This is very useful to create any kinds of lists. All WPF `ItemsControls` like `ComboBox`, `ListBox` or `Menu` use a StackPanel as their internal layout panel.

```
<StackPanel>
  <TextBlock Margin="10" FontSize="20">How do you like your
coffee?</TextBlock>
  <Button Margin="10">Black</Button>
  <Button Margin="10">With milk</Button>
  <Button Margin="10">Latte machiato</Button>
```

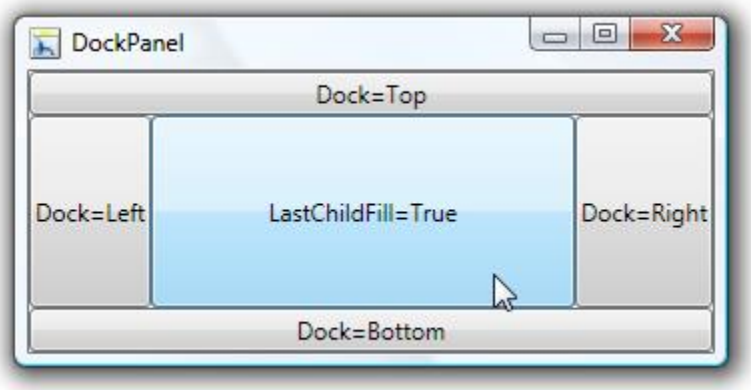
```
  <Button Margin="10">Chappuchino</Button>
</StackPanel>
```

Stack Items horizontally

A good example for a horizontal stack panel are the "OK" and "Cancel" buttons of a dialog window. Because the size of the text can change if the user changes the font-size or switches the language we should avoid fixed sized buttons. The stack panel aligns the two buttons depending on their desired size. If they need more space they will get it automatically. Never mess again with too small or too large buttons.


```
<StackPanel Margin="8" Orientation="Horizontal">
   <Button MinWidth="93">OK</Button>
   <Button MinWidth="93" Margin="10,0,0,0">Cancel</Button>
</StackPanel>
```

Dock Panel



Introduction

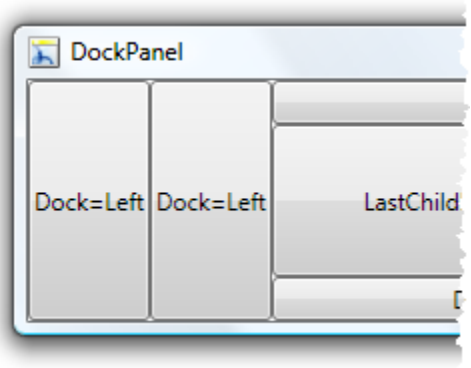
The dock panel is a layout panel, that provides an easy docking of elements to the left, right, top, bottom or center of the panel. The dock side of an element is defined by the attached property

`DockPanel.Dock`. To dock an element to the center of the panel, it must be the last child of the panel and the `LastChildFill` property must be set to true.

```
<DockPanel LastChildFill="True">
    <Button Content="Dock=Top" DockPanel.Dock="Top"/>
    <Button Content="Dock=Bottom" DockPanel.Dock="Bottom"/>
    <Button Content="Dock=Left"/>
    <Button Content="Dock=Right" DockPanel.Dock="Right"/>
    <Button Content="LastChildFill=True"/>
</DockPanel>
```

Multiple elements on one side

The dock panel layout supports multiple elements on one side. Just add two or more elements with the same dock side. The panel simply stacks them.

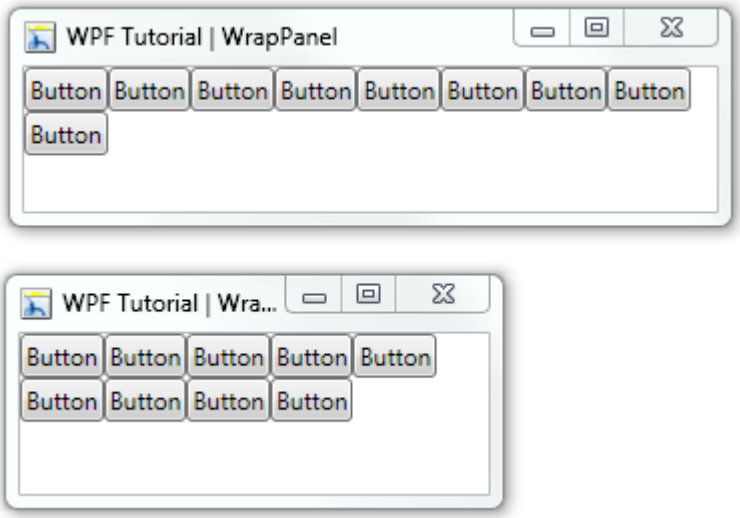



```
<DockPanel LastChildFill="True">
    <Button Content="Dock=Left"/>
    <Button Content="Dock=Left"/>
    <Button Content="Dock=Top" DockPanel.Dock="Top"/>
    <Button Content="Dock=Bottom" DockPanel.Dock="Bottom"/>
    <Button Content="Dock=Right" DockPanel.Dock="Right"/>
    <Button Content="LastChildFill=True"/>
</DockPanel>
```

Change the stacking order

The order of the elements matters. It determines the alignment of the elements. The first elements gets the whole width or height. The following elements get the remaining space.

Wrap Panel



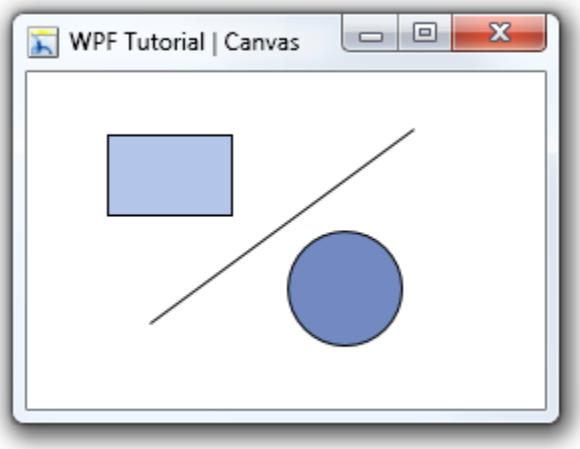
Introduction

The wrap panel is similar to the StackPanel but it does not just stack all child elements to one row, it wraps them to new lines if no space is left. The `Orientation` can be set to `Horizontal` or `Vertical`.

The StackPanel can be used to arrange tabs of a tab control, menu items in a toolbar or items in an Windows Explorer like list. The WrapPanel is often used with items of the same size, but its not a requirement.

```
<WrapPanel Orientation="Horizontal">
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
</WrapPanel>
```

Canvas Panel



Introduction

The Canvas is the most basic layout panel in WPF. It's child elements are positioned by **explicit coordinates**. The coordinates can be specified **relative to any side** of the panel usind the `Canvas.Left`, `Canvas.Top`, `Canvas.Bottom` and `Canvas.Right attached properties.`

The panel is typically used to group 2D graphic elements together and **not to layout user interface elements**. This is important because specifing absolute coordinates brings you in trouble when you begin to resize, scale or localize your application. People coming from WinForms are familiar with this kind of layout - but it's not a good practice in WPF.

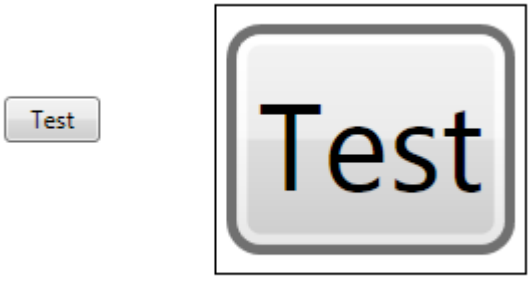
```
<Canvas>
    <Rectangle Canvas.Left="40" Canvas.Top="31" Width="63" Height="41"
Fill="Blue"  />
    <Ellipse Canvas.Left="130" Canvas.Top="79" Width="58" Height="58"
Fill="Blue"  />
    <Path Canvas.Left="61" Canvas.Top="28" Width="133" Height="98"
Fill="Blue"
          Stretch="Fill" Data="M61,125 L193,28"/>
</Canvas>
```

Override the Z-Order of Elements

Normally the Z-Order of elements inside a canvas is specified by the order in XAML. But you can override the natural Z-Order by explicity defining a `Canvas.ZIndex` on the element.


```
<Canvas>
    <Ellipse Fill="Green" Width="60" Height="60" Canvas.Left="30"
Canvas.Top="20"
             Canvas.ZIndex="1"/>
    <Ellipse Fill="Blue"  Width="60" Height="60" Canvas.Left="60"
Canvas.Top="40"/>
</Canvas>
```

How to use the ViewBox in WPF



Introduction

The ViewBox is a very useful control in WPF. If does nothing more than **scale to fit the content** to the available size. It does **not resize** the content, but **it transforms it**. This means that also all text sizes and line widths were scaled. Its about the same behavior as if you set the `Stretch` property on an Image or Path to `Uniform`.

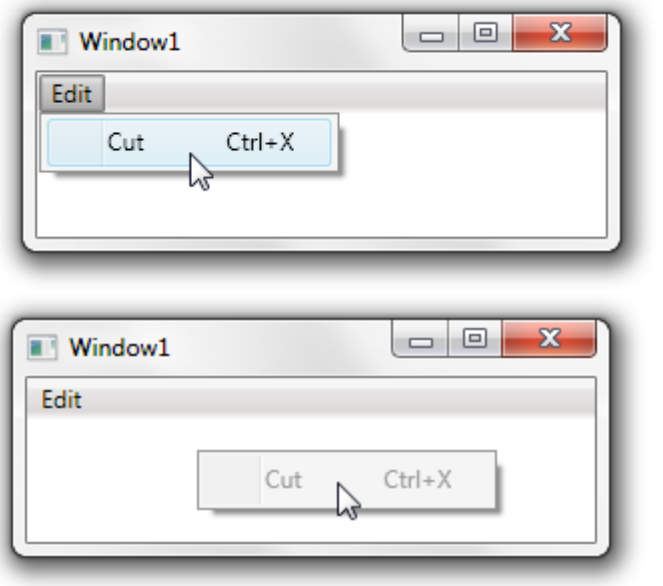
Although it can be used to fit any type of control, it's often used for 2D graphics, or to fit a scalable part of a user interface into an screen area.

```
<Button Content="Test" />

<Viewbox Stretch="Uniform">
    <Button Content="Test" />
</Viewbox>
```

How to Solve Execution Problems of RoutedCommands in a WPF ContextMenu

The Problem

I recently run into a problem, with RoutedCommands in a ContextMenu. The problem was, that the **commands could not be executed**, even if the CommandBinding on the parent window allowed it.



The following example shows the problem with simple window that has a Menu and a ContextMenu on it. Both menus contains a MenuItem with a "Cut" command set.

```
<Window x:Class="RoutedCommandsInPopups.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">
    <StackPanel x:Name="stack" Background="Transparent">
        <StackPanel.ContextMenu>
            <ContextMenu>
                <MenuItem Header="Cut" Command="Cut" />
            </ContextMenu>
        </StackPanel.ContextMenu>
        <Menu>
            <MenuItem Header="Edit" >
                <MenuItem Header="Cut" Command="Cut" />
            </MenuItem>
        </Menu>
    </StackPanel>
</Window>
```

In the codebehind of the Window I added a CommandBinding to handle the "Cut" command.

```
public Window1()
{
    InitializeComponent();

    CommandBindings.Add(
        new CommandBinding(ApplicationCommands.Cut, CutExecuted, CanCut));
}

private void CutExecuted(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Cut Executed");
}

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

The Reason

The reason is, that ContextMenus are separate windows with their own VisualTree and LogicalTree.

The reason is that the CommandManager searches for CommandBindings within the current focus scope. If the current focus scope has no command binding, it transfers the focus scope to the parent focus scope. When you startup your application **the focus scope is not set**. You can check this by calling `FocusManager.GetFocusedElement(this)` and you will receive `null`.

The Solution

Set the Logical Focus

The simplest solution is to **initially set the logical focus** of the parent window that is not null. When the CommandManager searches for the parent focus scope it finds the window and handels the CommandBinding correctly.

```
public Window1()
{
    InitializeComponent();

    CommandBindings.Add(
        new CommandBinding(ApplicationCommands.Cut, CutExecuted, CanCut));

    // Set the logical focus to the window
    Focus();
}
```

...or the same in XAML

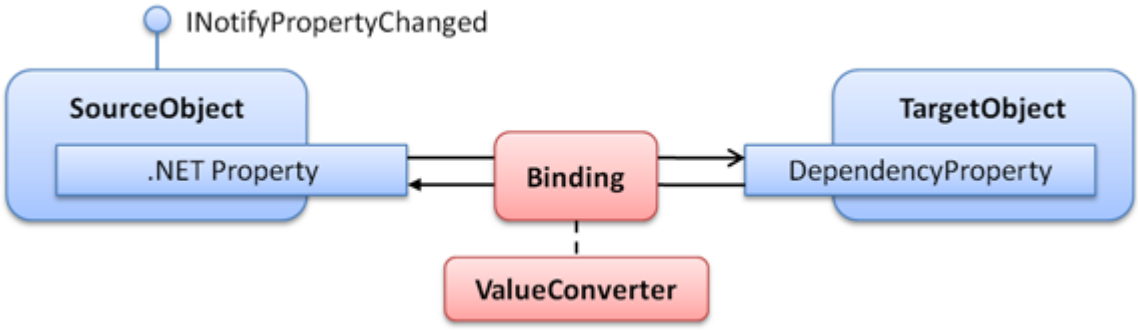
```
<Window x:Class="RoutedCommandsInPopups.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FocusManager.FocusedElement="
        {Binding RelativeSource={x:Static RelativeSource.Self},
Mode=OneTime}>
    ...
</Window>
```

Manually bind the CommandTarget

Another solution is to manually bind the `CommandTarget` to the parent ContextMenu.

```
<MenuItem Header="Cut" Command="Cut" CommandTarget="
          {Binding Path=PlacementTarget,
          RelativeSource={RelativeSource FindAncestor,
          AncestorType={x:Type ContextMenu}}}"/>
```

DataBinding in WPF



Introduction

WPF provides a simple and powerful way to **auto-update data** between the business model and the user interface. This mechanism is called **DataBinding**. Everytime when the data of your business model changes, it automatically reflects the updates to the user interface and vice versa. This is the preferred method in WPF to bring data to the user interface.

Databinding can be **unidirectional** (source -> target or target <- source), **or bidirectional** (source <-> target).

The source of a databinding can be a normal .NET property or a DependencyProperty. The **target property** of the binding **must be a DependencyProperty**.

To make the databinding properly work, both sides of a binding must provide a **change notification** that tells the binding when to update the target value. On normal .NET properties this is done by raising the `PropertyChanged` event of the **`INotifyPropertyChanged`** interface. On DependencyProperties it is done by the PropertyChanged callback of the property metadata

Databinding is **typically done in XAML** by using the **`{Binding}`** markup extension. The following example shows a simple binding between the text of a TextBox and a Label that reflects the typed value:

```
<StackPanel>
    <TextBox x:Name="txtInput" />
    <Label Content="{Binding Text, ElementName=txtInput,
                    UpdateSourceTrigger=PropertyChanged}" />
</StackPanel>
```

DataContext

Every WPF control derived from `FrameworkElement` has a `DataContext` property. This property is **meant to be set to the data object** it visualizes. If you don't explicity define a source of a binding, it takes the data context by default.

The `DataContext` property **inherits its value to child elements**. So you can set the `DataContext` on a superior layout container and its value is inherited to all child elements. This is very useful if you want to build a form that is bound to multiple properties of the same data object.

```
<StackPanel DataContext="{StaticResource myCustomer}">
    <TextBox Text="{Binding FirstName}"/>
    <TextBox Text="{Binding LastName}"/>
    <TextBox Text="{Binding Street}"/>
    <TextBox Text="{Binding City}"/>
</StackPanel>
```

ValueConverters

If you want to **bind two properties of different types** together, you need to use a **ValueConverter**. A ValueConverter converts the value from a source type to a target type and back. WPF already includes some value converters but in most cases you will need to write your own by implementing the `IValueConverter` interface.

A typical example is to bind a boolean member to the `Visibility` property. Since the visibility is an enum value that can be `Visible`, `Collapsed` or `Hidden`, you need a value converter.

```
<StackPanel>
    <StackPanel.Resources>
        <BooleanToVisibilityConverter x:Key="boolToVis" />
    </StackPanel.Resources>

    <CheckBox x:Name="chkShowDetails" Content="Show Details" />
    <StackPanel x:Name="detailsPanel"
                Visibility="{Binding IsChecked, ElementName=chkShowDetails,
                            Converter={StaticResource boolToVis}}">
```

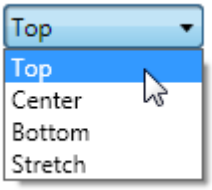
```
    </StackPanel>
</StackPanel>
```

The following example shows a simple converter that converts a boolen to a visibility property. Note that such a converter is already part of the .NET framework.

```
public class BooleanToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
                          CultureInfo culture)
    {
        if (value is Boolean)
        {
            return ((bool)value) ? Visibility.Visible : Visibility.Collapsed;
        }

        return value;
    }

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

Tipp: You can derive your value converter from `MarkupExtension` and return its own instance in the `ProvideValue` override. So you can use it directly without referencing it from the resources.

How to Bind to Values of an Enum



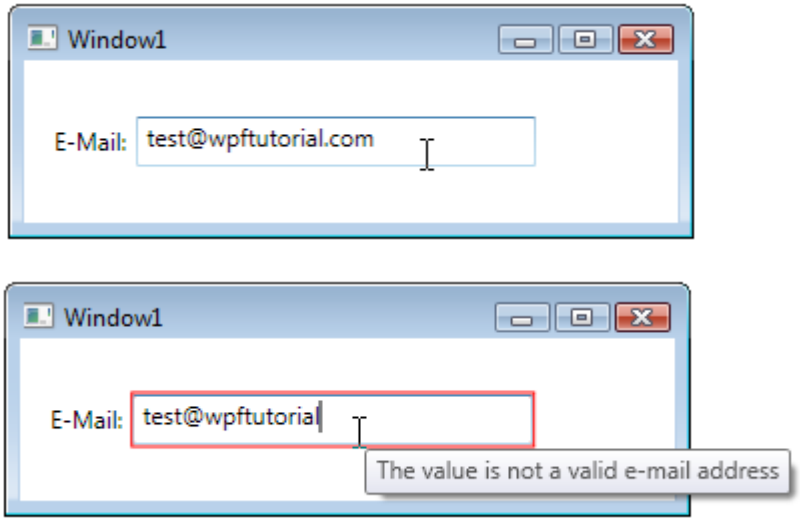
You **cannot directly bind the values of an enum** to a WPF list control, because the enum type does not provide a property that returns all values. The only way to get the names is to **call the GetNames() method**. But how to call a method from XAML?

The trick is to **use an ObjectDataProvider**, that allows us to specify the name of a method and their parameters and he invokes it from XAML. The result can be used by using a normal data binding

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

<Window.Resources>
    <ObjectDataProvider x:Key="aligmnments"
                        MethodName="GetNames" ObjectType="{x:Type sys:Enum}">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="VerticalAlignment" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</Window.Resources>



<ComboBox ItemsSource="{Binding Source={StaticResource aligmnments}}" />
```

Data Validation in WPF

What we want to do is a simple entry form for an e-mail address. If the user enters an invalid e-mail address, the border of the textbox gets red and the tooltip is showing the reason.



Implementing a ValidationRule (.NET 3.0 style)

In this example I am implementing an generic validation rule that takes a regular expression as validation rule. If the expression matches the data is treated as valid.

```
/// <summary>
/// Validates a text against a regular expression
/// </summary>
public class RegexValidationRule : ValidationRule
{
    private string _pattern;
    private Regex _regex;

    public string Pattern
    {
        get { return _pattern; }
        set
        {
            _pattern = value;
            _regex = new Regex(_pattern, RegexOptions.IgnoreCase);
        }
    }
```

```
    public RegexValidationRule()
    {
    }

    public override ValidationResult Validate(object value, CultureInfo
ultureInfo)
    {
        if (value == null || !_regex.Match(value.ToString()).Success)
        {
            return new ValidationResult(false, "The value is not a valid e-
mail address");
        }
        else
        {
            return new ValidationResult(true, null);
        }
    }
}
```

First thing I need to do is place a regular expression pattern as string to the windows resources

```
<Window.Resources>
        <sys:String x:Key="emailRegex">^[a-zA-Z][\w\.-]*[a-zA-Z0-9]@
        [a-zA-Z0-9][\w\.-]*[a-zA-Z0-9]\.[a-zA-Z][a-zA-Z\.]
        *[a-zA-Z]$</sys:String>
</Window.Resources>
```

Build a converter to convert ValidationErrors to a multi-line string

The following converter combines a list of `ValidationErrors` into a string. This makes the binding much easier. In many samples on the web you see the following binding expression:

```
{Binding RelativeSource={RelativeSource
Self},Path=(Validation.Errors)[0].ErrorContent}
```

This expression works if there is one validation error. But if you don't have any validation errors the data binding fails. This slows down your application and causes the following message in your debug window:

```
System.Windows.Data Error: 16 : Cannot get 'Item[]' value (type
'ValidationError') from '(Validation.Errors)' (type
'ReadOnlyObservableCollection`1').
BindingExpression:Path=(0).[0].ErrorContent; DataItem='TextBox'...
```

The converter is both, a value converter and a markup extension. This allows you to create and use it at the same time.

```
[ValueConversion(typeof(ReadOnlyObservableCollection<ValidationError>),
typeof(string))]
public class ValidationErrorsToStringConverter : MarkupExtension,
IValueConverter
{
    public override object ProvideValue(IServiceProvider serviceProvider)
    {
        return new ValidationErrorsToStringConverter();
    }

    public object Convert(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        ReadOnlyObservableCollection<ValidationError> errors =
            value as ReadOnlyObservableCollection<ValidationError>;

        if (errors == null)
        {
            return string.Empty;
        }

        return string.Join("\n", (from e in errors
                                  select e.ErrorContent as
string).ToArray());
    }
```

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

Create an ErrorTemplate for the TextBox

Next thing is to create an error template for the text box.

```
<ControlTemplate x:Key="TextBoxErrorTemplate" TargetType="Control">
    <Grid ClipToBounds="False" >
        <Image HorizontalAlignment="Right" VerticalAlignment="Top"
               Width="16" Height="16" Margin="0,-8,-8,0"
               Source="{StaticResource ErrorImage}"
               ToolTip="{Binding ElementName=adornedElement,
                         Path=AdornedElement.(Validation.Errors),
                         Converter={k:ValidationErrorsToStringConverter}}"/>
        <Border BorderBrush="Red" BorderThickness="1" Margin="-1">
            <AdornedElementPlaceholder Name="adornedElement" />
        </Border>
    </Grid>
</ControlTemplate>
```

The ValidationRule and the ErrorTemplate in Action

Finally we can add the validation rule to our binding expression that binds the `Text` property of a textbox to a EMail property of our business object.

```
<TextBox x:Name="txtEMail" Template={StaticResource TextBoxErrorTemplate}>
    <TextBox.Text>
```

```
        <Binding Path="EMail" UpdateSourceTrigger="PropertyChanged" >
            <Binding.ValidationRules>
                <local:RegexValidationRule Pattern="{StaticResource
emailRegex}"/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

How to manually force a Validation

If you want to force a data validation you can manually call `UpdateSource()` on the binding expression. A useful scenario could be to validate on `LostFocus()` even when the value is empty or to initially mark all required fields. In this case you cann call `ForceValidation()` in the `Loaded` event of the window. That is the time, when the databinding is established.

The following code shows how to get the binding expression from a property of a control.

```
private void ForceValidation()
{
  txtName.GetBindingExpression(TextBox.TextProperty).UpdateSource();
}
```

Themes in WPF

Introduction

This article will follow soon...

How to use specific Windows theme in a WPF application

WPF includes all common Windows themes. By default WPF loads the current Windows theme as your default style-set. But you can override these styles by loading a specific theme. To do this you first have to

add an reference to the style assembly you like to use and second you need to merge the theme resource dictionary into your app resources. This overrides the default style-set that has been loaded by WPF.

The following exmple shows how to load the Windows Vista Aero theme.

```
<App.Resources>
    <ResourceDictionary Source="/PresentationFramework.Aero, Version=3.0.0.0,
           Culture=neutral, PublicKeyToken=31bf3856ad364e35,
           ProcessorArchitecture=MSIL;component/themes/aero.normalcolor.xaml"
/>
</App.Resources>
```

Introduction to Styles in WPF

Introduction

Imagine you want to create an application with a unique design. All your buttons should have an orange background and an italic font. Doing this the conventional way means that you have to set the `Background` and the `FontStyle` property on every single button.


```
<StackPanel Orientation="Horizontal" VerticalAlignment="Top">
    <Button Background="Orange" FontStyle="Italic"
            Padding="8,4" Margin="4">Styles</Button>
    <Button Background="Orange" FontStyle="Italic"
            Padding="8,4" Margin="4">are</Button>
    <Button Background="Orange" FontStyle="Italic"
            Padding="8,4" Margin="4">cool</Button>
</StackPanel>
```

This code is neighter maintainable nor short and clear. The solution for this problem are styles.

The concept of styles let you remove all properties values from the individual user interface elements and combine them into a style. A style consists of a list of setters. If you apply this style to an element it sets all properties with the specified values. The idea is quite similar to Cascading Styles Sheets (CSS) that we know from web development.

To make the style accessible to your controls you need to add it to the resources. Any controls in WPF have a list of resources that is inherited to all controls beneath the visual tree. Thats the reason why we need to specify a `x:Key="myStyle"` property that defines a unique resource identifier.

To apply the style to a control we set the `Style` property to our style. To get it from the resources we use the `{StaticResource [resourceKey]}` markup extension.

```
<Window>
    <Window.Resources>
        <Style x:Key="myStyle" TargetType="Button">
           <Setter Property="Background" Value="Orange" />
           <Setter Property="FontStyle" Value="Italic" />
           <Setter Property="Padding" Value="8,4" />
           <Setter Property="Margin" Value="4" />
        </Style>
    </Window.Resources>

    <StackPanel Orientation="Horizontal" VerticalAlignment="Top">
        <Button Style="{StaticResource myStyle}">Styles</Button>
        <Button Style="{StaticResource myStyle}">are</Button>
        <Button Style="{StaticResource myStyle}">cool</Button>
    </StackPanel>
</Window>
```

What we have achived now is

- A maintainable code base
- Removed the redundancy
- Change the appearance of a set of controls from a single point
- Possibility to swap the styles at runtime.

Style inheritance

A style in WPF can base on another style. This allows you to specify a base style that sets common properties and derive from it for specialized controls.

```
<Style x:Key="baseStyle">
  <Setter Property="FontSize" Value="12" />
  <Setter Property="Background" Value="Orange" />
</Style>


<Style x:Key="boldStyle" BasedOn="{StaticResource baseStyle}">
  <Setter Property="FontWeight" Value="Bold" />
</Style>
```

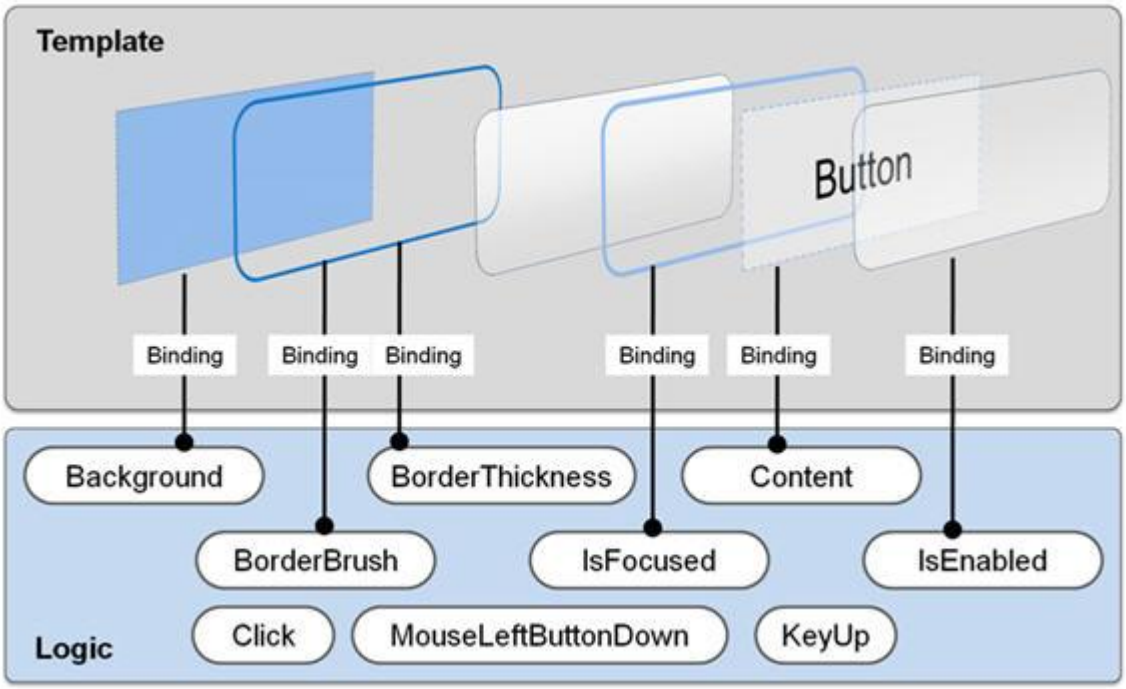
Control Templates

Introduction

Controls in WPF are separated into **logic**, that defines the states, events and properties and **template**, that defines the visual appearance of the control. The wireup between the logic and the template is done by DataBinding.

Each control has a default template. This gives the control a basic appearance. The default template is typically shipped together with the control and available for all common windows themes. It is by convention wrapped into a style, that is identified by value of the `DefaultStyleKey` property that every control has.

The template is defined by a dependency property called `Template`. By setting this property to another instance of a control template, you can completely replace the appearance (visual tree) of a control.



The control template is often included in a style that contains other property settings. The following code sample shows a simple control template for a button with an ellipse shape.

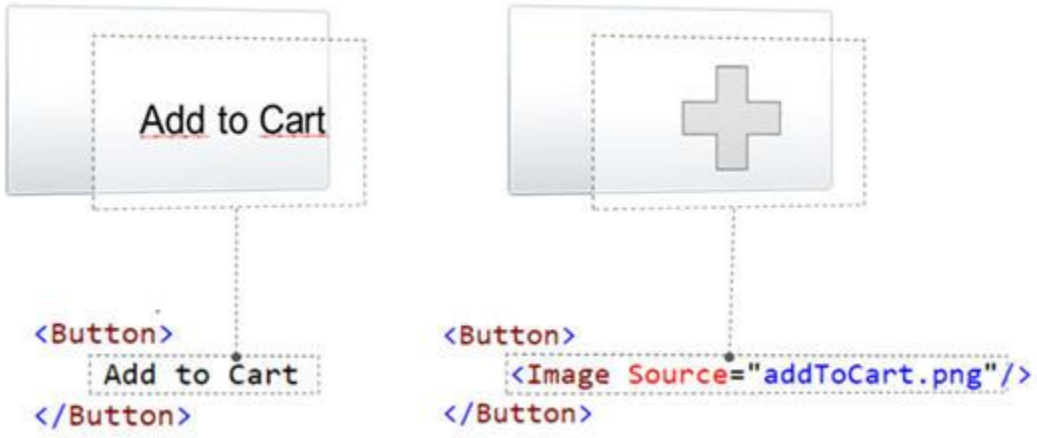
```
<Style x:Key="DialogButtonStyle" TargetType="Button">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type Button}">
                <Grid>
                    <Ellipse Fill="{TemplateBinding Background}"
                             Stroke="{TemplateBinding BorderBrush}"/>
                        <ContentPresenter HorizontalAlignment="Center"
                                          VerticalAlignment="Center"/>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

```
<Button Style="{StaticResource DialogButtonStyle}" />
```

A Button without and with a custom control template

ContentPresenter

When you create a custom control template and you want to define a placeholder that renders the content, you can use the `ContentPresenter`. By default it adds the content of the `Content` property to the visual tree of the template. To display the content of another property you can set the `ContentSource to the name of the property you like.`



Triggers

{RelativeSource TemplatedParent} not working in DataTriggers of a ControlTemplate

If you want to bind to a property of a property on your control like `Data.IsLoaded` you cannot use a normal Trigger, since it does not support this notation, you have to use a DataTrigger.

But when you are using a DataTrigger, with `{RelativeSource TemplatedParent}` it will not work. The reason is, that **`TemplatedParent` can only be used within the ControlTemplate**. It is not working in the Trigger section. You have to use the `{RelativeSource Self}` instead.

Data Templates

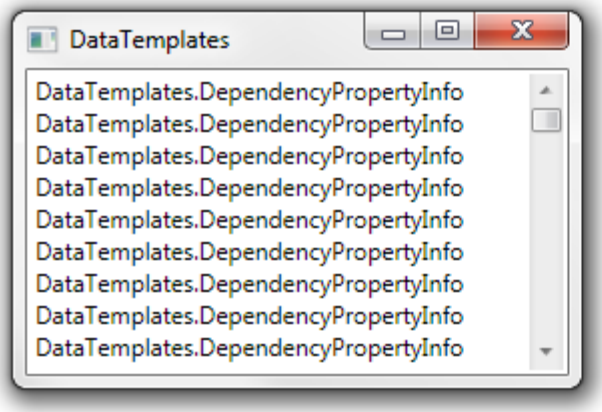
Introduction

Data Template are a similar concept as Control Templates. They give you a very flexible and powerful solution to **replace the visual appearance of a data item** in a control like ListBox, ComboBox or ListView. In my opinion this is one of the key success factory of WPF.

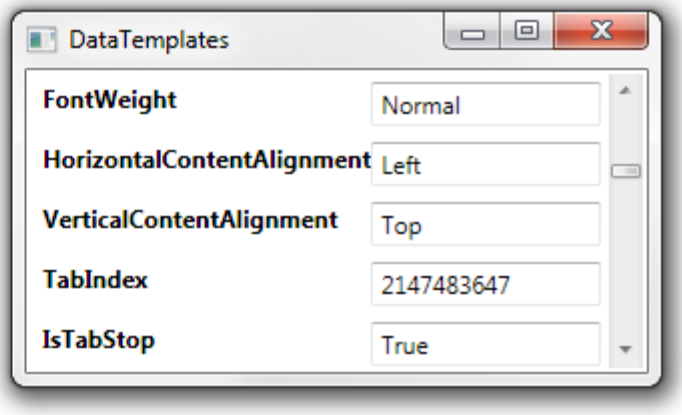
If you don't specify a data template, WPF takes the default template that is just a TextBlock. If you bind complex objects to the control, it just calls `ToString()` on it. Within a DataTemplate, the DataContext is set the data object. So you can easily bind against the data context to display various members of your data object

DataTemplates in Action: Building a simple PropertyGrid

Whereas it was really hard to display complex data in a ListBox with WinForms, its super easy with WPF. The following example shows a ListBox with a list of DependencyPropertyInfo instances bound to it. Without a DataTemplate you just see the result of calling `ToString()` on the object. With the data template we see the name of the property and a TextBox that even allows us to edit the value.



No DataTemplate



With Custom DataTemplate

```
<!-- Without DataTemplate -->
<ListBox ItemsSource="{Binding}" />

<!-- With DataTemplate -->
<ListBox ItemsSource="{Binding}" BorderBrush="Transparent"
         Grid.IsSharedSizeScope="True"
         HorizontalContentAlignment="Stretch">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid Margin="4">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" SharedSizeGroup="Key" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <TextBlock Text="{Binding Name}" FontWeight="Bold"  />
                <TextBox Grid.Column="1" Text="{Binding Value }" />
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

How to use a DataTemplateSelector to switch the Template depending on the data

Our property grid looks nice so far, but it would be much more usable if we could switch the editor depending on the type of the property.

The simplest way to do this is to use a `DataTemplateSelector`. The DataTemplateSelector has a single method to override: `SelectTemplate(object item, DependencyObject container)`. In this method we decide on the provided `item` which DataTemplate to choose.

The following exmple shows an DataTemplateSelector that decides between tree data templates:

```
public class PropertyDataTemplateSelector : DataTemplateSelector
{
    public DataTemplate DefaultnDataTemplate { get; set; }
    public DataTemplate BooleanDataTemplate { get; set; }
```

```
    public DataTemplate EnumDataTemplate { get; set; }

    public override DataTemplate SelectTemplate(object item,
                DependencyObject container)
    {
        DependencyPropertyInfo dpi = item as DependencyPropertyInfo;
        if (dpi.PropertyType == typeof(bool))
        {
            return BooleanDataTemplate;
        }
        if (dpi.PropertyType.IsEnum)
        {
            return EnumDataTemplate;
        }

        return DefaultnDataTemplate;
    }
}
```

```
<Window x:Class="DataTemplates.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:l="clr-namespace:DataTemplates"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">

    <Window.Resources>

        <!-- Default DataTemplate -->
        <DataTemplate x:Key="DefaultDataTemplate">
            ...
        </DataTemplate>

        <!-- DataTemplate for Booleans -->
        <DataTemplate x:Key="BooleanDataTemplate">
            ...
```

```
        </DataTemplate>

        <!-- DataTemplate for Enums -->
        <DataTemplate x:Key="EnumDataTemplate">
            ...
        </DataTemplate>

        <!-- DataTemplate Selector -->
        <l:PropertyDataTemplateSelector x:Key="templateSelector"
              DefaultnDataTemplate="{StaticResource DefaultDataTemplate}"
              BooleanDataTemplate="{StaticResource BooleanDataTemplate}"
              EnumDataTemplate="{StaticResource EnumDataTemplate}"/>
    </Window.Resources>
    <Grid>
        <ListBox ItemsSource="{Binding}" Grid.IsSharedSizeScope="True"
                 HorizontalContentAlignment="Stretch"
                 ItemTemplateSelector="{StaticResource templateSelector}"/>
    </Grid>
</Window>
```

Images in WPF

How to create a Thumbnail of an Image

```
private ImageSource GetThumbnail( string fileName )
{
   byte[] buffer = File.ReadAllBytes(fileName);
   MemoryStream memoryStream = new MemoryStream(buffer);

   BitmapImage bitmap = new BitmapImage();
   bitmap.BeginInit();
   bitmap.DecodePixelWidth = 80;
   bitmap.DecodePixelHeight = 60;
   bitmap.StreamSource = memoryStream;
   bitmap.EndInit();
```

```
    bitmap.Freeze();

    return bitmap;
}
```