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

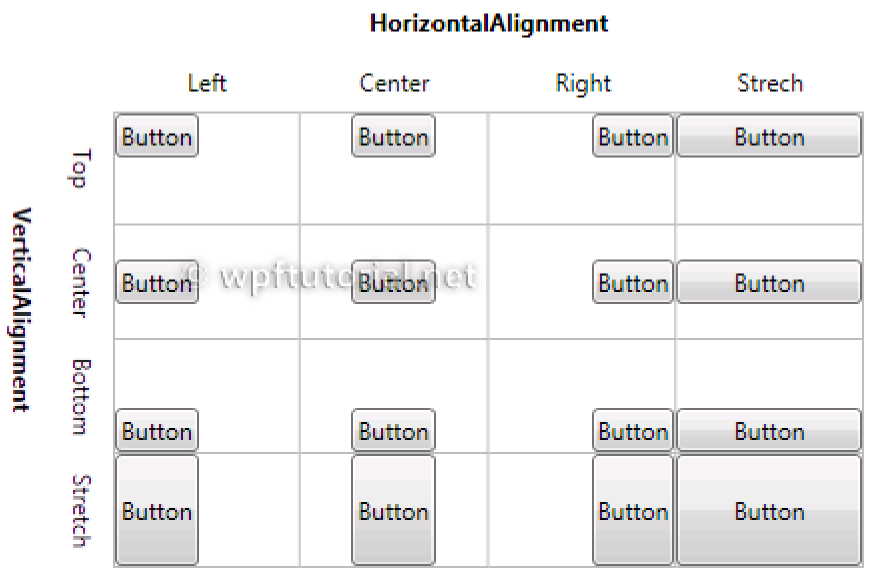
Best Practices

- 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

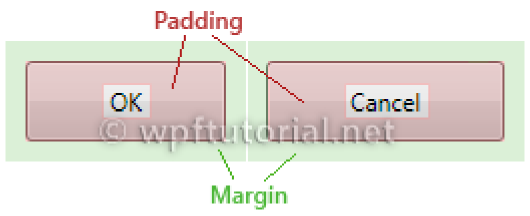
Use the `VerticalAlignment` and `HorizontalAlignmant` properties to dock the controls to one or multiple sides of the panel. The following illustrations show how the sizing behaves with the different combinations.



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.



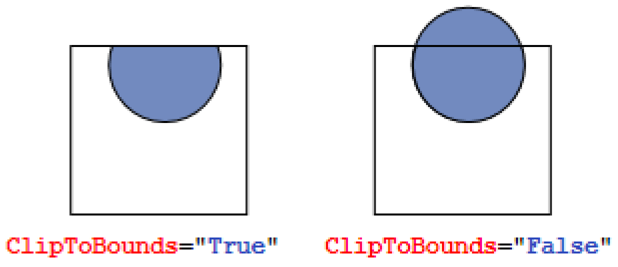
Height and Width

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.



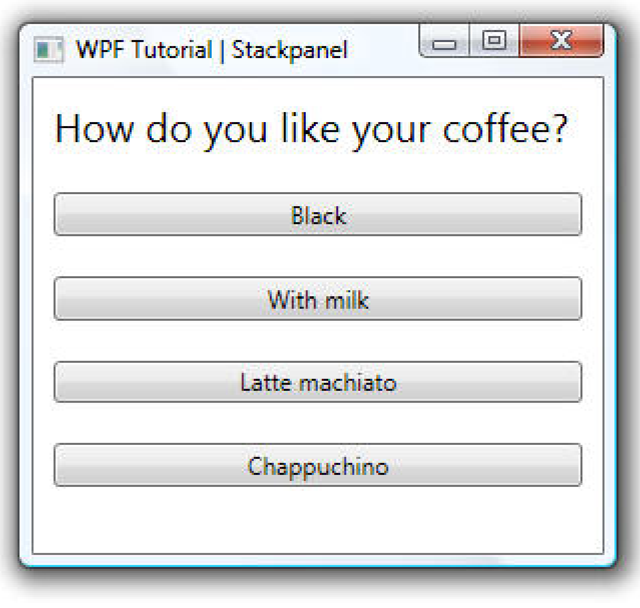
Scrolling

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>
```

WPF StackPanel



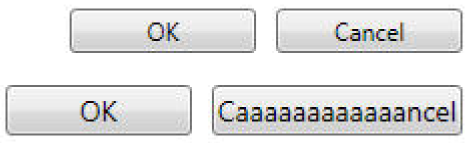
Introduction

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>
```

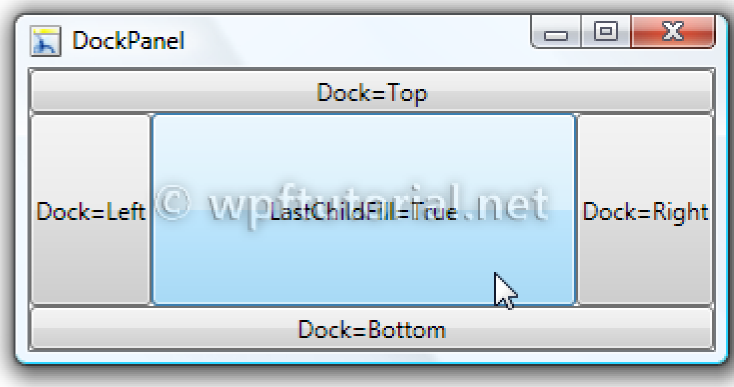
Stack Items horizontally

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

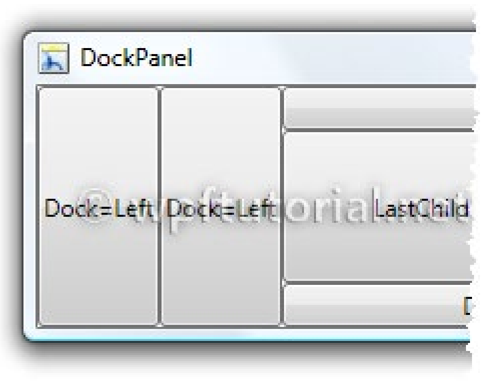
The dock panel is a layout panel, that provides an easy docking of elements to the left, right, top, bottom or center of the panel. The dock side of an element is defined by the attached property DockPanel.Dock. To dock an element to the center of the panel, it must be the last child of the panel and 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>
```

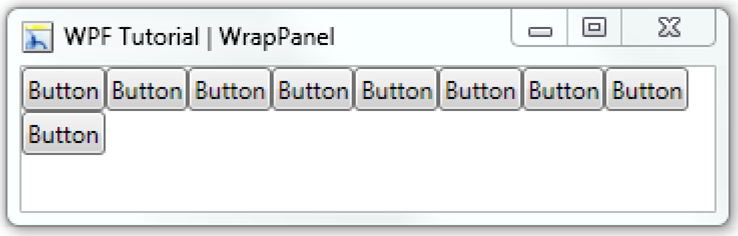
Multiple elements on one side

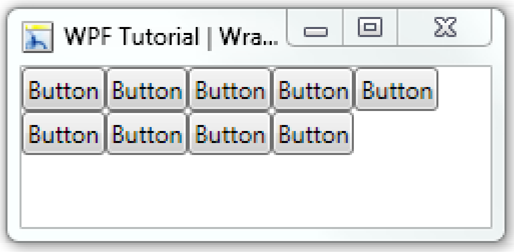
The dock panel layout supports multiple elements on one side. Just add two or more elements with the same dock side. The panel simply stacks them.



```
<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>
```

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 WrapPanel 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.

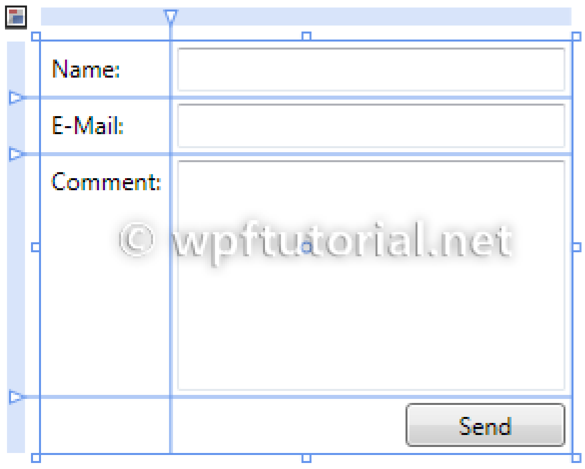
```
<WrapPanel Orientation="Horizontal">
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
</WrapPanel>
```

Grid Panel

How to share the width of a column over multiple grids

Using GridLenghts from code

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 (after filling all auto and fixed sized columns), proportionally divided over all star-sized columns. So 3*/5* means the same as 30*/50*. 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>
```

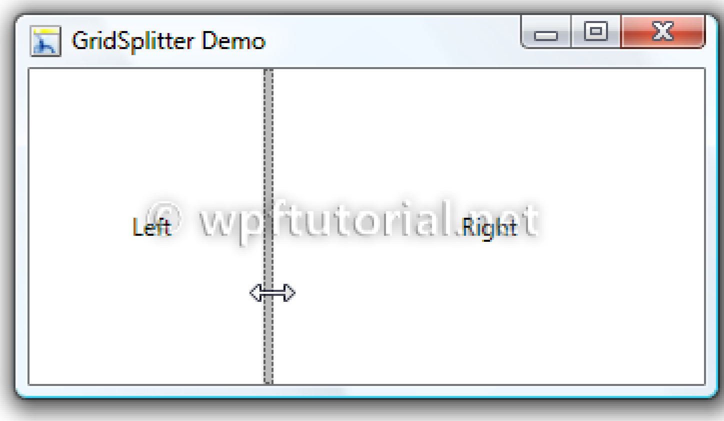
How to add controls to the grid

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 theResizeBehavior 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.

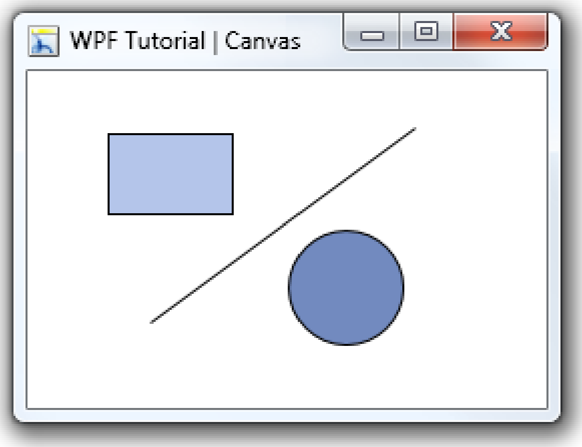
Auto sized	GridLength.Auto
Star sized	new GridLength(1,GridUnitType.Star)
Fixed size	new GridLength(100,GridUnitType.Pixel)

```
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);
```

Canvas Panel



Introduction

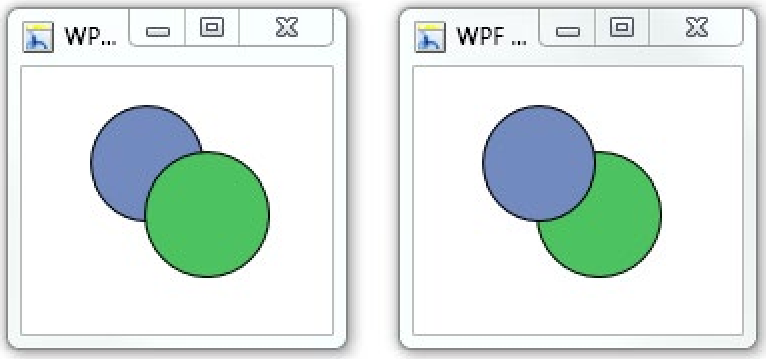
The Canvas is the most basic layout panel in WPF. It's child elements are positioned by **explicit coordinates**. The coordinates can be specified **relative to any side** of the panel usind the Canvas.Left, Canvas.Top, Canvas.Bottom andCanvas.Right attached properties.

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>
```

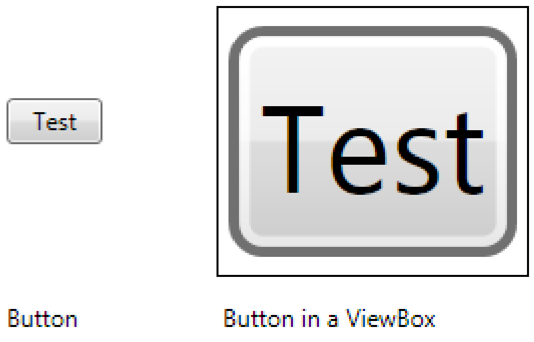
Override the Z-Order of Elements

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>
```

WPF already ships with a rich set of layout panels. Each of them fits to solve a particular problem.

- Canvas - To arrange elements by X,Y coordinates.
- Grid - To arrange elements based on rows and columns.
- StackPanel - To stack elements horizontally or vertically.
- DockPanel - To dock elements to a particular side.
- WrapPanel - To stack elements and automatically begin a new row.

But sometimes none of the included layout panels helps you to arrange child elements in the way you like it. In this case its easy to write your own layout panel. All you need to do is to create a new class that derives from `Panel`. There are two methods to override:

- **MeasureOverride** - to determine the required size of the panel according to the desired size of the child elements and the available space.

- **ArrangeOverride** - to arrange the elements in the finally available space. The final size can be smaller than the requested.

```
public class MySimplePanel : Panel
{
    // Make the panel as big as the biggest element
    protected override Size MeasureOverride(Size availableSize)
    {
        Size maxSize = new Size();

        foreach( UIElement child in InternalChildern)
        {
            child.Measure( availableSize );
            maxSize.Height = Math.Max( child.DesiredSize.Height,
maxSize.Height);
            maxSize.Width= Math.Max( child.DesiredSize.Width,
maxSize.Width);
        }
    }

    // Arrange the child elements to their final position
    protected override Size ArrangeOverride(Size finalSize)
    {
        foreach( UIElement child in InternalChildern)
        {
            child.Arrange( new Rect( finalSize ) );
        }
    }
}
```