# 2. WPF - Layout

## Content

When you create a user interface, you arrange your controls by location and size to form a layout. A key requirement of any layout is to **adapt to changes in window size and display settings**. Rather than forcing you to write the code to adapt a layout in these circumstances, WPF provides a first-class, extensible layout system for you [1].

The cornerstone of the layout system is relative positioning, which increases the ability to adapt to changing window and display conditions. The layout system also manages the negotiation between controls to determine the layout. The negotiation is a two-step process: first, a **control tells its parent what location and size it requires**. Second, the **parent tells the control what space it can have**.

The layout system is exposed to child controls through base WPF classes. For common layouts such as grids, stacking, and docking, WPF includes several layout controls:

- StackPanel: Child controls are stacked either vertically or horizontally.
- Grid: Child controls are positioned by rows and columns.
- Canvas: Child controls provide their own layout.
- DockPanel: Child controls are aligned to the edges of the panel.
- WrapPanel: Child controls are positioned in left-to-right order and wrapped to the next line when there isn't enough space. on the current line.

All these layout controls derive from the Panel base class.
More details on each layout panel are explained further on in this document.
But first, in the next section, some common layout properties are described.

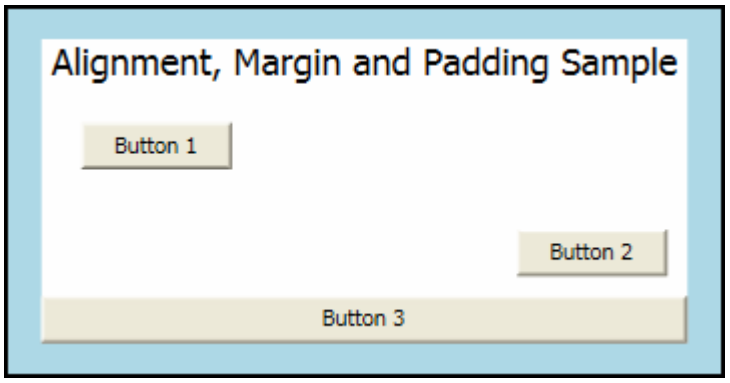
## 2.1 Common layout properties

The FrameworkElement class (base class for all WPF elements) exposes several properties that are used to precisely position child elements. This topic discusses four of the most important properties: HorizontalAlignment, Margin, Padding, and VerticalAlignment. The effects of these properties are important to understand, because they provide the basis for controlling the position of elements in Windows Presentation Foundation (WPF) applications.

### 2.1.1 Element Positioning

There are numerous ways to position elements using WPF. However, achieving ideal layout goes beyond simply choosing the right Panel element. Fine control of positioning requires an understanding of the HorizontalAlignment, Margin, Padding, and VerticalAlignment properties.

The following illustration shows a layout scenario that utilizes several positioning properties.

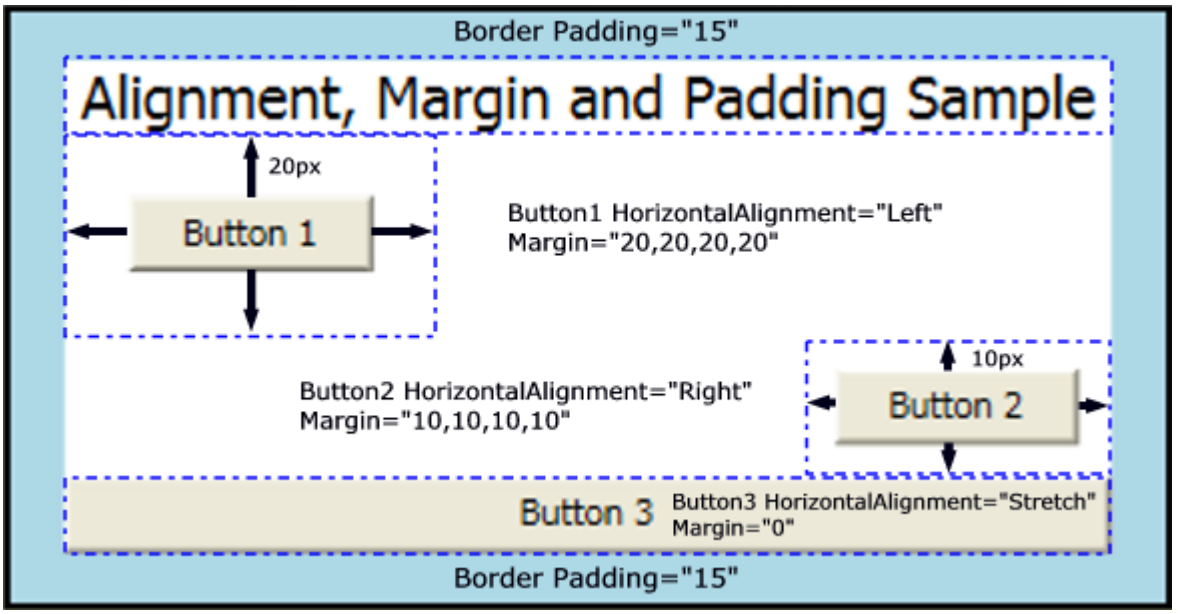



At first glance, the Button elements in this illustration may appear to be placed randomly. However, their positions are actually precisely controlled by using a combination of margins, alignments, and padding.

The following example describes how to create the layout in the preceding illustration. A Border element encapsulates a parent StackPanel, with a Padding value of 15 device independent pixels. This accounts for the narrow LightBlue band that surrounds the child StackPanel. Child elements of the StackPanel are used to illustrate each of the various positioning properties that are detailed in this topic. Three Button elements are used to demonstrate both the Margin and HorizontalAlignment properties.

```
// Create the application's main Window.
mainWindow = new Window ();
mainWindow.Title = "Margins, Padding and Alignment Sample";

// Add a Border
myBorder = new Border();
myBorder.Background = Brushes.LightBlue;
myBorder.BorderBrush = Brushes.Black;
myBorder.Padding = new Thickness(15);
myBorder.BorderThickness = new Thickness(2);

myStackPanel = new StackPanel();
myStackPanel.Background = Brushes.White;
myStackPanel.HorizontalAlignment = HorizontalAlignment.Center;
myStackPanel.VerticalAlignment = VerticalAlignment.Top;

TextBlock myTextBlock = new TextBlock();
myTextBlock.Margin = new Thickness(5, 0, 5, 0);
myTextBlock.FontSize = 18;
myTextBlock.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock.Text = "Alignment, Margin and Padding Sample";
Button myButton1 = new Button();
myButton1.HorizontalAlignment = HorizontalAlignment.Left;
myButton1.Margin = new Thickness(20);
myButton1.Content = "Button 1";
Button myButton2 = new Button();
myButton2.HorizontalAlignment = HorizontalAlignment.Right;
myButton2.Margin = new Thickness(10);
myButton2.Content = "Button 2";
Button myButton3 = new Button();
myButton3.HorizontalAlignment = HorizontalAlignment.Stretch;
myButton3.Margin = new Thickness(0);
myButton3.Content = "Button 3";

// Add child elements to the parent StackPanel.
myStackPanel.Children.Add(myTextBlock);
myStackPanel.Children.Add(myButton1);
myStackPanel.Children.Add(myButton2);
myStackPanel.Children.Add(myButton3);

// Add the StackPanel as the lone Child of the Border.
myBorder.Child = myStackPanel;

// Add the Border as the Content of the Parent Window Object.
mainWindow.Content = myBorder;
mainWindow.Show ();
```

The following diagram provides a close-up view of the various positioning properties that are used in the preceding sample. Subsequent sections in this topic describe in greater detail how to use each positioning property.




## 2.1.2 Understanding Alignment Properties

The HorizontalAlignment and VerticalAlignment properties describe how a child element should be positioned within a parent element's allocated layout space. By using these properties together, you can position child elements precisely. For example, child elements of a DockPanel can specify four different horizontal alignments: Left, Right, or Center, or to Stretch to fill available space. Similar values are available for vertical positioning.

Note:
Explicitly-set Height and Width properties on an element take precedence over the Stretch property value. Attempting to set Height, Width, and a HorizontalAlignment value of Stretch results in the Stretch request being ignored.

### 2.1.2.1 HorizontalAlignment Property

The HorizontalAlignment property declares the horizontal alignment characteristics to apply to child elements. The following table shows each of the possible values of the HorizontalAlignment property.

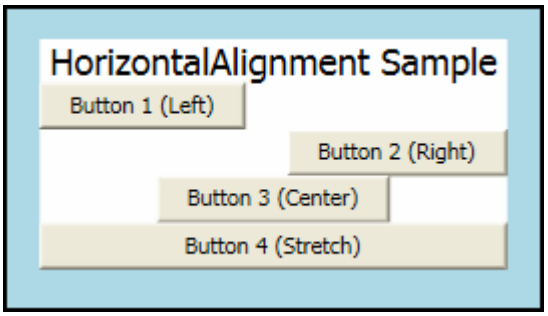
| Member | Description |
| --- | --- |

| | |
|---|---|
| Left | Child elements are aligned to the left of the parent element's allocated layout space. |
| Center | Child elements are aligned to the center of the parent element's allocated layout space. |
| Right | Child elements are aligned to the right of the parent element's allocated layout space. |
| Stretch (Default) | Child elements are stretched to fill the parent element's allocated layout space. Explicit Width and Height values take precedence. |

The following example shows how to apply the HorizontalAlignment property to Button elements. Each attribute value is shown, to better illustrate the various rendering behaviors.

```
Button myButton1 = new Button();
myButton1.HorizontalAlignment = HorizontalAlignment.Left;
myButton1.Content = "Button 1 (Left)";
Button myButton2 = new Button();
myButton2.HorizontalAlignment = HorizontalAlignment.Right;
myButton2.Content = "Button 2 (Right)";
Button myButton3 = new Button();
myButton3.HorizontalAlignment = HorizontalAlignment.Center;
myButton3.Content = "Button 3 (Center)";
Button myButton4 = new Button();
myButton4.HorizontalAlignment = HorizontalAlignment.Stretch;
myButton4.Content = "Button 4 (Stretch)";
```

The preceding code yields a layout similar to the following image. The positioning effects of each HorizontalAlignment value are visible in the illustration.




### 2.1.2.2 VerticalAlignment Property

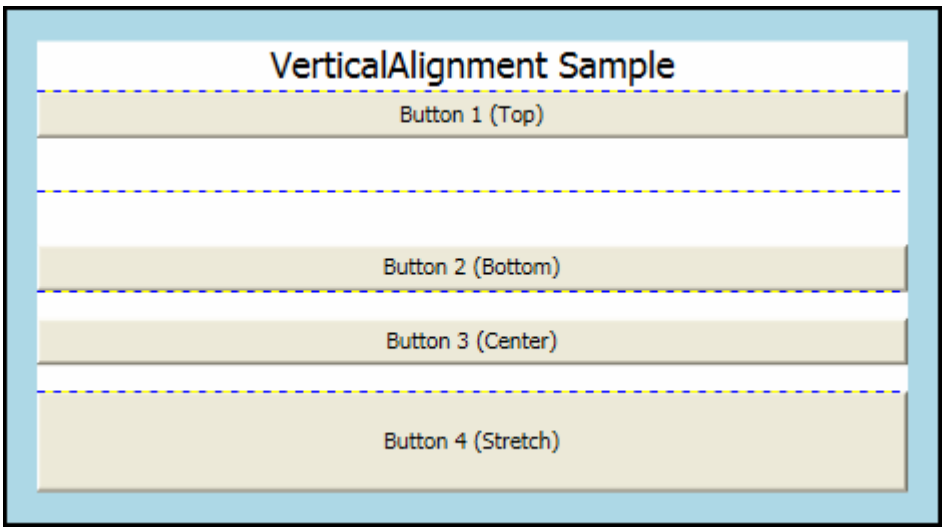
The VerticalAlignment property describes the vertical alignment characteristics to apply to child elements. The following table shows each of the possible values for the VerticalAlignment property.

| Member | Description |
|---|---|
| Top | Child elements are aligned to the top of the parent element's allocated layout space. |
| Center | Child elements are aligned to the center of the parent element's allocated layout space. |
| Bottom | Child elements are aligned to the bottom of the parent element's allocated layout space. |
| Stretch (Default) | Child elements are stretched to fill the parent element's allocated layout space. Explicit Width and Height values take precedence. |

The following example shows how to apply the VerticalAlignment property to Button elements. Each attribute value is shown, to better illustrate the various rendering behaviors. For purposes of this sample, a Grid element with visible gridlines is used as the parent, to better illustrate the layout behavior of each property value.

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      WindowTitle="VerticalAlignment Sample">
  <Border Background="LightBlue" BorderBrush="Black" BorderThickness="2" Padding="15">
    <Grid Background="White" ShowGridLines="True">
      <Grid.RowDefinitions>
        <RowDefinition Height="25"/>
        <RowDefinition Height="50"/>
        <RowDefinition Height="50"/>
        <RowDefinition Height="50"/>
        <RowDefinition Height="50"/>
      </Grid.RowDefinitions>
           <TextBlock Grid.Row="0" Grid.Column="0" FontSize="18" HorizontalAlignment="Center">VerticalAlignme
           <Button Grid.Row="1" Grid.Column="0" VerticalAlignment="Top">Button 1 (Top)</Button>
           <Button Grid.Row="2" Grid.Column="0" VerticalAlignment="Bottom">Button 2 (Bottom)</Button>
           <Button Grid.Row="3" Grid.Column="0" VerticalAlignment="Center">Button 3 (Center)</Button>
           <Button Grid.Row="4" Grid.Column="0" VerticalAlignment="Stretch">Button 4 (Stretch)</Button>
    </Grid>
  </Border>
</Page>
```

The preceding code yields a layout similar to the following image. The positioning effects of each VerticalAlignment value are visible in the illustration.




### 2.1.3 Understanding Margin Properties

The Margin property describes the distance between an element and its child or peers. Margin values can be uniform, by using syntax like `Margin="20"`. With this syntax, a uniform Margin of 20 device independent pixels would be applied to the element. Margin values can also take the form of four distinct values, each value describing a distinct margin to apply to the left, top, right, and bottom (in that order), like `Margin="0,10,5,25"`. Proper use of the Margin property enables very fine control of an element's rendering position and the rendering position of its neighbor elements and children.

Note

A non-zero margin applies space outside the element's ActualWidth and ActualHeight.

The following example shows how to apply uniform margins around a group of Button elements. The Button elements are spaced evenly with a ten-pixel margin buffer in each direction.

```
<Button Margin="10">Button 7</Button>
<Button Margin="10">Button 8</Button>
<Button Margin="10">Button 9</Button>
```

In many instances, a uniform margin is not appropriate. In these cases, non-uniform spacing can be applied. The following example shows how to apply non-uniform margin spacing to child elements. Margins are described in this order: left, top, right, bottom.

```
<Button Margin="0,10,0,10">Button 1</Button>
<Button Margin="0,10,0,10">Button 2</Button>
<Button Margin="0,10,0,10">Button 3</Button>
```

### 2.1.4 Understanding the Padding Property

Padding is similar to Margin in most respects. The Padding property is exposed on only on a few classes, primarily as a convenience: Block, Border, Control, and TextBlock are samples of classes that expose a Padding property. The Padding property enlarges the effective size of a child element by the specified Thickness value.

The following example shows how to apply Padding to a parent Border element.

```
<Border Background="LightBlue"
        BorderBrush="Black"
        BorderThickness="2"
        CornerRadius="45"
        Padding="25">
```

## 2.2 Width and Height

The width or height of a FrameworkElement can also be explicitly set.
The following properties can be used:

| Member | Description |
|---|---|
| Width | The width of the element, in device-independent units (1/96th inch per unit). |
| MinWidth | The minimum width of the element, in device-independent units (1/96th inch per unit). |
| MaxWidth | The maximum width of the element, in device-independent units (1/96th inch per unit). |
| Height | The height of the element, in device-independent units (1/96th inch per unit). |
| MinHeight | The minimum height of the element, in device-independent units (1/96th inch per unit). |
| MaxHeight | The maximum height of the element, in device-independent units (1/96th inch per unit). |

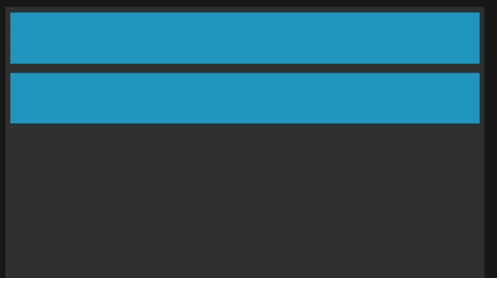
## 2.3 StackPanel

Arranges child elements into a single line that can be oriented horizontally or vertically (default).

A StackPanel contains a collection of UIElement objects, which are in the Children property.

The default value is stretch for both HorizontalAlignment and VerticalAlignment of content that is contained in a StackPanel.

```
<StackPanel>
  <Rectangle Fill="LightBlue"
    Height="20" Margin="2"/>
  <Rectangle Fill="LightBlue"
    Height="20" Margin="2"/>
</StackPanel>
```



The Orientation of the Children can be set to Horizontal as illustrated below:

```
<StackPanel Orientation="Horizontal">
  <Rectangle Fill="LightBlue"
    Width="20" Margin="2"/>
  <Rectangle Fill="LightBlue"
    Width="20" Margin="2"/>
</StackPanel>
```

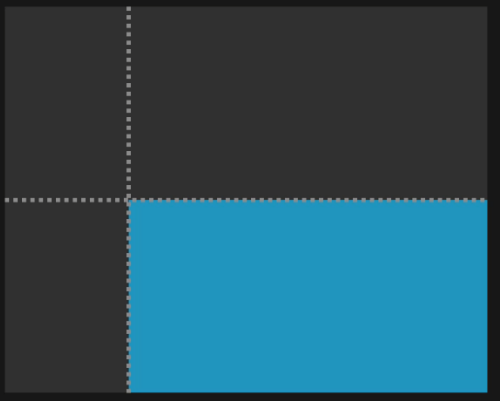
## 2.4 Grid

The Grid allows you to position your elements in rows and columns.

To define rows, you use the Grid's RowDefinitions property with a property element, *Grid.RowDefinitions*.
Creating columns in the Grid works the same way. You add ColumnDefinition objects to the Grid's ColumnDefinitions property.

In the following example a Grid with 2 rows and 2 columns is defined:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="40"/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Rectangle Fill="LightBlue"
    Grid.Column="1" Grid.Row="1"/>
</Grid>
```



A row definition has a Height property and the column definition has a Width property. The value for the width or height can be

- A Star size.
    - Star values are expressed as * or 2*. In the first case, the row or column would receive one share of the available space; in the second case, the row or column would receive two shares of the available space, and so on.
- The value Auto
    - The row or column will use the least amount of space necessary to display the children in that row.
- A number
    - The size is the specified number of pixels

The default width or height for a row or column is * (one Star, same as 1*). So in the example above the 2 rows both have a size of 1*, resulting in each row getting 50% of the available space.

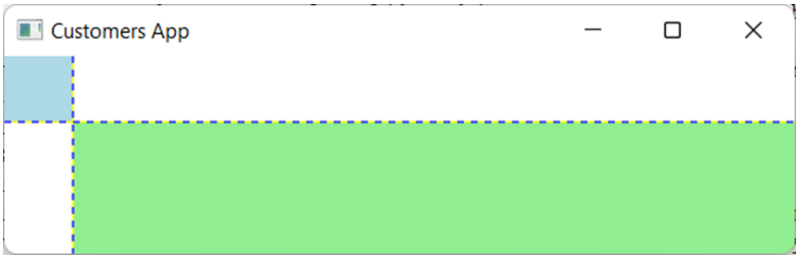
The first column in the example above, has a fixed width of 40 pixels. The second column has a 1* size, so it gets the remaining space.

A Grid contains a collection of UIElement objects, which are in the Children property. By default all children are positioned in the first column (0) of the first row (0).

To position child elements in another column or row, the attached properties *Grid.Column* and *Grid.Row* can be set on those elements.
The blue rectangle, in the example above, is placed in the second column of the second row.

Example:

```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Rectangle Fill="LightBlue" Width="40"></Rectangle>
    <Rectangle Fill="LightGreen" Grid.Row="1" Grid.Column="1"></Rectangle>
</Grid>
```




In the example above the blue rectangle is positioned in the first cell of the Grid (Grid.Row=0, Grid.Column=0 are the defaults).
The green rectangle is positioned in the second row and second column of the Grid (Grid.Row=1, Grid.Column=1)
The rows are star sized. The first row gets one share of the available space, the second row gets 2 shares. This results in the second row being twice as high as the first row, no matter what the size of the window is.
The first column has width *Auto* resulting in a width of 40 since the biggest child element in the first column needs 40 pixels to display itself.
The second column gets the remaining space available (1*).

When there are multiple child elements in the same Grid cell, You can precisely position them by using a combination of the Margin property and alignment properties.

The Grid  is quite often used as a root panel.
Often, other layout panels are nested in the Grid.

## 2.5 Canvas

Defines an area within which you can explicitly position child elements by using coordinates that are relative to the Canvas area.
Child elements of a Canvas are never resized, they are just positioned at their designated coordinates.

Example:

```
<Canvas>
  <Rectangle Fill="LightBlue"
    Height="50" Width="50“
    Canvas.Left="50" Panel.ZIndex="1"
    Canvas.Top="100"/>
  <Rectangle Fill="Orange"
    Height="50" Width="50"
    Canvas.Left="75"
    Canvas.Top="125"/>
</Canvas>
```

The Canvas in the example above contains a light blue rectangle with a height and a width of 50. On the blue rectangle the attached property Canvas.Left is set to 50 to move the rectangle 50 pixels away from the left side of the Canvas. The attached property Canvas.Top is set to 100, to move the rectangle 100 pixels down from the top.
The orange rectangle with the same size is moved 75 away from the left side (Canvas.Left), moved 125 away from the top (Canvas.Top).
Normally the  orange rectangle is rendered on top of the light blue rectangle. This is because the orange rectangle is added to the Canvas after the light blue rectangle. The XAML document is read from top to bottom. To get a different behavior, the attached property Panel.ZIndex is set on the blue rectangle to a value that is greater than 0. Zero is the default value for the ZIndex property. With this value, the blue rectangle is rendered on top of the orange rectangle.

The Canvas is a fantastic panel to position shapes like rectangles, ellipses, lines, and polygons. With the Canvas and these shape elements, you can create nice drawings and diagrams in your WPF application, but the Canvas is not a good choice to position UI elements like text boxes and buttons.
This is because the Canvas positions elements absolutely, which means it does not adjust an element size if there is more space available,
To make child elements adjust their size to the available space, use another Panel element, like Grid, instead of Canvas.

## 2.6 DockPanel

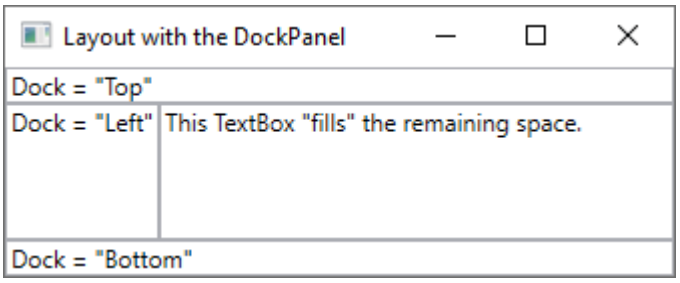
Defines an area where you can arrange child elements either horizontally or vertically, relative to each other.

The following example uses a DockPanel to lay out several TextBox controls:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.LayoutWindow"
    Title="Layout with the DockPanel" Height="143" Width="319">

  <!--DockPanel to layout four text boxes-->
  <DockPanel>
    <TextBox DockPanel.Dock="Top">Dock = "Top"</TextBox>
    <TextBox DockPanel.Dock="Bottom">Dock = "Bottom"</TextBox>
    <TextBox DockPanel.Dock="Left">Dock = "Left"</TextBox>
    <TextBox Background="White">This TextBox "fills" the remaining space.</TextBox>
  </DockPanel>

</Window>
```

The DockPanel allows the child TextBox controls to tell it how to arrange them. To do this, the DockPanel implements a `Dock` attached property that is exposed to the child controls to allow each of them to specify a dock style.

The following figure shows the result of the XAML markup in the preceding example:

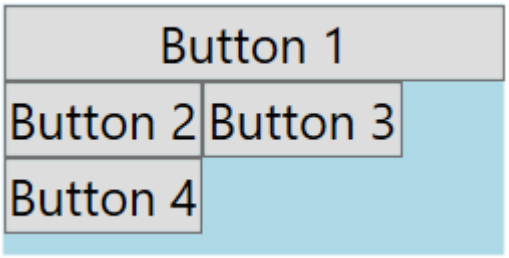



## 2.7 WrapPanel

Positions child elements in sequential position from left to right, breaking content to the next line at the edge of the containing box. Subsequent ordering happens sequentially from top to bottom or from right to left, depending on the value of the Orientation property.

Example:

```
<WrapPanel Background="LightBlue" Width="200" Height="100">
    <Button Width="200">Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
</WrapPanel>
```




## 2.8 Sources

[1] Microsoft, “Desktop Guide (WPF .NET)”, Available: https://docs.microsoft.com/nl-nl/dotnet/desktop/wpf