**Chapter 4.1: Layouts**

**Table of Contents**

* WPF Layout System – An Introduction
* WPF Layout – Size, Width, and Height
* [WPF Layout – Margins](http://www.c-sharpcorner.com/uploadfile/mahesh/wpf-layout-%E2%80%93-margins80/)
* WPF Layout – Horizontal and Vertical Alignment
* WPF Layout – Content Alignment
* WPF Layout – Managing Percentage Size
* WPF Layout – Panels
  + WPF Layout – Canvas Panel
  + WPF Layout – DockPanel
  + WPF Layout – Grid Panel
  + WPF Layout – StackPanel
  + WPF Layout – WrapPanel
  + WPF Layout – Border
  + WPF Layout – VirtualizingStackPanel

Proper layout and positioning play a vital role when it comes to making a UI interactive, high performance and user-friendly. In this chapter, we will discuss the layout process in WPF. The chapter begins with an understanding of the WPF layout process. Further along, will cover the basics of layout and positioning such as size, margin, padding, and alignment of elements. Later in this chapter, we will cover various panels and other parent controls available in WPF.

**WPF Layout System – An Introduction**

Take any UI framework, the first thing that grabs the attention are the layouts, every framework has its own set of panels to arrange data and elements on the screen. Layouts make sure everything is consistent concerning size, alignment, margin, position etc. WPF has a collection of rich UI Panels to take care of these arrangements. The layout process in WPF includes the selection of a suitable parent container, initial placement and positioning, setting margins, paddings, and alignments of parent and child elements. Usually, a parent container is a Window, a UserControl or a Page depending on the type of WPF application. Once a parent container is placed inside a Window, UserControl or Page, then they become the host of child container and other UI elements.

Figure 1 is an example of a typical Window with a title, a border, and minimize, maximize, close buttons. The blue area is a parent container element that usually is a panel. (In this example it is a Grid). The pink is a child of blue parent and the green area is the child of the pink area. That makes blue Grid parent of all other controls inside this window.

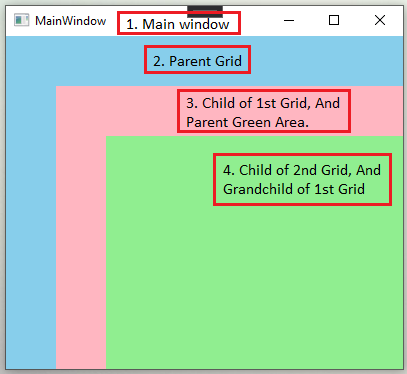


Figure 1

The sub-system that is responsible for the layout and positioning in WPF is called the ***Layout System*.** The layout system is not only responsible for designing user interfaces at design-time but also manages the rendering of elements at runtime. The layout system also manages the event processing of the elements.

**Control vs. Element**

By now it must be clear that what building UI at design-time and run-time is (We have covered this in chapter 1). WPF has two common ways to represent UI objects. Let’s take a Button, for example, you can represent a button with <Button> tag in XAML or Button class in C#. The run-time representation of a UI object in C# is often called as a control and the design-time representation of a UI object in XAML is often called as an element.

**Layout Slot and Layout Clip**

In WPF, each element is defined within a rectangle shape that represents the boundaries of an element. This rectangle is called *the* ***layout slot***. The actual size of this rectangle is calculated by the layout system at runtime after performing calculations based on the screen size, parent properties and element properties such as border, width, height, margin, and padding. After calculating the layout properties of the element, the final visible region of an element is called *the* ***layout clip***.

For example, Figure 2 and Figure 3 show the layout slot and layout clip of an element.

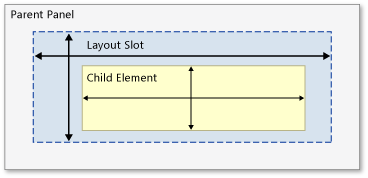


Figure 2

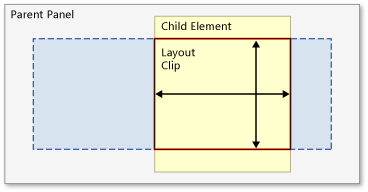


Figure 3

**Get Layout Information**

If you look at Figure 4, there is a Button and a TextBlock element placed on a Window. You may not notice the layout area of the TextBlock but if you look at Figure 5, you will notice a blue color, and it is an actual layout area of the TextBlock.

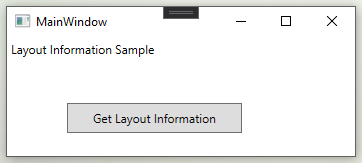


Figure 4

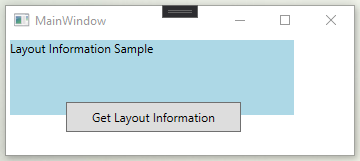


Figure 5

**The LayoutInformation class** is used to get information about the layout of a Window, UserControl or Page in WPF.

The LayoutInformation class has two static methods – GetLayoutClip and GetLayoutSlot. The **GetLayoutClip method** returns a Geometry that represents the visible region of an element. The **GetLayoutSlot method** returns the entire bounding rectangle of an element.

The code snippet in Listing 1 calls GetLayoutClip and GetLayoutSlot methods to get layout clip and layout slot of a TextBlock element.

Geometry clipGeometry = LayoutInformation.GetLayoutClip(textBlock1);

Rect layoutRectangle = LayoutInformation.GetLayoutSlot(textBlock1); textBlock1.Text = "Layout: " + layoutRectangle.ToString();

Listing 1

**WPF Layout – Size, Width, and Height**

This section focuses on the size of elements such as width and height etc.

Figure 6 shows a typical window with size, margin, alignment, and padding settings of various WPF elements.

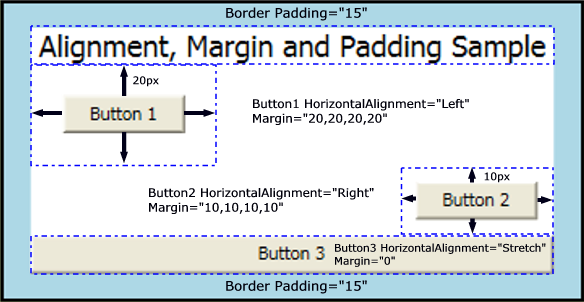


Figure 6

The size of a control is defined by the width and height properties of the control. When you create a Windows application in WPF, the default code looks like Listing 2. In Listing 2, a Window element is used to create a Window and a Grid panel is placed on this window. The Height and the Width of the window are set to 300 and 400 pixels respectively.

The type of Width and Height is a double device-independent unit (1/96th inch). This value can be followed by strings px, in, cm, or pt that is a pixel, inch, centimetre, or point respectively. Here is a listing of pixels and other measurements.

* 1 inch = 96 pixels
* 1 centimeter = 96/2.54 pixels
* 1 point = 96/72 pixels

The following code snippet sets the Width and Height of a grid to 300 and 200 pixels respectively.

<Grid x:Name="RootLayout"

Background="LightBlue"

Height="200px"

Width="300px" />

***Note****: If the Height and Width properties of an element are not set, it inherits the width and height of the container control.*

<Window x:Class="HelloWPFSample.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:HelloWPFSample"

mc:Ignorable="d"

Title="MainWindow" Height="300" Width="400">

<Grid Name="RootLayout">

</Grid>

</Window>

Listing 2

The output of Listing 2 looks like Figure 7.

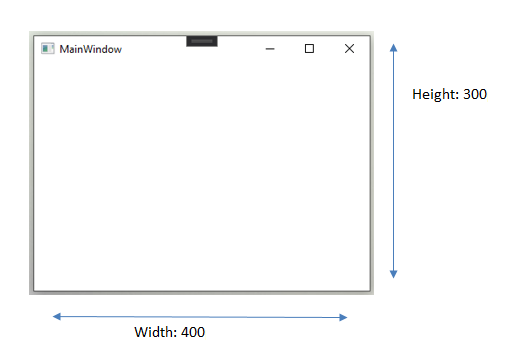


Figure 7

Now let’s set the width and height of Grid.

<Grid x:Name="RootLayout"

Background="LightBlue"

Height="200px"

Width="300px" />

New output with changed width and height of Grid looks like Figure 8.



Figure 8

* **MinHeight property** of an element represents the minimum height of an element. If you set Height property less than this value, it will be considered as MinHeight.

* **MaxHeight property** of an element represents the maximum height of an element. If you set Height property greater than this value, it will be considered as MaxHeight.
* **MinWidth property** of an element represents the minimum width of an element. If you set Width property less than this value, it will be considered as MinWidth.
* **MaxWidth property** of an element represents the maximum width of an element. If you set Width property greater than this value, it will be considered as MaxWidth.
* **ActualHeight property** of an element gets the actual rendered height of an element after calculating MinHeight, MaxHeight, and Height properties.
* **ActualWidth property** of an element gets the actual rendered width of an element after calculating MinWidth, MaxWidth, and Width properties.

*Note: MinWidth, MinHeight, MaxWidth, and MaxHeight properties are useful when you want to restrict your application’s size to a specific width and height.*

The code snippet in Listing 3 creates a Rectangle that has Width and Height properties set to 600 each but MaxHeight and MaxWidth are set to 200 each. That means the ActualHeight and ActualWidth of the rectangle that renders can’t be more than 200 each.

<Grid x:Name="GridMain">

<Rectangle x:Name="SizeRectangle"

Fill="LightBlue"

Height="600"

MaxHeight="200"

MaxWidth="200"

MinHeight="200"

MinWidth="200"

Width="600" />

</Grid>

Listing 3

The rectangle generated by Listing 3 looks like Figure 9 where actual height and actual width of the rectangle is 200 each, not 600.



Figure 9

**WPF Layout – Margins**

The margin is the space between an element and the parent element or other adjacent element on the same parent. Margin adds the extra space around the outside edges of an element.

The Margin property of FrameworkElement is used to specify the margin of an element. It is a type of Thickness structure. The Thickness structure has four properties – Left, Top, Right, and Bottom. You can either pass an only a double value or four double values. When margin is defined by a single value, it applies to all four sides of an element. If a margin is defined with values for all 4 sides, then specified 4 values are applied to the Left, Top, Right, and Bottom side of an element. If you don’t want to set a margin for one particular side, then you can define 0 in that place.

The code snippet in Listing 4 creates two rectangles and sets margin of the first rectangle by passing only one value 20 that sets margin for all four directions left, top, bottom, and right. The margin for the second rectangle is set 50,50,0,0.

<Rectangle x:Name="Rectangle1"

Fill="LightBlue"

Height="100"

HorizontalAlignment="Left"

Margin="20"

VerticalAlignment="Top"

Width="200" />

<Rectangle x:Name="Rectangle2"

Fill="LightGreen"

Height="100"

HorizontalAlignment="Left"

Margin="50,50,0,0"

VerticalAlignment="Top"

Width="200" />

Listing 4

The output of Listing 4 looks like Figure 10. In Figure 10, you see two rectangles. The blue rectangle has a margin set to 20,20,20,20. That shows that this extra space is added between the starting point (left and top corner) of the parent control and the rectangle.

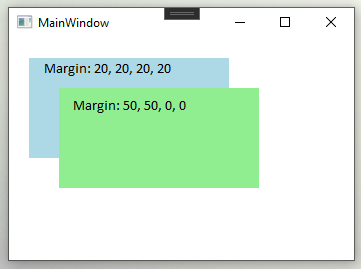


Figure 10

Similar to the Width and Height, the value of Margin can be set in px, in, cm and pt. The code snippet in Listing 5 sets Margin properties at runtime by creating a Thickness object.

Rectangle1.Margin = new Thickness(20);

Rectangle2.Margin = new Thickness(50, 50, 0, 0);

Listing 5

**WPF Layout - Padding**

The padding adds extra space from the inside part of an element, it is a space between the element and a border. It is represented by the Padding property of FrameworkElement and it is of type Thickness. The Padding property is not available to all elements in XAML and WPF. A few elements that have this property are Block, Border, Control, and TextBlock.

The code snippet in Listing 6 creates a Border and sets its Padding property in XAML.

<Border x:Name="BorderGrid"

Background="LightBlue"

BorderBrush="Black"

BorderThickness="5"

CornerRadius="20"

Padding="20" />

Listing 6

Listing 6 generates output looks like Figure 11.

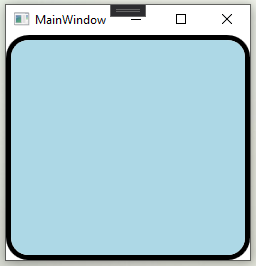


Figure 11

The code snippet in Listing 7 creates a Grid and adds a Border to the Grid and sets padding. The output looks exactly like Figure 11.

private void SetDynamicPadding()

{

Border brd = new Border();

brd.Background = new SolidColorBrush(Colors.LightBlue);

brd.BorderBrush = new SolidColorBrush(Colors.Black);

brd.BorderThickness = new Thickness(5);

brd.CornerRadius = new CornerRadius(20);

brd.Padding = new Thickness(20);

Grid1.Children.Add(brd);

}

Listing 7

**WPF Layout - Horizontal and Vertical Alignment**

The FrameworkElement has two alignment properties – HorizontalAlignment and VerticalAlignment.

**The HorizontalAlignment property** used to position child elements horizontally. HorizontalAlignment enumeration is used to set the values of HorizontalAlignment property. The HorizontalAlignment enumeration has four properties – Left, Center, Right, and Stretch. The Left, Center, and Right properties set a child element to the left, center, and right side of the parent element. The Stretch property stretches a child element to fill the parent element's allocated layout space. Which means both parent and child would have the same length.

*Note: If the Width property is set of an element, Stretch alignment does not have any effect.*

The code listed in Listing 8 places four Buttons elements on a StackPanel and set the HorizontalAlignment property to Left, Center, Right and Stretch.

<StackPanel x:Name="StackPanelOuter"

Background="LightGray">

<Button Name="ButtonFirst"

Background="LightBlue"

Content="Left Aligned"

Height="50"

HorizontalAlignment="Left"

Width="150" />

<Button x:Name="ButtonSecond"

Background="LightGreen"

Content="Center Aligned"

Height="50"

HorizontalAlignment="Center"

Width="150" />

<Button x:Name="ButtonThird"

Background="LightCyan"

Content="Right Aligned"

Height="50"

HorizontalAlignment="Right"

Width="150"/>

<Button Name="ButtonFourth"

Background="LightPink"

Content="Stretch Aligned"

Height="50"

HorizontalAlignment="Stretch"/>

</StackPanel>

Listing 8

Listing 8 generates output looks like Figure 12. Now no matter how much you resize or move the window, the buttons will stay aligned the same way.

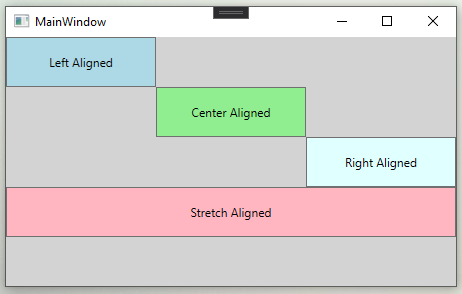


Figure 12

The code listed in Listing 9 creates a StackPanel and adds four Button controls through code-behind and sets their HorizontalAlignment property to Left, Center, Right, and Stretch dynamically. The output looks exactly like Figure 12.

private void DynamicallyHorizontalAlignment()

{

Button btn1 = new Button();

btn1.Width = 150;

btn1.Height = 50;

btn1.Background = new SolidColorBrush(Colors.LightBlue);

btn1.Content = "Left Aligned";

btn1.HorizontalAlignment = HorizontalAlignment.Left;

StackPanel1.Children.Add(btn1);

Button btn2 = new Button();

btn2.Width = 150;

btn2.Height = 50;

btn2.Background = new SolidColorBrush(Colors.LightGreen);

btn2.Content = "Center Aligned";

btn2.HorizontalAlignment = HorizontalAlignment.Center;

StackPanel1.Children.Add(btn2);

Button btn3 = new Button();

btn3.Width = 150;

btn3.Height = 50;

btn3.Background = new SolidColorBrush(Colors.LightCyan);

btn3.Content = "Right Aligned";

btn3.HorizontalAlignment = HorizontalAlignment.Right;

StackPanel1.Children.Add(btn3);

Button btn4 = new Button();

btn4.Height = 50;

btn4.Background = new SolidColorBrush(Colors.LightPink);

btn4.Content = "Stretch Aligned";

btn4.HorizontalAlignment = HorizontalAlignment.Stretch ;

StackPanel1.Children.Add(btn4);

}

Listing 9

**Visual Basic**

**Dim myButton1 As New Button()**

**myButton1.HorizontalAlignment = Windows.HorizontalAlignment.Left**

**myButton1.Margin = New Thickness(20)**

**myButton1.Content = "Button 1"**

**Dim myButton2 As New Button()**

**myButton2.HorizontalAlignment = Windows.HorizontalAlignment.Right**

**myButton2.Margin = New Thickness(10)**

**myButton2.Content = "Button 2"**

**Dim myButton3 As New Button()**

**myButton3.HorizontalAlignment = Windows.HorizontalAlignment.Center**

**myButton3.Margin = New Thickness(0)**

**myButton3.Content = "Button 3"**

**Dim myButton4 As New Button()**

**myButton4.HorizontalAlignment = Windows.HorizontalAlignment.Stretch**

**myButton4.Content = "Button 4 (Stretch)"**

**The VerticalAlignment property** is a type of VerticalAlignment enumeration and represents how vertically child elements are positioned within a parent element.

The VerticalAlignment enumeration has four properties – Top, Center, Bottom, and Stretch. The Top, Center, and Bottom properties set a child element to the top, center, and bottom side of the parent element. The Stretch property stretches a child element to fill the parent element's allocated layout space vertically.

*Note: If the Height property is a set, then Stretch alignment does not have any effect.*

The code listed in Listing 10 places four Button elements on a Grid and set the VerticalAlignment property to Top, Center, Bottom and Stretch.

<Grid x:Name="GridMain">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="100" />

<ColumnDefinition Width="100" />

<ColumnDefinition Width="100" />

<ColumnDefinition Width="100" />

</Grid.ColumnDefinitions>

<Button x:Name="ButtonFirst"

Background="LightBlue"

Content="Left Aligned"

Height="30"

VerticalAlignment="Top"

Width="100"/>

<Button x:Name="ButtonSecond"

Background="LightGreen"

Content="Center Aligned"

Height="30"

VerticalAlignment="Center"

Width="100"

Grid.Column="1" />

<Button x:Name="ButtonThird"

Background="LightCyan"

Content="Right Aligned"

Height="30"

HorizontalAlignment="Left"

VerticalAlignment="Bottom"

Width="100"

Grid.Column="2"/>

<Button x:Name="Button4"

Background="LightPink"

Content="Stretch Aligned"

HorizontalAlignment="Stretch"

Width="100"

Grid.Column="3" />

</Grid>

Listing 10

Listing 10 generates output looks like Figure 13. Now no matter how much you resize or move the window, the buttons will stay aligned the same way.

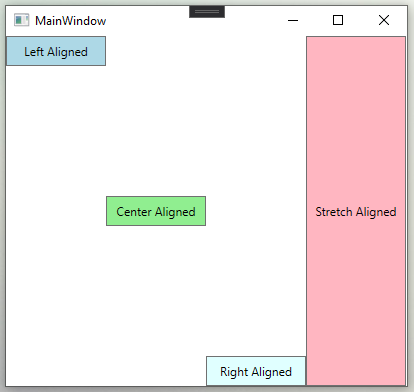


Figure 13

The code snippet in Listing 11 creates a Grid and adds four Button controls and sets their VerticalAlignment property to Top, Center, Bottom, and Stretch dynamically. The output looks exactly like Figure 13.

private void DynamicallyVerticalAlignment()

{

Button btn1 = new Button();

btn1.Width = 100;

btn1.Height = 30;

btn1.Background = new SolidColorBrush(Colors.LightBlue);

btn1.Content = "Top Aligned";

btn1.VerticalAlignment = VerticalAlignment.Top;

Grid1.Children.Add(btn1);

Button btn2 = new Button();

btn2.Width = 100;

btn2.Height = 30;

btn2.Background = new SolidColorBrush(Colors.LightGreen);

btn2.Content = "Center Aligned";

btn2.VerticalAlignment = VerticalAlignment.Center;

Grid1.Children.Add(btn2);

Button btn3 = new Button();

btn3.Width = 100;

btn3.Height = 30;

btn3.Background = new SolidColorBrush(Colors.LightCyan);

btn3.Content = "Bottom Aligned";

btn3.VerticalAlignment = VerticalAlignment.Bottom;

Grid1.Children.Add(btn3);

Button btn4 = new Button();

btn4.Width = 100;

btn4.Background = new SolidColorBrush(Colors.LightPink);

btn4.Content = "Stretch Aligned";

btn4.VerticalAlignment = VerticalAlignment.Stretch;

Grid1.Children.Add(btn4);

}

Listing 11

**WPF Layout - Content Alignments**

In Chapter 3, we learned what content is and how we can change the content of elements. The alignment of content can be set with two properties are HorizontalContentAlignment and VerticalContentAlignment. These properties are defined in the System.Windows.Controls.Control class that is parent class of all controls in WPF.

Similar to the HorizontalAlignment and VerticalAlignment, both HorizontalContentAlignment and VerticalContentAlignment properties are of type HorizontalAlignment and VerticalAlignment enumerations respectively.

Figure 14 shows a Windows which has a Button and a TextBox element. As you can see the default vertical and horizontal alignment of the content of a Button is center. The default vertical and horizontal alignment of content for a TextBox is left and top.

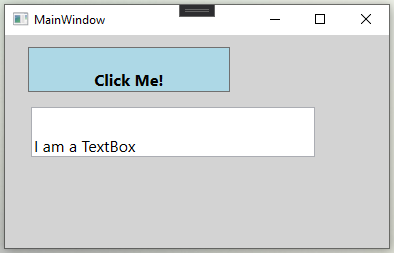


Figure 14

Now, what if you want to place contents of a Button and TextBox on different positions? Let’s say, you want to set the contents of the Button and TextBox to bottom and right. This comes handy when the size of elements is larger than the size of contents.

The code snippet in Listing 12 sets VerticalContentAlignment and HorizontalContentAlignment properties to bottom and right.

<Grid Name="GridRoot"

Background="LightGray" >

<Button x:Name="ButtonClickMe"

Background="LightBlue"

Content="Click Me!"

FontSize="16"

FontWeight="Bold"

Height="45"

HorizontalContentAlignment="Right"

Margin="23,12,159,0"

VerticalAlignment="Top"

VerticalContentAlignment="Bottom" />

<TextBox x:Name="TextBoxDetails"

FontSize="16"

Height="50"

HorizontalContentAlignment="Right"

Margin="26,72,74,0"

Text="I am a TextBox"

VerticalContentAlignment="Bottom"

VerticalAlignment="Top" />

</Grid>

Listing 12

Listing 12 generates an output which looks like Figure 15. As you notice, the contents of the Button and TextBox are aligned bottom and right.

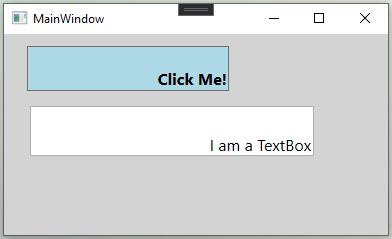


Figure 15

**WPF Layout - Dealing with Percentage Size in WPF**

In HTML, the percentage (%) symbol is used to define a uniform layout that keeps the same width and height ratio when a web page is resized. We did not have this feature in windows forms. However, WPF supports this feature differently by using an asterisk (\*) suffix with a double number. Unlike the percentage, an asterisk does not have a maximum limit of 100. An asterisk uses current width or height of an element and divides by the value associated with the asterisk and when a Window, UserControl or a Page is resized, the actual size of the element is calculated at runtime.

Let us explain to you the problem with an example. Figure 16 is a Window with a Grid panel and three Rectangle elements.

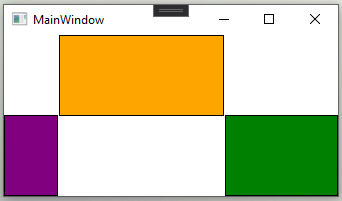


Figure 16

You can create this UI by simply placing 3 rectangles on a Grid element and moving them where you want. The XAML for the same is listed in the following listing.

<Grid Name="GridRoot">

<Rectangle x:Name="RectangleFirst"

Fill="Orange"

Margin="55,0,112,80"

Stroke="Black" />

<Rectangle x:Name="RectangleSecond"

Fill="Green"

HorizontalAlignment="Right"

Margin="0,80,0,0"

Stroke="Black"

Width="113" />

<Rectangle x:Name="RectangleThird"

Fill="Purple"

HorizontalAlignment="Left"

Margin="0,80,0,0"

Stroke="Black"

Width="54" />

</Grid>

Listing 13

The problem occurs when we try to resize this window, all these rectangles won’t have a size which is proportional to the current window size. Figure 15 shows what happens if we try to resize the same window.



Figure 17

So when we resize the window, we want the size of the rectangles to have the same ratio as the new size of the Window. We can achieve this using an asterisk with the size.

However, all elements in XAML do not support asterisk feature. So we have placed columns and rows in a Grid and fix their width and height with the asterisk.

The new code is listed in Listing 14.

<Window x:Class="HelloWPFSample.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:HelloWPFSample"

mc:Ignorable="d"

Title="MainWindow" Height="300" Width="400">

<Grid Name="GridRoot">

<Grid.RowDefinitions>

<RowDefinition Height="139\*" />

<RowDefinition Height="150\*" />

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="101\*" />

<ColumnDefinition Width="208\*" />

<ColumnDefinition Width="169\*" />

</Grid.ColumnDefinitions>

<Rectangle x:Name="RectangleFirst"

Fill="Orange"

Stroke="Black"

Grid.Column="1" />

<Rectangle x:Name="RectangleSecond"

Fill="Purple"

Stroke="Black"

Grid.Row="1" />

<Rectangle x:Name="RectangleThird"

Fill="Green"

Stroke="Black"

Grid.Column="2"

</Grid>

</Window>

Listing 14

Figure 18 shows how UI would look with Listing 14.

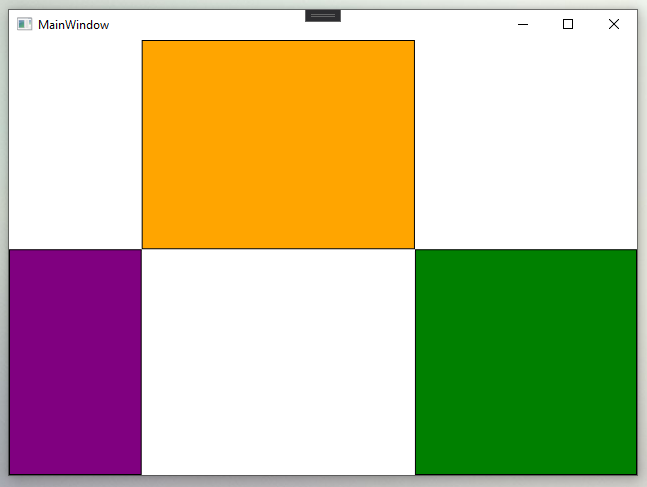


Figure 18

Now if you resize the window, the size of rectangles will be changed in proportional to the size of the window. You can notice the difference in Figure 19.



Figure 19

**WPF Layout - Panels**

All Panel controls are defined in the System.Windows.Controls namespace that resides in presentationframework.dll assembly. Besides the root Window, a Panel works as a parent control for other child controls. If you create a WPF application using Visual Studio 2019, the default code for the Window looks like Listing 15, where you can see a Grid panel is the default parent control that is placed inside a Window.

<Window x:Class="HelloWPFSample.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:HelloWPFSample"

mc:Ignorable="d"

Title="MainWindow" Height="300" Width="300">

<Grid>

</Grid>

</Window>

Listing 15

Listing 15 generates Figure 20.

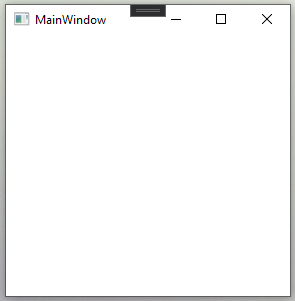


Figure 20

WPF comes with five built-in panels.

1. Canvas
2. DockPanel
3. Grid
4. StackPanel
5. WrapPanel

Each panel serves a different purpose. Each has a different way to position and reposition child elements. Similar to any other WPF control, Panel control is represented in two ways. First, at design-time using XAML elements and attributes and second, at run-time, using a WPF class and its properties.

The code snippet in Listing 16 creates a Grid panel at design-time using XAML.

<Window x:Class="HelloWPFSample.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:HelloWPFSample"

mc:Ignorable="d"

Title="MainWindow" Height="300" Width="300">

<Grid x:Name="GridPanel"

Background="Blue"

FlowDirection="LeftToRight"

Height="200"

HorizontalAlignment="Left"

VerticalAlignment="Top"

Width="250" />

</Window>

Listing 16

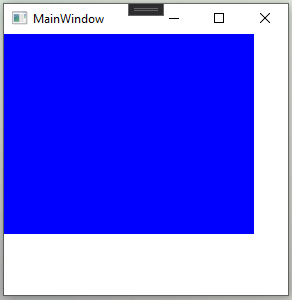
Output of Listing 16 showed in Figure 21.

Figure 21

The code snippet in Listing 17 creates the same Grid panel shown in Figure 21 at run-time using WPF classes.

private void CreateDynamicPanel()

{

// Create a Grid Panel control

Grid gridPanel = new Grid();

// Set Grid Panel properties

gridPanel.Background = new SolidColorBrush(Colors.Blue);

gridPanel.Width = 250;

gridPanel.Height = 200;

gridPanel.HorizontalAlignment = HorizontalAlignment.Left;

gridPanel.VerticalAlignment = VerticalAlignment.Top;

gridPanel.FlowDirection = FlowDirection.LeftToRight;

// Set Grid Panel as the content of the Window

RootWindow.Content = gridPanel;

}

Listing 17

All Panel controls are derived from the Panel class. Let’s take a look at the Panel class properties and methods.

**Panel Class Properties**

* **The Background** property represents the background colour of a panel which can be set with a Brush object.
* **The Children property** represents all child controls of a Panel.
* **The HasLogicalOrientation property** signifies whether a Panel arranges its descendants in a single dimension.
* **The InternalChildren property** represents all child controls of a Panel including items that are added directly in the code, and also items that are generated by data binding.
* **The IsItemHost property** signifies if this Panel is a container for UI items that are generated by an ItemsControl. An ItemsControl is a control that can be used to present a collection of items.
* **The LogicalChildrent property** returns an enumerator that iterate the logical child elements of this panel.
* **The LogicalOrientation property** signifies if a panel only supports layout in a single dimension.
* **The VisualChildrentCount property** represents several child Visual objects in this Panel.

**WPF Layout – Canvas**

We are familiar with a canvas used in a painting, it is used as a surface for oil painting. Artists can paint in any direction at any point, he can even overlap some of his art. The artist has complete control over the canvas. It is up him to decide where to put items on the canvas.

WPF canvas panel follows the same rules as real-life canvas, it gives complete control to the developer.  
It allows child elements to be overlapped or to be placed in any direction as per the developer's wish.

A Canvas panel is used to position child elements by using coordinates that are relative to the canvas area. Canvas has 4 attached properties that specify the position of the child element.  
Left, Right, Top, Bottom

Canvas.Left="10", Canvas.Top="10"

The Left property represents the distance between the left side of control and its parent container Canvas. The Top property represents the distance between the top of the control and its parent container Canvas.

Here, 10 specifies the margin of a specific coordinate, if Canvas.left is set to "10", then that element would be positioned on the left side by keeping a margin of 10.

Same goes for Right and Bottom.

Here are some of the properties of Canvas panels.

1. The default values of Height and Width properties are 0. If you do not set these values, you will not see a canvas unless child elements are automatically resizable.
2. Child elements on a Canvas can never be resized.
3. The vertical and horizontal alignments on child elements do not work. Child elements are placed on positions set by the Canvas Left, Top, Right, and Bottom properties.
4. The margin does work partially. If Left property of Canvas is set, Right property does not work. If Top property of Canvas is set, the Bottom property does not work.

Following is a syntax of a Canvas in XAML.

<Canvas/>

Let's see a basic example.  
We will draw some basic 2D shapes and see how to place them inside a canvas panel.  
Shapes in the following order:

* Ellipse
* Rectangle
* Rectangle with curved edges
* Path
* Path with heart shape

Order is very important in canvas because canvas overlaps last control over the previous ones, meaning the 2nd rectangle will be overlapped on the 1st ellipse in our case.  
The code snippet in Listing 18 covers all of the above requirements.

<Canvas x:Name="CanvasPanel"

Background="LightGray">

<Ellipse x:Name="TwoDEllipse"

Height = "100"

Margin="10 0 0 0"

Stroke="Black"

StrokeThickness="1"

Width = "100"

Canvas.Left="0"

Canvas.Top="18"

>

<Ellipse.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "AliceBlue"/>

<GradientStop Offset = "1" Color = "LightBlue"/>

<GradientStop Offset = "2" Color = "DarkBlue"/>

</RadialGradientBrush>

</Ellipse.Fill>

</Ellipse>

<Rectangle x:Name="TwoDRectangle"

Height="75"

Margin = "10 0 0 0"

Stroke="Black"

StrokeThickness="1"

Width="75"

Canvas.Left="50"

Canvas.Top="65">

<Rectangle.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "#f1ba82"/>

<GradientStop Offset = "1" Color = "Coral"/>

<GradientStop Offset = "2" Color = "Coral"/>

</RadialGradientBrush>

</Rectangle.Fill>

</Rectangle>

<Rectangle x:Name="TwoDRectangle2"

Height="75"

Margin = "10 0 0 0"

RadiusX="10"

RadiusY="10"

Stroke="Black"

StrokeThickness="1"

Width="75"

Canvas.Left="90"

Canvas.Top="100">

<Rectangle.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "Coral"/>

<GradientStop Offset = "1" Color = "#ff3f33"/>

<GradientStop Offset = "2" Color = "#ff5733"/>

</RadialGradientBrush>

</Rectangle.Fill>

</Rectangle>

<Path x:Name="TwoDPath"

Height = "80"

Margin = "10 0 0 0"

Stretch = "Fill"

Stroke="Black"

StrokeThickness="1"

Width="80"

Canvas.Left="124"

Canvas.Top="134">

<Path.Data>

<PathGeometry x:Name="PathGeoMetry">

<PathFigure StartPoint = "50,0" IsClosed = "True">

<LineSegment Point = "100,50"/>

<LineSegment Point = "50,100"/>

<LineSegment Point = "0,50"/>

</PathFigure>

</PathGeometry>

</Path.Data>

<Path.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "#e8e670"/>

<GradientStop Offset = "1" Color = "#eda619"/>

<GradientStop Offset = "2" Color = "#edea19"/>

</RadialGradientBrush>

</Path.Fill>

</Path>

<Path x:Name="PathHeart"

Data="M 241,200

A 20,20 0 0 0 200,240

C 210,250 240,270 240,270

C 240,270 260,260 280,240

A 20,20 0 0 0 239,200"

Stroke="Black"

StrokeThickness="1"

Canvas.Right="35"

Canvas.Bottom="30"

>

<Path.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "#e88270"/>

<GradientStop Offset = "1" Color = "#ee3514"/>

<GradientStop Offset = "2" Color = "#ee1414"/>

</RadialGradientBrush>

</Path.Fill>

</Path>

</Canvas>

Listing 18

Figure 22 shows the output of Listing 18.

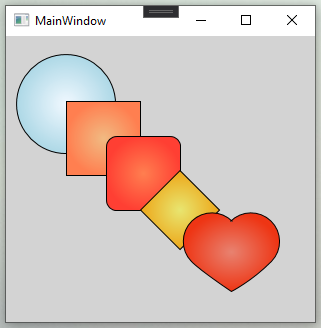


Figure 22

**Z-Index:** Here, we have limited shapes in our example. So we can manually adjust which control we want to display overlap by adding it at the end of the XAML.  
It gets messy when it has more than 20 or 30 shapes.  
To overcome this problem, the canvas panel has a property named Z-Index. Control with the higher z index overlaps the one with a lower z index.

*Note: Two controls can have the same z index, but then last defined control will overlap the previous one.*  
  
Let's make changes in our example by setting **Panel.ZIndex** for the Rectangle and Path to 2. Now, these 2 shapes will overlap our centered shape. To achieve that, set its **Panel.ZIndex = 1**.

So now, the order for ZIndex is **0 -> 2 -> 1 -> 2 -> 0.** why 0? Because by default, ZIndex is set to 0.

Listing 19 incorporates all these new changes.

<Canvas x:Name="CanvasPanel"

Background="LightGray">

<Ellipse x:Name="TwoDEllipse"

Height = "100"

Margin="10 0 0 0"

Stroke="Black"

StrokeThickness="1"

Width = "100"

Canvas.Left="0"

Canvas.Top="18"

>

<Ellipse.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "AliceBlue"/>

<GradientStop Offset = "1" Color = "LightBlue"/>

<GradientStop Offset = "2" Color = "DarkBlue"/>

</RadialGradientBrush>

</Ellipse.Fill>

</Ellipse>

<Rectangle x:Name="TwoDRectangle"

Height="75"

Margin = "10 0 0 0"

Stroke="Black"

StrokeThickness="1"

Width="75"

Canvas.Left="50"

Canvas.Top="65"

Panel.ZIndex="2">

<Rectangle.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "#f1ba82"/>

<GradientStop Offset = "1" Color = "Coral"/>

<GradientStop Offset = "2" Color = "Coral"/>

</RadialGradientBrush>

</Rectangle.Fill>

</Rectangle>

<Rectangle x:Name="TwoDRectangle2"

Height="75"

Margin = "10 0 0 0"

RadiusX="10"

RadiusY="10"

Stroke="Black"

StrokeThickness="1"

Width="75"

Canvas.Left="90"

Canvas.Top="100"

Panel.ZIndex="1">

<Rectangle.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "Coral"/>

<GradientStop Offset = "1" Color = "#ff3f33"/>

<GradientStop Offset = "2" Color = "#ff5733"/>

</RadialGradientBrush>

</Rectangle.Fill>

</Rectangle>

<Path x:Name="TwoDPath"

Height = "80"

Margin = "10 0 0 0"

Stretch = "Fill"

Stroke="Black"

StrokeThickness="1"

Width="80"

Canvas.Left="124"

Canvas.Top="134"

Panel.ZIndex="2">

<Path.Data>

<PathGeometry x:Name="PathGeoMetry">

<PathFigure StartPoint = "50,0" IsClosed = "True">

<LineSegment Point = "100,50"/>

<LineSegment Point = "50,100"/>

<LineSegment Point = "0,50"/>

</PathFigure>

</PathGeometry>

</Path.Data>

<Path.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "#e8e670"/>

<GradientStop Offset = "1" Color = "#eda619"/>

<GradientStop Offset = "2" Color = "#edea19"/>

</RadialGradientBrush>

</Path.Fill>

</Path>

<Path x:Name="PathHeart"

Data="M 241,200

A 20,20 0 0 0 200,240

C 210,250 240,270 240,270

C 240,270 260,260 280,240

A 20,20 0 0 0 239,200"

Stroke="Black"

StrokeThickness="1"

Canvas.Right="35"

Canvas.Bottom="30"

>

<Path.Fill>

<RadialGradientBrush>

<GradientStop Offset = "0" Color = "#e88270"/>

<GradientStop Offset = "1" Color = "#ee3514"/>

<GradientStop Offset = "2" Color = "#ee1414"/>

</RadialGradientBrush>

</Path.Fill>

</Path>

</Canvas>

Listing 19

Figure 23 is an output of Listing 19. Where you can see, how the second and the fourth shape are overlapping on others.

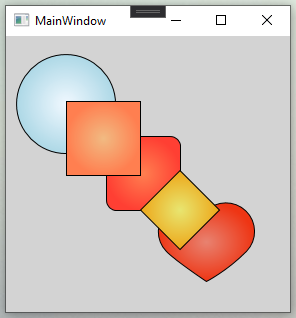


Figure 23

Canvas panel is the simplest amongst other panels in WPF. It is famous because of its flexibility and it is mostly used with shapes or animations.

**WPF Layout – DockPanel**

A Dock Panel is used for docking its child elements in the left, right, top, and the bottom position. The position of child elements is determined by the Dock property of the respective child elements and the relative order of those child elements. The default value of Dock property is left. The Dock property is a type of Dock enumeration that has Left, Right, Top, and Bottom values.

Here are some of the properties of DockPanels.

* **The LastChildFill property** specifies whether the last child element within a DockPanel stretches to fill the remaining available space. If it is set to true, then Dock property of the last element will be ignored.
* **Margin** is applicable on DockPanels.

The GetDock and SetDock methods are used to get and set dock value of an element.

The code snippet in Listing 22 creates a DockPanel with five Buttons. Four buttons are docked and the last one has no docking set that fills the entire remaining area.

<DockPanel Name="MainPanel">

<Button Name="ButtonTop"

Background="LightGreen"

Content="Top"

Height="50"

DockPanel.Dock="Top" />

<Button Name="ButtonLeft"

Background="LightBlue"

Content="Left"

Width="50"

DockPanel.Dock="Left" />

<Button Name="ButtonRight"

Background="LightSalmon"

Content="Right"

Width="50"

DockPanel.Dock="Right" />

<Button Name="ButtonBottom"

Background="LightCyan"

Content="Bottom"

Height="50"

DockPanel.Dock="Bottom" />

<Button Name="ButtonCentre"

Background="LightGray" />

</DockPanel>

Listing 20

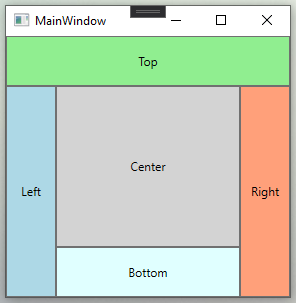


Figure 22

The DockPanel class in WPF represents a DockPanel control. The code listed in Listing 21 creates a Dock Panel dynamically, add five Button controls, and sets their docking properties by using SetDock method. The output of Listing 21 also generates the same output as in Figure 22.

private void CreateADockPanelDynamically()

{

// Create a DockPanel

DockPanel dcPanel = new DockPanel();

// Create a button

Button TopRect = new Button();

TopRect.Background = new SolidColorBrush(Colors.LightGreen);

TopRect.Height = 50;

TopRect.Content = "Top";

// Dock button to top

DockPanel.SetDock(TopRect, Dock.Top);

// Add docked button to DockPanel

dcPanel.Children.Add(TopRect);

// Create a button

Button LeftRect = new Button();

LeftRect.Background = new SolidColorBrush(Colors.LightBlue);

LeftRect.Width = 50;

LeftRect.Content = "Left";

// Dock button to left

DockPanel.SetDock(LeftRect, Dock.Left);

// Add docked button to DockPanel

dcPanel.Children.Add(LeftRect);

// Create a button

Button RightRect = new Button();

RightRect.Background = new SolidColorBrush(Colors.LightSalmon);

RightRect.Width = 50;

RightRect.Content = "Right";

// Dock button to left

DockPanel.SetDock(RightRect, Dock.Right);

// Add docked button to DockPanel

dcPanel.Children.Add(RightRect);

// Create a button

Button BottomRect = new Button();

BottomRect.Background = new SolidColorBrush(Colors.LightCyan);

BottomRect.Height = 50;

BottomRect.Content = "Bottom";

// Dock button to left

DockPanel.SetDock(BottomRect, Dock.Bottom);

// Add docked button to DockPanel

dcPanel.Children.Add(BottomRect);

// Create a fill button

Button FillRect = new Button();

FillRect.Background = new SolidColorBrush(Colors.LightGray);

FillRect.Content = "Center";

// Add docked button to DockPanel

dcPanel.Children.Add(FillRect);

RootWindow.Content = dcPanel;

}

Listing 21

**WPF Layout - Grid**

We all are surrounded by a grid-like structure. To give you an idea let me brief you a bit. A most popular example of a grid is the relational database where data is divided into rows and columns. Grids are so powerful that, they not only used for storing data but for displaying the data as well. Grid panel is WPF's one of the widely used panel.

It consists of columns and rows.  Each field is occupied by a single row-column combination. To give you an idea, this is what the grid looks like:

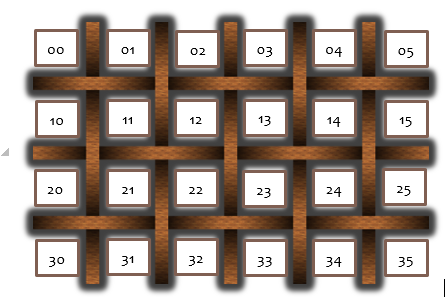


Figure 23

Grid Panel may be complicated but versatile panel among all panels. It can be used to design complicated UIs where we need to place multiple elements in a tabular format of rows and columns.

The following code snippet creates a Grid element and sets its background, width, height, vertical and horizontal alignment properties.

<Grid x:Name="GridPanel"

Background="Blue"

Height="220"

HorizontalAlignment="Center"

VerticalAlignment="Top"

Width="280"/>

Listing 22

The output of Listing 22 looks like Figure 24.



Figure 24

**Grid Properties**

The Grid has three major properties – RowDefinitions, ColumnDefinitions, and ShowGridLines. The **RowDefinitions property** is a collection of RowDefintion. **The ColumnDefinitions property** represents a collection of ColumnDefinition. **The ShowGridLines property** decides whether the grid lines of a Grid panel should be visible or not.

**Create Grid**

The following code snippet creates a Grid control, sets it width and foreground color and make sure grid lines are visible.

<Grid Name="MainGrid" Width="400" Background="LightSteelBlue" ShowGridLines="True" \>

Listing 23

The ColumnDefinitions property is used to add columns and the RowDefinitions property is used to add rows to a Grid. The following code snippet in Listing 24 adds three columns and three rows to a grid.

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="45" />

<RowDefinition Height="45" />

<RowDefinition Height="45" />

</Grid.RowDefinitions>

Listing 24

There are 2 attach properties of Grid. One is Grid.Row and another is Grid.Column. These properties are used by Grid’s child elements. Any child element can use Grid.Row and Grid.Column properties to specify their position inside a Grid. The values of rows and columns start with 0. That means, if there are three columns in a grid, the first column would be represented by number 0.

The following code snippet in Listing 25 puts a TextBlock control in the second row and third column.

<TextBlock Grid.Row="1" Grid.Column="2" Foreground="Green"

Text="Age" Height="20" VerticalAlignment="Top" />

Listing 25

Listing 26 merges code from Listing 23,24, and 25, with few more TextBlocks to cover all cells in the Grid.

<Grid x:Name="MainGrid"

Background="LightSteelBlue"

ShowGridLines="True"

Width="400">

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="45" />

<RowDefinition Height="45" />

<RowDefinition Height="45" />

</Grid.RowDefinitions>

<TextBlock x:Name="TextBlockHeaderOne"

FontSize="14"

FontWeight="Bold"

Foreground="Green"

Height="20"

Text="Model Name"

VerticalAlignment="Top"/>

<TextBlock x:Name="TextBlockHeaderSecond"

FontSize="14"

FontWeight="Bold"

Foreground="Green"

Height="20"

Text="OS"

VerticalAlignment="Top"

Grid.Column="1" />

<TextBlock x:Name="TextBlockHeaderThird"

FontSize="14"

FontWeight="Bold"

Foreground="Green"

Text="Company"

Height="20"

VerticalAlignment="Top"

Grid.Column="2" />

<TextBlock x:Name="TextBlockModelOne"

FontSize="12"

Text="iPhone 12"

Grid.Row="1"/>

<TextBlock x:Name="TextBlockOSOne"

FontSize="12"

Text="ios 14"

Grid.Column="1"

Grid.Row="1" />

<TextBlock x:Name="TextBlockCompanyOne"

FontSize="12"

Text="Apple"

Grid.Row="1"

Grid.Column="2"/>

<TextBlock x:Name="TextBlockModelTwo"

FontSize="12"

Text="Note 20"

Grid.Row="2" />

<TextBlock x:Name="TextBlockMOSTwo"

FontSize="12"

Text="Android"

Grid.Column="1"

Grid.Row="2" />

<TextBlock x:Name="TextBlockCompanyTwo"

FontSize="12"

Text="Samsung"

Grid.Column="2"

Grid.Row="2" />

</Grid>

Listing 26

The output of Listing 26 looks like this Figure 25.

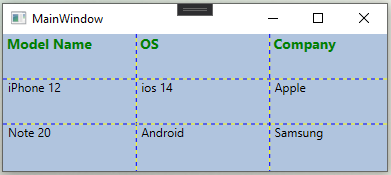


Figure 25

**Create a Grid Dynamically**

The Grid class in WPF represents a Grid control. The following code snippet in Listing 27 creates a Grid control, sets its width, horizontal alignment, vertical alignment, show grid lines, and background color.

Grid DynamicGrid = new Grid();

DynamicGrid.Width = 400;

DynamicGrid.HorizontalAlignment = HorizontalAlignment.Left;

DynamicGrid.VerticalAlignment = VerticalAlignment.Top;

DynamicGrid.ShowGridLines = true;

DynamicGrid.Background = new SolidColorBrush(Colors.LightSteelBlue);

Listing 27

The following code snippet in Listing 28 adds three columns and three rows to Grid.

// Create Columns

ColumnDefinition gridCol1 = new ColumnDefinition();

ColumnDefinition gridCol2 = new ColumnDefinition();

ColumnDefinition gridCol3 = new ColumnDefinition();

DynamicGrid.ColumnDefinitions.Add(gridCol1);

DynamicGrid.ColumnDefinitions.Add(gridCol2);

DynamicGrid.ColumnDefinitions.Add(gridCol3);

// Create Rows

RowDefinition gridRow1 = new RowDefinition();

gridRow1.Height = new GridLength(45);

RowDefinition gridRow2 = new RowDefinition();

gridRow2.Height = new GridLength(45);

RowDefinition gridRow3 = new RowDefinition();

gridRow3.Height = new GridLength(45);

DynamicGrid.RowDefinitions.Add(gridRow1);

DynamicGrid.RowDefinitions.Add(gridRow2);

DynamicGrid.RowDefinitions.Add(gridRow3);

Listing 28

Once rows and columns are added to the Grid, you can add any contents to Grid cells by using SetRow and SetColumn methods. SetRow and SetColumn methods take control name as the first parameter, and row number or column number as the second parameter. The following code snippet in Listing 29 creates a TextBlock control and displays it in Cell(0,0) that represents the first row and first column of Grid.

// Add first column header

TextBlock txtBlock1 = new TextBlock();

txtBlock1.Text = "Model Name";

txtBlock1.FontSize = 14;

txtBlock1.FontWeight = FontWeights.Bold;

txtBlock1.Foreground = new SolidColorBrush(Colors.Green);

txtBlock1.VerticalAlignment = VerticalAlignment.Top;

Grid.SetRow(txtBlock1, 0);

Grid.SetColumn(txtBlock1, 0);

Listing 29

Once control is created and its position within Grid is set, next step is to add control to Grid by using Grid.Children.Add method. This code snippet in Listing 30 adds a TextBlock to Grid.

DynamicGrid.Children.Add(txtBlock1);

Listing 30

The complete code is listed in Listing 31 which generates the same output as shown in Figure 25.

private void CreateDynamicWPFGrid()

{

// Create the Grid

Grid DynamicGrid = new Grid();

DynamicGrid.Width = 400;

DynamicGrid.HorizontalAlignment = HorizontalAlignment.Left;

DynamicGrid.VerticalAlignment = VerticalAlignment.Top;

DynamicGrid.ShowGridLines = true;

DynamicGrid.Background = new SolidColorBrush(Colors.LightSteelBlue);

// Create Columns

ColumnDefinition gridCol1 = new ColumnDefinition();

ColumnDefinition gridCol2 = new ColumnDefinition();

ColumnDefinition gridCol3 = new ColumnDefinition();

DynamicGrid.ColumnDefinitions.Add(gridCol1);

DynamicGrid.ColumnDefinitions.Add(gridCol2);

DynamicGrid.ColumnDefinitions.Add(gridCol3);

// Create Rows

RowDefinition gridRow1 = new RowDefinition();

gridRow1.Height = new GridLength(45);

RowDefinition gridRow2 = new RowDefinition();

gridRow2.Height = new GridLength(45);

RowDefinition gridRow3 = new RowDefinition();

gridRow3.Height = new GridLength(45);

DynamicGrid.RowDefinitions.Add(gridRow1);

DynamicGrid.RowDefinitions.Add(gridRow2);

DynamicGrid.RowDefinitions.Add(gridRow3);

// Add first column header

TextBlock txtBlock1 = new TextBlock();

txtBlock1.Text = "Model Name";

txtBlock1.FontSize = 14;

txtBlock1.FontWeight = FontWeights.Bold;

txtBlock1.Foreground = new SolidColorBrush(Colors.Green);

txtBlock1.VerticalAlignment = VerticalAlignment.Top;

Grid.SetRow(txtBlock1, 0);

Grid.SetColumn(txtBlock1, 0);

// Add second column header

TextBlock txtBlock2 = new TextBlock();

txtBlock2.Text = "OS";

txtBlock2.FontSize = 14;

txtBlock2.FontWeight = FontWeights.Bold;

txtBlock2.Foreground = new SolidColorBrush(Colors.Green);

txtBlock2.VerticalAlignment = VerticalAlignment.Top;

Grid.SetRow(txtBlock2, 0);

Grid.SetColumn(txtBlock2, 1);

// Add third column header

TextBlock txtBlock3 = new TextBlock();

txtBlock3.Text = "Company";

txtBlock3.FontSize = 14;

txtBlock3.FontWeight = FontWeights.Bold;

txtBlock3.Foreground = new SolidColorBrush(Colors.Green);

txtBlock3.VerticalAlignment = VerticalAlignment.Top;

Grid.SetRow(txtBlock3, 0);

Grid.SetColumn(txtBlock3, 2);

//// Add column headers to the Grid

DynamicGrid.Children.Add(txtBlock1);

DynamicGrid.Children.Add(txtBlock2);

DynamicGrid.Children.Add(txtBlock3);

// Create first Row

TextBlock authorText = new TextBlock();

authorText.Text = "iPhone 12";

authorText.FontSize = 12;

authorText.FontWeight = FontWeights.Bold;

Grid.SetRow(authorText, 1);

Grid.SetColumn(authorText, 0);

TextBlock ageText = new TextBlock();

ageText.Text = "ios 14";

ageText.FontSize = 12;

ageText.FontWeight = FontWeights.Bold;

Grid.SetRow(ageText, 1);

Grid.SetColumn(ageText, 1);

TextBlock bookText = new TextBlock();

bookText.Text = " Apple ";

bookText.FontSize = 12;

bookText.FontWeight = FontWeights.Bold;

Grid.SetRow(bookText, 1);

Grid.SetColumn(bookText, 2);

// Add first row to Grid

DynamicGrid.Children.Add(authorText);

DynamicGrid.Children.Add(ageText);

DynamicGrid.Children.Add(bookText);

// Create second row

authorText = new TextBlock();

authorText.Text = "Note 20";

authorText.FontSize = 12;

authorText.FontWeight = FontWeights.Bold;

Grid.SetRow(authorText, 2);

Grid.SetColumn(authorText, 0);

ageText = new TextBlock();

ageText.Text = "Android";

ageText.FontSize = 12;

ageText.FontWeight = FontWeights.Bold;

Grid.SetRow(ageText, 2);

Grid.SetColumn(ageText, 1);

bookText = new TextBlock();

bookText.Text = "Samsung";

bookText.FontSize = 12;

bookText.FontWeight = FontWeights.Bold;

Grid.SetRow(bookText, 2);

Grid.SetColumn(bookText, 2);

// Add second row to Grid

DynamicGrid.Children.Add(authorText);

DynamicGrid.Children.Add(ageText);

DynamicGrid.Children.Add(bookText);

// Display grid into a Window

RootWindow.Content = DynamicGrid;

}

Listing 31

**Managing Column Width and Row Height**

The ColumnDefinition has three properties that are used to manage the width of a column in a Grid. These properties are Width, MaxWidth, and MinWidth. The Width property represents the width of a column. The MaxWidth and MinWidth are used to set maximum and minimum width of a column.

The RowDefinition has three properties that are used to manage the height of a row in a Grid. These properties are Height, MaxHeight, and MinHeight. The Height property represents the height of a row. The MaxHeight and MinHeight are used to set maximum and minimum height of a row.

The code snippet in Listing 32 sets the width of columns and height of rows in a Grid panel at design-time using XAML.

<Grid x:Name="MainGrid"

Background="LightSteelBlue"

ShowGridLines="False"

Width="400">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="150"/>

<ColumnDefinition Width="120" />

<ColumnDefinition Width="200"/>

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="25" />

<RowDefinition Height="22" />

<RowDefinition Height="22" />

<RowDefinition Height="22" />

<RowDefinition Height="22" />

<RowDefinition Height="22" />

</Grid.RowDefinitions>

<TextBlock x:Name="TextBlockHeaderOne"

FontSize="14"

FontWeight="Bold"

Foreground="Green"

Height="20"

Text="Model Name"

VerticalAlignment="Top"/>

<TextBlock x:Name="TextBlockHeaderSecond"

FontSize="14"

FontWeight="Bold"

Foreground="Green"

Height="20"

Text="OS"

VerticalAlignment="Top"

Grid.Column="1" />

<TextBlock x:Name="TextBlockHeaderThird"

FontSize="14"

FontWeight="Bold"

Foreground="Green"

Text="Company"

Height="20"

VerticalAlignment="Top"

Grid.Column="2" />

<TextBlock x:Name="TextBlockModelOne"

FontSize="12"

Text="iPhone 12"

Grid.Row="1"/>

<TextBlock x:Name="TextBlockOSOne"

FontSize="12"

Text="ios 14"

Grid.Column="1"

Grid.Row="1" />

<TextBlock x:Name="TextBlockCompanyOne"

FontSize="12"

Text="Apple"

Grid.Row="1"

Grid.Column="2"/>

<TextBlock x:Name="TextBlockModelTwo"

FontSize="12"

Text="Note 20"

Grid.Row="2" />

<TextBlock x:Name="TextBlockMOSTwo"

FontSize="12"

Text="Android"

Grid.Column="1"

Grid.Row="2" />

<TextBlock x:Name="TextBlockCompanyTwo"

FontSize="12"

Text="Samsung"

Grid.Column="2"

Grid.Row="2" />

<TextBlock x:Name="TextBlockModelThird"

Text="BlackBerry Bold 9000"

Grid.Row="3" />

<TextBlock x:Name="TextBlockOSThird"

Text="BlackBerry OS 6"

Grid.Column="1"

Grid.Row="3" />

<TextBlock x:Name="TextBlockCompanyThird"

Text="BlackBerry"

Grid.Column="2"

Grid.Row="3" />

<TextBlock x:Name="TextBlockModelFourth"

Text="Nokia Lumia 630"

Grid.Row="4"/>

<TextBlock x:Name="TextBlockOSFourth"

Text="Windows 8.1"

Grid.Column="1"

Grid.Row="4" />

<TextBlock x:Name="TextBlockCompanyFourth"

Text="Nokia"

Grid.Column="2"

Grid.Row="4" />

<TextBlock x:Name="TextBlockModelFifth"

Text="Sony Ericsson Walkman"

Grid.Row="5" />

<TextBlock x:Name="TextBlockOSFifth"

Text="Symbian"

Grid.Column="1"

Grid.Row="5" />

<TextBlock x:Name="TextBlockCompanyFifth"

Text="Sony Ericsson"

Grid.Column="2"

Grid.Row="5" />

</Grid>

Listing 31

The output of Listing looks like Figure 26.

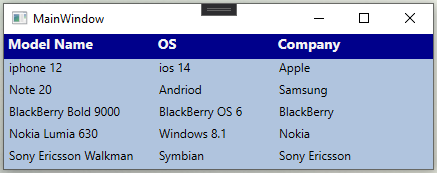


Figure 26

**Add and Remove Columns**

* **The Add method** of Grid.ColumnDefinitions adds a new column to Grid.

DynamicGrid.ColumnDefinitions.Add(new ColumnDefinition());

* **The Grid.ColumnDefinitions.Insert method** adds a column at a given position. The following code adds a new column at position 3 in a Grid.

DynamicGrid.ColumnDefinitions.Insert(3, new ColumnDefinition());

* **The RemoveAt method** of Grid.ColumnDefinitions delete a column from the given position.

DynamicGrid.ColumnDefinitions.RemoveAt(3);

* **The Clear method** of Grid.ColumnDefinitions delete all columns in a Grid.

DynamicGrid.ColumnDefinitions.Clear();

**Add and Remove Rows**

* **The Add method** of Grid.RowDefinitions adds a new row to the Grid.

DynamicGrid.RowDefinitions.Add(new RowDefinition());

* **The Grid.RowDefinitions.Insert** method adds a row at a given position.

DynamicGrid.RowDefinitions.Insert(3, new RowDefinition ());

* **The RemoveAt method** of Grid.RowDefinitions deletes a row from the given position.

DynamicGrid.RowDefinitions.RemoveAt(3);

* **The Clear method** of Grid.RowDefinitions deletes all rows in a Grid.

DynamicGrid.RowDefinitions.Clear();

**Resize Grid Rows with a GridSplitter**

The following code snippet in Listing 32 adds a Grid splitter to a Grid that you can use to resize a Grid row.

<Grid x:Name="MainGrid"

Background="LightSteelBlue"

Canvas.Top="119"

Height="200"

ShowGridLines="True"

Width="466"

Canvas.Left="8" >

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="50\*" />

<RowDefinition Height="Auto" />

<RowDefinition Height="50\*" />

</Grid.RowDefinitions>

<GridSplitter x:Name="GridSplitterCentre"

Background="Green"

Height="3"

HorizontalAlignment="Stretch"

ResizeDirection="Rows"

VerticalAlignment="Stretch"

Width="Auto"

Grid.ColumnSpan="10"

Grid.Row="1" />

</Grid>

Listing 32

**Formatting Grid**

The Background property of Grid sets the background colors of a Grid. The following code snippet uses linear gradient brushes to draw the background of a Grid.

<Grid.Background>

<LinearGradientBrush StartPoint="0,0" EndPoint="1,1" >

<GradientStop Color="Blue" Offset="0.1" />

<GradientStop Color="Orange" Offset="0.25" />

<GradientStop Color="Green" Offset="0.75" />

<GradientStop Color="Red" Offset="1.0" />

</LinearGradientBrush>

</Grid.Background>

Listing 33

The output of Listing 33 looks like Figure 27.



Figure 27

**Setting Image as Background of a Grid**

To set an image as the background of a Grid, The following code snippet sets the background of a Grid to an image. The code also sets the opacity of the image.

<Grid.Background>

<ImageBrush ImageSource="GridImage.jpg" Opacity="0.6"/>  
</Grid.Background>

Listing 34

The new output looks like Figure 28.

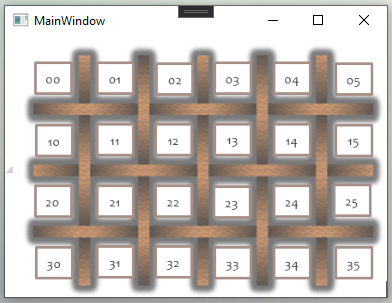
****

Figure 28

Let's see one last example by designing a user registration form.

Our final screen would look like Figure 29

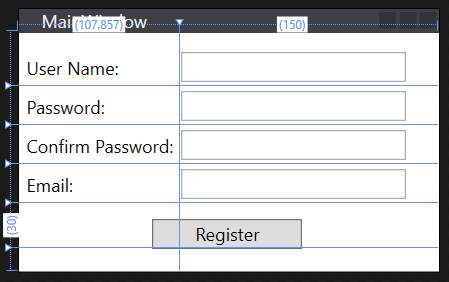


Figure 29

This screen is consisting of 4 labels, 4 Textboxes and 1 button. This Grid is consisting of 5 Rows and 2 Columns which are defined inside a RowDefinations and ColumnDefinations.

The height of rows can be defined with Height property, where the width of the columns is defined with Width property in the following three ways:

1. Fixed value: To assign a fixed size to element in pixels.
2. Auto: It will take up the default space of control. If the length of the text is 10 then 10, If it is 100 then 100. Auto calculates length or height of control and assigns available space.
3. Star (\*): It will take the remaining space once Auto or fixed-sized are filled.

The code snippet in Listing 35 fulfils all above requirements.

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Label x:Name="LabelUserName"

Content="User Name:"

Margin="0 10 0 0"/>

<Label x:Name="LabelPassword"

Content="Password:"

Grid.Row="1"/>

<Label x:Name="LabelConfirmPassword"

Content="Confirm Password:"

Grid.Row="2"/>

<Label x:Name="LabelEmailId"

Content="Email:"

Grid.Row="3"/>

<TextBox x:Name="TextBoxUserName"

Text="{Binding UserName}"

Height="20"

Margin="0 10 0 0"

Width="150"

Grid.Column="1"/>

<PasswordBox x:Name="TextBoxPassword"

Height="20"

Width="150"

Grid.Column="1"

Grid.Row="1"/>

<PasswordBox x:Name="TextBoxConfirmPassword"

Height="20"

Width="150"

Grid.Column="1"

Grid.Row="2"/>

<TextBox x:Name="TextBoxEmail"

Height="20"

Width="150"

Grid.Column="1"

Grid.Row="3"/>

<Button x:Name="ButtonLogin"

Content="Register"

Height="20"

HorizontalAlignment="Center"

Margin="20 10 0 0"

Width="100"

Grid.ColumnSpan="2"

Grid.Row="4"/>

</Grid>

Listing 35

Figure 30 is a result of Listing 35.

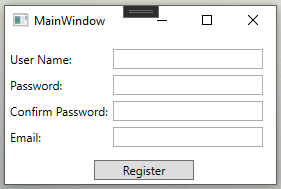


Figure 30

**Placement is always -1**

In the following code, for PasswordBox, we have set Grid.Row property to 1 but it is technically on the 2nd row, this is possible because the Index of column and rows starts with 0 and also notice that we did not even set Grid.Row property for our UserName label, this is because, by default, a Grid panel is created with one row and one column so Grid.Row = 0 and Grid.Column = 0 are set by default.

* Have a look at the way we have set Column properties in the above example for these TextBoxes. No Grid.Row for the first TextBox.

<TextBox x:Name="TextBoxUserName"

Text="{Binding UserName}"

Height="20"

Width="150"

Margin="0 10 0 0"

Grid.Column="1"/>

<PasswordBox x:Name="TextBoxPassword"

Height="20"

Width="150"

Grid.Column="1"

Grid.Row="1"/>

Listing 36

* What if you want to merge 2 columns? We can use **ColumnSpan property**, then specify the number of columns you wish to merge. So it merges 2 columns from its position. Login Button’s column position is 0 so it merges 0 and 1 together. For row, there is **RowSpan property**.

Have a look at the code snippet from Listing 37, which was part of our example. How we have used ColumnSpan property on Button.

<Button x:Name="ButtonLogin"

Height="20"

Width="100"

HorizontalAlignment="Center"

Command="{Binding RegisterButtonClicked}"

Grid.Row="4"

Grid.ColumnSpan="2"/>

Listing 37

**WPF Layout – StackPanel**

Stack panel is one of the simplest panels to use. It stacks its child elements in a single line, either horizontally or vertically. In real life, we have seen a stack of books, arranged vertically one below another. Stack Panel follows the same concept.

The following code snippet creates a StackPanel at design-time using XAML.

<StackPanel x:Name="StackPanelRoot"

Background="LightBlue"

Height="200"

Width="300" />

Listing 38

The Orientation property specifies the direction of children that can be vertical or horizontal. The default orientation of the StackPanel is Vertical, meaning if you don't specify the orientation property, by default child elements will be stacked vertically. Here’s how you can set the orientation property to Horizontal.

<StackPanel x:Name="StackPanelRoot"

Background="LightBlue"

Height="200"

Orientation="Horizontal"

Width="300" />

Listing 39

***Note****: If child elements on a StackPanel do not fit in the StackPanel’s area, then they go outside of the visible area. If you wish to wrap the child elements then you can use WrapPanel instead.*

***Note****: Also by default StackPanel stretches its elements. We can use the HorizontalAlignment and the VerticalAlignment properties of child elements to get rid of this stretch behaviour.*

First, let’s see how Canvas control positions five ellipse elements on UI. The code snippet in Listing 40 shows the same.

<Canvas >

<Ellipse Width="100" Height="100" Fill="Red" />

<Ellipse Width="80" Height="80" Fill="Orange" />

<Ellipse Width="60" Height="60" Fill="Yellow" />

<Ellipse Width="40" Height="40" Fill="Green" />

<Ellipse Width="20" Height="20" Fill="Blue" />

</Canvas>

Listing 40

Figure 31 illustrates the output of Listing 40.

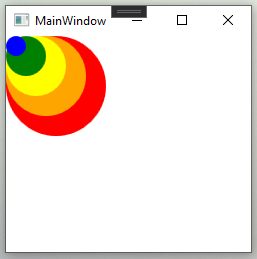


Figure 31

Now, in Listing 40, let’s replace Canvas with a StackPanel. The new code looks like Listing 41.

<StackPanel>

<Ellipse Width="100" Height="100" Fill="Red" />

<Ellipse Width="80" Height="80" Fill="Orange" />

<Ellipse Width="60" Height="60" Fill="Yellow" />

<Ellipse Width="40" Height="40" Fill="Green" />

<Ellipse Width="20" Height="20" Fill="Blue" />

</StackPanel>

Listing 41

The new output looks like Figure 32, where you can see all elements are stacked in a vertical direction.

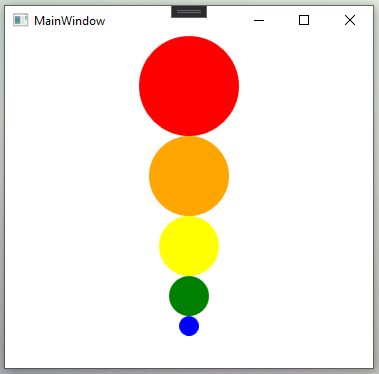


Figure 32

Now let’s change the Orientation property to horizontal by doing the following.

<StackPanel Orientation="Horizontal">

The new output would look like Figure 33.

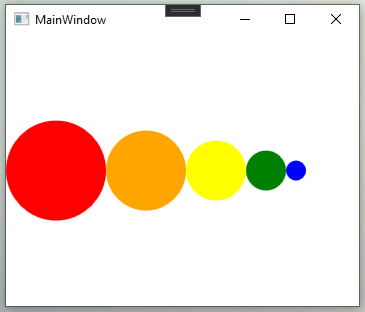


Figure 33

The code snippet in Listing 42 creates a StackPanel dynamically, then sets its properties and add five ellipses.

private void CreateDynamicStackPanel()

{

// Create a StackPanel and set its properties

StackPanel dynamicStackPanel = new StackPanel();

dynamicStackPanel.Background = new SolidColorBrush(Colors.LightBlue);

dynamicStackPanel.Orientation = Orientation.Horizontal;

// Create Ellipses and add to StackPanel

Ellipse redCircle = new Ellipse();

redCircle.Width = 100;

redCircle.Height = 100;

redCircle.Fill = new SolidColorBrush(Colors.Red);

dynamicStackPanel.Children.Add(redCircle);

Ellipse orangeCircle = new Ellipse();

orangeCircle.Width = 80;

orangeCircle.Height = 80;

orangeCircle.Fill = new SolidColorBrush(Colors.Orange);

dynamicStackPanel.Children.Add(orangeCircle);

Ellipse yellowCircle = new Ellipse();

yellowCircle.Width = 60;

yellowCircle.Height = 60;

yellowCircle.Fill = new SolidColorBrush(Colors.Yellow);

dynamicStackPanel.Children.Add(yellowCircle);

Ellipse greenCircle = new Ellipse();

greenCircle.Width = 40;

greenCircle.Height = 40;

greenCircle.Fill = new SolidColorBrush(Colors.Green);

dynamicStackPanel.Children.Add(greenCircle);

Ellipse blueCircle = new Ellipse();

blueCircle.Width = 20;

blueCircle.Height = 20;

blueCircle.Fill = new SolidColorBrush(Colors.Blue);

dynamicStackPanel.Children.Add(blueCircle);

// Display StackPanel into a Window

RootWindow.Content = dynamicStackPanel;

}

Listing 42

The output of Listing 42 generates output as Figure 34.

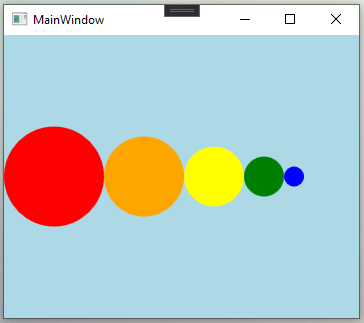


Figure 34

If there isn’t enough space on a StackPanel vertically or horizontally, you may add scrolling feature to it. The CanHorizontallyScroll and CanVerticallyScroll properties are used to add scrolling functionality to a StackPanel.

**WPF Layout – WrapPanel**

Wouldn't it be great to have a panel that arranges its child elements by itself? Well, WPF got you covered. The answer is WrapPanel: The meaning is in the name itself, it wraps the contents until there is no empty place left. It arranges elements automatically.

The WrapPanel arranges its child elements next to one other, either horizontally or vertically, unlike a stack panel by default WrapPanel's orientation is horizontal. While wrapping these child elements, one need to have fix height or width for every child elements.  
If the orientation of WrapPanel is Horizontal, then child elements will share the same height as the highest element. If the orientation of WrapPanel is Vertical, then child elements will share the same width as the widest element.

The following code snippet creates a WrapPanel and sets its height, width, and background properties at design-time using XAML.

<WrapPanel x:Name="PanelRoot"

Background="LightBlue"

Height="200"

Width="300" />

Listing 43

The ItemHeight and ItemWidth properties of WrapPanel are used to set a fixed uniform height and width for all items that are contained within a WrapPanel. The following code snippet sets Orientation, ItemHeight, and ItemWidth properties of a WrapPanel in XAML. Every item within a WrapPanel will have the height and width of 100 each.

<WrapPanel x:Name="PanelRoot"

Background="LightBlue"

Height="200"

ItemHeight="100"

ItemWidth="100"

Orientation="Vertical"

Width="300" />

Listing 44

The code snippet in Listing 45 creates a WrapPanel control and keeps its orientation to horizontal by default.

<WrapPanel>

<Ellipse Width="100" Height="100" Fill="Red" />

<Ellipse Width="90" Height="90" Fill="Orange" />

<Ellipse Width="80" Height="80" Fill="Yellow" />

<Ellipse Width="70" Height="70" Fill="LightGreen" />

<Ellipse Width="60" Height="60" Fill="Green" />

<Ellipse Width="50" Height="50" Fill="LightBlue" />

<Ellipse Width="40" Height="40" Fill="Blue" />

<Ellipse Width="30" Height="30" Fill="Black" />

</WrapPanel>

Listing 45

Figure 35 is an output of Listing 36 where all child controls are wrapped horizontally.



Figure 35

Now if you change orientation to vertical like the following code, the output would look like Figure 36.

<WrapPanel Orientation="Vertical">

As you can see Figure 36, the new controls are aligned vertically.



Figure 36

The code listed in Listing 37 creates a StackPanel dynamically, then sets its properties and add five ellipses.

private void CreateDynamicWrapPanel()

{

// Create a WrapPanel and set its properties

WrapPanel dynamicWrapPanel = new WrapPanel();

dynamicWrapPanel.Orientation = Orientation.Horizontal;

// Create Ellipses and add to StackPanel

Ellipse redCircle = new Ellipse();

redCircle.Width = 100;

redCircle.Height = 100;

redCircle.Fill = new SolidColorBrush(Colors.Red);

dynamicWrapPanel.Children.Add(redCircle);

Ellipse orangeCircle = new Ellipse();

orangeCircle.Width = 80;

orangeCircle.Height = 80;

orangeCircle.Fill = new SolidColorBrush(Colors.Orange);

dynamicWrapPanel.Children.Add(orangeCircle);

Ellipse yellowCircle = new Ellipse();

yellowCircle.Width = 60;

yellowCircle.Height = 60;

yellowCircle.Fill = new SolidColorBrush(Colors.Yellow);

dynamicWrapPanel.Children.Add(yellowCircle);

Ellipse greenCircle = new Ellipse();

greenCircle.Width = 40;

greenCircle.Height = 40;

greenCircle.Fill = new SolidColorBrush(Colors.Green);

dynamicWrapPanel.Children.Add(greenCircle);

Ellipse blueCircle = new Ellipse();

blueCircle.Width = 20;

blueCircle.Height = 20;

blueCircle.Fill = new SolidColorBrush(Colors.Blue);

dynamicWrapPanel.Children.Add(blueCircle);

// Display WrapPanel into a Window

RootWindow.Content = dynamicWrapPanel;

}

Listing 46

Let's see the one more example with default orientation: horizontal.  
The screen will have 3 labels and 3 Textboxes and 1 Register button.

<WrapPanel x:Name="WrapPanelRoot"

Grid.IsSharedSizeScope="True">

<Grid x:Name="GridInnerUserName">

<Grid.ColumnDefinitions>

<ColumnDefinition SharedSizeGroup="FirstColumn"/>

<ColumnDefinition SharedSizeGroup="SecondColumn"/>

</Grid.ColumnDefinitions>

<Label x:Name="LabelUserName"

Content="User Name:"

Margin="0 10 0 0"/>

<TextBox x:Name="TextBoxUserName"

Text="{Binding UserName}"

Height="20"

Width="150"

Grid.Column="1"/>

</Grid>

<Grid x:Name="GridInnerPassword">

<Grid.ColumnDefinitions>

<ColumnDefinition SharedSizeGroup="FirstColumn"/>

<ColumnDefinition SharedSizeGroup="SecondColumn"/>

</Grid.ColumnDefinitions>

<Label x:Name="LabelPassword"

Content="Password:" />

<PasswordBox x:Name="TextBoxPassword"

Height="20"

Width="150"

Grid.Column="1"/>

</Grid>

<Grid x:Name="GridInnerEmail">

<Grid.ColumnDefinitions>

<ColumnDefinition SharedSizeGroup="FirstColumn"/>

<ColumnDefinition SharedSizeGroup="SecondColumn"/>

</Grid.ColumnDefinitions>

<Label x:Name="LabelEmailId"

Content="Email:" />

<TextBox x:Name="TextBoxEmail"

Height="20"

Width="150"

Grid.Column="1"/>

</Grid>

<Button x:Name="ButtonLogin"

Content="Register"

Height="20"

HorizontalAlignment="Center"

Margin="20 10 0 0"

Width="100" />

</WrapPanel>

Listing 47

Now if you start dragging screen horizontally, You’ll notice that elements will start occupying empty space as we drag more.

# 

Dragging horizontally

Figure 37

# 

Figure 38

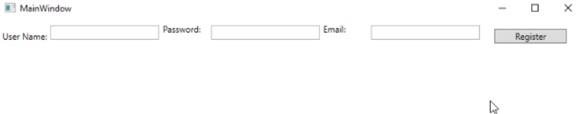


Figure 39



Dragging vertically upwards

Figure 40

To see its horizontal wrapping behaviour, change the wrap panel's orientation to vertical.

<WrapPanel x:Name="GridOuter" Orientation="Vertical" >

Alright, once you are done with making these changes, run the project and see the alternate magic:

# 

Figure 41

# 

Dragging vertically downwards

Figure 42

# 

Dragging vertically downwards

Figure 43

# 

Figure 44

In layman terms, items are horizontally arranged when the window is stretch horizontally and items are vertically arranged when the window is stretch vertically.

**WPF Layout – Border**

Elements in WPF do not have a border property. To place a border around an element, WPF provides the Border element. Similar to other WPF elements, the Border has Width, Height, Background, and HorizontalAlignment and VerticalAlignment properties.

Besides these common properties, Border has two properties that make a border the Border, which are BorderThickness and BorderBrush. **The BorderBrush property** represents the brush that is used to draw the border. **The BorderThickness property** represents the thickness of the border.

**The CornerRadius property** represents the degree to which the corners of a Border will be rounded. The following code snippet creates a Border element and sets its properties.

<Border x:Name="BorderGreen"

BorderThickness="5"

BorderBrush="Green"

Background="LightGray"

CornerRadius="10"

Height="250"

HorizontalAlignment="Left"

VerticalAlignment="Top"

Width="270" />

Listing 48

The code snippet in Listing 40 creates a Border around a Canvas element and sets its properties.

The output looks like Figure 1 where all child controls are wrapped horizontally.

<Border x:Name="BorderGreen"

BorderThickness="5"

BorderBrush="Green"

Background="LightGray"

CornerRadius="10"

Height="250"

HorizontalAlignment="Left"

VerticalAlignment="Top"

Width="270">

<Canvas Background="LightCyan" >

<Rectangle x:Name="RectangleInner"

Canvas.Top="20"

Fill="Red"

Height="200"

Stroke="Black"

StrokeThickness="10"

Width="200"

Canvas.Left="30"/>

</Canvas>

</Border>

Listing 49

The output looks like Figure 45 where you can see the green border with rounded corners.

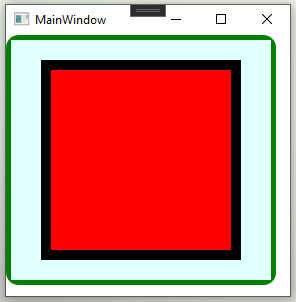


Figure 45

The Border class in WPF represents a Border element. The code snippet listed in Listing 41 creates a Border, sets its properties, and places it around a Canvas element.

private void CreateDynamicBorder()

{

Border border = new Border();

border.Background = new SolidColorBrush(Colors.LightGray);

border.BorderThickness = new Thickness(5);

border.BorderBrush = new SolidColorBrush(Colors.Green);

border.CornerRadius = new CornerRadius(15);

border.Width = 270;

border.Height = 250;

Canvas cnvas = new Canvas();

Rectangle rect = new Rectangle();

rect.Width = 200;

rect.Height = 200;

rect.Fill = new SolidColorBrush(Colors.Black );

rect.StrokeThickness = 10d;

cnvas.Children.Add(rect);

border.Child = cnvas;

RootLayout.Content = border;

}

Listing 50

***Note****. Border can have only one child element. If you need to place a border around multiple elements, you must place a border around each element.*

**WPF Layout – VirtualizingStackPanel**

Virtualization technique in WPF improves the rendering performance of UI elements. By applying virtualization, the layout system ensures that only the visible items of a container are rendered on the screen. For example, a ListBox element may have thousands of items but virtualization will reduce the rendering to the visible items only.

**VirtualizingStackPanel**

The VirtualizingStackPanel control in WPF is used to implement virtualization. **The IsVirtualizing property** of the VirtualizingStackPanel initiates the virtualization. By default, the IsVirtualizing property is set to true. When IsVirtualizing is set to false, a VirtualizingStackPanel behaves the same as an ordinary StackPanel.

The VirtualizingStackPanel calculates the number of visible items and works with the ItemContainerGenerator from an ItemsControl to create UI elements only for visible items. The following code snippet creates a VirtualizingStackPanel.

<VirtualizingStackPanel x:Name="VirtualizingStackPanelRoot"

Height="200"

Width="300" />

The VirtualizingStackPanel.VirtualizationMode property has two values, **Standard and Recycling**. The default value of VirtualizationMode is Standard that means that the VirtualizingStackPanel creates an item container for each visible item and discards it when it is no longer needed (such as when the item is scrolled out of view). When an ItemsControl contains a lot of items, the process of creating and discarding item containers can negatively affect performance. In that case, use the Recycling, which reuses item containers instead of creating a new one each time.

The following code snippet creates a VirtualizingStackPanel and sets its VirtualizationModel to Recycling.

<VirtualizingStackPanel x:Name="VirtualizingStackPanelRoot"

Height="200"

VirtualizationMode="Recycling"

Width="300" />

Virtualization can be applied to a container control by directly setting the VirtualizingStackPanel.VirtualizationMode property. The code snippet in Listing 42 creates a ListBox control and sets its VirtualizationMode to Recycling.

<StackPanel x:Name="PanelRoot">

<ListBox x:Name="ListBoxParent"

Background="#FFF5BDBD"

Height="193.333"

HorizontalAlignment="Left"

Margin="20,20,0,0"

VerticalAlignment="Top"

Width="400"

VirtualizingStackPanel.VirtualizationMode="Recycling"/>

<Button x:Name="ButtonLoad"

Content="Load Data"

Height="29"

HorizontalAlignment="Left"

Margin="20,18.666,0,0"

VerticalAlignment="Top"

Width="120"

Click="LoadDataButton\_Click"/>

</StackPanel>

Listing 51

The supporting code is listed in Listing 43 that creates a data collection and binds the ListBox.ItemsSource property to the data collection.

private void LoadDataButton\_Click(object sender, RoutedEventArgs e)

{

listBox.ItemsSource = GetDataSet();

}

public ArrayList GetDataSet()

{

ArrayList items = new ArrayList();

for (var count = 0; count < 10000; ++count)

{

items.Add(string.Format("Item {0}", count));

}

return items;

}

}

Listing 52

**Summary**

At the beginning of this chapter, we covered a basic understanding of the WPF layout system. Later we discussed the layout clip and the layout slot. We covered the size, margins, paddings, alignments of elements in the WPF layout system. of WPF elements. Later, we switched our focus on panels. We discussed the basics of the panels in WPF. We covered how to use different panels such as canvas, dock, grid, stack and wrap panel. We also discuss their purpose and behaviour. Finally, we covered Border and VirtualizingStackPanel.