UWP-008 - XAML Layout with Grids

In this lesson, I want to begin talking about layout, or rather, the process of positioning visual controls and other elements on your application's user interface. There are several different XAML controls that exist for the purpose of layout, and cover most of the popular ones in this series of lessons.

In the past, layout was relatively simple. After all, you were typically only laying out an application for a single form factor -- a single device like a phone or a desktop application. However, there are a few new wrinkles introduced as we begin to build applications that can adaptively resize based on the device that we run our app on. And this is one of the new key features in app development on the Windows platform.

We'll start with simple concepts then build up to the more challenging concepts in the lessons that follow.

Before we begin in earnest, I want to point out one thing regarding all XAML controls that are intended for the purpose of layout. Most controls have a Content property, so your Button control has a Content property, for example. And the Content property can only be set to an instance of another object. So in other words, I can set the Content property of a Button to a TextBlock:

```
9
10      <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11          <Button>
12              <TextBlock>Hi</TextBlock>
13              <Image Source="Assets/Square44x44Logo.scale-200.png" />
14          </Button>
15      </Grid>
16  </Page>
```

… but then I also have added an Image inside of that Button control's default property as well. Whenever I attempt to put more than one control inside of the Content property, I get the error: The property "Content" can only be set once.

However, layout controls are intended to host more than one control. And so as a result, they do not have a Content property. Instead, they usually have a Children property that is of a special data type, a collection data type that can hold XAML controls called UIElementCollection.

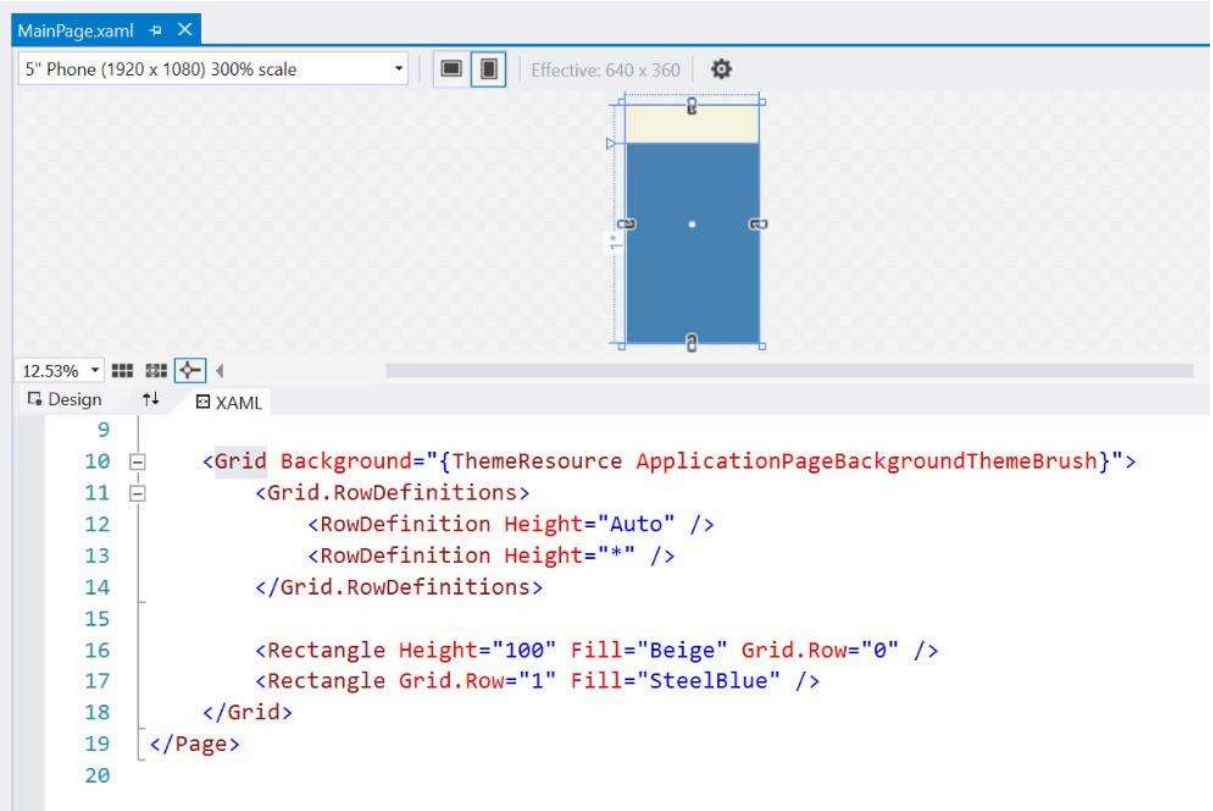
In XAML, as we add new instances of controls inside of the definition of our layout control, we're actually calling the Add method of our layout control's UIElementCollection, or rather, just the collection property. So here again, XAML hides a lot of the complexity for us and makes our code very concise by inferring our intent by how we write our XAML.

So we will begin learning about layout in this lesson by looking at the Grid control. Like any grid, it allows you to define both rows and columns to create cells. And then each of the controls that are used by your application can request which row and which column that they want to be placed inside of. So whenever you create a new app using the blank app template, you're provided very little guidance. You get a single empty Grid with no rows or no columns defined.

```
 9
10      <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11
12      </Grid>
13  </Page>
```

However, by default, there is always one row definition and one column definition, even if it is not explicitly defined in your XAML. These take up the full vertical and horizontal space available to represent one large cell in the grid. Any items that are placed between that opening and closing Grid element are understood to be inside of that single, implicitly defined cell.

I created a quick example of a grid that defines two rows, just to illustrate two primary ways of creating rows and setting their heights. See the associated project called: RowDefinitions.



So here you can see that I want you to notice that I have two rectangles. There is an upper rectangle and a lower rectangle. And those are defined through a series of RowDefinition objects here. So you can see that I have inside of the grid this property element syntax to define a collection of RowDefinitions with instances of RowDefinition created with their Height property set.

The first RowDefinition has its Height property set to Auto and its second RowDefinition object has its Height set to star (*).

And then there are two Rectangle objects. Notice how I'm setting the Row property that this Rectangle object wants to put itself inside of. It wants to put itself inside of the row zero, the first row (since when referencing Rows and Columns you apply a zero-based counting.)

The second Rectangle wants to put itself inside of the Grid.Row="1".

Notice is how the Rectangles are putting themselves into the various Grid Rows, and then also how you reference both Rows and Columns using a zero-based numbering scheme.

Next, observe the weird syntax: Grid.Row and Grid.Column. And these are called "Attached Properties". Attached Properties enable an object (in this case, a rectangle) to assign a value for a property (in this case, the row property, but it could apply to the column property as well), to assign a value for a property that its own class does not define. So, nowhere in the Rectangle class definition is there a Grid.Row property, or even a Row property. These are all defined inside of the Grid object.

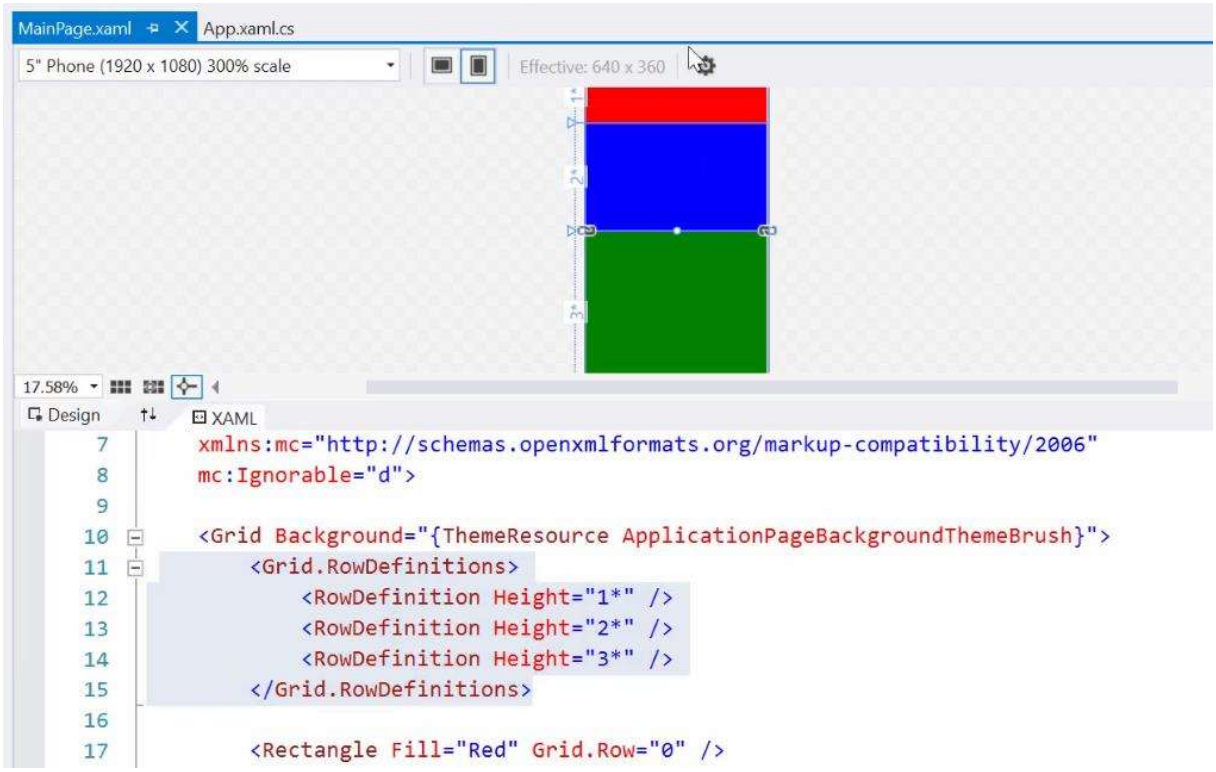
The reason why Attached Properties exist is an advanced XAML topic that is not actionable at this point in your introduction to XAML. If you want to get deeper into the internals of XAML, then you should search MSDN for articles about both "Attached Properties", as well as the loosely related topic of "Dependency Properties". In a nutshell, Attached Properties keep your XAML simple.

Third, notice in this example that there are the two different row heights. The first height, we set to Auto and the second row height we set to star (*). There are three syntaxes that you can use to help persuade the sizing for each row and each column. I use the term "persuade" intentionally. With XAML layout, heights and widths are relative and can be influenced by a number of different factors. All these factors are considered by the layout engine at run time to determine the actual placement of items on your given page, or your screen.

So for example, the term "Auto" means that the height for the row should be tall enough to accommodate all of the controls that are placed inside of that row. If the tallest control (in this case, you can see the Rectangle has its height explicitly set to 100 pixels) is 100 pixels tall, then that's the actual height of the row, 100 pixels. If we were to change the height of the Rectangle to 50 pixels, you can see that the height of the Row changes now to be 50 as well. "Auto" means that the height is relative to the controls that are inside of that given row or column, or whatever the case might be.

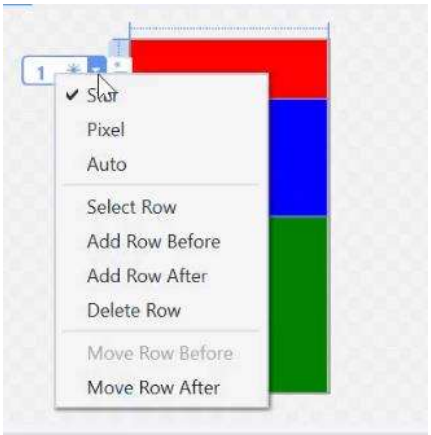
Secondly, the asterisk character is also known as "star sizing", and it means that the height of the row should take up all of the rest of the available height available.

I created a separate project called StarSizing that has three Rows defined in the Grid.



Notice the heights of each one of them. By adding a number before the asterisk I am saying of all the available space, give me one "share" of all the available space, or two or three "shares" of all the available space. The sum of all of those rows adds up to six. So each "one star" is equivalent to 1/6th of the height that is currently available. Therefore, three star would get half of the height that is available as depicted in the output of this example.

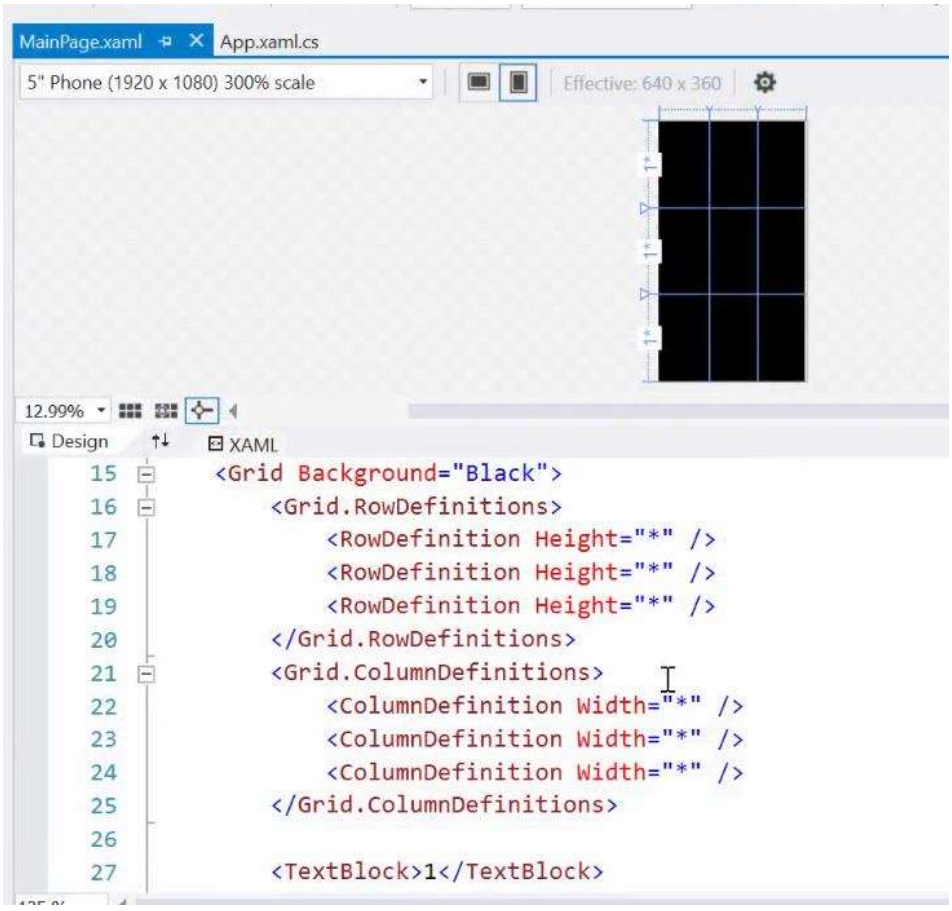
Also notice off to the left-hand side of the visual designer's depiction of the Grid that there are some visual tools that we can use to change the sizing. For example, I can change from star sizing to auto, or to pixel. I can also just type in the given value here, and that would set the Height property.



Besides auto and star sizing, you can also specify widths and heights, as well as margins in terms of pixels. So in fact, when only numbers are present, it represents that number of pixels for the width or the height. Generally, it is **not** a good idea to use exact pixels in layouts for widths and heights because of the likelihood that various screens will be larger or smaller, so there are several different types of phones or several different form factors for tablets and desktops. You do not want to specify exact numbers or else it is not going to look correct on a different form factor. Instead, it is preferable to use relative layouts like auto and star sizing for layout.

Notice that the Widths and Heights, are assumed to be 100% unless otherwise specified, especially for rectangles. And while that's generally true for many XAML controls, other controls like the Button are not treated this way. Their size is based on the content inside of them.

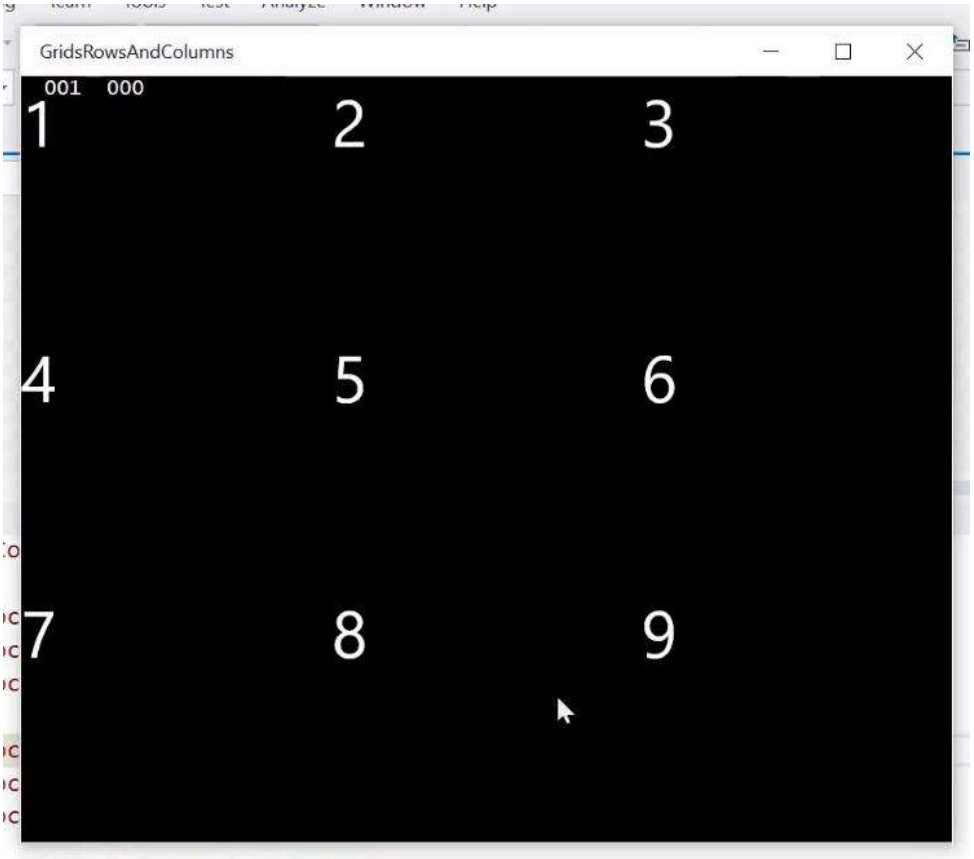
A Grid can have a collection of ColumnDefinitions. In another example named “GridsRowsAndColumns” you can see that I created a 3 x 3 grid, three RowDefinitions and a ColumnDefinitions collection that contains three ColumnDefinitions.



Furthermore, I added a TextBlock inside of each of the cells.

```
26
27          <TextBlock>1</TextBlock>
28          <TextBlock Grid.Column="1">2</TextBlock>
29          <TextBlock Grid.Column="2">3</TextBlock>
30
31          <TextBlock Grid.Row="1">4</TextBlock>
32          <TextBlock Grid.Row="1" Grid.Column="1">5</TextBlock>
33          <TextBlock Grid.Row="1" Grid.Column="2">6</TextBlock>
34
35          <TextBlock Grid.Row="2">7</TextBlock>
36          <TextBlock Grid.Row="2" Grid.Column="1">8</TextBlock>
37          <TextBlock Grid.Row="2" Grid.Column="2">9</TextBlock>
```

Now unfortunately, you the designer is not displaying them correctly, but if we were to run the application, you would be able to see that we get a different number in each cell.

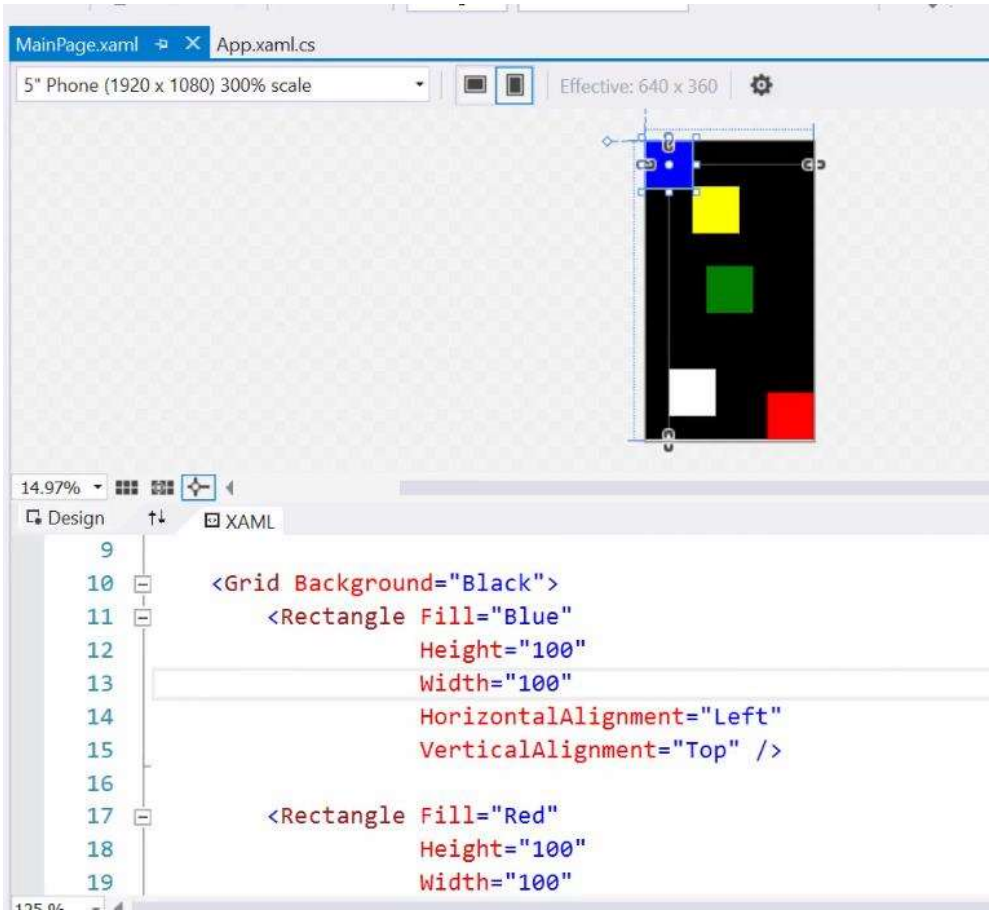


In this very first cell in the upper left-hand corner I am not setting a row, nor am I setting a column. By default, if you do not supply that information, it is assumed to be zero. So we're assuming that we're putting a TextBlock containing “1” in row zero, column zero.

Furthermore, if you take a look at the next TextBlock containing “2”, I am setting the grid column equal to one, but I'm not setting the row, meaning that I am assuming that to be zero.

Relying on defaults keeps your code, again, more concise, but you have to understand that there is a convention being used.

I have yet another example called AlignmentAndMargins.



This example illustrates how VerticalAlignment and HorizontalAlignment work even in a given grid cell. And this will hold true in a StackPanel as well as we talk about it in the next lesson.

The VerticalAlignment or HorizontalAlignment property attributes pull controls towards their boundaries. By contrast, the margin attributes push controls away from their boundaries.

In this example, you can see that the HorizontalAlignment is pulling this blue rectangle towards the left-hand side, and the VerticalAlignment is pulling it towards the top side.

Next, look at the White Rectangle. The HorizontalAlignment is pulling it towards the left, and the VerticalAlignment is pulling it towards the bottom. But then I'm setting the Margins equal to 50, 0, 0 and 50. As a result you can see that the margin will now push the rectangle away from the left-hand boundary by 50 pixels and away from the bottom boundary by 50 pixels as well.

Also notice the odd way in which margins are defined. Margins are represented as a series of numeric values that are separated by commas. This convention was borrowed from Cascading Style Sheets. So the numbers represent the margin pixel values in a clockwise fashion, starting at the left-hand side.

A bit earlier, I said that it is generally a better idea to use relative sizes like auto or star sizing whenever you want to define heights and widths. So why is it then that margins are defined in exact pixels? Usually margins are just small values to provide spacing or padding between two relative values and so they can be a fixed size without negatively impacting the overall layout of the page. If you want a small amount of spacing between two rectangles 50 pixels should suffice whether you have a large or a smaller size. And if it is not, then you can change it through other techniques that I'll demonstrate in this series.

To recap, in this lesson, we talked about layout controls and how they allow you to define areas of your application where other visual XAML controls will be hosted. In this lesson, we specifically learned about the Grid and how to define columns and rows, how to define their relative sizes using star and auto, and then how to specify which row and column a given control would request to be inside of by setting Attached Properties (I.e., Grid.Row, Grid.Column) on that given XAML Control.

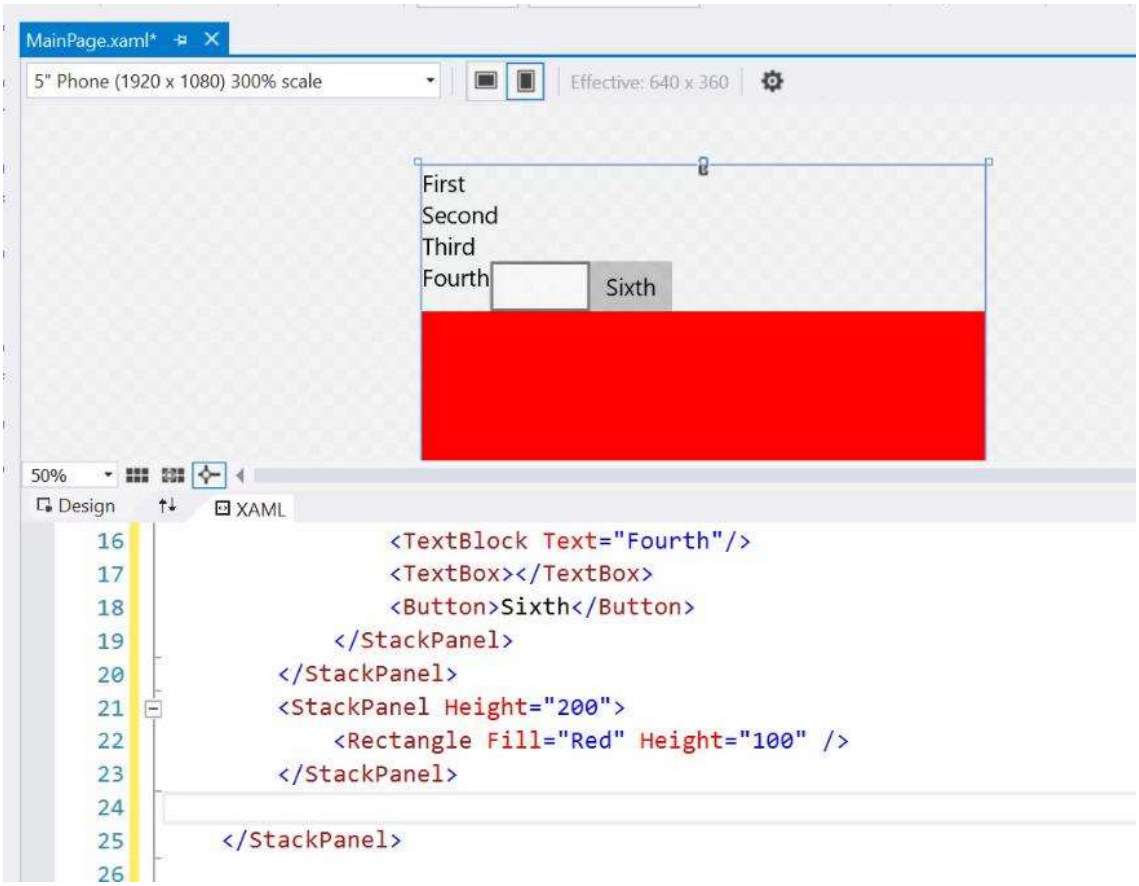
We also talked about how to set the alignment and the margins of those controls inside of a given cell and more.

UWP-009 - XAML Layout with StackPanel

The StackPanel layout control allows you to arrange your XAML Controls in a flow from top to bottom by default. Or we can change the Orientation property to flow from left to right in a horizontal fashion. We can even change the flow to go from left-to-right to right-to-left.

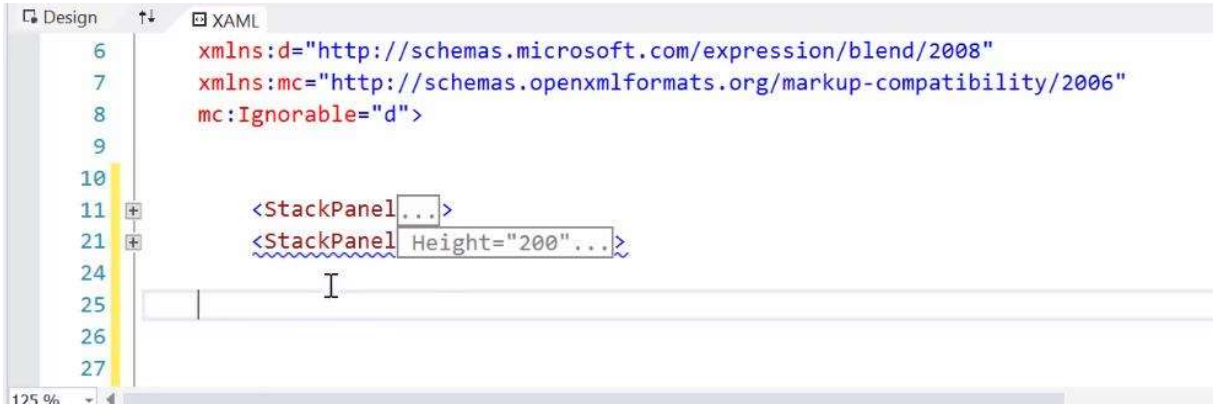
Creaing a StackPanel is simple. It's simply a panel (almost like a div tag, if you're familiar with HTML development), and inside of the opening and closing tags you can add XAML controls which will be stacked on top of each other or side by side.

I created a very simple application named SimpleStackPanel.



As a side note: notice that I use a little plus and minus symbols in the left-most column to roll up my code so we can see the overall structure of this application.

And you might be wondering, well why do we have StackPanels inside of StackPanels? Why can't we just do something like this?



If I remove the outer-most StackPanel I get the blue squiggly line with the familiar exception that the property "Content" is set more than once. A Page only has a Content property and it does not have a Children property collection. So that's why we need an outermost StackPanel and then inside of that we can put as many StackPanels or other XAML controls as needed.

Inside of the first StackPanel, notice that I have three TextBlocks that are stacked on top of each other.

```
Design  XAML
10      <StackPanel>
11          <StackPanel>
12              <TextBlock>First</TextBlock>
13              <TextBlock>Second</TextBlock>
14              <TextBlock>Third</TextBlock>
15              <StackPanel Orientation="Horizontal">
16                  <TextBlock Text="Fourth"/>
17                  <TextBox></TextBox>
18                  <Button>Sixth</Button>
19              </StackPanel>
20          </StackPanel>
```

Notice they take up the full width because we haven't specified a width and by default, and that they're arranged in a vertical fashion. The item at the top will be the first item stacked, the item at the bottom will be the last item stacked. So the stacking is performed in order.

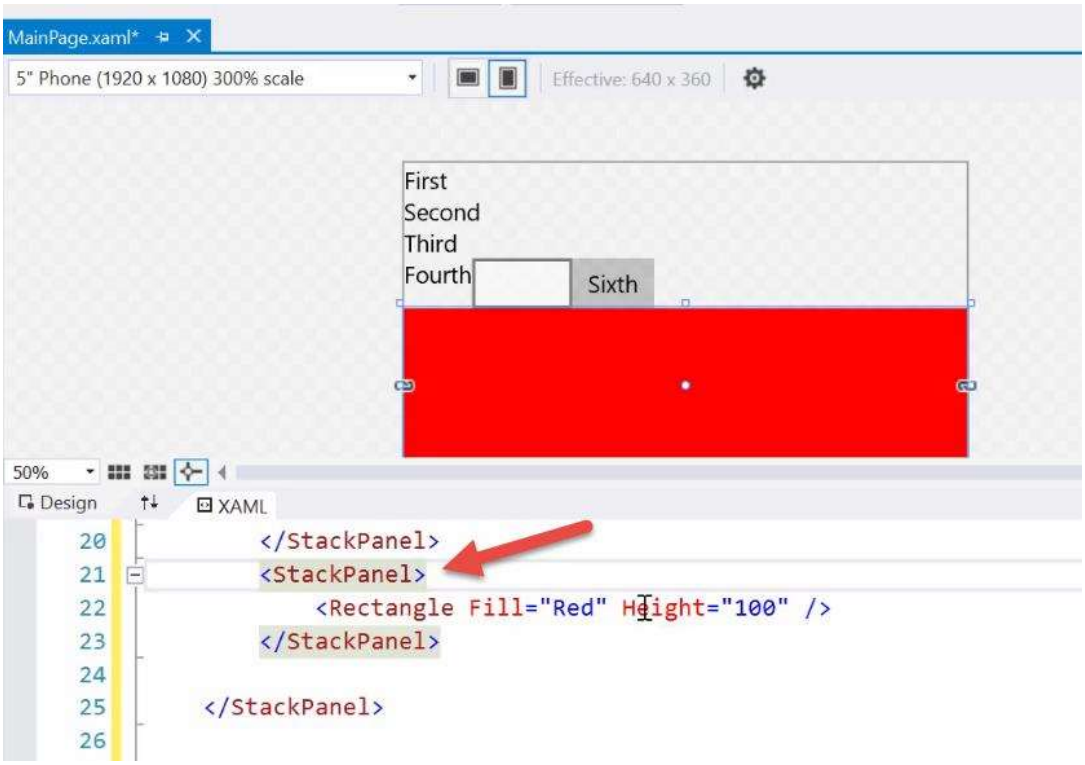
Furthermore, underneath the third TextBlock, I've added a second StackPanel. I set its Orientation to Horizontal. I'm using the StackPanel here as another "row" that's stacked beneath the TextBlocks. Inside of the inner-most StackPanel, I stacked several XAML Controls in a horizontal fashion. The leftmost stacked item is a TextBlock that has the word "Fourth" in it, the next item that is stacked next to it is a TextBox, then the next item is a button with the content, "Sixth", in it.

We can achieve almost anything by simply using this technique of stacking StackPanels inside of StackPanels. I tend to use StackPanel actually more than I use the Grid for layout. Personally, I think that it allows me to get the design that I want and it gives me the flow that I want regardless of the size of the device that the app is actually running on.

You can also see that I've got a StackPanel defined beneath it with a simple rectangle.

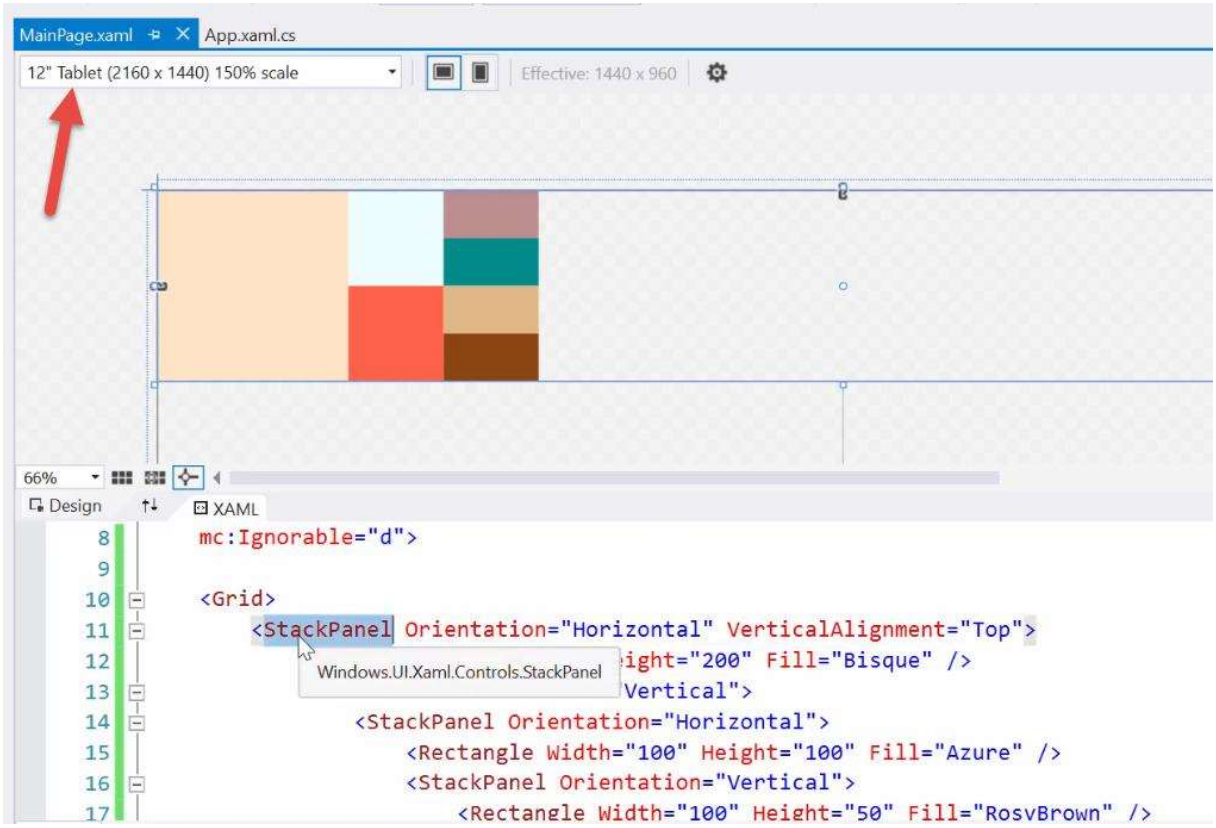
```
Design  XAML
20          </StackPanel>
21          <StackPanel Height="200">
22              <Rectangle Fill="Red" Height="100" />
23          </StackPanel>
24
25      </StackPanel>
```

The only thing I want to illustrate here is that the height of the StackPanel can be set, and yet items inside of the StackPanel can have their own height independent of the parent. If I were to remove the height, notice what happens:



Essentially, the Height of the StackPanel is set to Auto. If we were try to set it to star (*), it wouldn't work but we can set it to a numerical pixel value or we can just leave it at the default, which is Auto.

I created another example called ComplexStackPanel. Note that I have changed from the default viewer, that would be a five-inch phone, to a 12" Tablet to accommodate the larger width of this design that I created.



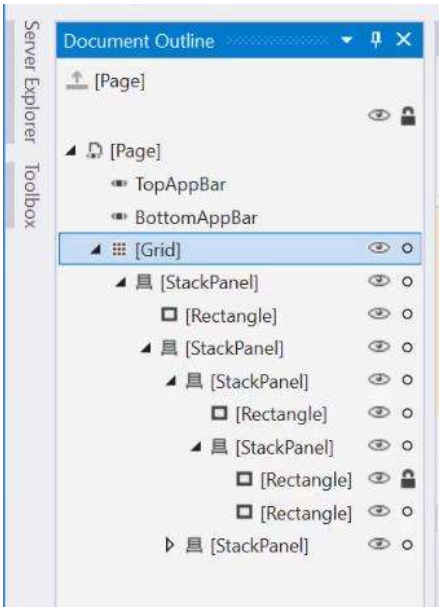
This example takes the concept of adding StackPanels inside of StackPanels to an extreme. Admittedly, it can become confusing to read the code and understand what code is producing a given StackPanel, however, hopefully, between the color descriptions and holding your mouse cursor on it and seeing the little boundary box that is selected you can make heads or tails of how this works.

The topmost StackPanel defines a “row” of sorts. The height of the top-most StackPanel is actually dictated by the items that are inside of it … specifically rectangle with the Fill="Bisque". Its height is set to 200 so that sets the height for the entire StackPanel since nothing else inside of it has a greater height.

Also, you'll notice that I set the VerticalAlignment of this top-most StackPanel, and the reason I did that, because if you change that, it will now get moved to the vertical middle of the grid cell, and that's just the difference in how grids work and StackPanels work.

So the result of StackPanels inside of StackPanels produces the desired intricacy until I have the series of rectangles that resemble something like an ornate pane of Frank Lloyd Wright designed glass.

One of the things that can help you out whenever you are working through an intricate amount of XAML and it's difficult to find your way around is this little Document Outline. If you do not see it by default on this left-most next to the Toolbox, you can go to the View menu > Other Windows > Document Outline.



You can expand each node in the Document Outline to reveal the hierarchy of items in the Designer. This is a representation of a “visual tree”. By selecting a given leaf in the tree you can see the little boundary box selection around the associated XAML Control in the Designer.

Furthermore, the Document Outline is very convenient whenever you want to make changes in the Properties window and it's difficult to find the exact item that you are looking for just by clicking around the Designer and the XAML is too intricate.

The Document Outline also can be used to hide items by clicking the left-column on the right-hand side to remove certain items from view. We can also lock items which means that they can't be selected and therefore they cannot easily be changed in the Properties window. You can see that it adds this IsLocked="True" XAML to the design time experience.

The final example project named GridAndStackPanel demonstrates a “gotcha” when using StackPanels. Furthermore it should help us to have a better understanding of the difference between Grids and StackPanels, and something that you need to watch out for and completely understand.

```
 9
10      <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11          <StackPanel...>
21          <StackPanel Height="200"...>
24      </Grid>
25  </Page>
```

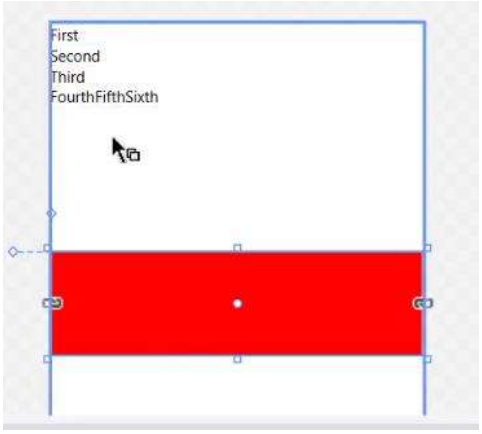
This is almost identical to the first example that we used earlier in this lesson. This time, we have a Grid that surrounds two StackPanels. The top-most StackPanel contains three TextBlocks, first, second and third, and then another embedded StackPanel whose Orientation property is set to Horizontal. This time I just have three TextBlocks that will be stacked horizontally, so we got fourth, fifth and sixth.

```
11          <StackPanel>
12              <TextBlock>First</TextBlock>
13              <TextBlock>Second</TextBlock>
14              <TextBlock>Third</TextBlock>
15              <StackPanel Orientation="Horizontal">
16                  <TextBlock>Fourth</TextBlock>
17                  <TextBlock>Fifth</TextBlock>
18                  <TextBlock>Sixth</TextBlock>
```

Below that, we have another StackPanel with a rectangle inside of it.

```
20          </StackPanel>
21          <StackPanel Height="200">
22              <Rectangle Fill="Red" Height="100" />
23          </StackPanel>
24      </Grid>
```

Now in this case, you'll look at it and say, why is there so much space in between this Rectangle and the bottom of this StackPanel?



And you might attempt to resolve this by setting the VerticalAlignment=”Top”. It would appear as though you lost the top-most StackPanel. You didn't lose it, actually. It is sitting behind the second StackPanel (the one with the red Rectangle).

There are three things going on here at the same time, and this will hopefully help you to understand the difference between Grids and StackPanels.

First, some controls like image and rectangle controls that set themselves inside of a grid cell will be set to 100 width and 100 height by default. So that is also true of StackPanels. When you put a StackPanel inside of a grid cell, in this case the default grid cell, cell zero or row zero, column zero, then the two StackPanels are essentially, well, the first StackPanel has set itself to 100 height and 100 width.

Now the second StackPanel has set itself to 200 height, but the second thing that you need to understand that by default the content in a cell of a grid will be vertically and horizontally centered, so

we have the second StackPanel, and let's remove the VerticalAlignment. The default here would be center. And so you can see that it is actually sitting, the entire StackPanel is in the center of that cell.

Third, you can easily overlap items in a grid cell. So if two or more controls are set to reside in the exact same grid cell, and their HorizontalAlignment equals top and their VerticalAlignment equals top, then they will literally sit on top of each other. We're actually looking at two StackPanels. This first StackPanel takes up the entire length, and the items inside of it, however, are aligned to the top of that StackPanel, but the StackPanel itself takes up the whole frame. And then the second StackPanel is set into the middle of that cell, but it's only 200 tall. However, whenever we change the VerticalAlignment and now we move that up to the very top of that single cell inside the grid, that's when we overlapped.

So I just want you to understand that about grids and about StackPanels, that StackPanels will never ever allow you to put two of them on top of each other where you cannot see the one that is underneath it. Whereas grid cells will absolutely allow you to do that.

There are a couple of different ways that we could rectify this. The easiest one is just to use StackPanels as the outermost container here in this particular layout. However, you could also create two rows instead of just one row and put this first StackPanel in the top row, put the second StackPanel in the bottom row, and That will ensure that they don't overlap. You could also set the StackPanels, you can set its VerticalAlignment to the top but then use a margin to push it down. That seems a little more fragile, I don't know if I like that idea.

There are a couple of different ways to still make this work even though you are working with a grid. But ideally, you'll probably switch to just using StackPanels. In fact, like I said earlier, I typically use the StackPanel just about for everything. I don't use Grids as much as I used to, except to give the page an overall structure. And with everything else, I try to use StackPanels.

There is this special technique, however, that I learned when we talk about adaptive triggers and adaptive layout that utilizes Grids to adapt from a desktop to a mobile layout, that wouldn't be possible with a StackPanel. The Grid still does have its uses. I just prefer, in most cases, to stick with the StackPanel.

UWP-010 - Cheat Sheet Review: XAML and Layout Controls

Let you in on a little secret, nobody memorizes all this stuff, at least not at first. It would probably take you several months of working and building applications in order to really internalize a lot of this. IntelliSense is awesome, and it really helps you through a lot of the rough spots. Little code snippets online are great, I use cheat sheets, which allow me to, they're just basically notes that I've gathered from watching a video series, or from reading articles online. It helps me to organize the information, see it visually, and then organize it in my mind as well. We will use this as a form of, of review throughout this series of lessons.

This is what I promote on LearnVisualStudio.NET, and it's a great way to index in to the content to remind yourself of what the major ideas where, the crux of those videos. You can always remove the content you don't need and you can add content that you might think might be helpful in the future to customize this resource for how you will use it.

Note: In the video I narrate through the creation of the cheat sheet, however in this PDF version I will simply paste in the cheat sheet review content for the sake of brevity.

UWP-04 - What is XAML?

XAML - XML Syntax, create instances of Classes that define the UI.

UWP-05 - Understanding Type Converters

Type Converters - Convert literal strings in XAML into enumerations, instances of classes, etc.

UWP-06 - Understanding Default Properties, Complex Properties and the Property Element Syntax

Default Property ... Ex. sets Content property:

<Button>Click Me</Button>

Complex Properties - Break out a property into its own element syntax:

```
<Button Name="ClickMeButton"
    HorizontalAlignment="Left"
    Content="Click Me"
    Margin="20,20,0,0"
    VerticalAlignment="Top"
    Click="ClickMeButton_Click"
    Width="200"
    Height="100">
```

```
        <Button.Background>
          <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="Black" Offset="0"/>
            <GradientStop Color="Red" Offset="1"/>
          </LinearGradientBrush>
        </Button.Background>
      </Button>
```

UWP-07 - Understanding XAML Schemas and Namespace Declarations

Don't touch the schema stuff - it's necessary!

Schemas define rules for XAML, for UWP, for designer support, etc.

Namespaces tell XAML parser where to find the definition / rules for a given element in the XAML.

UWP-08 - XAML Layout with Grids

Layout controls don't have a content property ... they have a Chidren property of type UIElementCollection.

By embedding any control inside of a layout control, you are implicitly calling the Add method of the Children collection property.

```
<Grid Background="Black">
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
</Grid>
```

Sizes expressed in terms of:

Explicit pixels - 100

Auto - use the largest value of elements it contains to define the width / height

* (Star Sizing) - Utilize all the available space

1* - Of all available space, give me 1 "share"

2* - Of all available space, give me 2 "shares"

3* - Of all available space, give me 3 "shares"

6 total shares ... 3* would be 50% of the available width / height.

Elements put themselves into rows and columns using attached property syntax:

```
...
  ...
  <Button Grid.Row="0" />
</Grid>
```

- When referencing Rows and Columns ... 0-based.
- There's always one default implicit cell: Row 0, Column 0
- If not specified, element will be in the default cell

UWP-09 - XAML Layout with StackPanel

```
<StackPanel>
  <TextBlock>Top</TextBlock>
  <TextBlock>Bottom</TextBlock>
</StackPanel>
```

- Vertical Orientation by default.
- Left-to-right flow by default when Horizontal orientation.
- Most layouts will combine multiple layout controls.
- Grid will overlap controls. StackPanel will stack them.