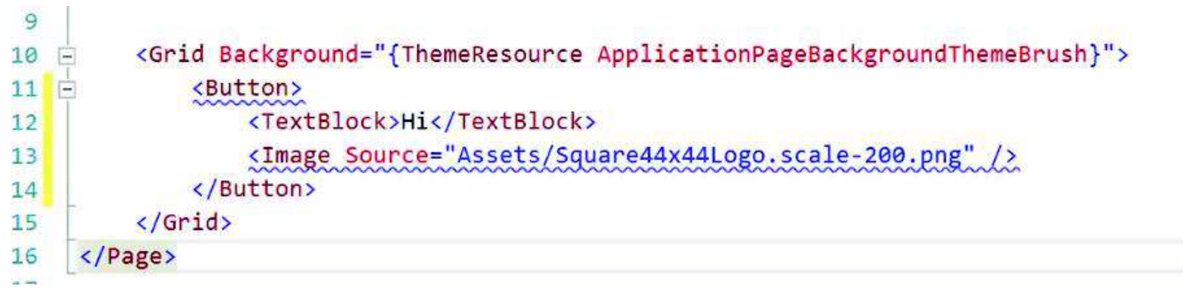
UWP-008 - XAML Layout with Grids

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:



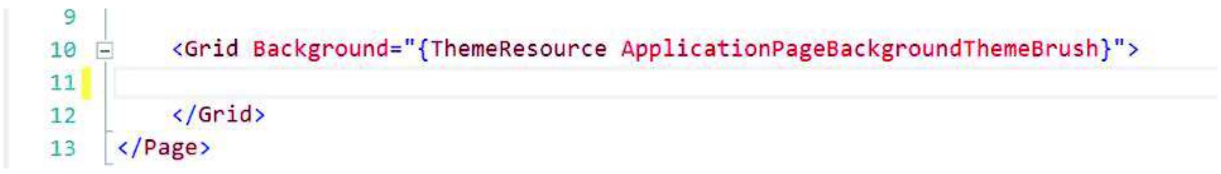
```
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Button>
            <TextBlock>Hi</TextBlock>
            <Image Source="Assets/Square44x44Logo.scale-200.png" />
        </Button>
    </Grid>
</Page>
```

… 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.

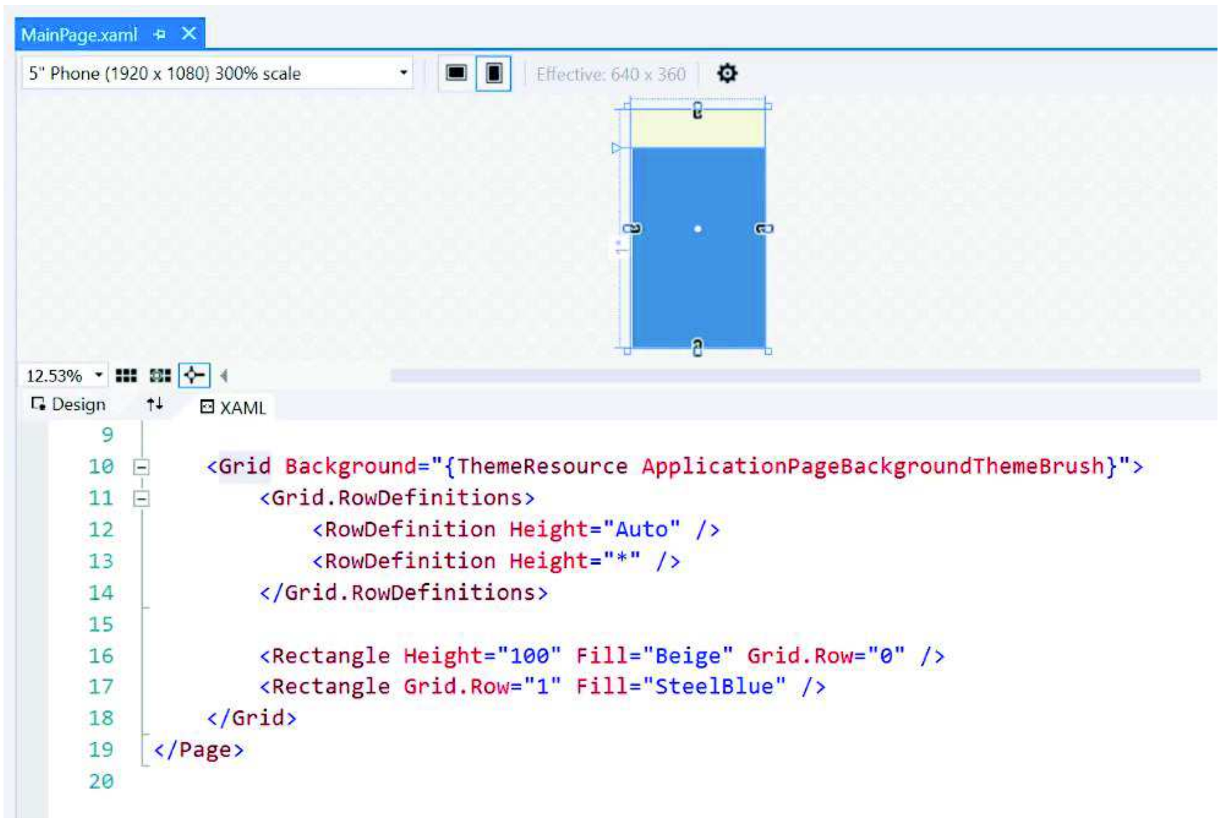


```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

</Grid>
</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.



```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Rectangle Height="100" Fill="Beige" Grid.Row="0" />
    <Rectangle Grid.Row="1" Fill="SteelBlue" />
</Grid>
</Page>
```

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.

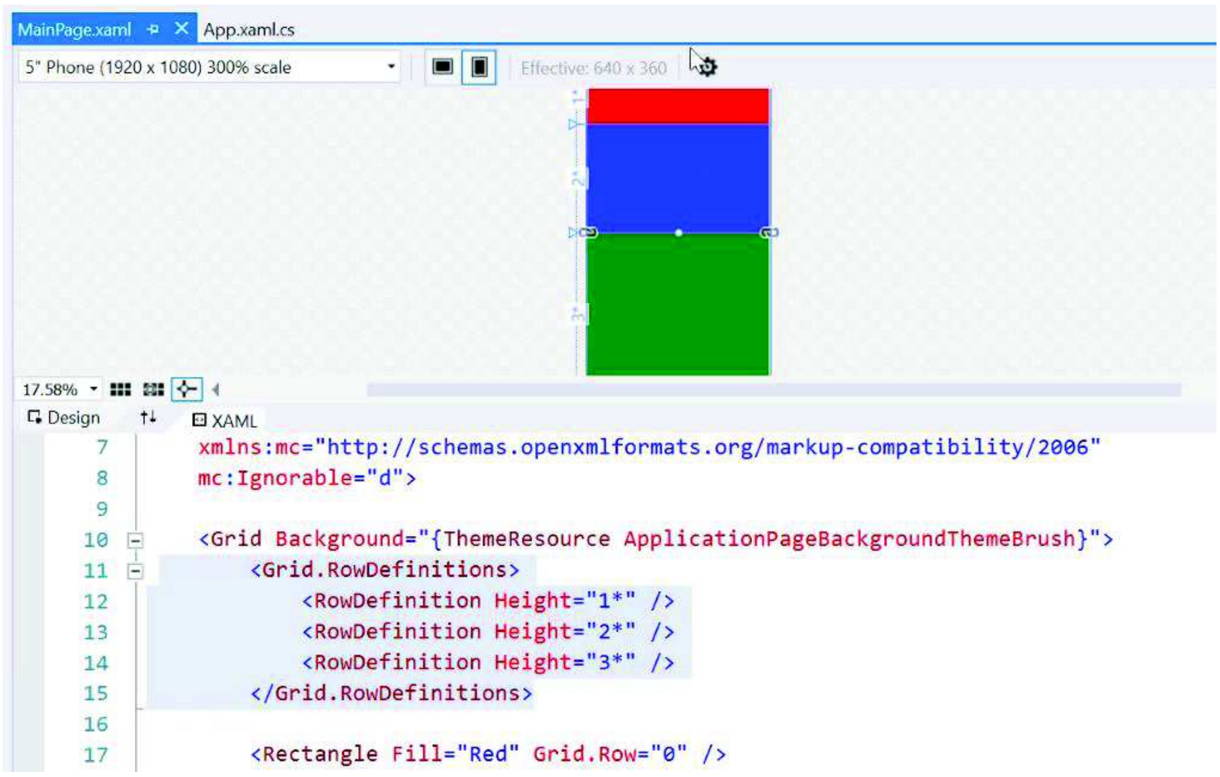
The reason why Attached Properties exist is an advanced XAML topic that is not actionable at this point in your introduction to XAML. If you want to get deeper into the internals of XAML, then you should search MSDN for articles about both "Attached Properties", as well as the loosely related topic of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



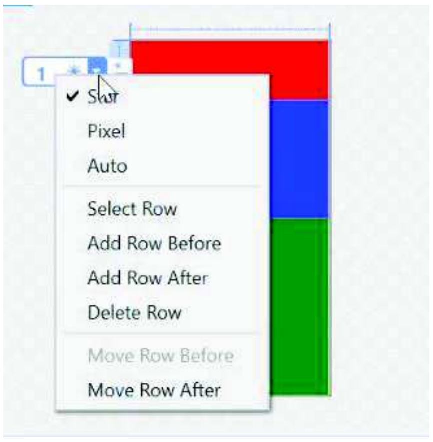
```
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d">

        <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
            <Grid.RowDefinitions>
                <RowDefinition Height="1*" />
                <RowDefinition Height="2*" />
                <RowDefinition Height="3*" />
            </Grid.RowDefinitions>

            <Rectangle Fill="Red" Grid.Row="0" />
```

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.

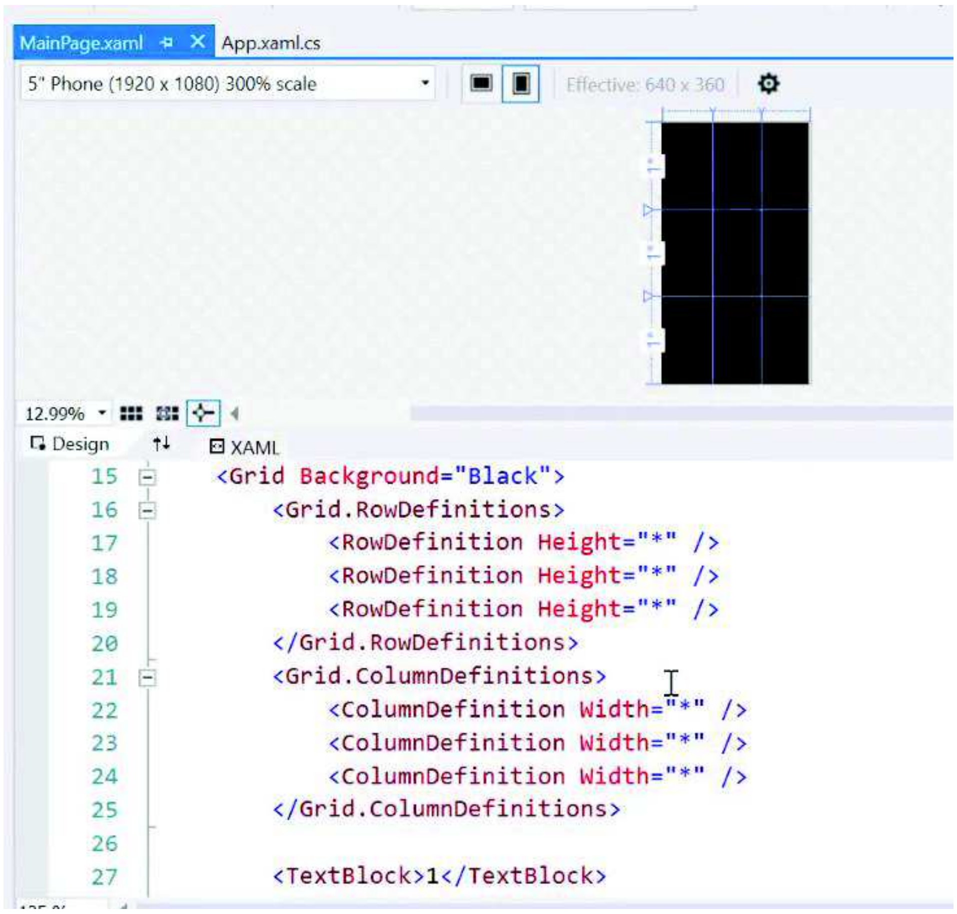
Also notice off to the left-hand side of the visual designer’s depiction of the Grid that there are some visual tools that we can use to change the sizing. For example, I can change from star sizing to auto, or to pixel. I can also just type in the given value here, and that would set the Height property.



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.

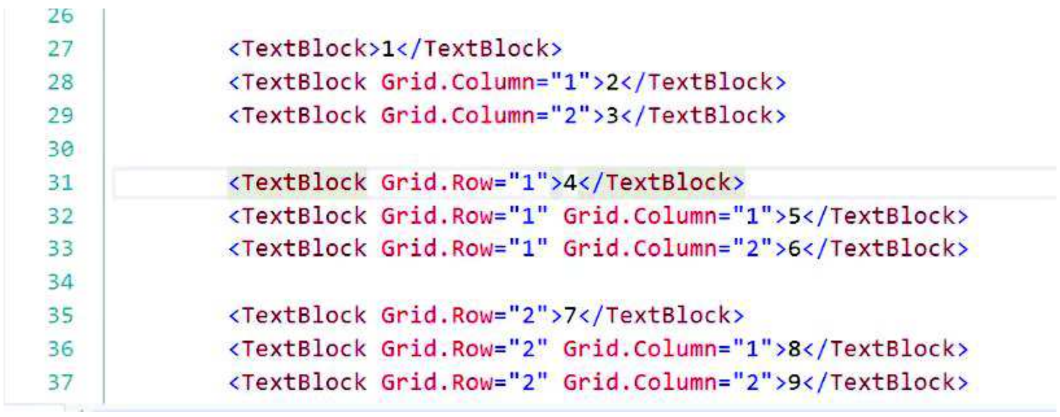
A Grid can have a collection of ColumnDefinitions. In another example named "GridsRowsAndColumns" you can see that I created a 3 x 3 grid, three RowDefinitions and a ColumnDefinitions collection that contains three ColumnDefinitions.



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

    <TextBlock>1</TextBlock>
```

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

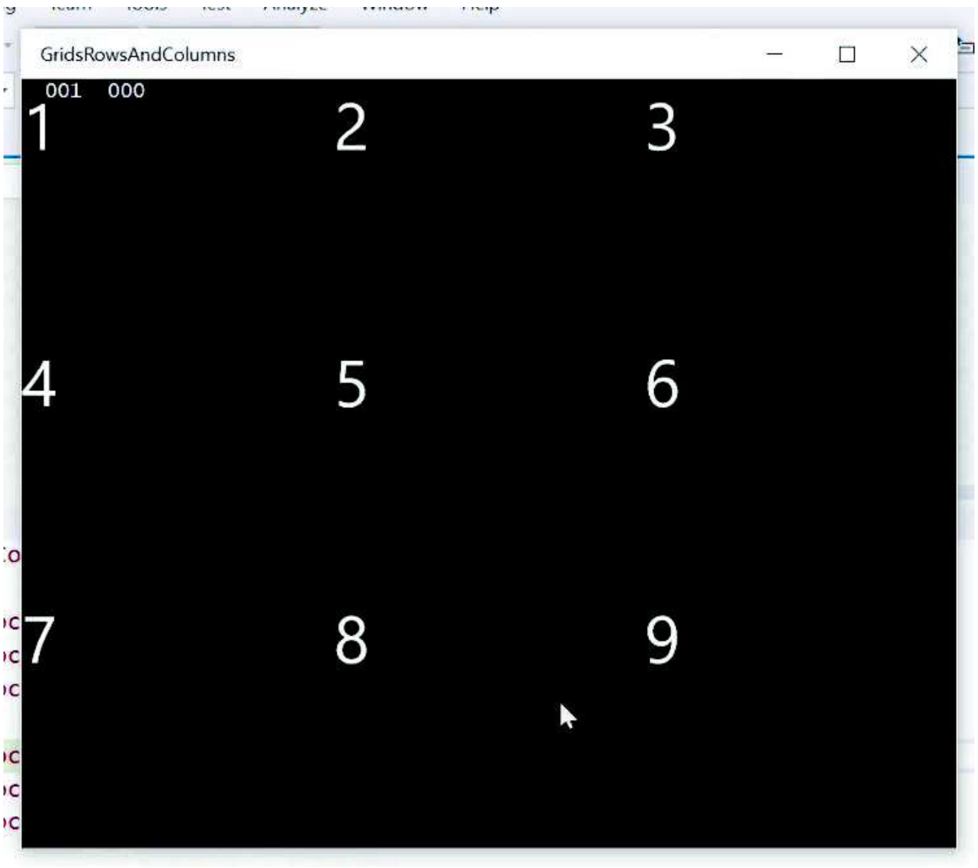


```
        <TextBlock>1</TextBlock>
        <TextBlock Grid.Column="1">2</TextBlock>
        <TextBlock Grid.Column="2">3</TextBlock>

        <TextBlock Grid.Row="1">4</TextBlock>
        <TextBlock Grid.Row="1" Grid.Column="1">5</TextBlock>
        <TextBlock Grid.Row="1" Grid.Column="2">6</TextBlock>

        <TextBlock Grid.Row="2">7</TextBlock>
        <TextBlock Grid.Row="2" Grid.Column="1">8</TextBlock>
        <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.

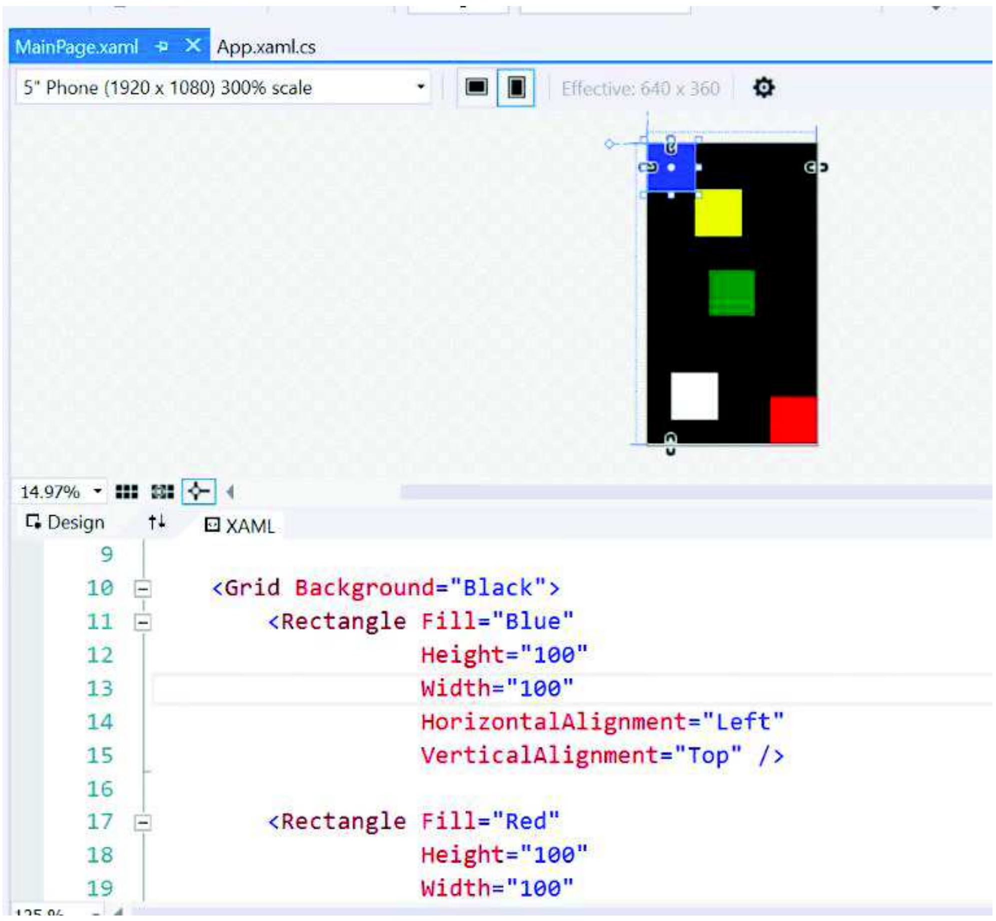


In this very first cell in the upper left-hand corner I am not setting a row, nor am I setting a column. By default, if you do not supply that information, it is assumed to be zero. So we're assuming that we're putting a TextBlock containing “1” in row zero, column zero.

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.



```
<Grid Background="Black">
    <Rectangle Fill="Blue"
               Height="100"
               Width="100"
               HorizontalAlignment="Left"
               VerticalAlignment="Top" />

    <Rectangle Fill="Red"
               Height="100"
               Width="100"
```

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.