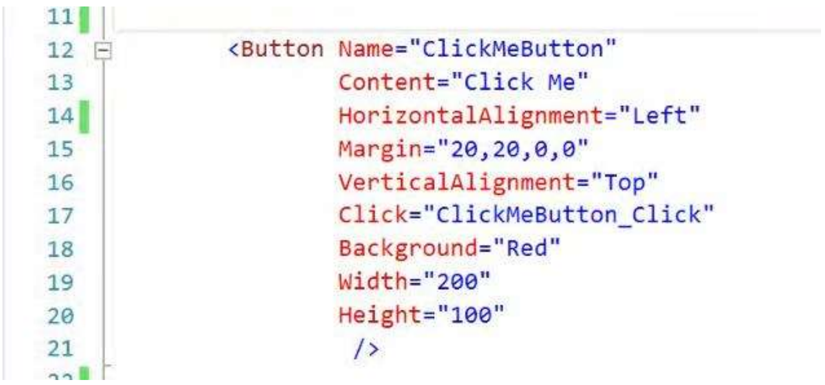
UWP-005 - Understanding Type Converters

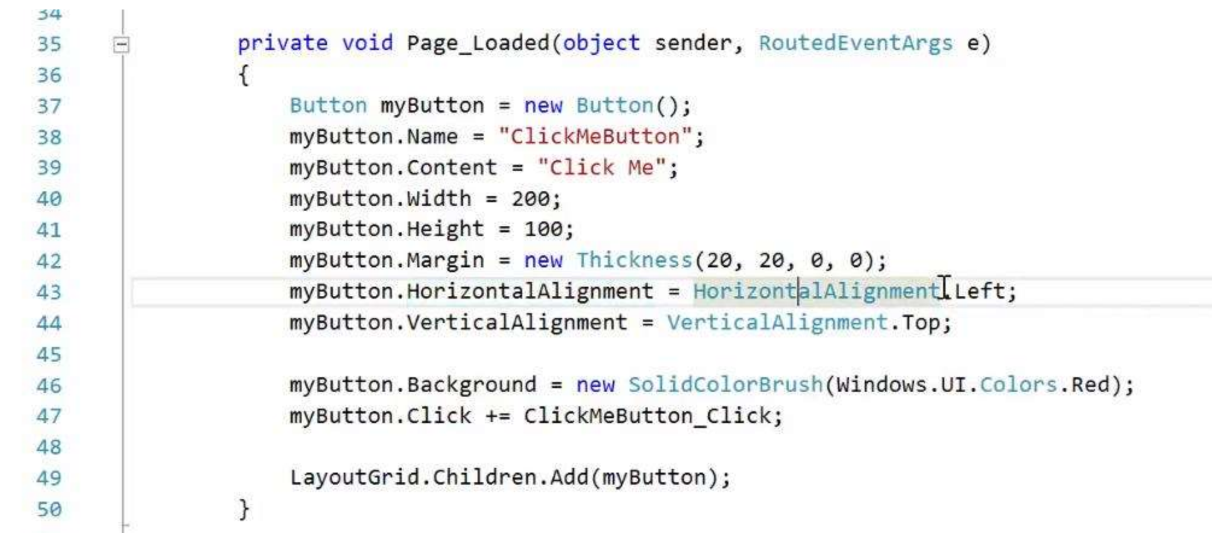
In this lesson I want to explain one curious little feature called Type Converters that we see in play in the example we created in the previous lesson.

If you took a few moments to examine the project, and if you employed a keen eye, you may have noticed that the HorizontalAlignment property is set to a string with the value "Left".



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

However, the C# version is a little bit different. (You see a similar situation when observing the VerticalAlignment property set to "Top".)



```
        private void Page_Loaded(object sender, RoutedEventArgs e)
        {
            Button myButton = new Button();
            myButton.Name = "ClickMeButton";
            myButton.Content = "Click Me";
            myButton.Width = 200;
            myButton.Height = 100;
            myButton.Margin = new Thickness(20, 20, 0, 0);
            myButton.HorizontalAlignment = HorizontalAlignment.Left;
            myButton.VerticalAlignment = VerticalAlignment.Top;

            myButton.Background = new SolidColorBrush(Windows.UI.Colors.Red);
            myButton.Click += ClickMeButton_Click;

            LayoutGrid.Children.Add(myButton);
        }
```

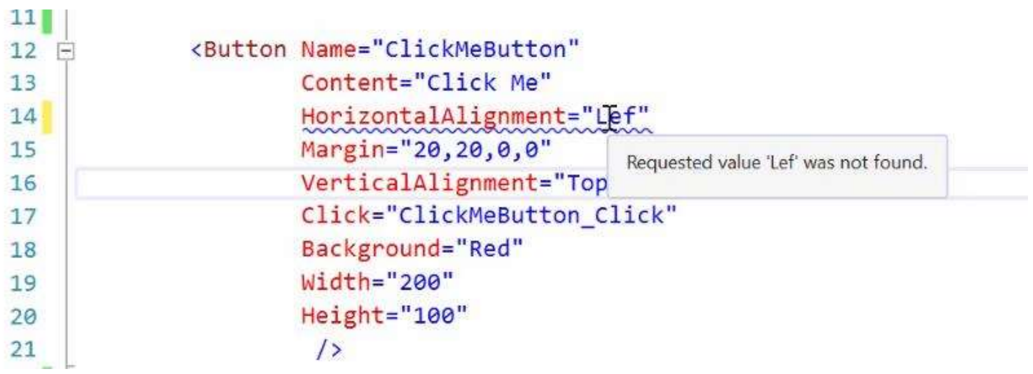
If you take a look at when we set that HorizontalAlignment property, we're actually using a strongly typed enumeration (not a string) of type Windows.UI.Xaml.HorizontalAlignment and the particular enumeration is also Left -- however it is strongly typed.

How does this work? Why is it that we can use a string here, but we have to use a strong type in C#? The reason why this works is because the XAML parser will perform a conversion to turn this string value into a strongly typed version of that value of type Windows.UI.Xaml.HorizontalAlignment.left through the use of a feature called a Type Converter.

A Type Converter is simply a class that has one function and that is to translate a string value into a strong type. And there are several of these that are built into the Universal Windows Platform API that we use throughout this series.

So, in this example of the HorizontalAlignment property, when it was developed by Microsoft's developers it was marked with a special attribute in the source code, which signals to the XAML parser that looks through our code and makes sure everything is okay and that it would compile to run that string value “Left” through a Type Converter and try to match the literal string “Left” with one of the enumeration values defined in this Windows.UI.Xaml.HorizontalAlignment enumeration.

Just for fun, let's take a look at what happens if we were to misspell the word "Left". Let's get rid of the "t" on the end.



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

Requested value 'Lef' was not found.

Immediately the XAML parser says, "Wait a second. "I can't find a match for this literal string, lef, (without the T), with any of the enumerations that are defined in this Windows.UI.Xaml.HorizontalAlignment enumeration."

This will cause a compilation error because the Type Converter can't find the exact match to convert it to a strong type.

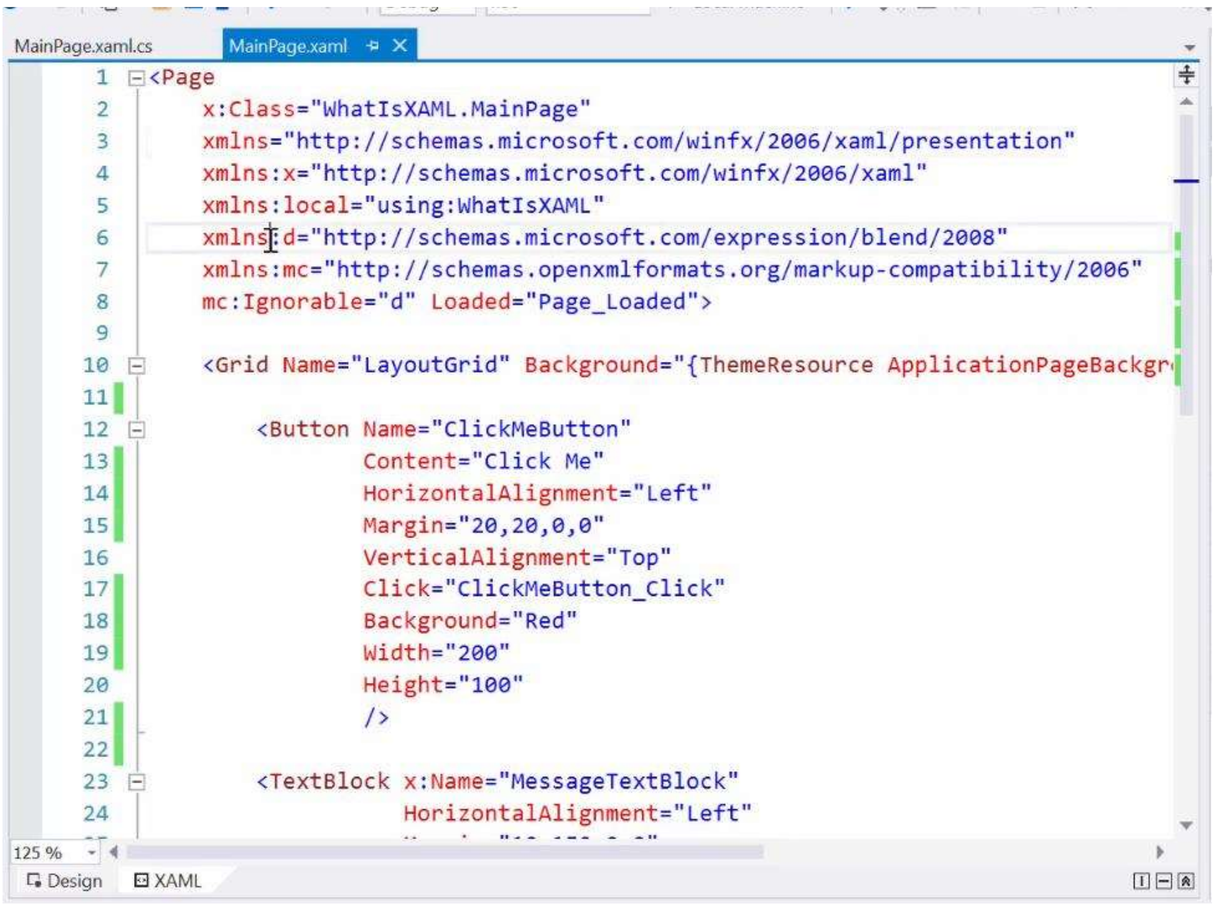
So, one of the first characteristics of XAML is that it is a succinct means of creating instances of classes and setting properties in Type Converters. This is one little trick that XAML uses to accomplish this so that we don't have to waste characters and that we can actually just use a very succinct syntax to set values that might be actually representing longer class and enumeration names behind the scenes.

UWP-006 - Understanding Default Properties, Complex Properties and the Property Element Syntax

This lesson will discuss other XAML syntax features, specifically Default Properties, Complex Properties, and the Property Element Syntax. Finally, it will discuss how an intelligent XAML parser reduces extraneous XAML code because it can infer the relationships between the elements by their context

To begin, notice the containment relationship between objects defined in XAML. Since XAML is essentially an XML document, we can embed elements inside of other elements and that implies a containment relationship.

Observing the MainPage.xaml at the top-most level there's Page object and inside of the opening and closing Page tag, there's a Grid object, and inside of the opening and closing Grid element, there's a both a Button and a TextBlock.



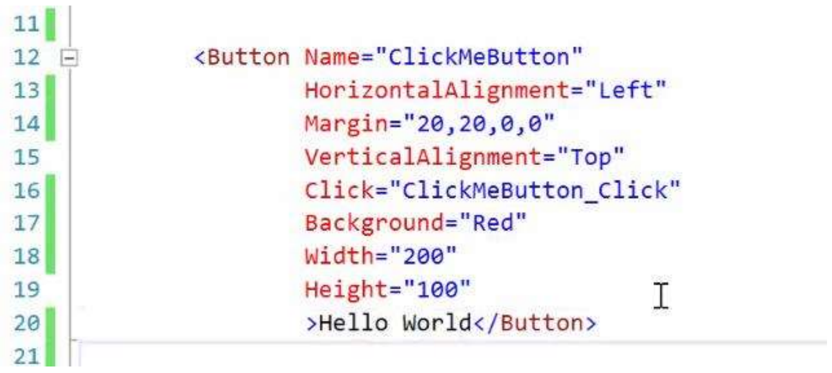
```
<Page
    x:Class="WhatIsXAML.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WhatIsXAML"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Loaded="Page_Loaded">

    <Grid Name="LayoutGrid" Background="{ThemeResource ApplicationPageBackgr

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

        <TextBlock x:Name="MessageTextBlock"
                   HorizontalAlignment="Left"
```

The proper parlance in XAML is that the Page's content property is set to a new instance of the Grid control. The Grid control has a property collection called Children. In this case two instances a Button and a TextBlock are added to the Children collection.

Admittedly those properties (I.e., Content, Children) are not explicitly defined in the XAML. However, what is going on behind the scenes is that those properties are being set. I'll address how this is possible at the very end of this lesson. For now, understand that there's more than meets the eye.

By using XAML to indicate containment we're actually setting the Grid's Children property and we're setting the Pages' content property. If you take a look at the C# version of our code, it more accurately shows that relationship between the LayoutGrid and the Button and TextBlock. The LayoutGrid has a Children collection of controls, and we're adding to that Children collection an instance of the Button, OK. We just don't see that here in our XAML.

Next, Default Properties are employed in the example's XAML. Depending on the type of control that you're working with, the Default Property for that control can be populated by using this embedded style syntax where we have XAML elements defined inside of other XAML elements. For example, the Button control's Default Property is Content, so I can remove that property from the properties or the attributes of that element, and then I can also just remove that self-enclosing syntax there for the element, and just create a proper closing tag for the Button.

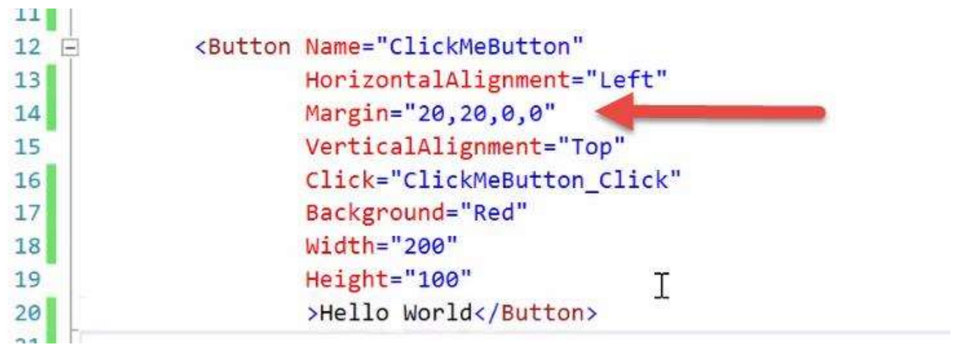


```
        <Button Name="ClickMeButton"
                HorizontalAlignment="Left"
                Margin="20,20,0,0"
                VerticalAlignment="Top"
                Click="ClickMeButton_Click"
                Background="Red"
                Width="200"
                Height="100"
                >Hello World</Button>
```

By using that Default Property syntax for the Button, we can actually see in the Design View that we've changed the Content property for the Button.

Next, let's revisit Type Converters for a moment. In last lesson, I only highlighted some simple Type Converters like the HorizontalAlignment property's string value to perform the type conversion to an instance of an enumeration Windows.UI.Xaml.HorizontalAlignment.Left.

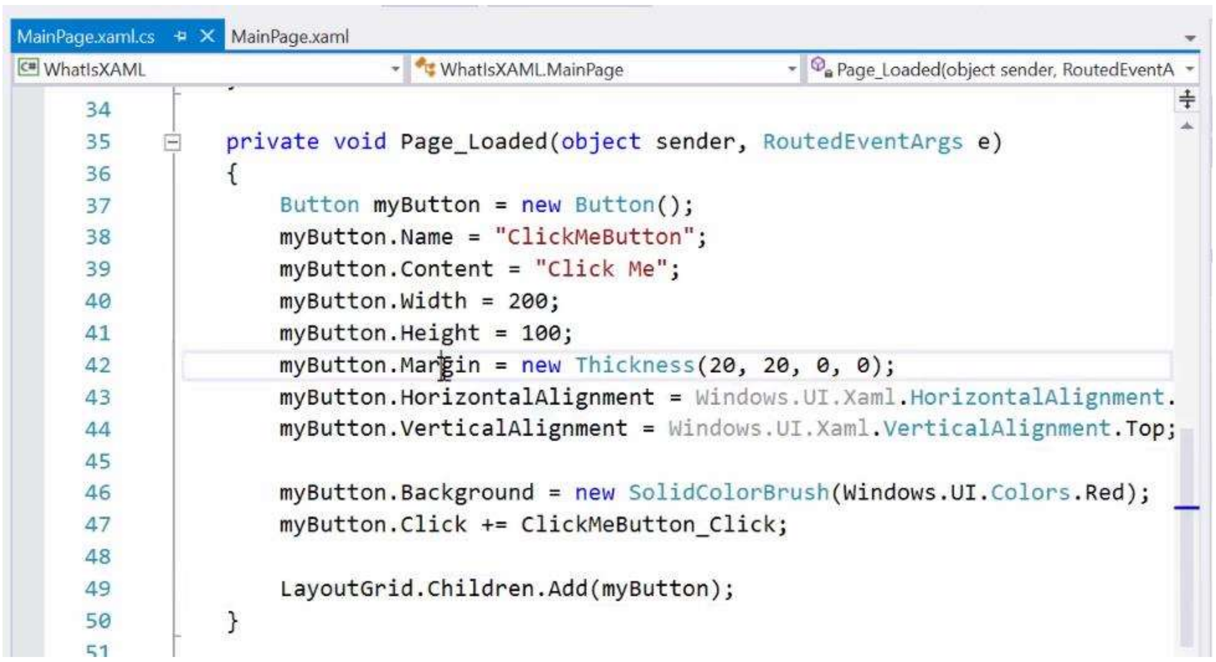
However, there are more complex versions of Type Converters, like we see here in the Margin.



```
        <Button Name="ClickMeButton"
                HorizontalAlignment="Left"
                Margin="20,20,0,0"
                VerticalAlignment="Top"
                Click="ClickMeButton_Click"
                Background="Red"
                Width="200"
                Height="100"
                >Hello World</Button>
```

Here, our Margin property in XAML is set to a literal string of "20, 20, 0, 0", and if you take a look at the Background property, it also is set to just a simple string, a literal string of "Red".

But if you take a look at the C# version, there are a little bit more going on here. The Type Converters have to work a little bit harder than they did with the Horizontal and Vertical Alignment. In the case of the Margin, notice that we're just not giving it an enumeration but we're actually creating a new instance of an object called Thickness, and that Thickness has a constructor that accepts four input parameters that happen to be the left margin, top margin, right margin, bottom margin.

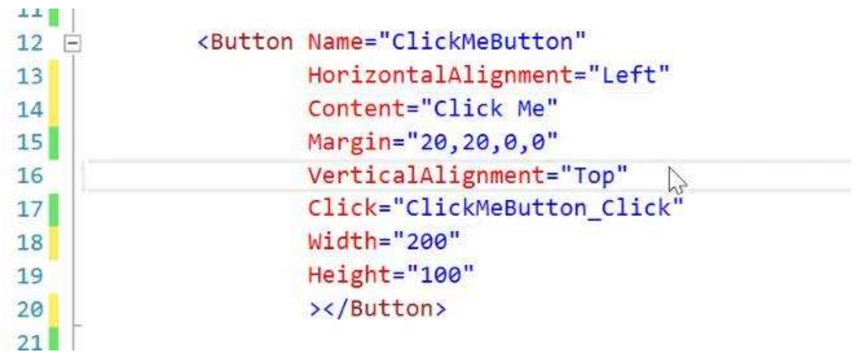


```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
    Button myButton = new Button();
    myButton.Name = "ClickMeButton";
    myButton.Content = "Click Me";
    myButton.Width = 200;
    myButton.Height = 100;
    myButton.Margin = new Thickness(20, 20, 0, 0);
    myButton.HorizontalAlignment = Windows.UI.Xaml.HorizontalAlignment.
    myButton.VerticalAlignment = Windows.UI.Xaml.VerticalAlignment.Top;

    myButton.Background = new SolidColorBrush(Windows.UI.Colors.Red);
    myButton.Click += ClickMeButton_Click;

    LayoutGrid.Children.Add(myButton);
}
```

Likewise, if you take a look at the Background property here, we're actually setting it to a new instance of an object called a SolidColorBrush, and we're passing in as an input parameter to the constructor, a enumeration of type, Windows.UI.Colors.Red.

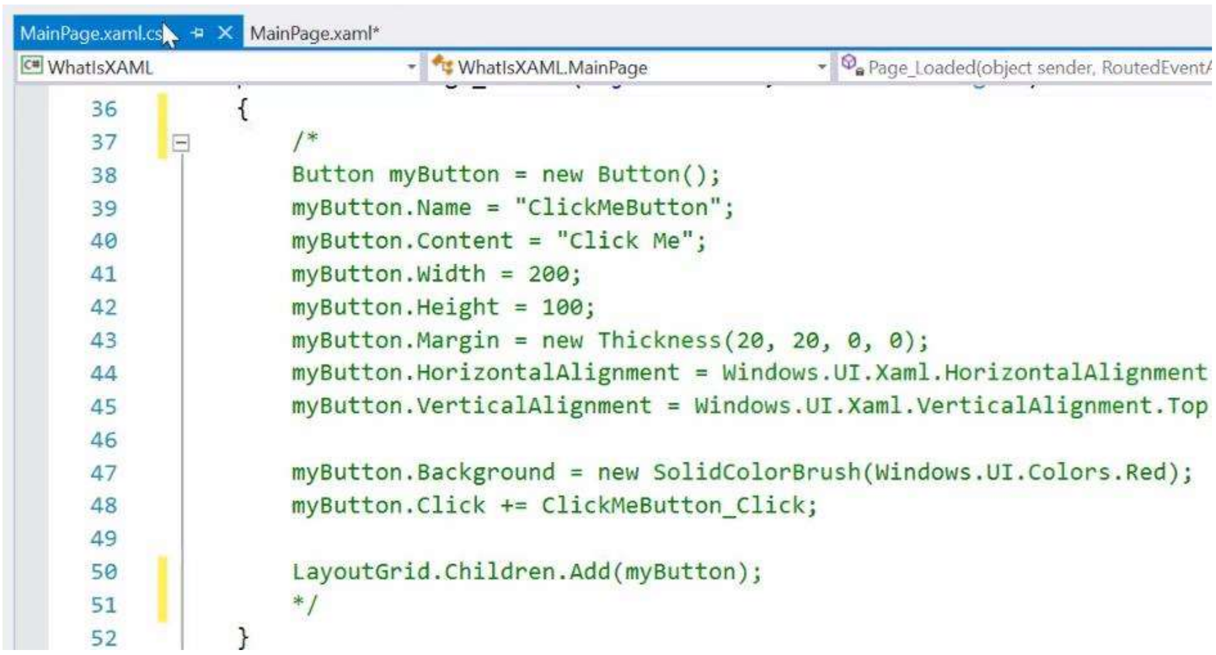
In some situations, there are propert values that are simply too complex to be represented merely by Type Converters, or handled by Type Converters. When a control's property is not easily represented by just a simple XAML attribute like we see in these examples that we've been looking at, then it's referred to as a Complex Property.

To demonstrate this, I will remove the Background attribute from the Button definition. I will also remove this Default Property and I'll reset the Content property here back to what it was.



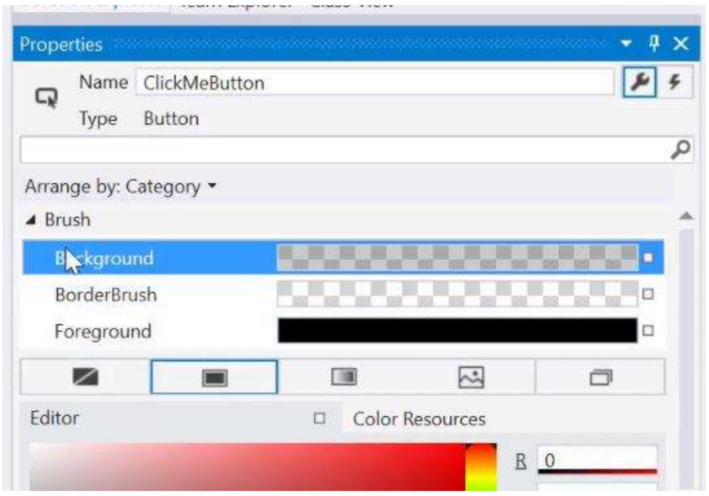
```
<Button Name="ClickMeButton"
        HorizontalAlignment="Left"
        Content="Click Me"
        Margin="20,20,0,0"
        VerticalAlignment="Top"
        Click="ClickMeButton_Click"
        Width="200"
        Height="100"
        ></Button>
```

I'll comment out the C# code in the Page_Loaded event:

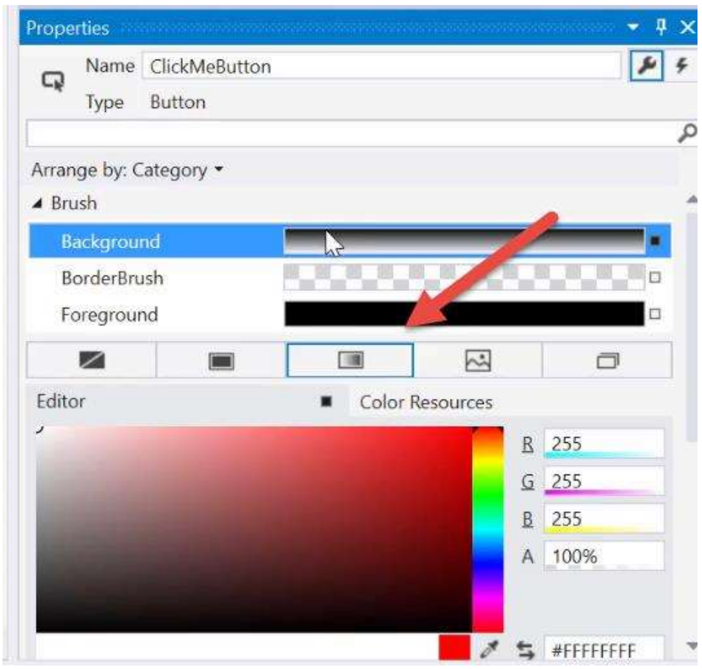


```
{
    /*
    Button myButton = new Button();
    myButton.Name = "ClickMeButton";
    myButton.Content = "Click Me";
    myButton.Width = 200;
    myButton.Height = 100;
    myButton.Margin = new Thickness(20, 20, 0, 0);
    myButton.HorizontalAlignment = Windows.UI.Xaml.HorizontalAlignment
    myButton.VerticalAlignment = Windows.UI.Xaml.VerticalAlignment.Top

    myButton.Background = new SolidColorBrush(Windows.UI.Colors.Red);
    myButton.Click += ClickMeButton_Click;

    LayoutGrid.Children.Add(myButton);
    */
}
```

Selecting the Button with my mouse cursor, I look at the Properties window and ensure that the Name is set to ClickMeButton. That lets me know that I'm in the right context for the Properties window. I want to set that Background property again using the Property dialog here, and I want to change the Background property.

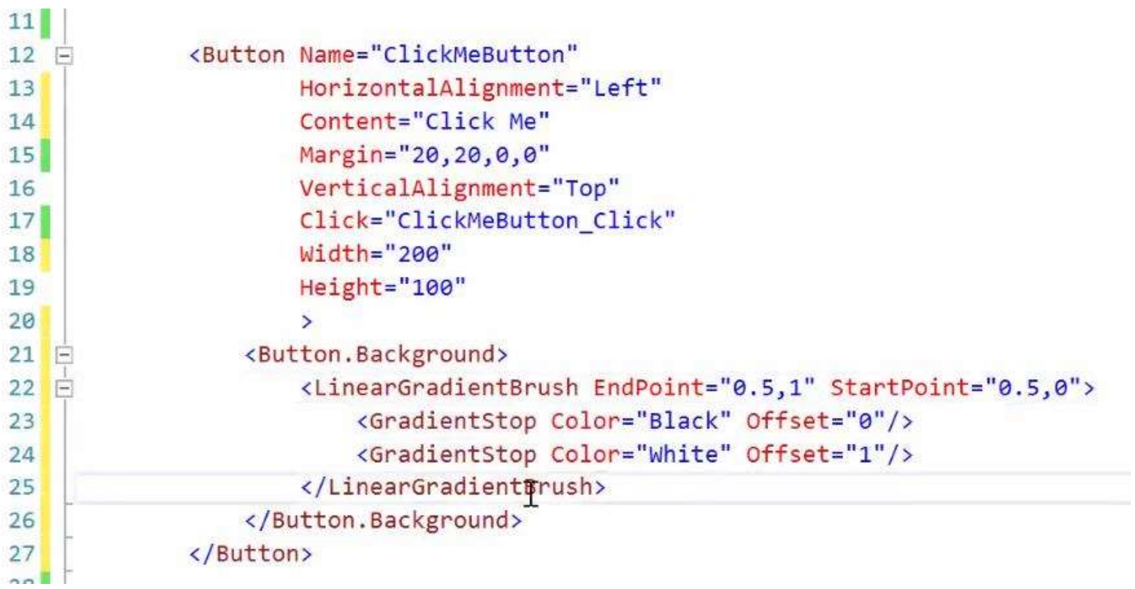


Now since I removed the XAML, the Background property is not being set at all. I will change from a SolidColorBrush that to a GradientBrush to use a gradient.



By default, the gradient starts out as black and then it slowly fades into the color white.

Notice the XAML that was generated:



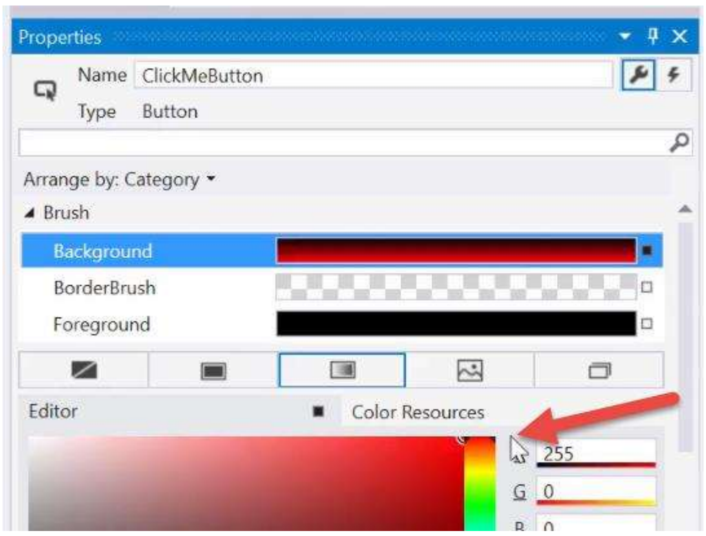
```
        <Button Name="ClickMeButton"
                HorizontalAlignment="Left"
                Content="Click Me"
                Margin="20,20,0,0"
                VerticalAlignment="Top"
                Click="ClickMeButton_Click"
                Width="200"
                Height="100"
                >
            <Button.Background>
                <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                    <GradientStop Color="Black" Offset="0"/>
                    <GradientStop Color="White" Offset="1"/>
                </LinearGradientBrush>
            </Button.Background>
        </Button>
```

Quite a bit of XAML has been added to the project between the opening and closing Button tags. Here, the Button.Background property is set to an instance of LinearGradientBrush.

So whenever you see this type of syntax -- this type of element embedded inside of another element -- this is called a Property Element Syntax. In other words, there's an Object.Property and XAML is added inside of the Property Element Syntax that will define the values for this Complex Property.

You may wonder what is a LinearGradientBrush? First whenever you see the term "brush" think "paint brush"; you're just thinking about color or colors that will be painted on an object. In this case, we're looking at a paintbrush that can paint a color starting at the top of the wall, at black, and by the time we paint the bottom of the wall, it'll be white.

However, I want to modify the colors. Making sure that I'm working with the Button control, I go to the Property window. I want it to change the color White to the color Red. I select the little circle in the Editor panel currently in the upper left hand corner and I drag it all the way to the right hand side.



Properties

Name ClickMeButton

Type Button

Arrange by: Category

Brush

Background

BorderBrush

Foreground

Editor

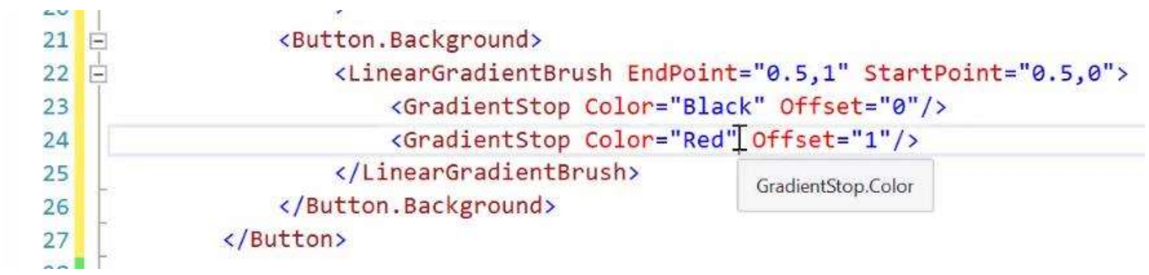
Color Resources

R 255

G 0

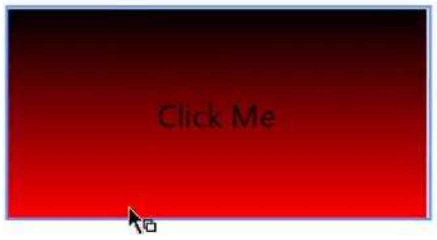
B 0

Notice that it changed the GradientStop from White to Red.



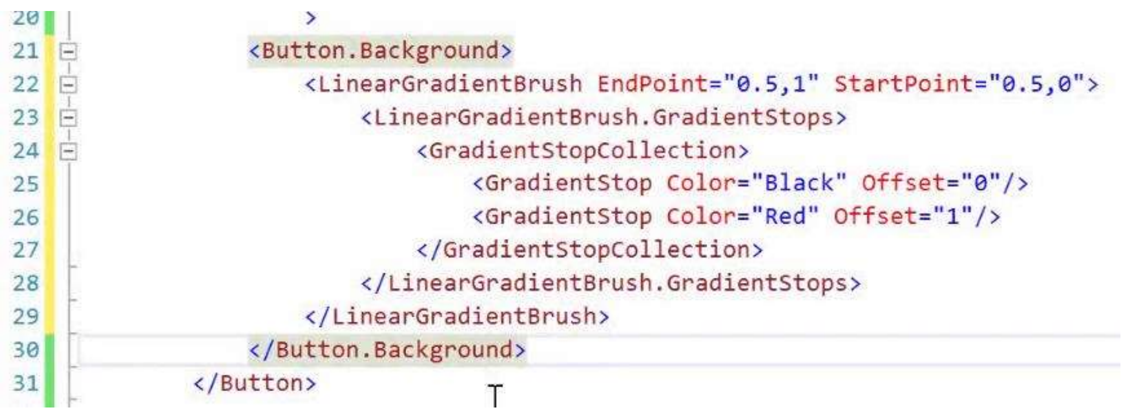
```
        <Button.Background>
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                <GradientStop Color="Black" Offset="0"/>
                <GradientStop Color="Red" Offset="1"/>
            </LinearGradientBrush>
        </Button.Background>
    </Button>
```

Let's save that and take a look at what it looks like here in the designer, and you can see how that's represented. At the top of the Button, it's black, the bottom of the button is red, great.



Note: You probably never, ever want to do this because it just doesn't follow the same aesthetic as the rest of the applications that your users are going to see in Windows 10, but let's pretend for now that you want to express your individuality or there is some branding that you want to do that your company is known for and you need that gradient.

In the XAML editor, if you do want to define a LinearGradientBrush, you have to supply quite a bit of information here. You have to not only give it the colors that you want to use, you also have to give it a collection of GradientStops and their Offsets, where one Color starts and the other Color stops. While this does look like a lot of additional XAML just to represent a Color, the code snippet here that I have highlighted is actually shortened automatically by Visual Studio. Let me take just a moment here and type out what the full XAML should be if Visual Studio didn't try to compact it for us



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

I've added a couple lines of code here in Line 23 and 24, and then the closing tags in Lines number 28 and 29. I'm setting the Button's Background to a new instance of the LinearGradientBrush class. The LinearGradientBrush class has a property called GradientStops. The GradientStops property is of type GradientStopCollection, so we create a new instance of GradientStopCollection and add two instances of the GradientStop object to that collection.

So, as you can see, that we were able to omit lines 23, 24, as well as 27, or rather 28, or yeah, 27 and 28, and we were able to omit this, or actually, Visual Studio, when it generated the XAML for us, was able to omit it because it wanted to make the XAML more concise and more compact, and it's made possible by an intelligent XAML parser.

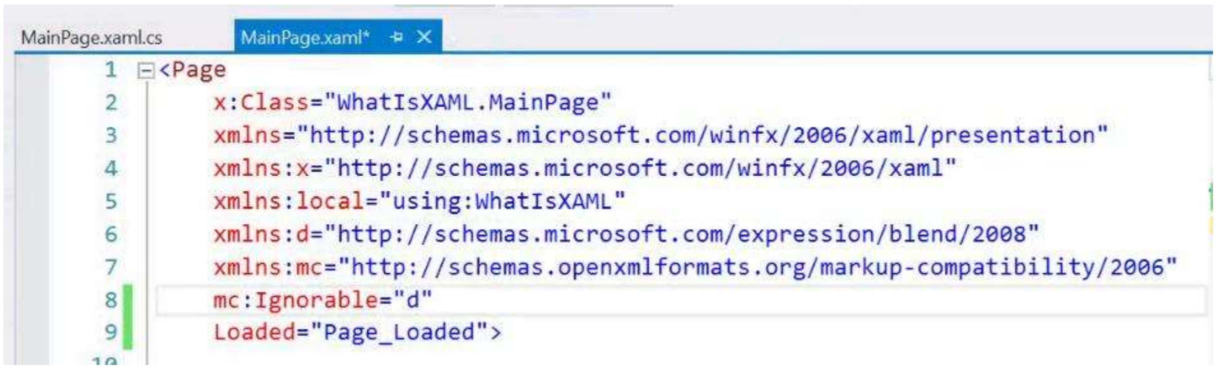
We talked about Default Properties at the very outset of this lesson. The GradientStops property is the Default Property for the LinearGradientBrush. The GradientStops property is of type GradientStopCollection, so the only thing that we can put in a GradientStopCollection are instances of GradientStop. So since we just put two GradientStop objects inside of a LinearGradientBrush, it knows that we're dealing with the Default Property and that Default Property is of type GradientStopCollection, so we don't even need to put that in there. That's already implied. We can supply the two GradientStops and it can infer the rest that it needs from its context.

So the moral of the story is that the XAML parser is intelligent, and it doesn't require us to include redundant code that it can infer from the context. As long as it has enough information to create that object graph correctly, it'll do it and we don't have to give it any more than it needs. And furthermore, Visual Studio will emit concise code if we use the Properties window or other tooling support inside of Visual Studio.

So just to recap this lesson, we talked about a number of different things again that all fit together. We talked about Default Properties, we talked about Complex Properties, then we talked about the Property Element Syntax like we saw here, like we saw, we've actually removed some of those lines of code between Line 22 and Line 25, and we also talked about how an intelligent XAML parser allowed us to remove those lines of code because it allows us to keep our XAML compact by inferring from context what inner elements should be used for. And finally, whenever possible, Visual Studio will generate concise XAML for us.

UWP-007 - Understanding XAML Schemas and Namespace Declarations

Up to now we've avoided all of this ugly code here at the very top of our MainPage.xaml.



```
1  <Page
2      x:Class="WhatIsXAML.MainPage"
3      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5      xmlns:local="using:WhatIsXAML"
6      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
7      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
8      mc:Ignorable="d"
9      Loaded="Page_Loaded">
```

A few lessons ago I said that XAML is actually just a flavor of XML, or rather that XAML uses XML to implement its syntax. I also said that in order to use XML (or at least use it properly) you have to create schemas. And a schema is like a contract that both the producer of the XML and the consumer of the XML agree upon so that they can work together.

So if that's the case, then where do we find the schema for XAML, for the page that we're currently working on? Well, actually, you might have guessed it, that's lines number three through seven, specifically, here at the very top of our file.

There are actually five schemas that the MainPage.xaml adheres to and it promises that it will follow the rules set forth in those schemas. Furthermore, it associates each schema with a namespace. A XAML namespace is just like we would use in a C# namespace.

For example, the very first schema is defined in the namespace associated with the letter "x", and that adheres to a schema defined here at http://schemas.microsoft.com/winfx/2006/xaml.

So when we prefix some XAML in our code with a namespace, we're saying that the rules for this particular XAML is specified by the associated schema.

That also means that everything else in this document, everything that doesn't have any prefix in front of it, whether it be the "x", the "local", the "d", the "mc", and so on, that means it adheres to this default namespace that's defined at the very top: http://schemas.microsoft.com/winfx/2006/xaml/presentation

You may wonder if you can see the schema if you were to open up that URL (above) into a web browser. However, you'll get a 404 error "Page Not Found". Schemas are not really defined at location in the sense that you can go out and you can look at the schemas online. It's more of just a unique name, just like namespaces are unique names, just intended to disambiguate. When Microsoft implemented Visual Studio and Blend and the XAML parser and compiler, they adhered to these "unwritten" rules -- or they might be written somewhere, but we can't get to them, at least not from this particular URL. In fact, it's not really even a URL, it's more of a URI, a Uniform Resource Indicator that's used as a namespace, in a sense, to define schemas that we can use in our document.

The intricacies of this may be confusing, however there is really just one main takeaway in this lesson: everything in our XAML Page follows a namespace, or rather a schema, an XML schema that's defined in one of these URIs.

You might wonder what each of these schemas are used for. For example, what's really the difference between this topmost schema and the second schema? The difference is really subtle, but essentially the top schema defines all of the UI elements, so the Grid, the Button, and all their attributes whereas the second schema is for the general rules for XAML.

Next up you see that we have a local namespace. This is actually not a schema, per se. It's actually a namespace just like we would use a using statement in C# -- so that we can reference classes without their full namespace hierarchy. So say for example I create a Car class in my application. I would be able to reference it inside of my XAML because as long as I prefix that with local:car, I'd be able to reference it here in my XAML document.

The last two are used by the Designer tools of Visual Studio and Blend. Actually you can see that the schema's URI actually uses: expression/blend/2008.

However, if you look at this little line number 8, it said that the mc: namespace prefix is ignorable. So at runtime, just ignore this namespace - we don't need it. The only reason why this ignorable attribute exists is because of its definition at the URI http://schemas.openxmlformats.org/markup-compatibility/2006.

Now that you understand what these do, you can completely forget this. They have no actionable result until we get a little bit deeper into our software development careers. We may have to use these namespace prefixes, especially the local one, and we may have to create one when we get into working with data, when we get to working with custom classes that we've created that represent data. But that's later. Just know that you should never ever modify this code here at the very top of a XAML file unless you have a really good reason to do so. And you almost never will.

The moral of the story is that, yes, XAML is XML. These are the schemas that it adheres to. Don't make any changes to this. But it's what makes everything work together seamlessly between Visual Studio and the parser and the compiler.