

UWP-028 - XAML Styles

As you already know Windows 10 has a very distinctive look and feel. I'm not just talking about things like the hamburger-style navigation, I'm also talking about the colors, the background/foreground colors, and the typography.

Here's a great resource that deals with the aesthetics of building Universal Windows Platform apps.

<http://dev.windows.com/en-us/design>

It talks about the aesthetics, approaches to navigation, etc. related to the design and user experience. There are probably other great resources available on www.microsoft.com as well, or www.windows.com. But even inside of those guidelines, there is room for some creative expression, for example, your company may need its branding, its colors, its font styles to be represented in the application. So, you'll not only want to know what the rules are, but then you'll also want to know why and when to break those rules.

In this lesson, I'll talk about the technical aspects of working with XAML to style your app. Primarily, I'll talk creating reusable styles and resources that can be shared between many different elements across a single Page, multiple Pages in your entire app, or even across multiple applications. Then in the next lesson, we will talk about utilizing the pre-built Themes that are available that will help enforce consistency across all apps on a given user's device.

Suppose that you have a particular color, or any property setting that you know that you're going to want to use throughout the entire page of your application, or the entire app. Perhaps a color or a property setting that you potentially may change occasionally, and you just want to make that change in one place and then have it reflected everywhere that particular style is being utilized, similar to Cascading Style Sheets. In that case, you'll create a resource and then bind to that resource as necessary.

So I've created a new project called XAMLResources and I'll create the simplest possible example of creating and using resources.

In the MainPage.xaml I add a Page.Resources section. Inside of that Page.Resources section I'll add a new Resource – a SolidColorBrush named “brush” and that will simply be used to create a solid color brush with the color brown. Again, an extremely simple example, just to illustrate the concept at first.

Now that I've created a resource in this manner, whenever I want to use that brush that I've defined anywhere in my application, I can reference it using a binding statement. In the following code example, I'll set the Foreground of my TextBlock to the resource using a binding statement.

```

10 <Page.Resources>
11     <ResourceDictionary>
12         <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
13     </ResourceDictionary>
14 </Page.Resources>
15
16 <StackPanel>
17     <TextBlock Text="Hello World"
18         Foreground="{StaticResource MyBrush}" />
19 </StackPanel>
20

```

You can see the open and close curly braces and the word `StaticResource`, and then I give it the name of the resource that I want to bind to, in this case, `MyBrush`. The operative part of this example is the binding syntax, which we see often for different purposes in XAML, binding to a resource takes the form of the binding expression syntax of the open and close curly brace and then the word `StaticResource` and then whatever the resource name is. The curly braces indicate binding, so that first word, `StaticResource`, defines the type of binding that we're engaging in. We're binding to a resource that's defined in XAML and it's only evaluated once as the application first starts. There are other kinds of binding expressions that allow you to continually evaluate the information that will be bound, or pre-compile the Binding statements that we'll talk about as well, and those will come into play at various points during the remainder of this series of lessons.

In this case, I've only created a `SolidColorBrush`, however we can use this pattern to create lots of different types of resources. For example, I'll create a Resource that's a string named "greeting" and I use that greeting resource in the Text attribute of my `TextBlock`.

```

9
10 <Page.Resources>
11     <ResourceDictionary>
12         <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
13         <x:String x:Key="greeting">Hello world</x:String>
14     </ResourceDictionary>
15 </Page.Resources>
16
17 <StackPanel>
18     <TextBlock Text="{StaticResource greeting}"
19         Foreground="{StaticResource MyBrush}" />
20 </StackPanel>
21

```

Beyond simple resources, Styles allow you to collect one or more settings that can be reused across a Page, or across an entire app, for a specific type of Control. Then, whenever you want to reuse that Style, you set a given Control's Style attribute and essentially bind it to the Style that you've defined. So virtually every Control, and even the Page itself has features of its appearance that can be customized, whether it be fonts, or colors or border thickness, or width, or height. These attributes can be set on an individual basis on each Control in your XAML, however that would make your XAML quite wordy and maybe even difficult to parse through visually. The other downside is, obviously, if you miss a setting and you're trying to keep things consistent across all intended uses of those settings, you might forget to set one of the properties.

A better approach would be to define a style once then reusing the style when you need to apply all of those property settings to a given control.

```

10 <Page.Resources>
11     <ResourceDictionary>
12         <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
13         <x:String x:Key="greeting">Hello world</x:String>
14
15         <Style TargetType="Button" x:Key="MyButtonStyle">
16             <Setter Property="Background" Value="Blue" />
17             <Setter Property="FontFamily" Value="Arial Black" />
18             <Setter Property="FontSize" Value="36" />
19         </Style>
20
21     </ResourceDictionary>
22 </Page.Resources>
23
24 <StackPanel>
25     <TextBlock Text="{StaticResource greeting}"
26         Foreground="{StaticResource MyBrush}" />
27     <Button Content="My Button Style Example"
28         Height="100"
29         Style="{StaticResource MyButtonStyle}" />
30 </StackPanel>
31

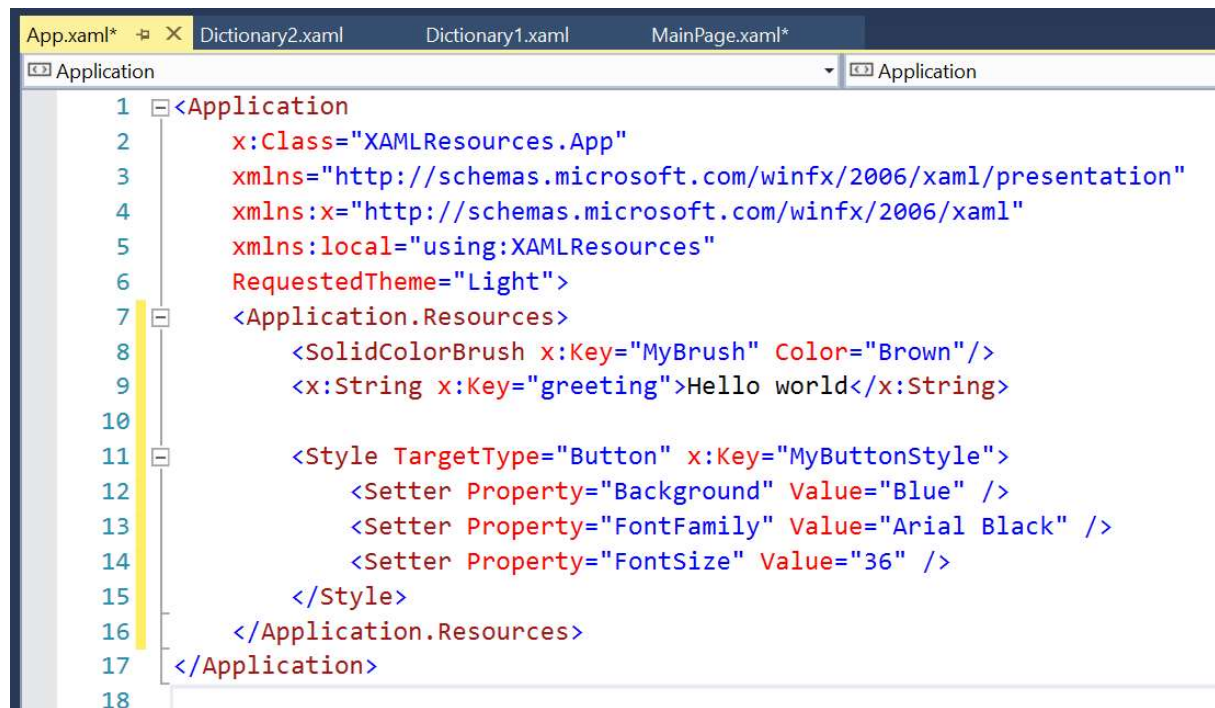
```

In lines 15 through 19 I create a new style called "MyButtonStyle" that targets the Button control. I have three setter elements that set the values of several properties. Later, in line 27 through 29, I apply that style using the binding syntax and the keyword StaticResource.

These are very simple examples to illustrate the idea, and as a result, it may not be readily apparent why you would actually want to utilize Styles and Resources. What's the value of this approach? As your application grows larger, as you add more Pages or many more Controls to your application, you might find this approach to be quite handy. It would require a lot of effort to reapply even those three Properties (from the previous example) every single time you have a Button Control in your application. Using Styles and Resources will help keep your XAML compact and concise. It will also be easier to

manage. In that regard, if you ever need to make a change, you change it in one spot, in the Style itself, and then it's applied everywhere.

In the previous example I created these as local Resources on the Page where they're being used, meaning that their scoped to just this MainPage.xaml. But what if I wanted to share these Resources across all of the Pages in my application? In that case, I would remove those Resources and the Style from this MainPage.xaml, and I'd put them into an application Resources Element on the App.xaml.



```
1 <Application
2     x:Class="XAMLResources.App"
3     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5     xmlns:local="using:XAMLResources"
6     RequestedTheme="Light">
7     <Application.Resources>
8         <SolidColorBrush x:Key="MyBrush" Color="Brown"/>
9         <x:String x:Key="greeting">Hello world</x:String>
10
11         <Style TargetType="Button" x:Key="MyButtonStyle">
12             <Setter Property="Background" Value="Blue" />
13             <Setter Property="FontFamily" Value="Arial Black" />
14             <Setter Property="FontSize" Value="36" />
15         </Style>
16     </Application.Resources>
17 </Application>
18
```

I've removed the resources and styles from the MainPage.xaml and added them to App.xaml in the Application.Resources section. However at runtime, nothing should change about the application whatsoever.

Before moving on, the Application.Resources section can be used to add Static Resources, Control templates, animations and even more features of XAML and UWP that we've not learned about up until now. Just keep in mind that if you to reuse something – almost anything -- across the entire application, it may make sense to put it into the Application.Resources section.

Finally, you can also create “merged resource dictionaries” that allow you to define your Resource dictionaries in multiple files and then combine them together at runtime. You may choose to put your style and resource definitions into separate files to help you manage the complexity, to reuse your Dictionary files in other projects, etc.

To add a new Resource Dictionary, I'll go to the Project menu > Add New Item then select the Resource Dictionary template. I'll leave the name as Dictionary1.xaml and click Add.

In the new file, I'll add the following:

```
1 <ResourceDictionary
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4     xmlns:local="using:XAMLResources">
5     <SolidColorBrush x:Key="brush" Color="Red"/>
6 </ResourceDictionary>
7
```

I'm creating a SolidColorBrush resource named "brush", setting its color to "Red".

Now we need to merge this ResourceDictionary into the Page's ResourceDictionary, and to do that, we will go to the MainPage's Page.Resources and create a ResourceDictionary element, and then a ResourceDictionary.MergedDictionaries attribute, and here we will create a ResourceDictionary and set the Source="Dictionary1.xaml".

```
10 <Page.Resources>
11     <ResourceDictionary>
12         <ResourceDictionary.MergedDictionaries>
13             <ResourceDictionary Source="Dictionary1.xaml" />
14         </ResourceDictionary.MergedDictionaries>
15     </ResourceDictionary>
16 </Page.Resources>
17
```

To see whether this works, we can change the resource we're binding to from "MyBrush" to just the "brush" resource:

```
18 <StackPanel>
19     <TextBlock Text="{StaticResource greeting}"
20               Foreground="{StaticResource brush}" />
21     <Button Content="My Button Style Example"
22            Height="100"
23            Style="{StaticResource MyButtonStyle}" />
24 </StackPanel>
25
```

With this change the TextBlock's text should be red.

We can actually merge multiple Dictionaries together. I'll go to the Project menu > Add New Item, select Resource Dictionary, keep the default name Dictionary2.xaml and click Add.

I'll remove the resource named greeting from App.xaml and paste it into Dictionary2, and then merge that into my MainPage.xaml.

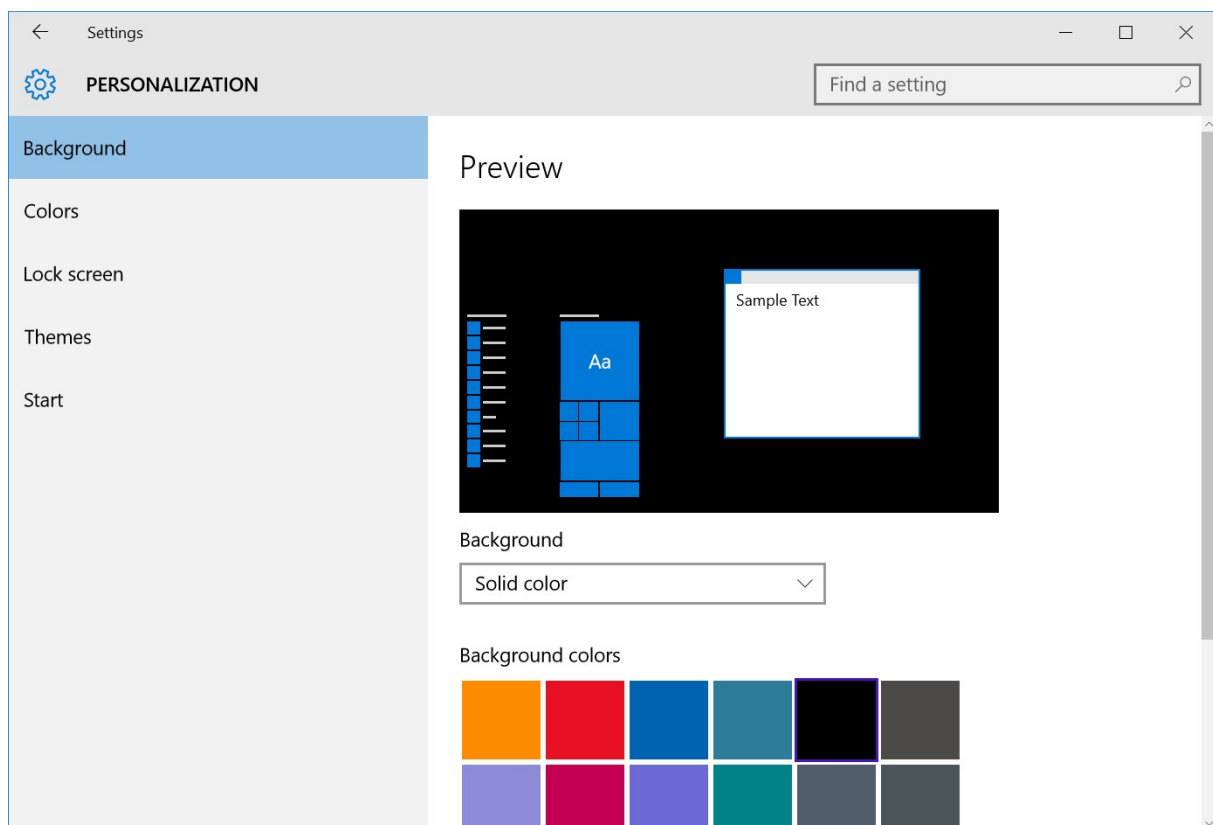
```
10 <Page.Resources>
11     <ResourceDictionary>
12         <ResourceDictionary.MergedDictionaries>
13             <ResourceDictionary Source="Dictionary1.xaml" />
14             <ResourceDictionary Source="Dictionary2.xaml" />
15         </ResourceDictionary.MergedDictionaries>
16     </ResourceDictionary>
17 </Page.Resources>
```

After making this change the TextBlock's text should remain "Hello World".

UWP-029 - XAML Themes

In the previous lesson we created Styles and Resources and used Binding expression to bind them to Controls. We used the keyword `StaticResource` which is a type of binding that only happens once when the app first starts. That's why it's a "StaticResource"; it won't change throughout the life of the app.

However, there are other examples of Binding statements that are not static. For example, there are themed resources which are similar to the `StaticResource`, but the resource lookup is evaluated when the theme changes. What's a theme? A theme is a collection of colors that are selected by the end user at an operating system level. In the Windows 10 Desktop you can pop open the Settings App, go to personalization, and here you can choose a background color and I think also an accent color, alright, and so in this case I've got black as my background and then this blue accent color.



You can also personalize the phone and the Xbox One as well by choosing a background and an accent, now admittedly, each have a different set of options, but in general the end user can personalize their colors in all the Windows 10 flavors.

As a developer you can choose to utilize these color selections in your app so that your app honors the user's choices. You're not required to do this, but you probably should unless you have a particular branding goal in mind. There are a set of styles that allow you to utilize those colors that were selected

by the user. Getting to the ResourceDictionary where those Styles are defined is a bit tricky and and I'll show you how to find it in the next lesson.

I've created a new project named "ThemeResources". On the MainPage.xaml, the default page template contains a Grid and that Grid utilizes a ThemeResource called ApplicationPageBackgroundThemeBrush.

```
9
10 <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
11
12 </Grid>
13 </Page>
14
```

If you put your mouse cursor just anywhere in that ThemeResource name and press the F12 key on your keyboard, it will open up a preview of a file named generic.xaml. If I hover my mouse cursor over the tab you can see the full path of that file on my copy of Windows is

Program Files (x86)\Windows Kits\10\DesignTime\CommonConfiguration\Neutral\UAP (version number)\Generic\generic.xaml Program Files (x86)\Windows Kits\10\DesignTime\CommonConfiguration\Neutral\UAP (version number)\Generic\generic.xaml

In the file itself you can see here that's it's defined a SolidColorBrush with that Key (ApplicationPageBackgroundThemeBrush) that's automatically being used in our Grid and that sets it to this color #FFFFFF. I recognize this to be the color white.

In this lesson I'm primarily going to refer to colors defined as ThemeResources, however there are many different types of Styles that are defined in this generic.xaml file, including Font Faces and weights and sizes and thicknesses and there are even default Styles, behavior and layout for all the basic XAML Controls. So if you ever wonder why something is styled a certain way or why it behaves in a certain way, this generic.xaml file is where you can look and you can use that technique that I used just a moment ago to get it. Generic.xaml is 14,000 lines of code so you'll probably want to use the search function of Visual Studio to find what you're looking for.

The way that you utilize those Styles is to reference them inside of your own Styles. Back in MainPage.xaml, I'll create Page.Resources element and a ResourceDictionary element. I may create a SolidColorBrush that I call AccentBrush. I set the color equal to a ThemeResource called SystemAccentColor. That SystemAccentColor will give me access to the accent colors that the user selects.

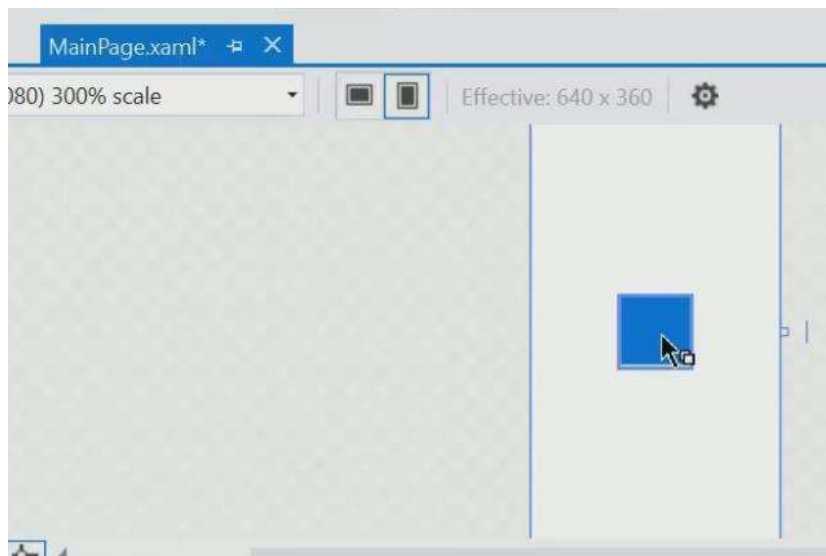
To use this resource, I'll add a Rectangle, set the Width="100", the Height="100" and the Fill equal to that StaticResource that I called AccentBrush.


```

9      <Page.Resources>
10         <SolidColorBrush x:Key="AccentBrush" Color="{ThemeResource SystemAccentColor}" />
11     </Page.Resources>
12
13     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
14         <Rectangle Width="100" Height="100" Fill="{StaticResource AccentBrush}" />
15     </Grid>
16 </Page>
17

```

A peek at the designer and you can see that it chose that same blue color that you see in the tab of Visual Studio because Visual Studio's using that that blue color for the selected tab.



I could skip the part where I create a resource first and just use that theme resource directly in the Fill property:

```

13     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
14         <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
15     </Grid>

```

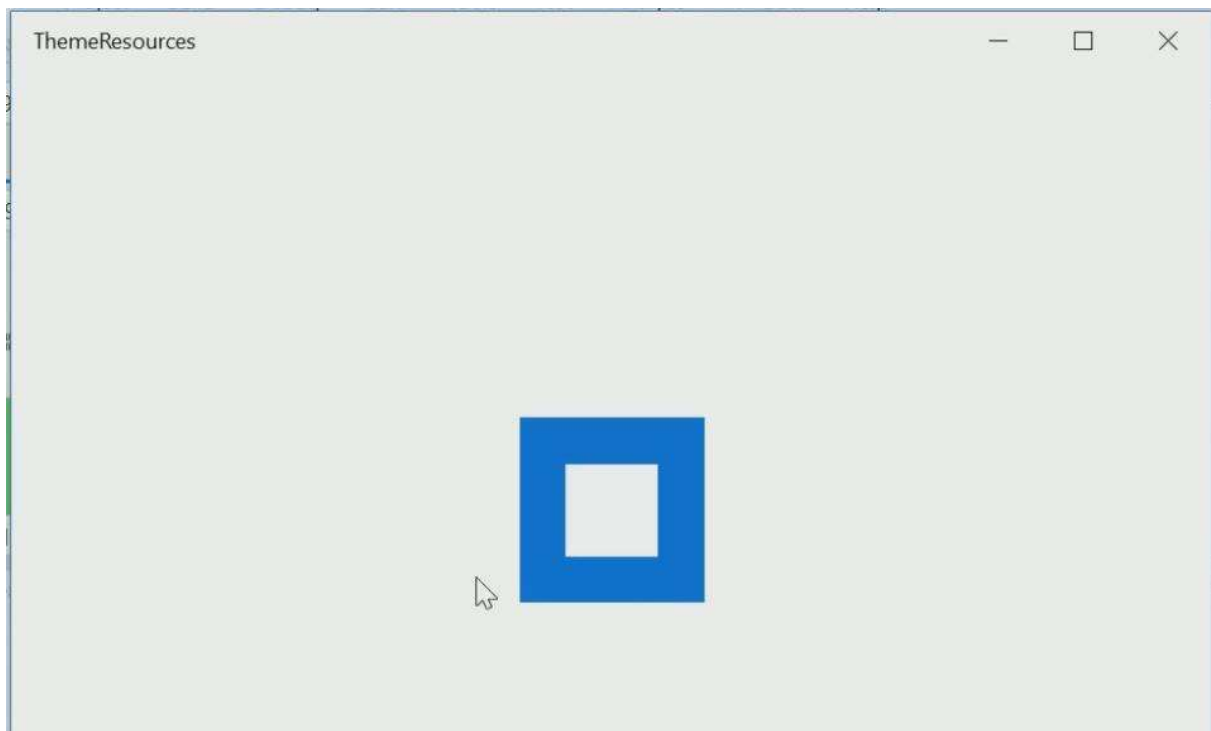
I can also access the color of the window like so:

```

13     <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
14         <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
15         <Rectangle Width="50" Height="50" Fill="{ThemeResource SystemColorWindowColor}" />
16     </Grid>
17 </Page>

```

SystemColorWindowColor gives us a light-colored area inside of the blue box.



Again, that matches that light color and the rest of the window. This will become important as we talk about the preferred Theme in just a moment.

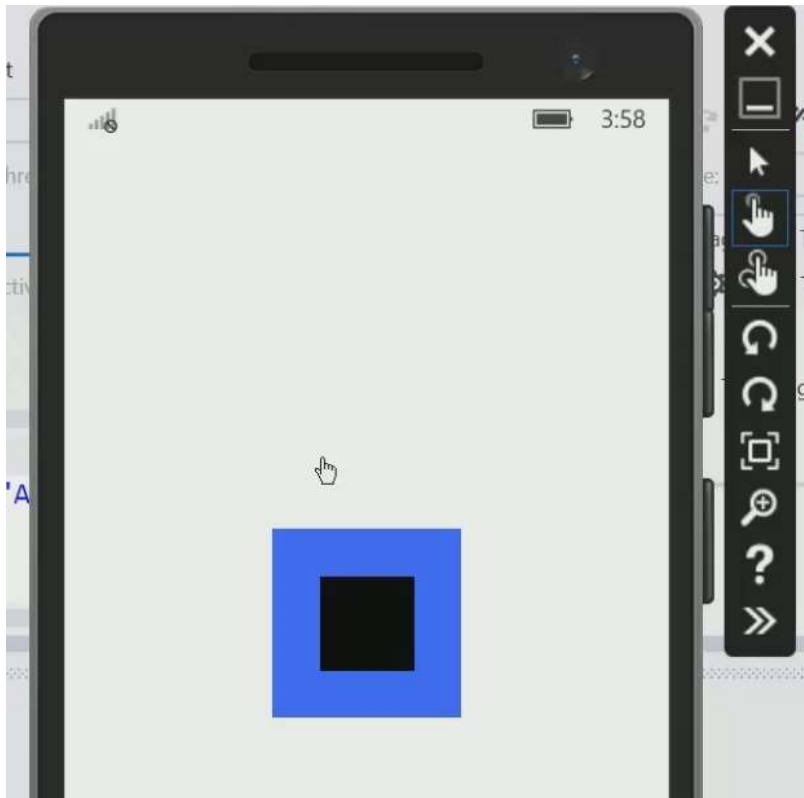
If you want an entire list of all of these Themes, take a look at “XAML ThemeResource” at:

bit.do/theme-resources

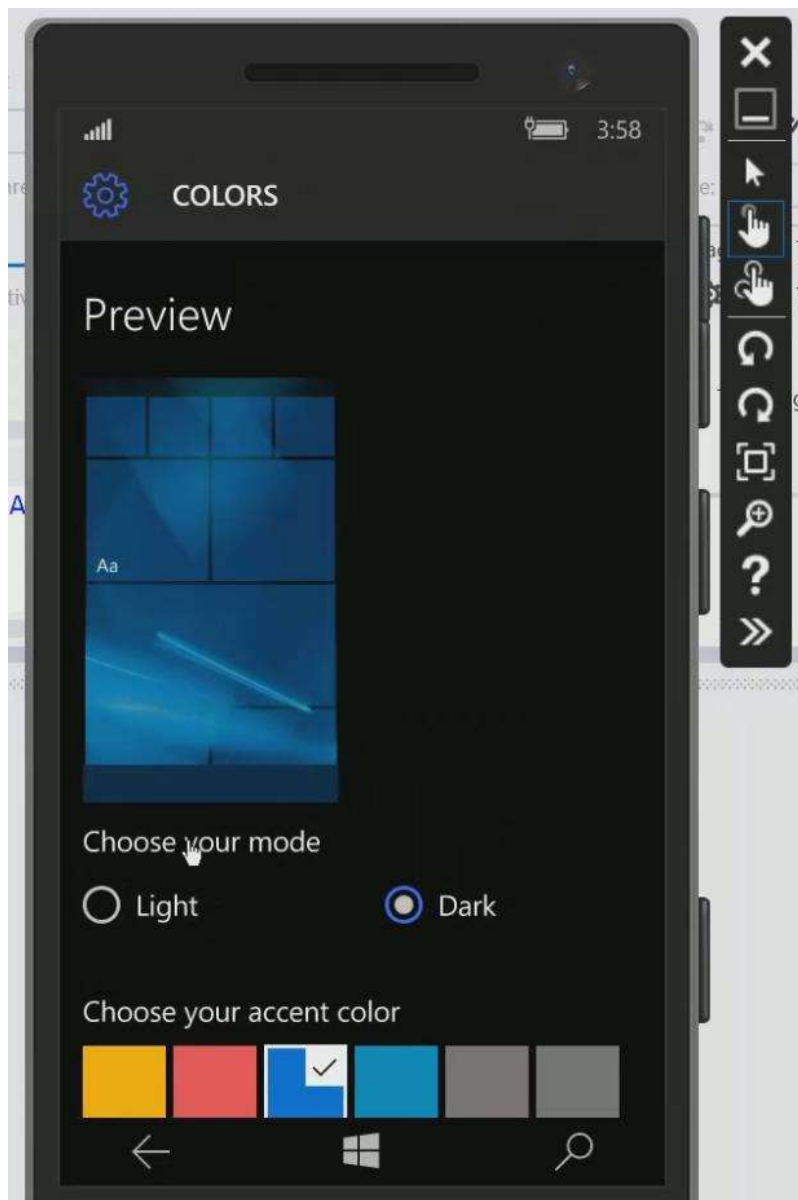
This contains long list of the different theme brushes, colors for buttons, text, etc.

On the Windows Phone this all works a little bit differently.

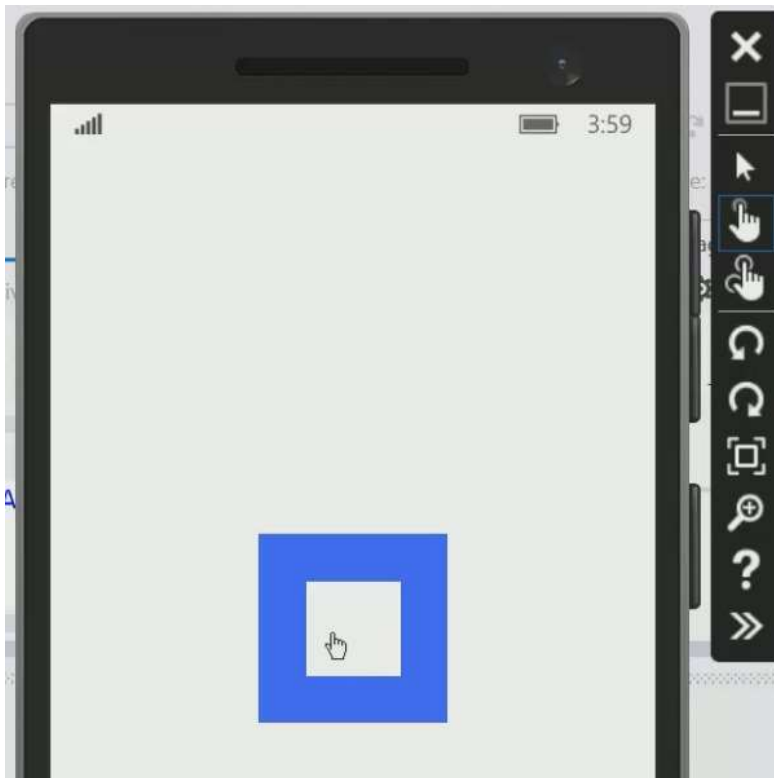
On the Windows 10 Phone, the user can select both a theme and an accent color, just like whenever you're building for the Windows 10 Desktop, you the developer get to decide whether to use those colors or not. On the phone, the user can select a “Dark Theme” or a “Light Theme” and you can see here in this particular case that I've chosen the Dark Theme. How do I know that? Because this `SystemColorWindowColor` is not white on the phone, but rather black.



In the simulator go down to Settings > Personalization > Colors. Here you can choose your mode, either Light or Dark. I'll change from Dark to Light ...

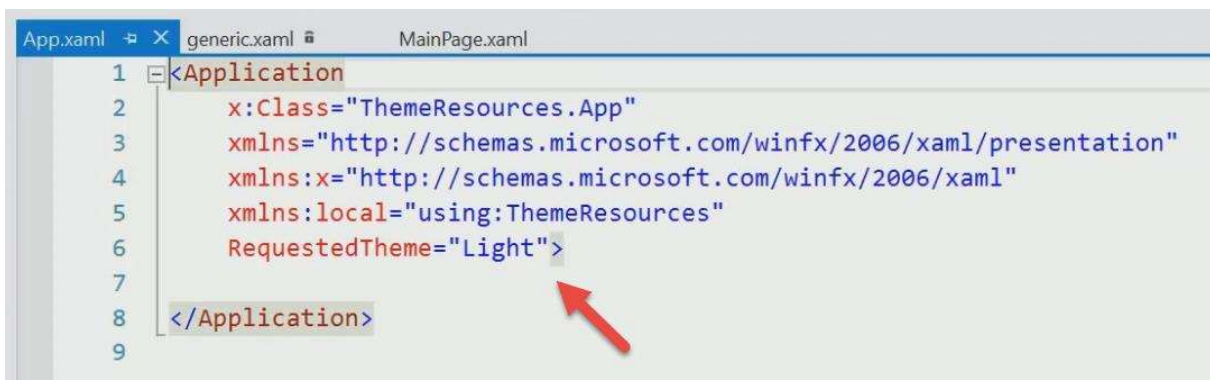


and go back to my ThemeResources app again and notice that that system color changed from Black to White.

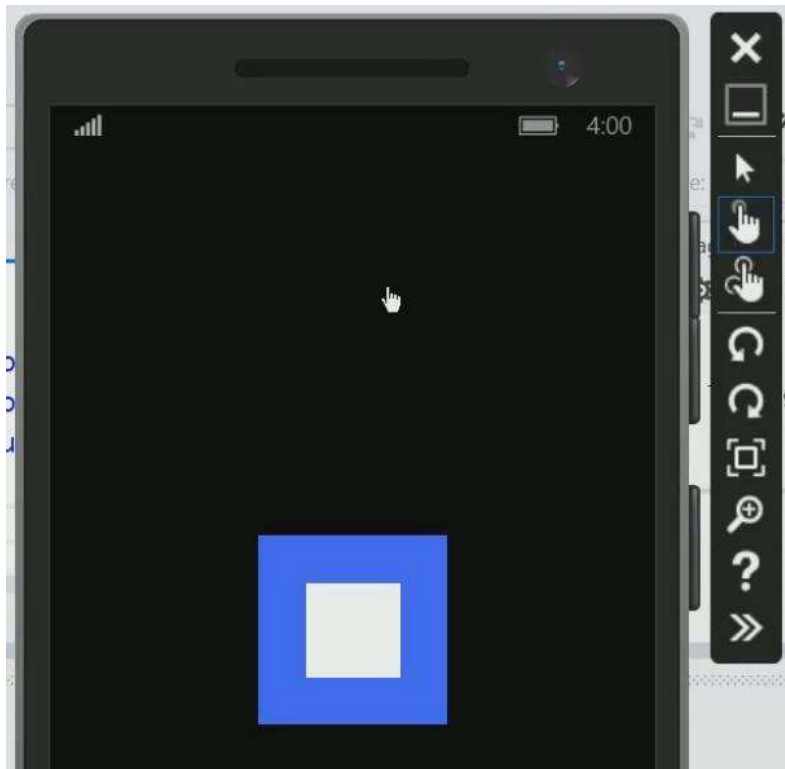


The intent of that Light and Dark Theme was really to change the “shell”, the colors that are used by the operating system, but again, as the developer you can choose to actually utilize those colors and each control is already Themed for both Light and Dark and new phones are typically configured with the black theme.

However, on an application level, you can see by default the RequestedTheme is Light which accounts for the light color window of our app.



If I change it to Dark then re-run the app, background color for the window is black



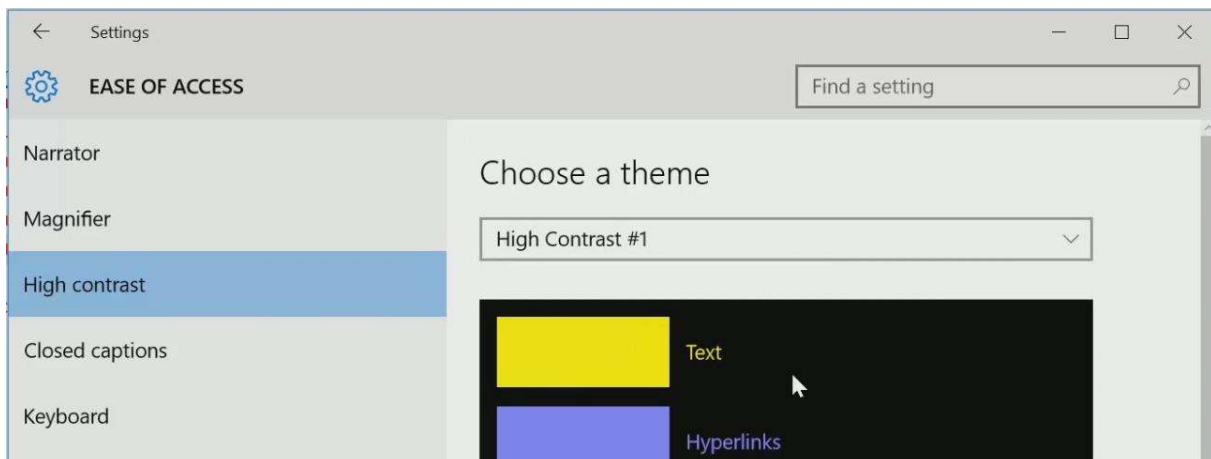
This works the same way inside of our Desktop apps as well.

I'm not going to show you how to override the Styles that are defined in this generic.xaml, but you can read about it in that help article titled "XAML ThemeResources".

Finally, a moment ago I said that you could request a Theme and I wanted to emphasize the fact that it's merely a request and even the property says RequestedTheme. If the user is using one of the high-contrast Themes on the desktop, it's going to override the RequestedTheme as well as any Styles that you create for your Control.

High-contrast Themes are for accessibility -- for people that are vision impaired -- and that takes precedence over any aesthetic that you might want to use for your app.

In Settings I'll search for "High Contrast". That setting is part of the Ease of Access settings.



In the Ease of Access section I'll choose "High contrast", then choose "High Contrast #1", then select the "Apply" button. The system does a quick log-out / log-in and now you can see that the Settings App looks completely different, Visual Studio looks different, when we run our app, this time we get the high contrast theme colors.



ThemedResources should be leveraged to keep your app looking like they belong on the user's desktop or device, so you should always resist the urge to use custom colors or fonts and things of that nature unless you have a really good reason to do so.

One of the good reasons would be for branding purposes for your company.

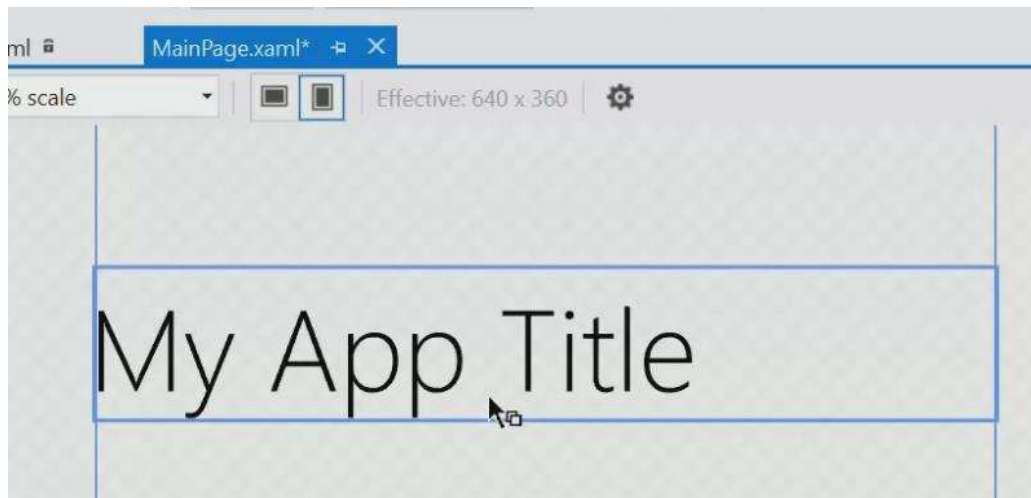
In addition to ThemeResources, there are built in Styles that are available to your app defined in generic.xaml.

First, I'll switch to use a StackPanel for layout, then I'll going to add a TextBlock and set the Text="My App Title". I'll bind the Style to "StaticResource HeaderTextBlockStyle".

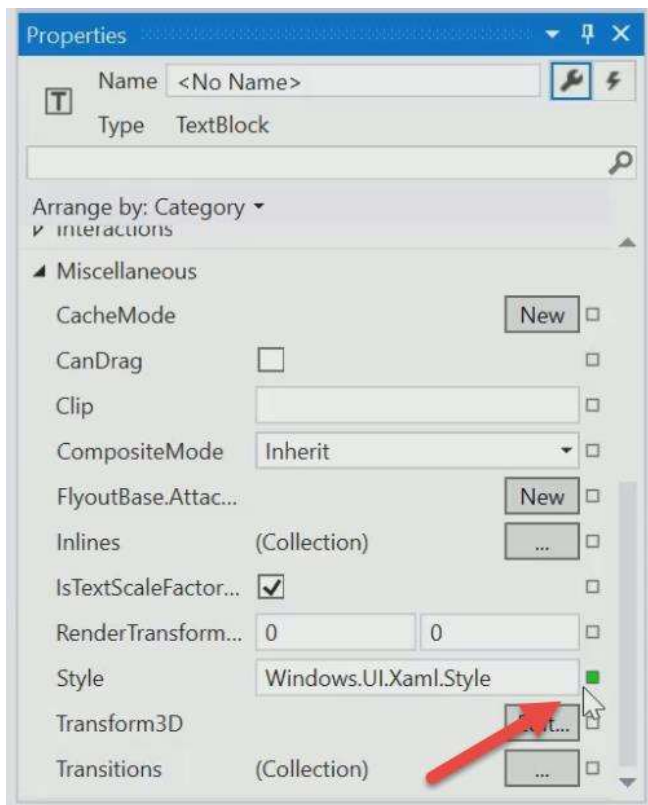
```
13 <StackPanel>
14     <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
15     <Rectangle Width="50" Height="50" Fill="{ThemeResource SystemColorWindowColor}" />
16     <TextBlock Text="My App Title" Style="{StaticResource CaptionTextBlockStyle}" />
17 </StackPanel>
18 </Page>
19
```



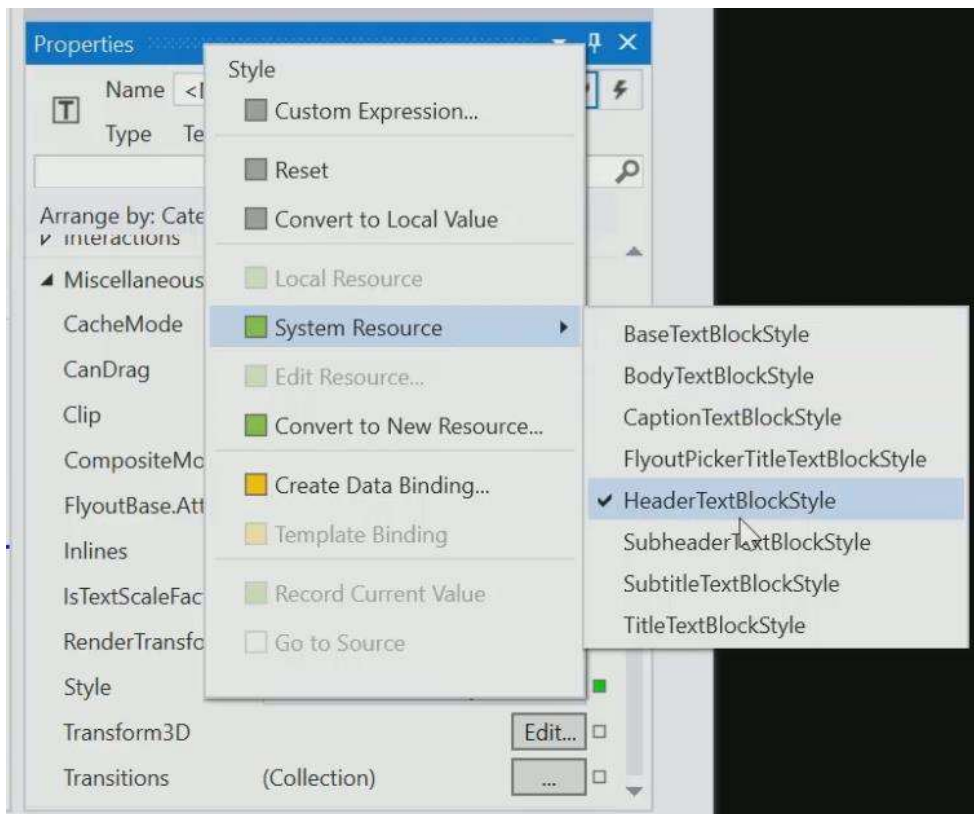
This style utilizes one of the Segoe fonts used often in applications.



If you ever are wondering where you can actually find these Styles, make sure the given control you're styling is selected then go to the Properties window, Miscellaneous section, and focus on the Style property:



The value of Style is set to Windows.UI.Xaml.Style. If I click this the green square, you can see that I can choose from a number of options. For this discussion, we'll expand System Resources which displays all of the available styles defined as System Resources.



I'll choose "CaptionTextBlockStyle" which is defined using a much smaller font.

Back in my XAML, If I put my mouse cursor on the word "CaptionTextBlockStyle" and hit F12, you'll notice that it brings us to line 12,400 in the generic.xaml file.



If you scroll over to the right, you'll see that this Style has a BasedOn Attribute. This particular Style builds on something called BaseTextBlockStyle. It's based on it, but it adds a couple of modifications to it. This works a lot like Cascading Style Sheets where you take a Style and you keep building on it and adding changes to it to create new styles.

If I search for the “BaseTextBlockStyle” I can find that it's not based on anything else.

```
12400 <!-- FRAMEWORK STYLES -->
12401
12402 <Style x:Key="BaseTextBlockStyle" TargetType="TextBlock">
12403     <Setter Property="FontFamily" Value="Segoe UI"/>
12404     <Setter Property="FontWeight" Value="SemiBold"/>
12405     <Setter Property="FontSize" Value="15"/>
12406     <Setter Property="TextTrimming" Value="None"/>
12407     <Setter Property="TextWrapping" Value="Wrap"/>
12408     <Setter Property="LineStackingStrategy" Value="MaxHeight"/>
12409     <Setter Property="TextLineBounds" Value="Full"/>
12410 </Style>
12411
```

These style properties can be overridden or added to in order to create new styles.

As developers we can utilize the BasedOn attribute in our own Styles, again, to give us that Cascading Style Sheet approach to Styling the app.

UWP-030 - Cheat Sheet Review: Controls, ScrollViewer, Canvas, Shapes, Styles, Themes

UWP-025 - Common XAML Controls - Part 2

```
<TimePicker ClockIdentifier="12HourClock" />
```

```
<CalendarDatePicker PlaceholderText="choose a date" />
```

```
<CalendarView SelectionMode="Multiple"
    SelectedDatesChanged="MyCalendarView_SelectedDatesChanged" />
```

```
private void MyCalendarView_SelectedDatesChanged(CalendarView sender,
CalendarViewSelectedDatesChangedEventArgs args)
{
    var selectedDates = sender.SelectedDates.Select(p => p.Date.Month.ToString() + "/" +
p.Date.Day.ToString()).ToArray();
    var values = string.Join(", ", selectedDates);
    CalendarViewResultTextBlock.Text = values;
}
```

```
<Button Content="Flyout">
    <Button.Flyout>
        <Flyout x:Name="MyFlyout">

            </Flyout>
        </Button.Flyout>
    </Button>
```

```
MyFlyout.Hide();
```

```
<Button Content="FlyoutMenu">
    <Button.Flyout>
        <MenuFlyout Placement="Bottom">
            <MenuFlyoutItem Text="Item 1" />
            <MenuFlyoutItem Text="Item 2" />
            <MenuFlyoutSeparator />
```



```

<MenuFlyoutSubItem Text="Item 3">
  <MenuFlyoutItem Text="Item 4" />
  <MenuFlyoutSubItem Text="Item 5">
    <MenuFlyoutItem Text="Item 6" />
    <MenuFlyoutItem Text="Item 7" />
  </MenuFlyoutSubItem>
</MenuFlyoutSubItem>
<MenuFlyoutSeparator />
<ToggleMenuFlyoutItem Text="Item 8" />
</MenuFlyout>
</Button.Flyout>
</Button>

<!-- You can apply this to anything ... ex. Image: -->
<!-- https://msdn.microsoft.com/en-us/library/windows/apps/xaml/dn308516.aspx -->

```

```

<AutoSuggestBox Name="MyAutoSuggestBox"
  QueryIcon="Find"
  PlaceholderText="Search"
  TextChanged="MyAutoSuggestBox_TextChanged" />

```

```

private string[] selectionItems = new string[] { "Ferdinand", "Frank", "Frida", "Nigel", "Tag", "Tanya",
"Tanner", "Todd" };

```

```

private void MyAutoSuggestBox_TextChanged(AutoSuggestBox sender,
AutoSuggestBoxTextChangedEventArgs args)
{
  var autoSuggestBox = (AutoSuggestBox)sender;
  var filtered = selectionItems.Where(p => p.StartsWith(autoSuggestBox.Text)).ToArray();
  autoSuggestBox.ItemsSource = filtered;
}

```

```

<Slider Maximum="100" Minimum="0" />

```

```

<ProgressBar Maximum="100" Value="{x:Bind MySlider.Value, Mode=OneWay}" />

```

```

<ProgressRing IsActive="True" />

```

UWP-026 - Working with the ScrollViewer

```
<ScrollViewer
    HorizontalScrollBarVisibility="Auto"
    VerticalScrollBarVisibility="Auto">

</ScrollViewer>
```

You can put anything inside of it, however, don't put it inside of a StackPanel!

UWP-027 - Canvas and Shapes

Canvas allows you to do absolute positioning via attached properties.

```
<Line X1="10"
      X2="200"
      Y1="10"
      Y2="10"
      Stroke="Black"
      Fill="Black"
      StrokeThickness="5"
      StrokeEndLineCap="Triangle" />
```

```
<Polyline Canvas.Left="150"
          Canvas.Top="0"
          Stroke="Black"
          StrokeThickness="5"
          Fill="Red"
          Points="50,25 0,100 100,100 50,25"
          StrokeLineJoin="Round"
          StrokeStartLineCap="Round"
          StrokeEndLineCap="Round" />
```

```
<Rectangle />
```

```
<Ellipse />
```

```
Canvas.ZIndex="100"
```

The higher the ZIndex, the higher in the stack it appears (covering what is below it).

UWP-028 - XAML Styles

<https://dev.windows.com/en-us/design>

```

<Page.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="Brown" />
  <Style TargetType="Button" x:Key="MyButtonStyle">
    <Setter Property="Background" Value="Blue" />
    <Setter Property="FontFamily" Value="Arial Black" />
    <Setter Property="FontSize" Value="36" />
  </Style>
</Page.Resources>

```

Binding: {StaticResource ResourceName}

```

<Button Content="OK" Style="{StaticResource MyButtonStyle}" />

```

Create Page or Application level resource dictionaries

```

<Application.Resources>
</Application.Resources>

```

Split up your styles into Resource Dictionary files:

```

<!-- Dictionary1.xaml -->
<ResourceDictionary>
  <SolidColorBrush x:Key="brush" Color="Red"/>
</ResourceDictionary>

```

```

<Page>
  <Page.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Dictionary1.xaml"/>
        <ResourceDictionary Source="Dictionary2.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Page.Resources>
  <TextBlock Foreground="{StaticResource SomeStyle}" Text="Hi" />
</Page>

```

UWP-29 - XAML Themes

<http://bit.do/theme-resources>

Put your mouse on a style, hit F12 to open generic.xaml

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Rectangle Width="100" Height="100" Fill="{ThemeResource SystemAccentColor}" />
  <Rectangle Width="50" Height="50" Fill="{ThemeResource SystemColorWindowColor}" />
</Grid>

<App RequestedTheme="Light">
```

High Contrast themes override styles.

Lots of different styles of system styles defined:

```
<TextBlock Text="page name" Style="{StaticResource HeaderTextBlockStyle}" />
```

Many styles defined in generic.xaml used BasedOn attribute ... and you can too!