FlashlightColor

FlashlightColors app will provide the user options to change the flashlight color on the screen based on predefined colors.

Requirements

- Visual Studio 2012
- Windows Phone 8 SDK
- Windows 8
- Flashlight Tutorial

Create a new Windows Phone 8 Project and set the name as `FlashlightColors` .****

Ensure that `Create directory for solution` is checked.

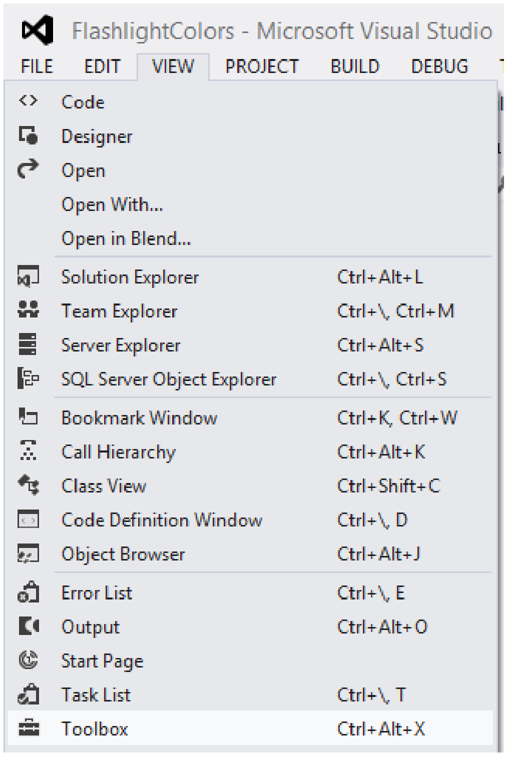
Leave the default value of `Windows Phone OS 8.0` .

Removing Unnecessary Elements

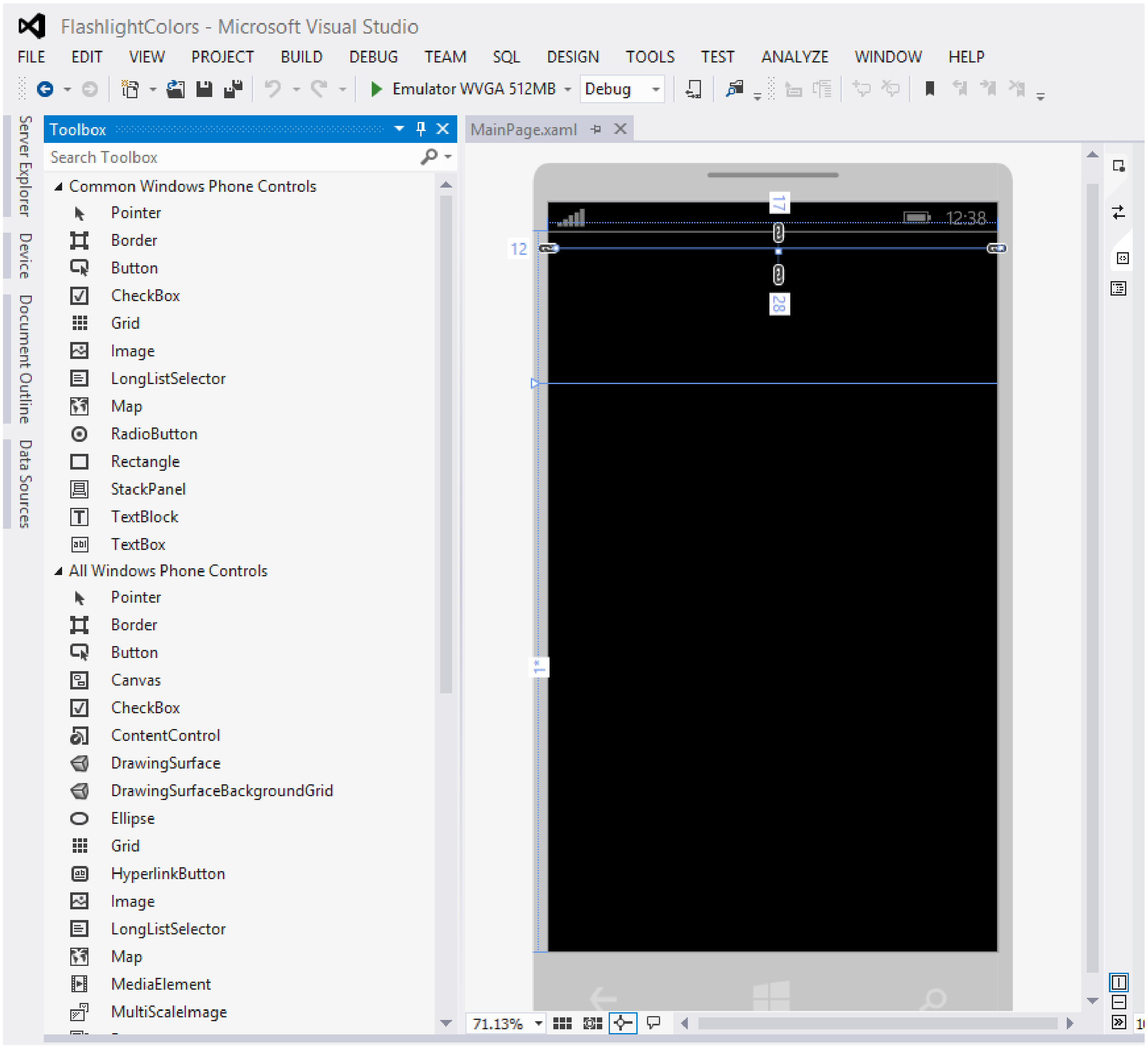
Remove the `<StackPanel>` element and all its children `<TextBLock>` elements.

Setup your project panes

After saving the project, the designer will be updated with what seems to be a blank screen. Now we will open up the `Toolbox` menu by selecting the `View` menu and then selecting `Toolbox` as shown below:


FlashlightColors - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG
Code
Designer
Open
Open With...
Open in Blend...
Solution Explorer Ctrl+Alt+L
Team Explorer Ctrl+\, Ctrl+M
Server Explorer Ctrl+Alt+S
SQL Server Object Explorer Ctrl+\, Ctrl+S
Bookmark Window Ctrl+K, Ctrl+W
Call Hierarchy Ctrl+Alt+K
Class View Ctrl+Shift+C
Code Definition Window Ctrl+\, D
Object Browser Ctrl+Alt+J
Error List Ctrl+\, E
Output Ctrl+Alt+O
Start Page
Task List Ctrl+\, T
Toolbox Ctrl+Alt+X

The `Toolbox` will appear and you can drag it to an area of the screen which is most convinient for you. I prefer to the left of the `Designer` view.

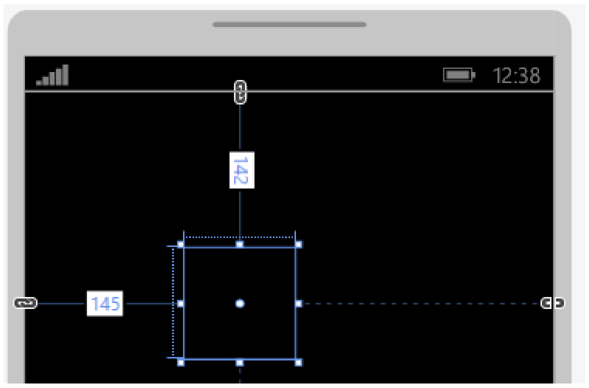


Likewise, do the same for `View` and then selecting `Properties` to display the `Properties` pane.

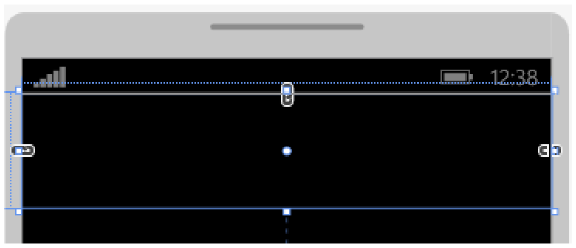
Set the UI Stage

In the `Toolbox` tab, expand the `Common Windows Phone Controls` tree.

Drag a `Grid` element onto the Designer canvas.



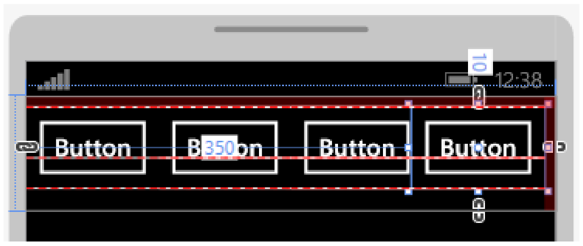
Reposition the `Grid` element on the canvas in the upper area. It will be about 100px in height which will be identified by the rulers and auto guides. The edges of the `Grid` element will snap to the canvas edges and the bottom edge of the rectangle grid will snap with the existing `<ContentPanel>` Grid element.



In the `Properties` panel, name this upper grid `ButtonPanel`.

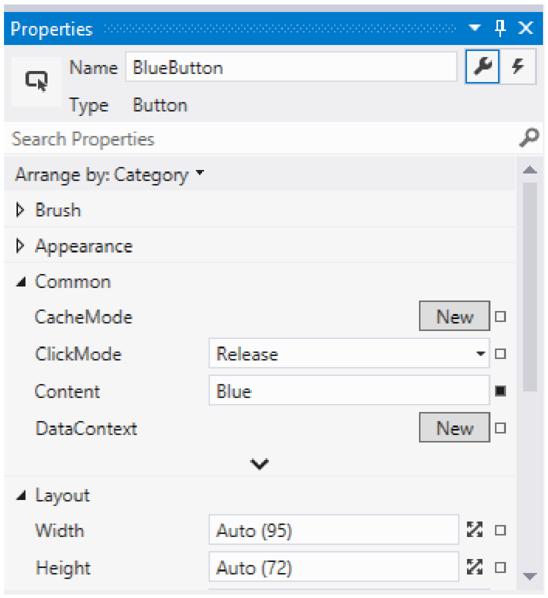
Now drag four separate `Button` controls onto the newly created `ButtonPanel` grid and position them left to right (horizontally).

When positoning the buttons, they will automatically try to snap in place and their outer margin will be auto calculated based on the grid that they are placed on. Their heights will be automatically determined so that does not need to be adjusted.



Button Attributes

Click a button on the Designer View and in the `Properties` pane, enter the value `Blue` for the `Content` attribute and `BlueButton` for the `Name` attribute.


Properties

Name BlueButton

Type Button

Search Properties

Arrange by: Category

Brush

Appearance

Common

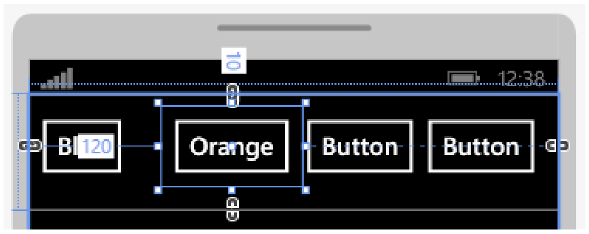
CacheMode	New
ClickMode	Release
Content	Blue
DataContext	New

Layout

Width	Auto (95)
Height	Auto (72)

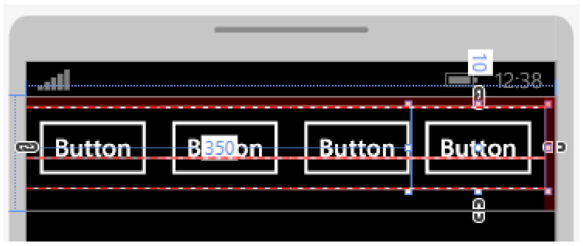
For the remaining three buttons, do the same simiarily for `Orange`, `Red`, `White`.

When renaming the button `Content` attributes, the widths of the buttons will auto adjust. You can easily snap them back so thaty they are evenly spaced apart.



Final Button Placements

Here is a preview of what your buttons can look like once they are aligned.



Here is a sample of what your MainPage.xaml would look like:

```
<Grid x:Name="ButtonPanel" HorizontalAlignment="Left" Height="103" Grid.RowSpan="2" VerticalAlignment="Top" Width="480">
    <Button x:Name="BlueButton" Content="Blue" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>
    <Button x:Name="OrangeButton" Content="Orange" HorizontalAlignment="Left" Margin="110,10,0,0" VerticalAlignment="Top"/>
    <Button x:Name="RedButton" Content="Red" HorizontalAlignment="Left" Margin="242,10,0,0" VerticalAlignment="Top"/>
    <Button x:Name="WhiteButton" Content="White" HorizontalAlignment="Left" Margin="337,10,0,0" VerticalAlignment="Top"/>
</Grid>
```

Flashlight Area

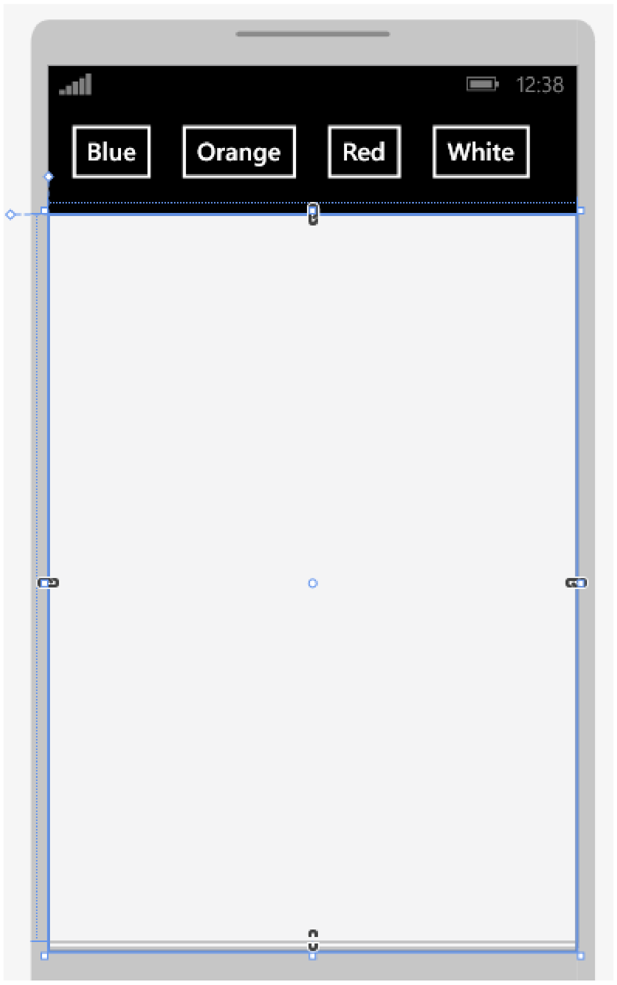
The lower grid area is called `ContentPanel`. Within this region, we will place the canvas which will serve as the Flashlight "light source".

Drag a new `Rectangle` element from the `Common Windows Phone Elements` group and place it on the lower Grid area. Maximize the size of the Rectangle grid so that it takes the entire size of the grid it is placed in and also name the rectangle shape `Flashlight` in the properties pane.

Your `ContentPanel` grid xml will look similar to the following:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="0,103,0,5">
    <Rectangle x:Name="Flashlight" Fill="#FFF4F4F5" Height="670" Stroke="Black" VerticalAlignment="Top" Margin="0,0,0,-10"/>
</Grid>
```

Your designer preview will look similar to this:



Button Events

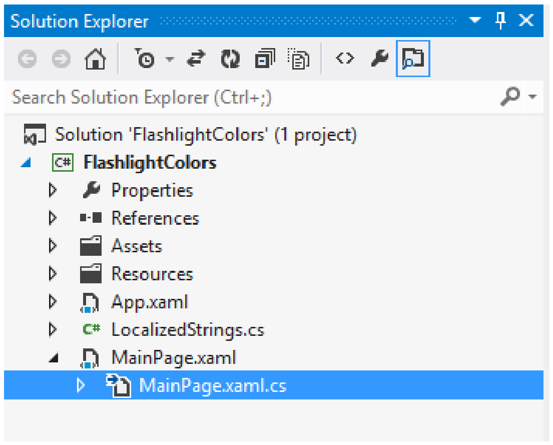
In Windows Phone, common elements such as Buttons can respond to user created events. In this case, we want to capture the event of the user clicking on each of the buttons. What result do we want to accomplish when the user clicks the button? We want the Rectangle to change to the appropriate color.

The default event behavior for buttons are calld `click events`. To create this behavior, double click the `Blue` button on the Designer UI. It will bring up to the `MainPage.xaml.cs` and auto generate this function:

```
        private void BlueButton_Click(object sender, RoutedEventArgs e)
        {

        }
```

The `MainPage.xaml.cs` file is grouped `MainPage.xaml` and is an extension of the XML document that generates the UI.

You can find this file within the `Solution Explorer` pane on the right hand side of the screen inside the `MainPage.xaml` file tree.



Return back to the `MainPage.xaml` file by doulble clicking it in the `Solution Explorer` pane. Perform the same double click on each of the buttons to have the rest of the click events generated.

You will notice that your `MainPage.xaml` file would have updated the four `<Button>` elements with a new attribute: `Click="BlueButton_Click"`. This tells the Button element to invoke the `BlueButton_Click` inside the associated C# file which is `MainPage.xaml.cs`.

```
<Grid x:Name="ButtonPanel" HorizontalAlignment="Left" Height="103" Grid.RowSpan="2" VerticalAlignment="Top" Width="480">
    <Button x:Name="BlueButton" Content="Blue" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" Click="
    <Button x:Name="OrangeButton" Content="Orange" HorizontalAlignment="Left" Margin="110,10,0,0" VerticalAlignment="Top" Cl
    <Button x:Name="RedButton" Content="Red" HorizontalAlignment="Left" Margin="242,10,0,0" VerticalAlignment="Top" Click="R
    <Button x:Name="WhiteButton" Content="White" HorizontalAlignment="Left" Margin="337,10,0,0" VerticalAlignment="Top" Clic
</Grid>
```

You should also have four methods generated within your C# file.

```
private void BlueButton_Click(object sender, RoutedEventArgs e)
{

}

private void OrangeButton_Click(object sender, RoutedEventArgs e)
{

}

private void RedButton_Click(object sender, RoutedEventArgs e)
{

}

private void WhiteButton_Click(object sender, RoutedEventArgs e)
{

}
```

Programming the Events

Now we have the events wired up with the associated buttons on the page. The next step is to execute code that will change the background fill of the rectangle we have on the main UI.

To do so we need to discover how to fill the Rectangle shape. Go to the MainPage.xaml file and hover your mouse over the `<Rectangle>` element. You will see that it points to the `System.Windows.Shapes.Rectangle` class. This is an easy way to figure out what namespace a certain class is contained in so that you can find the correct documentation online to utilizie the class.

The documentation for the class can be found at Rectangle Class. After a click glance, you can find the `Fill` property which is what we are interested in: Shape.Fill Property.

Scrolling through the class documentation, you will notice at the bottom a section called `More Code` and provides some quick examples how to accomplish common tasks using this class. The one we are interested in i called How to: Paint an Area with a Solid Color and the excerpt is found here:

> To paint an area with a solid color, you can use a predefined system brush, such as Red or Blue, or you can create a new SolidBrushColor and describe its Color using alpha, red, green, and blue values. In XAML, you may also paint an area with a solid color by using hexidecimal notation.

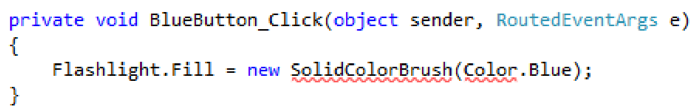
MSDN, *Microsoft Developer Network* provides a lot of resources, examples, and all the class documentation for the Windows Phone 8 and Windows 8 SDKs.

After reading through the documentation you will notice that you can simply create a new `SolidBrushColor` object and pass a predefinied `Colors` object as the constructor. This newly created object can be applied to the `Fill` property of our `Flashlight` Rectangle in our project.

Add the following to thew `BlueButton_Click` method.

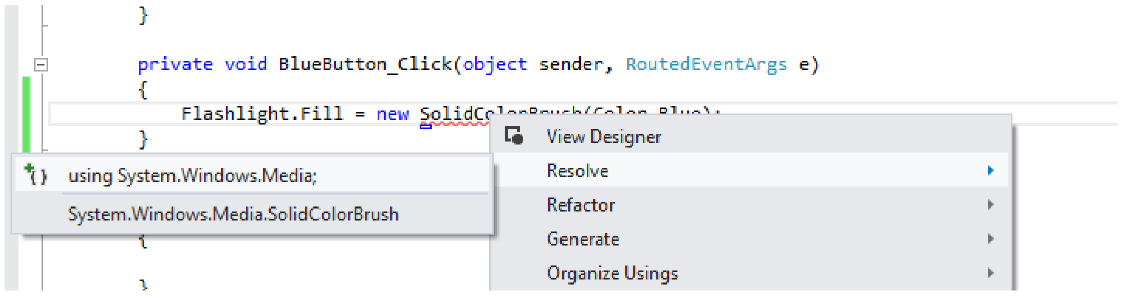
```
            Flashlight.Fill = new SolidColorBrush(Colors.Blue)
```

You will notice that `SolidColorBrush` as well as the `Colors` text has a red squiggly line underneath. This is Visual Studio's way of telling you that it is unable to find, or resolve those names. What we need to do is tell our `MainPage.xaml.cs` file to use resources from other packages in order to resolve those names.



```
private void BlueButton_Click(object sender, RoutedEventArgs e)
{
    Flashlight.Fill = new SolidColorBrush(Color.Blue);
}
```

Right click `SolidColorBrush`, in the flyout menu, select `Resolve` -> `using System.Windows.Media;`.



```
        }

        private void BlueButton_Click(object sender, RoutedEventArgs e)
        {
            Flashlight.Fill = new SolidColorBrush(Color.Blue);
        }
```

using System.Windows.Media;
System.Windows.Media.SolidColorBrush
View Designer
Resolve
Refactor
Generate
Organize Usings

You will notice that the red squiggly line has disappeared under both names because the `Colors` object could also be resolved under the same namespace.

Now do the same for the other three events using the appropate color from the `Colors` class.

```
        private void BlueButton_Click(object sender, RoutedEventArgs e)
        {
            Flashlight.Fill = new SolidColorBrush(Colors.Blue);
        }

        private void OrangeButton_Click(object sender, RoutedEventArgs e)
        {
            Flashlight.Fill = new SolidColorBrush(Colors.Orange);
        }

        private void RedButton_Click(object sender, RoutedEventArgs e)
        {
            Flashlight.Fill = new SolidColorBrush(Colors.Red);
        }
        private void WhiteButton_Click(object sender, RoutedEventArgs e)
        {
            Flashlight.Fill = new SolidColorBrush(Colors.White);
        }
```

Easily Build

Now at this point, we have completed our basic functionality for the Colored Flashlight app. Build and Deploy the application and try clicking the various buttons to see the colors changing on the flashlight canvas.

Refactoring Code

This part of the tutorial will go into how to refactor the code and coupling common events together.

The thought process behind this is that we have four buttons that fires on a specific event and does a simple action, change the rectangle color. The only changing factor is the color. Our objective is to have one event that multiple objects can respond to.

First what we will do is delete all the event methods except `BlueButton_Click` method.

We will now rename our single event method to `ColorButton_Click`.

Inspecting the method further we see two generic parameters: `object sender` and `RoutedEventArgs e`.

The basic way to explain this generic event is that `sender` is the object that invoked the event. This is where we can get any related objects. `e` contains any other special arguments in more complex events.

```
            var color = Colors.White;
            var button = (Button)sender;

            Flashlight.Fill = new SolidColorBrush(color);
```

What I have done is create a default color, white and type casted the sender object to a Button type. How do I know it is a button type? Because I only will be sending button objects to this event so it is a safe assumption to make.

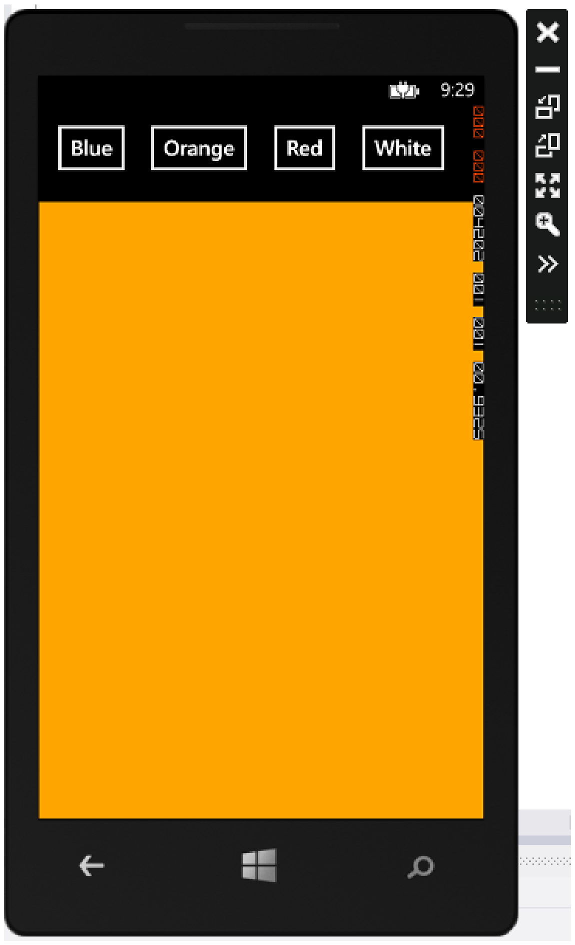
Now that the sender object has been casted to a `Button` type, we can access attributes derived from the xaml definition.

```
        private void ColorButton_Click(object sender, RoutedEventArgs e)
        {
            var color = Colors.White;
            var button = (Button)sender;

            switch (button.Name)
            {
                case "WhiteButton":
                    color = Colors.White;
                    break;

                case "BlueButton":
                    color = Colors.Blue;
                    break;

                case "RedButton":
                    color = Colors.Red;
                    break;

                case "OrangeButton":
                    color = Colors.Orange;
                    break;
            }

            Flashlight.Fill = new SolidColorBrush(color);
        }
```

The next step is to update the UI with the proper click events. Go back to the xaml code and update the `Click` events to the new method call.

```
    <Grid x:Name="ButtonPanel" HorizontalAlignment="Left" Height="103" Grid.RowSpan="2" VerticalAlignment="Top" Width="480">
        <Button x:Name="BlueButton" Content="Blue" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top" Click="
        <Button x:Name="OrangeButton" Content="Orange" HorizontalAlignment="Left" Margin="110,10,0,0" VerticalAlignment="Top" Cl
        <Button x:Name="RedButton" Content="Red" HorizontalAlignment="Left" Margin="242,10,0,0" VerticalAlignment="Top" Click="C
        <Button x:Name="WhiteButton" Content="White" HorizontalAlignment="Left" Margin="337,10,0,0" VerticalAlignment="Top" Clic
    </Grid>
```

Now build and see that your app has the same functionality, but by using one function, we can couple together events that have the similar expectation and outcome.



Last edited by JamieChung, a minute ago