Organizing your XAML

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Sharing is good

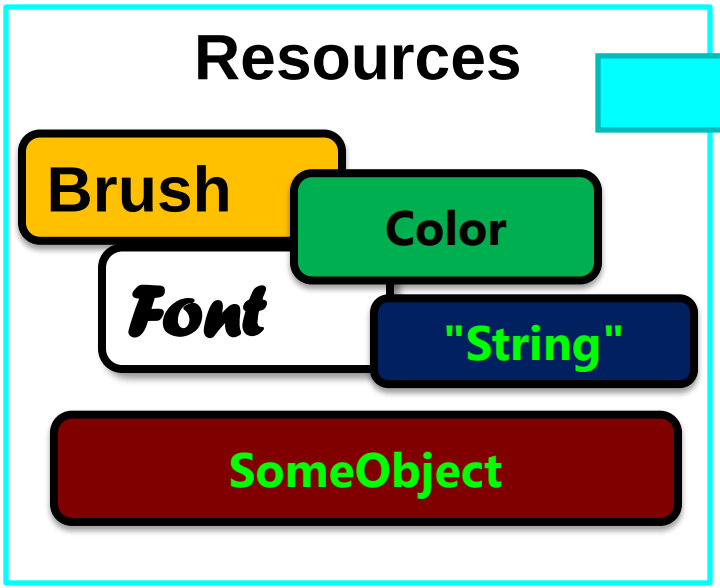
- Repeated XAML code is inefficient
  - poor use of memory
  - error-prone to change globally
  - difficult to keep consistent

```
<Window ...>
   <Grid>
     ...
     <Border Background="#FFd03933"> ... </Border>
     <Ellipse Fill="#FFd03933" ... />
     <TextBlock Foreground="#FFd03933" />
     <Button BorderBrush="#FFd03933" ... />
   ...
```

four different brushes are created here to paint the same color

Sharing objects through resources

- WPF extends the concept of "resource" to be *any object*
  - anything can be classified as a "resource"
  - allows objects to be easily shared throughout UI



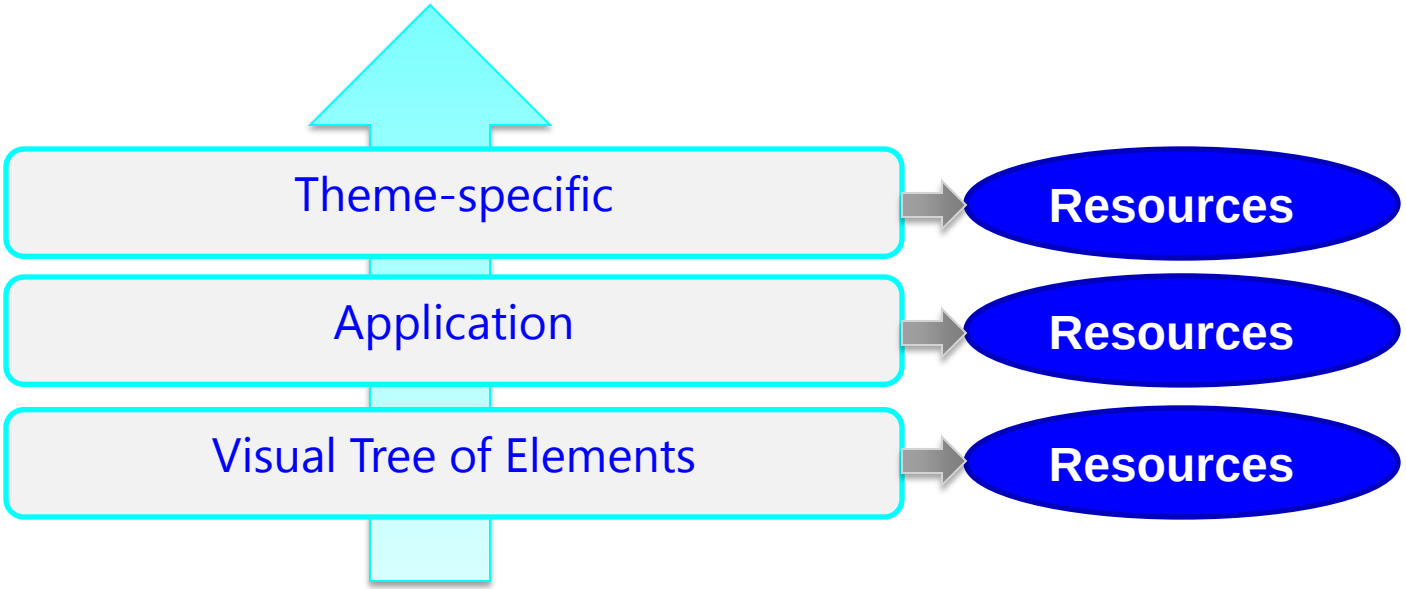

```
<Window FontFamily="... Font">

    <TextBlock
        Text="... String" />
    <Button
        Foreground="... Brush"
        Background="... Brush"
        Content="... SomeObject" />
</Window>
```

can place colors, brushes, fonts, strings and even full XAML blocks into resources

Defining Resources

- Resources can be defined at multiple levels
  - allows scoping of resources to most appropriate level



resources located by searching visual tree – just like routed events, starting at the element and moving outward

Adding new resources

- `FrameworkElement.Resources` holds assets
  - associates a single object with a unique key
  - can be added via XAML (preferred[1]) or code behind

```
<Window>
  <Window.Resources>
     <SolidColorBrush
        x:Key="redBrush"
        Color="#DD0000" />
  </Window.Resources>
</Window>
```

`x:Key` identifies the key in XAML

```
public partial class MainWindow
{
   MainWindow()
   {
      InitializeComponent();
      this.Resources.Add(
          "redBrush",
          new SolidColorBrush(
       Color.FromRgb(0xdd,0,0))
      );
  }
}
```

Using Resources in XAML

- `{StaticResource}` extension used to locate resources
  - parameter identifies the key
  - resource value applied once, during property setter

```
<StackPanel>

   <TextBlock Foreground="{StaticResource redBrush}"
              Text="{StaticResource caption}"/>

   <Button Foreground="{StaticResource whiteBrush}"
           Background="{StaticResource redBrush}" ... />

</StackPanel>
```

Using Resources in XAML [2]

- {DynamicResource} extension defers binding until usage
  - allows forward references
  - dynamically updates UI when value stored in dictionary changes

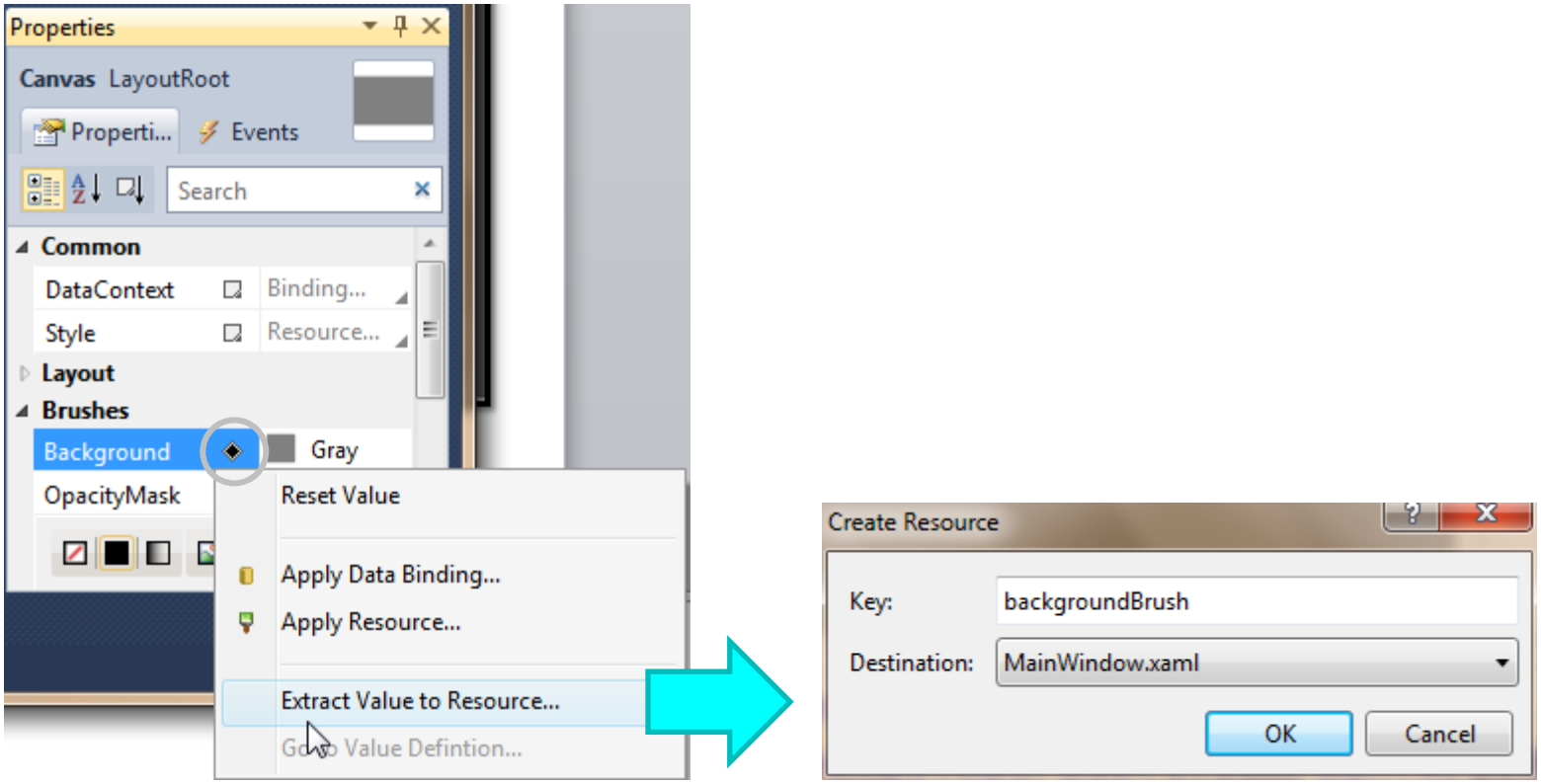
```
<Window Background="{DynamicResource background}">
   <Window.Resources>
      <SolidColorBrush x:Key="background" Color="White" />
   </Window.Resources>
         ...
   <Button Content="Change Color" Click="OnChangeColor" />
</Window>
```

```
void OnChangeColor(object sender, RoutedEventArgs e)
{
   this.Resources["background"] = Brushes.Yellow;
}
```

color of window will change at runtime when the brush is changed

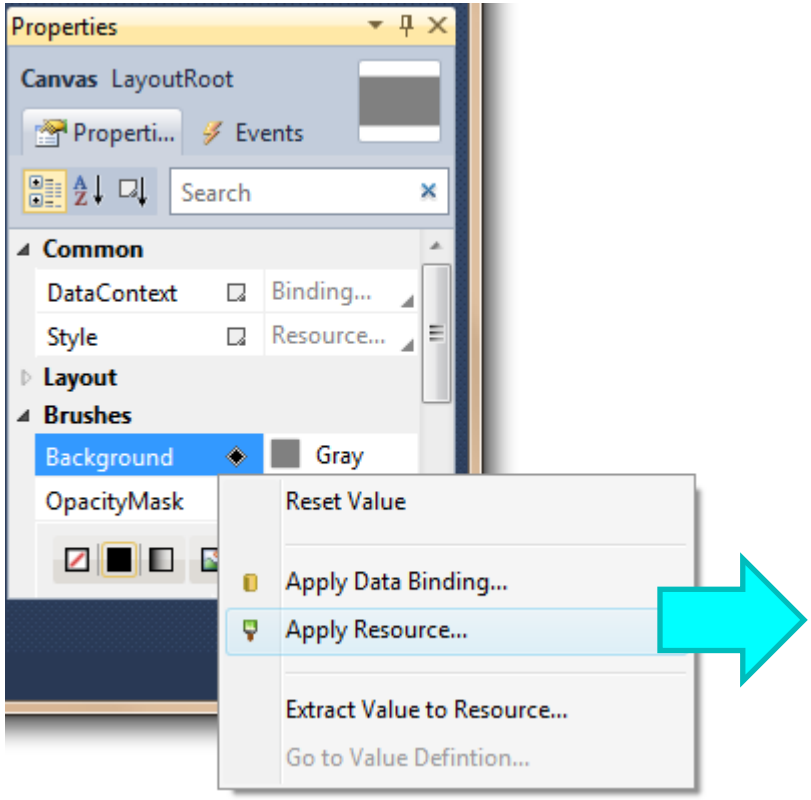
Creating general resources in Visual Studio

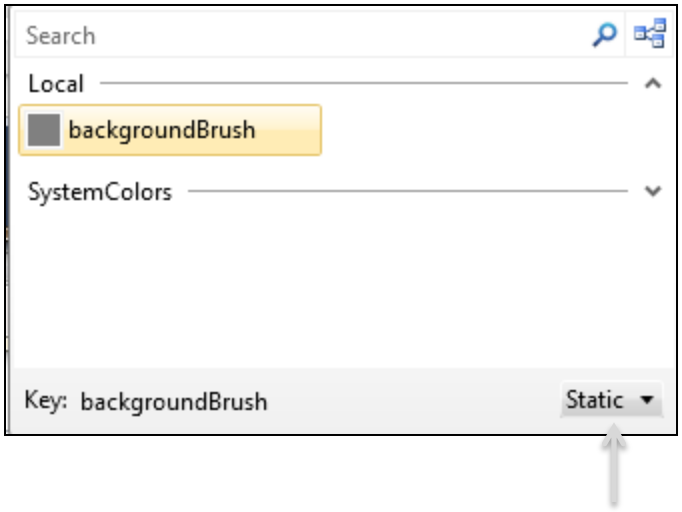
- Most properties can be moved to resources
  - can select what level to add resource to (destination)



Applying existing resources to properties

- Context Menu allows selection of existing resources
  - lists matching resource types





can select static vs. dynamic lookup

Using resources from code behind

- `Resources` property provides indexer access to dictionary
  - can be used to get or change existing resources
- `FindResource` is the preferred way to locate resources
  - performs "bubble-up" search to find key
  - throws exception if resource does not exist
  - use **`TryFindResource`** if resource might not exist

```
this.Resources.Add("redBrush", Brushes.Red);
Brush redBrush = (Brush) this.Resources["redBrush"];
...

button1.Background = (Brush) this.FindResource("redBrush");

textBlock1.Foreground = textBlock1.TryFindResource(
                            "blueBrush") as Brush;
```

Using System defined resources

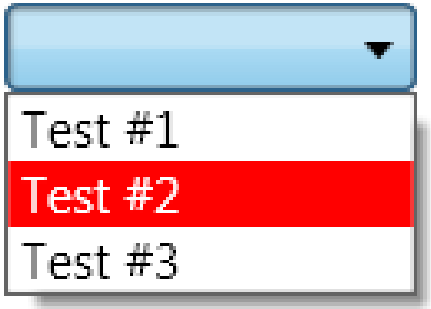
- WPF stored system resources in resource dictionaries
  - each property has a **xxxKey** property also defined on class
  - prefer to use **DynamicResource** to locate in case of changes
- Most controls use this technique to wire up colors and fonts
  - perform lookup in dictionary to get appropriate resource

```
<StackPanel>
   <Button Content="Some Button"
       Background="{DynamicResource
          {x:Static SystemColors.ActiveCaptionBrushKey}}"
       Width="{DynamicResource
          {x:Static SystemParameters.IconHeightKey}}" />
</StackPanel>
```

Replacing system resources

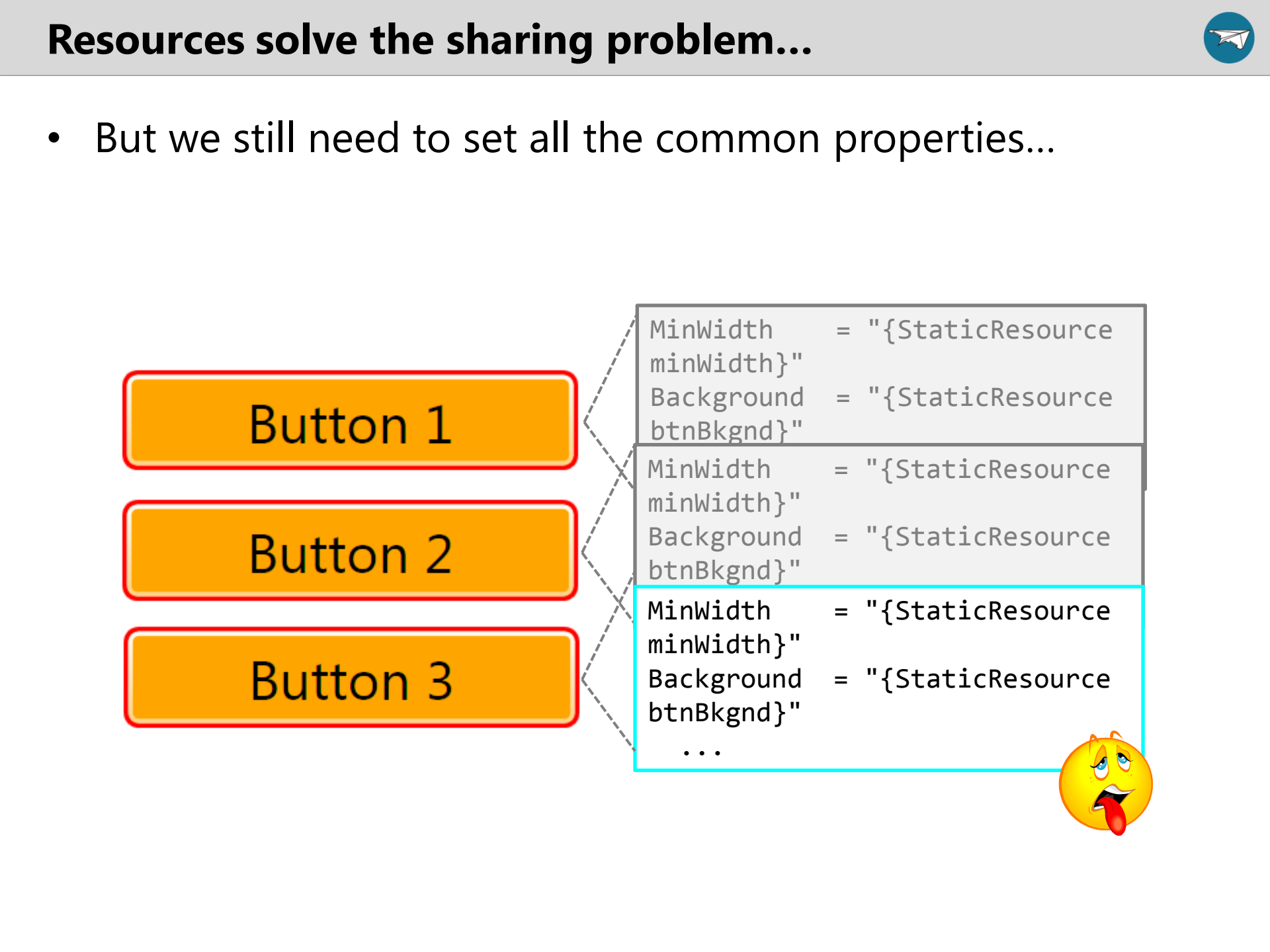
- Application can replace system values by adding new value to local dictionary using system key
  - resource is located through bubble-up search
  - does not work for all controls

override selection color of combo box by adding a brush resource with proper key




```
<ComboBox>
   <ComboBox.Resources>
      <SolidColorBrush Color="Red"
           x:Key="{x:Static SystemColors.HighlightBrushKey}" />
   </ComboBox.Resources>
   ...
</ComboBox>
```

Resources solve the sharing problem…



- But we still need to set all the common properties…

Button 1

Button 2

Button 3

```
MinWidth    = "{StaticResource
minWidth}"
Background  = "{StaticResource
btnBkgnd}"
```

```
MinWidth    = "{StaticResource
minWidth}"
Background  = "{StaticResource
btnBkgnd}"
```

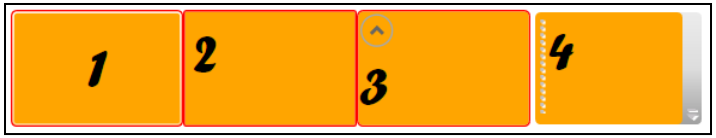
```
MinWidth    = "{StaticResource
minWidth}"
Background  = "{StaticResource
btnBkgnd}"
  ...
```

Introducing Styles

- Styles group together common property values ("setters")
  - applied to **`Framework[Content]Element.Style`**

```
<Style x:Key="CommonStyle">
   <Setter Property="Control.MinWidth" Value="100" />
   <Setter Property="Control.Background" Value="Orange" />
   <Setter Property="Control.BorderBrush" Value="Red" />
</Style>
```

`Property` identifies fully-qualified property name – if it doesn't exist on specific control it is ignored




```
<Button   Style="{StaticResource CommonStyle}">1</Button>
<TextBox  Style="{StaticResource CommonStyle}">2</TextBox>
<Expander Style="{StaticResource CommonStyle}">3</Expander>
<ToolBar  Style="{StaticResource CommonStyle}">4</ToolBar>
```

Targeting a specific control type

- Generally better to target a specific type
  - allows property specification to be shortened
  - constrains usage to specified or derived type

```
<Style x:Key="ButtonStyle"
       TargetType="{x:Type ButtonBase}">
   <Setter Property="MinWidth"    Value="100" />
   <Setter Property="Background"  Value="Orange" />
   <Setter Property="BorderBrush" Value="Red" />
</Style>
```

```
<Button      Style="{StaticResource ButtonStyle}" />
<RadioButton Style="{StaticResource ButtonStyle}" />
<CheckBox    Style="{StaticResource ButtonStyle}" />
```

Style setter precedence

- Property values are chosen from closest setter
  - **`Style`** itself is located using resource lookup

can omit `{x:Type}`as type converter assumes it is a Type

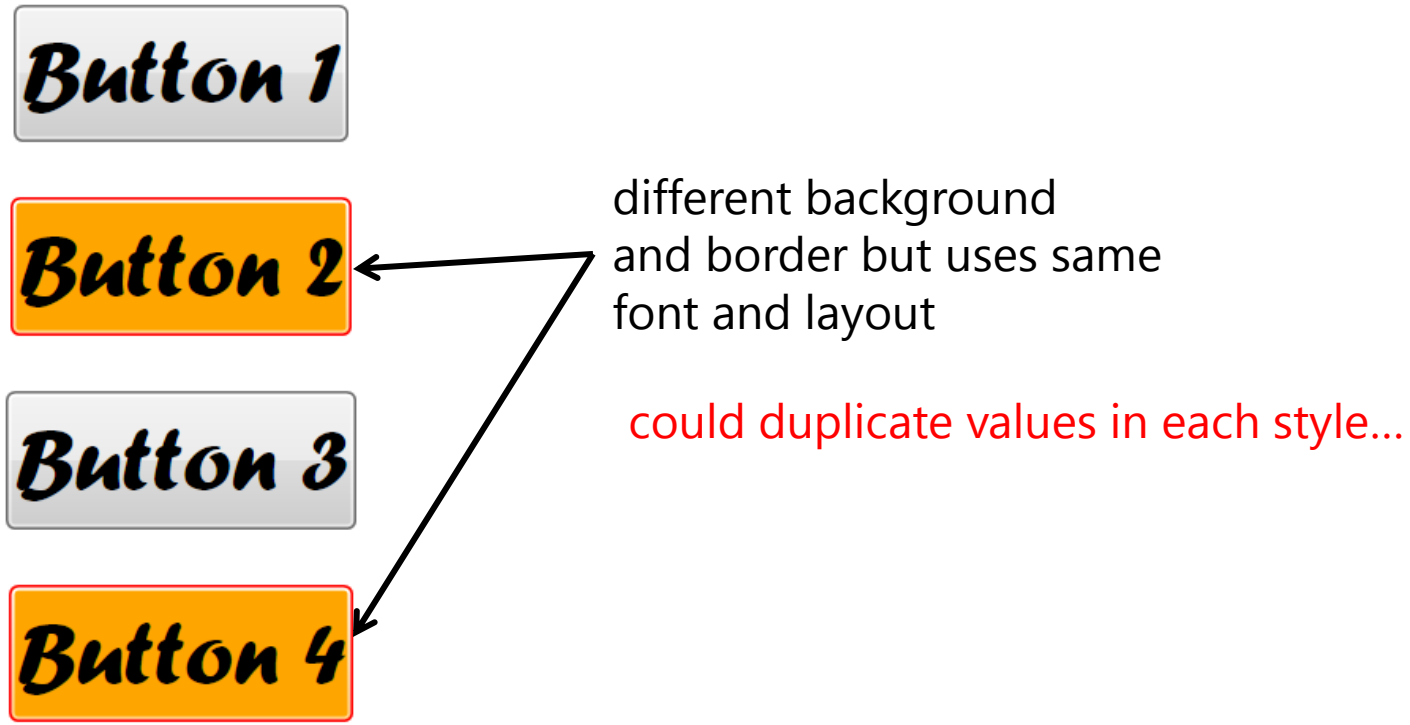
```
<Style x:Key="buttonStyle" TargetType="Button" >
   <Setter Property="Background" Value="Green" />
   <Setter Property="FontSize" Value="16pt" />
</Style>
```

```
<Button Style="{StaticResource buttonStyle}" Content="B"
        Background="Orange" />
```

direct value applied to the button overrides the style value

Grouping styles

- UI may be composed of "groups" that need different styles
  - some properties may be identical in the style



Style inheritance

- Styles can be derived from other styles
  - **BasedOn** property allows derived style to "inherit" values
  - requires **{StaticResource}**

```
<Window.Resources>
   <Style x:Key="allButtons" TargetType="Button">
      <Setter Property="FontSize" Value="24pt" />
      <Setter Property="FontFamily" Value="Forte" />
         ...
      </Style>
   <Style x:Key="orangeButtons" TargetType="Button"
          BasedOn="{StaticResource allButtons}">
      <Setter Property="Background" Value="Orange" />
      <Setter Property="BorderBrush" Value="Red" />
   </Style>
</Window.Resources>
```

Setting a global style

- Styles can be automatically applied by omitting the `x:Key`
  - default style is located by using the **`TargetType`** as the key[1]
  - can use **`BasedOn`** to keep any other default style settters too

```
<Application.Resources>
   <Style TargetType="Button"
          BasedOn="{StaticResource {x:Type Button}}">
      <Setter Property="FontSize" Value="24pt" />
      <Setter Property="Background" Value="Orange" />
      <Setter Property="BorderBrush" Value="Red" />
      <Setter Property="MinWidth" Value="100" />
   </Style>
</Application.Resources>
```

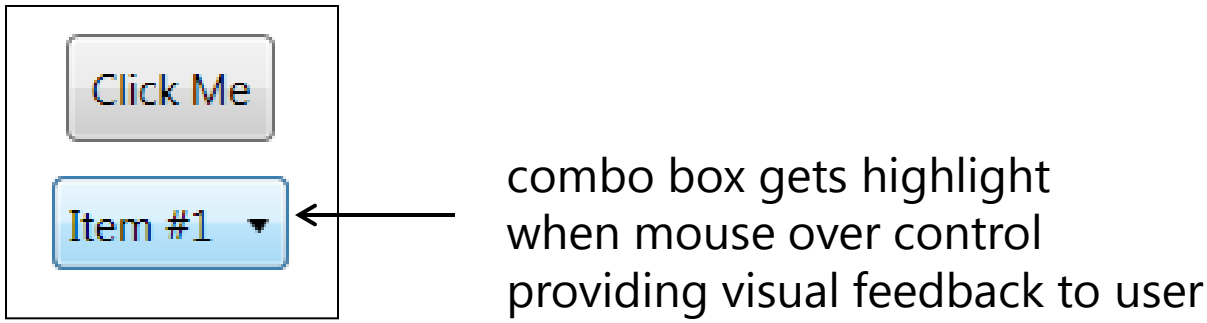
requires `TargetType` be specified

```
<StackPanel>
   <Button>A</Button>
   <Button>B</Button>
   <Button>C</Button>
</StackPanel>
```

no longer necessary to specify style

Providing visual behavior within the view

- UI often provides visual feedback as user interacts with it
  - traditionally requires procedural code for custom controls
  - designer needs ability to "trigger" visual changes



In a Win32 application this would involve handling the WM_MOUSENTER and WM_MOUSELEAVE messages

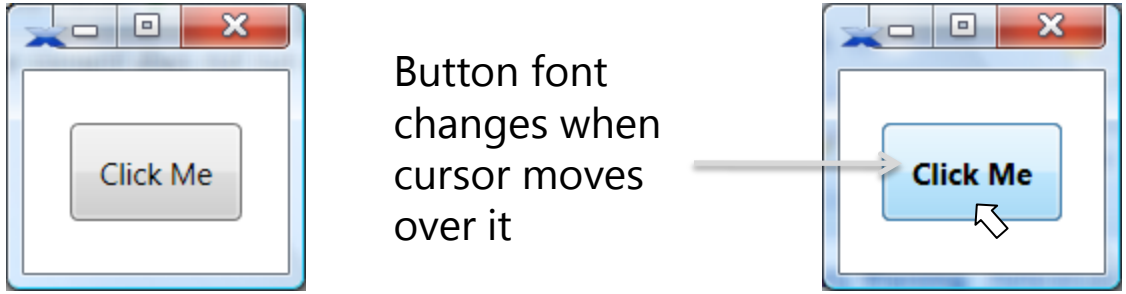
Introducing: Triggers

- Triggers define behavior for controls in XAML
  - i.e. what happens when an 'event' occurs
- Triggers are generally created as part of styles
  - each trigger defines condition and collection of setters
- Conditions that drive triggers can be
  - property value changes
  - routed event raised

```
<Window.Resources>
   <Style TargetType="Button">
      <Style.Triggers>
         <Trigger />
      </Style.Triggers>
   </Style>
   ...
</Window.Resources>
```

Property triggers

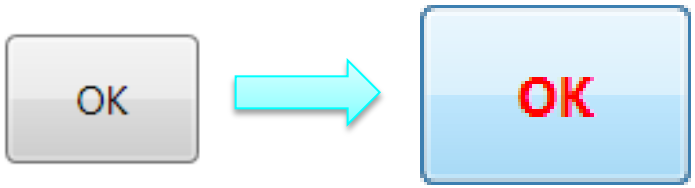
- Property trigger based on DependencyProperty value
  - applied when property = value, reset when property != value




```
<Style TargetType="{x:Type Button}">
   <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
         <Setter Property="FontWeight" Value="Bold" />
      </Trigger>
   </Style.Triggers>
</Style>
```

Multiple setters

- Triggers can contain multiple setters
  - applied in order




```
<Style TargetType="{x:Type Button}">
   <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
         <Setter Property="FontWeight" Value="Bold" />
         <Setter Property="Foreground" Value="Red" />
         <Setter Property="RenderTransform">
           <Setter.Value>
              <ScaleTransform ScaleX="1.5" ScaleY="1.5" />
           </Setter.Value>
         </Setter>
      </Trigger>
   </Style.Triggers>
</Style>
```

Expressing OR trigger relationships

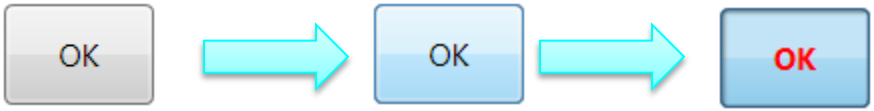
- Styles can contain multiple trigger definitions
  - setters can be same with different conditions

```
<Style TargetType="{x:Type Button}">
   <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
         <Setter Property="FontWeight" Value="Bold" />
      </Trigger>
      <Trigger Property="IsKeyboardFocused" Value="True">
         <Setter Property="FontWeight" Value="Bold" />
      </Trigger>
   </Style.Triggers>
</Style>
```

Font is bold if mouse is over button or if button has focus

Expressing AND property trigger relationships

- `MultiTriggers` allow multiple conditions to be applied
  - trigger only executed when all conditions are true



Setters only apply if mouse is over button and button is pressed

```
<Style x:Key="button" TargetType="{x:Type Button}">
   <Style.Triggers>
      <MultiTrigger>
         <MultiTrigger.Conditions>
            <Condition Property="IsMouseOver" Value="True" />
            <Condition Property="IsPressed" Value="True" />
         </MultiTrigger.Conditions>
         <Setter Property="FontWeight" Value="Bold" />
         <Setter Property="Foreground" Value="Red" />
      </MultiTrigger>
   </Style.Triggers>
</Style>
```

When the trigger does not work…

- Local values always take precedence
  - triggers cannot override local properties

```
<Style TargetType="Button">
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="FontWeight" Value="Bold" />
    </Trigger>
  </Style.Triggers>
</Style>
  ...
<Button FontWeight="Normal" Content="A Button" />
```

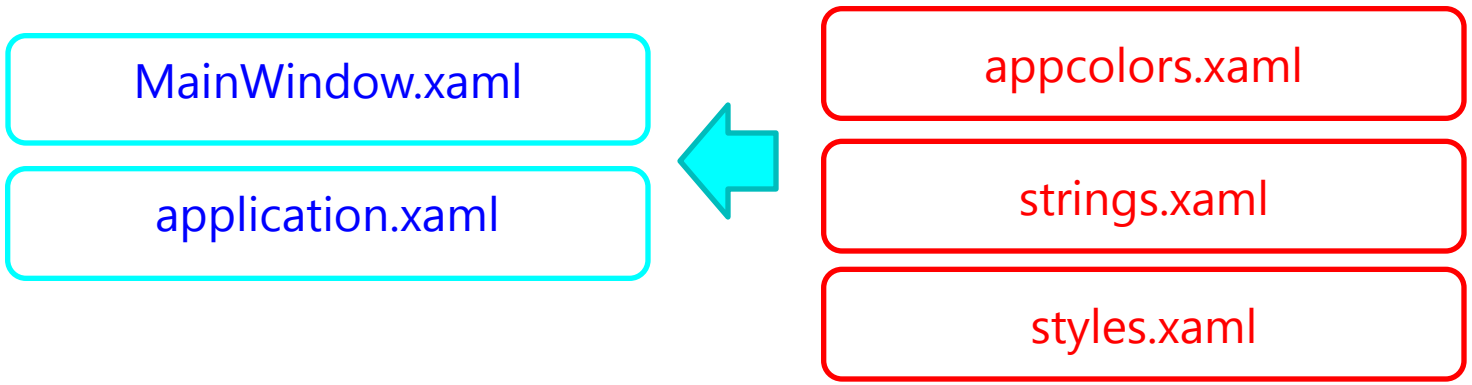
FontWeight is specified directly and trigger will not change

- e style

```
<Style TargetType="Button">
  <Setter Property="FontWeight" Value="Normal" />
    ...
</Style>
```

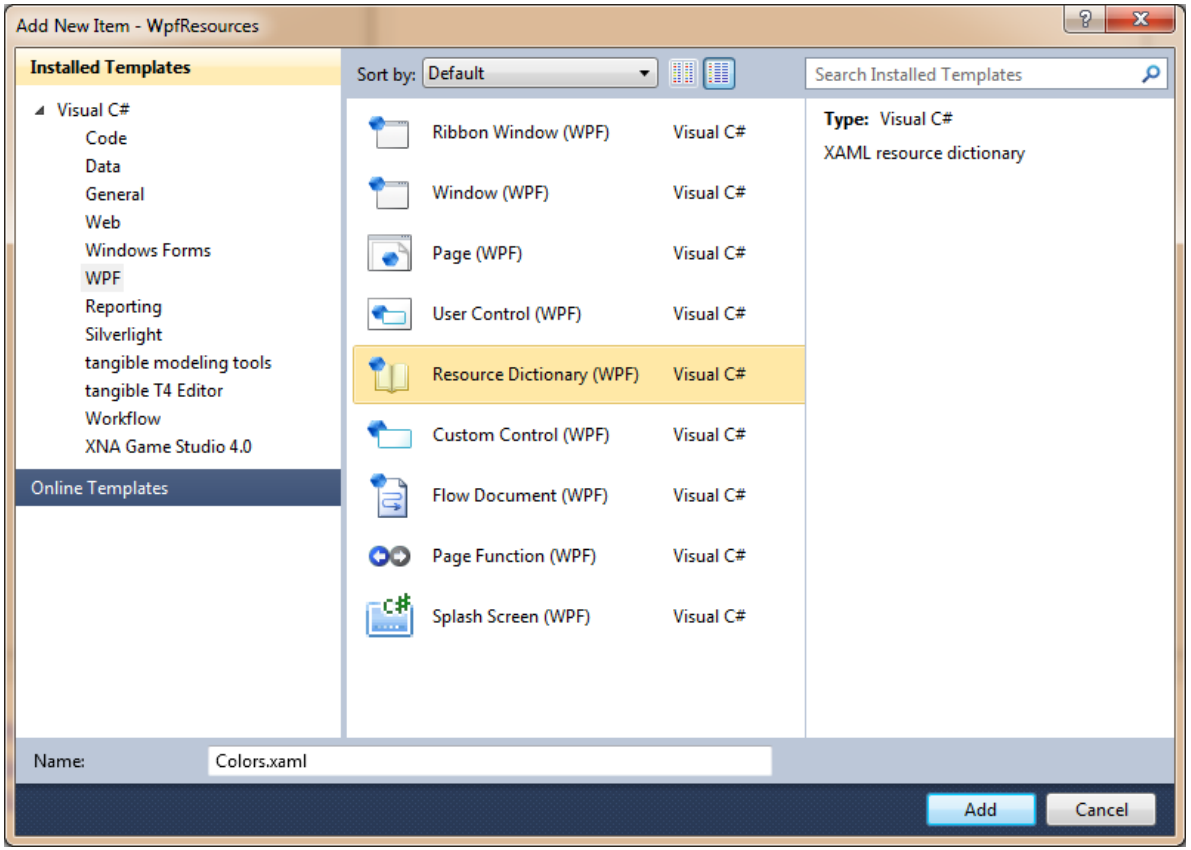
Organizing resources

- Resource Dictionaries are used to group assets
  - can be grouped by type and located in separate files
  - visual designers can bundle up visual styles this way
- Resource files must be *merged* into application resource tree
  - can be merged in at any point (application, window, element)
  - allows WPF to locate resources through normal search



Step 1: Create a ResourceDictionary file

- Contained in standalone `ResourceDictionary`
  - add "Resource Dictionary (WPF)" to project



Step 2: Add resources to dictionary

- Add each resource to root `ResourceDictionary`
  - can also name dictionary for easier access

Colors.xaml

```
1  <ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
2                      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
3                      xmlns:s="clr-namespace:System;assembly=mscorlib">
4
5      <SolidColorBrush x:Key="redBrush" Color="#FF0000" />
6      <SolidColorBrush x:Key="backgroundBrush" Color="Gold" />
7      <SolidColorBrush x:Key="borderBrush" Color="#FFd0f023" />
8      <s:String x:Key="copyright">(C) 2010 Some Company</s:String>
9
10 </ResourceDictionary>
```

Step 3: merging in resource dictionaries

- Resource dictionaries must be merged in to be used
  - add each dictionary to **MergedDictionaries** collection
  - automatically done if added with Blend

```
<Application.Resources>
   <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
         <ResourceDictionary Source="colors.xaml" />
         <ResourceDictionary Source="styles.xaml" />
      </ResourceDictionary.MergedDictionaries>
   </ResourceDictionary>
</Application.Resources>
```

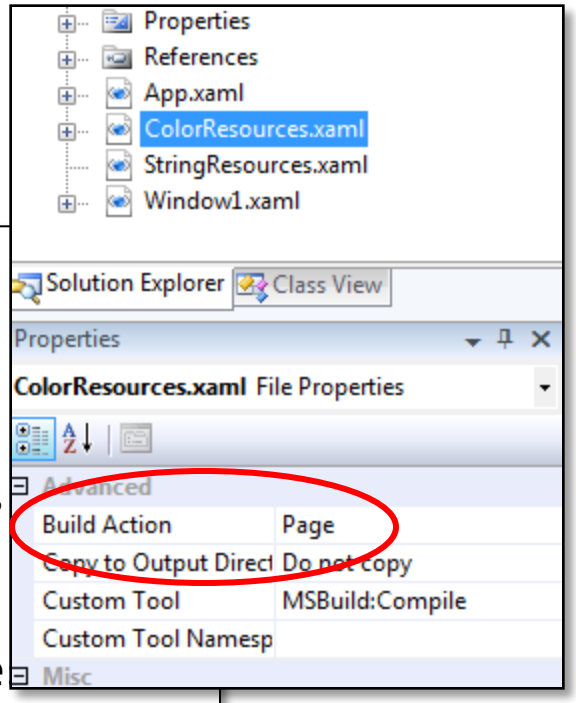
```
<ResourceDictionary xmlns="..." xmlns:x="...">
   <SolidColorBrush x:Key="redBrush" ... />
</ResourceDictionary>
```

colors.xaml

Using strongly-typed resource dictionaries

- Can also add resource dictionaries by name
  - provides compile-time safety
  - must be compiled into assembly[1]

```
<Application ...
   xmlns:app="clr-namespace:VideoPlayer">
   <Application.Resources>
      <ResourceDictionary>
         <ResourceDictionary.MergedDictionaries
            <app:ColorResources />
            <app:StyleResources />
         </ResourceDictionary.MergedDictionarie
      </ResourceDictionary>
   </Application.Resources>
</Application>
```



Sharing resources across applications

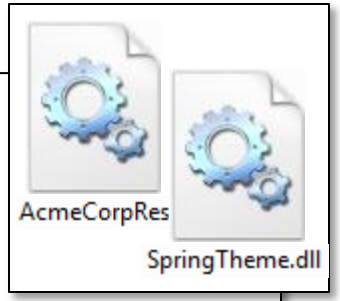
- `ResourceDictionary` can be moved to secondary assembly
  - can then be referenced by multiple applications
  - use either Pack URI or strongly-typed dictionaries

```
<Application.Resources>
   <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
         <ResourceDictionary
              Source="/acmecorp;Component/colors.xaml" />
         <acmecorp:StringResources />
      </ResourceDictionary.MergedDictionaries>
   </ResourceDictionary>
</Application.Resources>
```

Pack URI syntax used to locate resources in other assemblies – takes the format:
`/assembly;Component/name.xaml`

Dynamically merging resources

- Can dynamically manipulate resources in code behind
  - allows resources to be decided at runtime




```
partial class App : Application
{
   public App()
   {
      this.Resources.MergedDictionaries.Add(
               new AcmeCorpRes.CommonResources());

      string themeName = GetUsersSelectedTheme();
      var rd = Application.LoadComponent(
         new Uri(@"/" + themeName + @";Component/colors.xaml",
                  System.UriKind.Relative))
             as ResourceDictionary
      this.Resources.MergedDictionaries.Add(rd);
   }
}
```

Summary

- Resources allow sharing of visual properties
  - anything can be a resource
  - located by string-based key
- Styles are useful to set common "look" across controls
  - provide a global definition of fonts, colors, etc.
  - used by designers to separate volatile elements out
- Triggers allow actions to be defined in styles
  - setters are applied when action is detected
  - used extensively in controls to define behavior
- `ResourceDictionary` allows assets to be shared
  - WPF searches for appropriate resource using key
  - can be placed into separate XAML files for reuse