WPF201-B

Module 2

Using WPF Layout Panels & Resource Dictionaries

1 Windows & Panels

1.1 Windows

A **Window** is basically a **ContentControl** that has a window frame. Even though the **Content** property can contain anything, it can only be one thing. To contain multiple UI elements you can use a **Panel** control. A Panel has a **Children** property that can be added zero or more UI elements.

Using panel to host multiple elements: Windows1\MainWindow.xaml

```
<StackPanel>
    <Button x:Name="Button1" Click="Button1_Click" Content="StackPanel" />
    <Button x:Name="Button2" Click="Button2_Click" Content="WrapPanel" />
    <Button x:Name="Button3" Click="Button3_Click" Content="DockPanel" />
    <Button x:Name="Button4" Click="Button4_Click" Content="Canvas" />
    <Button x:Name="Button5" Click="Button5_Click" Content="UniformGrid" />
    <Button x:Name="Button6" Click="Button6_Click" Content="Grid" />
    <Button x:Name="Button7" Click="Button7_Click" Content="Close All" />
</StackPanel>
```

In one application, you can add as many **Window** classes as required. You can now add six additional window classes named *Window1* to *Window6*. You can now program button **Click** handlers for *Button1* to *Button6* to show them.

Instancing and showing windows

```
private void Button1_Click(object sender, ...) { new Window1().Show(); }
private void Button2_Click(object sender, ...) { new Window2().Show(); }
private void Button3_Click(object sender, ...) { new Window3().Show(); }
private void Button4_Click(object sender, ...) { new Window4().Show(); }
private void Button5_Click(object sender, ...) { new Window5().Show(); }
```

Even if the shutdown mode was **OnLastWindowClosed,** you can easily write code to close all windows in the application. Access the **Windows** property of the application object to get a collection of opened windows. If you do not wish to close the main window, you can find out which window is the main by comparing it against the **MainWindow** property and bypass that window when calling the **Close** method for each window. If you are sure that the current window is the main window, you can compare against **this** instead and you would not need to obtain it from the application object.

Closing all windows except the main window

```
private void Button7_Click(object sender, RoutedEventArgs e) {
    Application App = Application.Current;
    Window mainWindow = App.MainWindow;
    WindowCollection windows = App.Windows;
    foreach (Window window in windows)
        if (window != mainWindow) window.Close();
}
```

Simpler code to close all other windows from main window

```
private void Button7_Click(object sender, RoutedEventArgs e) {
    foreach (Window window in Application.Current.Windows)
        if (window != this) window.Close();
}
```

Each window can be owned by another window by using the **Owner** property. You can then access the collection of windows owned using **OwnedWindows** property. It is then possible to close only those windows owned and not every window opened. If we made the main window own all other windows, we would not have to access the application object to close the windows.

Make current window owner of instantiated window

```
private void Button1_Click(object sender, C) {
    Window window = new Window1(); window.Owner = this; window.Show();
}
```

Close all owned windows

```
private void Button7_Click(object sender, RoutedEventArgs e) {
    foreach (Window window in OwnedWindows) window.Close();
}
```

Sometimes you need may want a single instance of a window. To do this you should declare a field to hold on to a single instance. You will then re-use the instance until it is closed. Detect the **Closed** event to release the instance.

Implementing one instance per window class

```
private Window1 window1;
private void Button1_Click(object sender, RoutedEventArgs e) {
        if (window1 != null) { window1.Focus(); return; }
        window1 = new Window1();
        window1.Closed += (ws, we) => window1 = null;
        window1.Owner = this; window1.Show();
}
```

Rather than calling **Show** method, you can call **ShowDialog** method instead that will only return when the window is closed. You can optionally capture a result by default is not **true** but can be changed from within the window.

Using Window as a dialog box

```
private void Button8_Click(object sender, RoutedEventArgs e) {
    Window window = new Window6().Show();
    window.Owner = this; bool? result = window.ShowDialog();
 // Process user confirmation
    if (result == true) { }
}
```

In the window, you can provide buttons to close the window. For a button to close the window and return a **DialogResult** of **false**, set **IsCancel** property to **true**. Setting **IsDefault** does not automatically close the window nor return a **true** result. You still need to attach an event handler to set **DialogResult** and close the window. You can also use an event handler for the cancel button as well if you need to do operations as well when cancelled.

Implement confirm and cancel in dialog window: Window6.xaml

```
<StackPanel>
    <Button x:Name="btnOK" IsDefault="True" Click="BtnOK_Click">OK</Button>
    <Button x:Name="btnCancel" IsCancel="True">Cancel</Button>
</StackPanel>
```

Setting DialogResult and closing window: Window6.xaml.cs

```
private void BtnOK_Click(object sender, RoutedEventArgs e) {
    DialogResult = true;
    Close();
}
```

1.2 StackPanel

A panel is a layout control to manage and arrange a set of child elements. Since itself is an UI element it can be managed and arranged inside other panels. WPF provides a basic set of panels but you can always use also those created by other developers or implement your own panel. **StackPanel** layout child elements one after another. Use the **Orientation** property to determine whether controls will be stacked vertically or horizontally where default value is **Vertical** but changed be changed to **Horizontal**. A vertical StackPanel allow UI elements to be stretched horizontally only but horizontal StackPanel allow elements to be stretched vertically only.

Using StackPanel to layout elements: Window1.xaml

```
<StackPanel>
    <Button Content="Button1" />
    <Button Content="Button2" />
    <Button Content="Button3" />
    <Button Content="Button4" />
</StackPanel>
```

Change StackPanel orientation

```
<StackPanel Orientation="Horizontal">
    <Button Content="Button1" />
    <Button Content="Button2" />
    <Button Content="Button3" />
    <Button Content="Button4" />
</StackPanel>
```

1.3 WrapPanel

A **WrapPanel** attempts to fit as many elements as possible in the direction that is set using the **Orientation** property. If an element cannot fit it is moved to the next row if orientation is **Horizontal** which is the default and moved to the next column instead if the orientation is **Vertical**. Elements on the same column or same row will have the same height or width if they size have not been fixed. A WrapPanel does not allow any element to stretch horizontally or vertically.

Using the WrapPanel panel: Window2.xaml

```
<WrapPanel>
    <Button Content="Button1" />
    <Button Content="Button2" />
    <Button Content="Button3" />
    <Button Content="Button4" />
</WrapPanel>
```

Changing WrapPanel orientation

```
<WrapPanel Orientation="Vertical">
        :
```

1.4 DockPanel

A **DockPanel** provides five areas to arrange elements. Use **Dock** attached property provided by **DockPanel** to dock it to the **Top**, **Left**, **Right** or **Bottom** of the panel. Last elements that does not use this attached property is automatically docked into the central area that will have all the remaining space. In the panel add 5 buttons and dock 4 to each side of the panel. Last button automatically fills up the central area of the panel.

Using DockPanel panel: Window3.xaml

```
<DockPanel>
    <Button DockPanel.Dock="Top" Content="Top" />
    <Button DockPanel.Dock="Bottom" Content="Bottom" />
    <Button DockPanel.Dock="Left" Content="Left" />
    <Button DockPanel.Dock="Right" Content="Right" />
    <Button Content="Center" />
</DockPanel>
```

Elements docked to top and bottom can stretch horizontally and those docked to left and right can stretch vertically. Elements in the center uses the remaining space and can stretch in both directions. Multiple elements can be docked to any area. The order of elements is important to determine which elements get the shared corner space as shown in the following example.

Docking multiple controls to one area: panel3b.xaml

```
<DockPanel>
    <Button DockPanel.Dock="Top" Content="Top" />
    <Button DockPanel.Dock="Top" Content="Top2" />
    <Button DockPanel.Dock="Bottom" Content="Bottom" />
    <Button DockPanel.Dock="Left" Content="Left" />
    <Button DockPanel.Dock="Top" Content="Top3" />
    <Button DockPanel.Dock="Right" Content="Right" />
    <Button DockPanel.Dock="Top" Content="Top4" />
</DockPanel>
```

1.5 Canvas

Canvas is the most basic panel in WPF and does not do automatic layout of elements. Elements added to the **Canvas** have to positioned using attached properties **Left** and **Top**. You may also use **Right** and **Bottom** properties to offset child elements from the right and bottom sides instead. Stretching is not available as alignment properties are ignored. Elements are auto-sized to content unless a specific **Width** and **Height** is assigned.

Using Canvas panel: Window4.xaml

```
<Canvas>
    <Button Canvas.Left="8" Canvas.Top="8"
        Width="96" Height="24" Content="Button1" />
    <Button Canvas.Left="108" Canvas.Top="22"
        Width="96" Height="24" Content="Button2" />
    <Button Canvas.Left="8" Canvas.Top="36"
        Width="96" Height="24" Content="Button3" />
    <Button Canvas.Left="108" Canvas.Top="50"
        Width="96" Height="24" Content="Button4" />
    <Button Canvas.Left="8" Canvas.Top="64"
        Width="96" Height="24" Content="Button5" />
</Canvas>
```

1.6 Grid Panels

A **UniformGrid** is a simple panel of rows and columns where each cell in the grid has the same size. Use **Rows** and **Columns** properties to specify the number of rows and columns in the grid. Each element in the panel will be placed into a single cell from left to right, top to bottom. Elements without a cell will not appear.

Using UniformGrid panel: Window5.xaml

```
<UniformGrid Rows="2" Columns="3">
    <Image Margin="4" Source="Photos\andrew_fuller.jpg" />
    <Image Margin="4" Source="Photos\anne_dodsworth.jpg" />
    <Image Margin="4" Source="Photos\janet_leverling.jpg" />
    <Image Margin="4" Source="Photos\robert_king.jpg" />
    <Image Margin="4" Source="Photos\michael_suyama.jpg" />
    <Image Margin="4" Source="Photos\nancy_davolio.jpg" />
</UniformGrid>
```

A **Grid** is a panel that can be separated into multiple rows and columns. By default it has 1 row and 1 column. Use **RowDefinitions** and **ColumnDefinitions** for multiple rows and columns. Below is a grid with 3 rows and 4 columns. Spacing is distributed between cells if height of the row and width of column is undefined. Use **Height** and **Width** to fix the size for rows and columns or use **MinWidth**, **MinHeight**, **MaxWidth** and **MaxHeight**. Use **Auto** to size based on the contents of the cell.

Creating a grid of rows and columns: Window6.xaml

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
</Grid>
```

You can also use weighted size instead of absolute size by using an asterisk (*). The size is relative to sizes of other rows and columns. A row that has a relative height of 2 will have four times the height of other rows that has a relative height of 0.5 but have only half the height of rows that has a relative height of 4.

Setting row height and column width

```
<Grid.RowDefinitions>
    <RowDefinition Height="32" />
    <RowDefinition />
    <RowDefinition MinHeight="64" MaxHeight="128" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition MinWidth="64" />
    <ColumnDefinition MaxWidth="128" />
    <ColumnDefinition />
    <ColumnDefinition Width="128" />
</Grid.ColumnDefinitions>
```

Using weighted size instead of absolute size

```
<Grid.RowDefinitions>
    <RowDefinition Height="0.5*" />
    <RowDefinition Height="1*"/>
    <RowDefinition Height="0.5*" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="200" />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
```

Elements can use the **Row** and **Column** attached properties provided by **Grid** class to assign them to each cell in a grid. It is also possible to occupy more than one row and column using **RowSpan** and **ColumnSpan** properties.

Placing controls in cells

```
<Button Content="1" Grid.Row="0" Grid.Column="0" Grid.RowSpan="3" />
<Button Content="2" Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="3" />
<Button Content="3" Grid.Row="1" Grid.Column="1"
    Grid.RowSpan="2" Grid.ColumnSpan="2" />
<Button Content="4" Grid.Row="2" Grid.Column="3" />
```

You can add multiple elements in one cell where the later elements overlap the earlier elements occupying the same cells. You can still make all elements in the same cell visible as long as they have a different size, position or alignment. You can also set elements to be partially transparent using **Opacity** property where **1** is fully opaque and **0** is fully transparent.

Placing multiple controls in one cell

```
<Rectangle Grid.Row="1" Grid.Column="3" Fill="Black" />
<Ellipse Grid.Row="1" Grid.Column="3" Margin="12" Fill="Red" />
<Button Grid.Row="1" Grid.Column="3" Width="32" Height="32"
    HorizontalAlignment="Left" Opacity="0.7" Content="5" />
<Button Grid.Row="1" Grid.Column="3" Width="32" Height="32"
    HorizontalAlignment="Right" Opacity="0.7" Content="6" />
```

When elements occupy the same cells or across multiple cells it is hard to make out the rows and columns of a table. You can set **ShowGridLines** property to true during the design phase so that you can clearly see the rows and columns that form a grid. The lines are purely for display only and cannot be used for adjusting row and column size. Use **GridSplitter** to allow a user to manually resize the cells. **ResizeDirection** and **ResizeBehavior** will be used to determine what cells will be resized. To ensure that a GridSplitter does not over an element at the back, adjust the margin of the back element or you place the GridSplitter into its own cell.

Enabling row and column lines on Grid

```
<Grid ShowGridLines="True">
```

Using GridSplitter elements

```
<GridSplitter Grid.RowSpan="3"
    HorizontalAlignment="Right"
    ResizeDirection="Columns"
    ResizeBehavior="CurrentAndNext"
    Width="4" Background="Red" />

<GridSplitter Grid.Row="1" Grid.ColumnSpan="4"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Top"
    ResizeDirection="Rows"
    ResizeBehavior="PreviousAndCurrent"
    Height="4" Background="Blue" />
```

2 Brushes & Styles

2.1 Using Brushes

Brushes are used in WPF to draw all graphics except images. There are many types of brushes available. When you assign a string containing a color name or code, it will be converted automatically into a **SolidColorBrush** that is a brush with a single color.

Assigning SolidColorBrush in XAML: Brushes1\MainWindow.xaml

```
<StackPanel x:Name="Panel1">
    <TextBlock Text="Welcome to XAML!" Foreground="LightSteelBlue"
        FontFamily="Georgia" FontSize="36" Margin="8"/>
    <TextBlock Text="Welcome to XAML!" Foreground="#A37CA5"
        FontFamily="Georgia" FontSize="36" Margin="8"/>
</StackPanel>
```

A **Brushes** class is available that contains a standard set of solid color brushes. The following shows how to construct the above element and assign a brush by writing code. You can also create a new brush by using a color from the **Colors** class or construct your own custom color.

Assigning a standard SolidColorBrush in code

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
      var tb1 = new TextBlock(); tb1.Text = "Welcome to XAML!";
      tb1.Margin = new Thickness(8); tb1.FontSize = 36;
      tb1.Foreground = Brushes.LightSteelBlue;
      tb1.FontFamily = new FontFamily("Georgia");
      Panel1.Children.Add(tb1);
}
```

Creating a new SolidColorBrush from standard colors

```
tb1.Foreground = new SolidColorBrush(Colors.LightSteelBlue);
```

Creating a new SolidColorBrush using a custom color

```
tb1.Foreground = new SolidColorBrush(Color.FromRgb(0xA3, 0x7C, 0xA5));
```

A brush is an object but you can still assign it to a property by just using a color name or color code because there is a value converter available to convert them into a **SolidColorBrush**. If you wish to assign the brush explicitly you will have to expand the property as shown below. Even though it is not compulsory to do it this way it does allow you to assign a name to the brush so that you can access and control the brush from your programming code.

Assigning a brush object explicitly

```
<TextBlock Text="Welcome to XAML!"
    FontFamily="Georgia" FontSize="36" Margin="8">
    <TextBlock.Foreground>
        <SolidColorBrush x:Name="brush1" Color="LightSteelBlue" />
    </TextBlock.Foreground>
</TextBlock>
```

However there is no value converter available for every type of brush so you will need to use the above format to assign more complex brushes. For example the following shows how to use a **LinearGradientBrush** instead where you can paint with as many colors as you like by adding **GradientStop** elements to the default **GradientStops** collection property.

Using a LinearGradientBrush: Brushes2\MainWindow.xaml

```
<TextBlock Text="Welcome to XAML!"
    FontFamily="Georgia" FontSize="36" Margin="8">
    <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
            <GradientStop Color="Red" Offset="0" />
            <GradientStop Color="Blue" Offset="0.35" />
            <GradientStop Color="Green" Offset="0.65" />
            <GradientStop Color="Yellow" Offset="1" />
        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

A brush is not a UI element but a resource that can be assigned to UI elements. It will be troublesome to redefine the same brush multiple times if you need to use the same brush for many elements. The same resource can be defined once and shared by many elements by placing it into a resource dictionary.

Using the same brush for many elements

```
<StackPanel Orientation="Horizontal">
    <StackPanel.Resources>
        <LinearGradientBrush x:Key="brush1" StartPoint="0,0" EndPoint="1,1">
                :
        </LinearGradientBrush>
    </StackPanel.Resources>
</StackPanel>
    <Rectangle Stroke="Black" Fill="{StaticResource brush1}"
        Width="100" Height="100" />
    <TextBlock
        Foreground="{StaticResource brush1}" Text="Welcome to XAML!"
        FontFamily="Georgia" FontSize="36" Margin="8" />
    <Button Foreground="White" Background="{StaticResource brush1}"
        Width="96" Height="32" Margin="8" Content="Button" />
</StackPanel>
```

If you use a resource in many different locations in one **Window** it would make sense to assign it to **Window.Resources** instead. To use the same resource across all the windows, assign to **Application.Resources** instead.

Assign application resources: App.xaml

```
<Application.Resources>
    <LinearGradientBrush x:Key="brush1" StartPoint="0,0" EndPoint="1,1">
            :
    </LinearGradientBrush>
</Application.Resources>
```

WPF resources can be frozen by calling a **Freeze** method. Once a resource is frozen you cannot change anything in the resource. Freezing a resource can reduce memory use and provide faster access but the resource becomes read-only. No property can of the resource can be changed. Note that all application resources are automatically frozen.

Changing WPF resource properties at runtime

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    var brush1 = (LinearGradientBrush)FindResource("brush1");
//  brush1.Freeze();// WPF resources can be frozen
    brush1.GradientStops[0].Color = Colors.Fuschia; // will fail if frozen
}
```

The **StartPoint** and **EndPoint** properties of the LinearGradientBrush determine the direction of shading where point (0, 0) is the top left corner and point (1, 1) is the bottom right corner. Following table shows all the possible directions.

Directions

```
    Diagonal down       StartPoint (0, 0) EndPoint (1, 1)
    Diagonal up         StartPoint (1, 1) EndPoint (0, 0)
    Left to right       StartPoint (0, 0) EndPoint (1, 0)
    Right to left       StartPoint (1, 0) EndPoint (0, 0)
    Top to bottom       StartPoint (0, 0) EndPoint (0, 1)
    Bottom to top       StartPoint (0, 1) EndPoint (0, 0)
```

Paint a rectangle from top to bottom: Brushes3\MainWindow.xaml

```
<StackPanel Orientation="Horizontal">
      <Rectangle Margin="8" Width="64" Height="64">
            <Rectangle.Fill>
                  <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                        <GradientStop Color="Black" Offset="0" />
                        <GradientStop Color="White" Offset="1" />
                  </LinearGradientBrush>
            </Rectangle.Fill>
      </Rectangle>
</StackPanel>
```

If the brush does not fill up the entire area, the **SpreadMethod** property will be used to determine what happens to the empty space. The default value is **Pad** where the last color will be used to fill up the remaining space. You can change it to **Reflect** or **Repeat** to generate interesting patterns. The less space a brush fills up the more times the pattern will be repeated.

A brush that fills up only half the horizontal space

```
<LinearGradientBrushStartPoint="0,0" EndPoint="0,0.5">
```

Reflecting the brush to fill up the empty space

```
<LinearGradientBrushStartPoint="0,0" EndPoint="0,0.5" SpreadMethod="Reflect">
```

Creating a repeating pattern by reducing brush size

```
<LinearGradientBrush StartPoint="0,0"
    EndPoint="0.1,0.1" SpreadMethod="Reflect">
```

Repeating without reflection

```
<LinearGradientBrushStartPoint="0,0" EndPoint="0.1,0.1" SpreadMethod="Repeat">
```

While LinearGradientBrush can create interesting patterns, it cannot enhance the look of circles and ellipses. Use **RadialGradientBrush** instead that paints from a position set using the **GradientOrigin** property, outwards in circular steps. Compare these two brushes by first using LinearGradientBrush to paint an ellipse and then changing it to RadialGradientBrush.

Using LinearGradientBrush to paint ellipse

```
<Ellipse Margin="8" Width="64" Height="64">
    <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
            <GradientStop Color="White" Offset="0" />
            <GradientStop Color="Red" Offset="0.5" />
            <GradientStop Color="DarkRed" Offset="1" />
        </LinearGradientBrush>
    </Ellipse.Fill>
</Ellipse>
```

Using RadialGradientBrush to paint ellipse

```
<Ellipse Margin="8" Width="64" Height="64">
    <Ellipse.Fill>
        <RadialGradientBrush GradientOrigin="0.25,0.25">
            <GradientStop Color="White" Offset="0" />
            <GradientStop Color="Red" Offset="0.5" />
            <GradientStop Color="DarkRed" Offset="1" />
        </RadialGradientBrush>
    </Ellipse.Fill>
</Ellipse>
```

2.2 ImageBrush

A bitmap image can be used as a brush by using **ImageBrush**. Following example shows using a bitmap image to paint an **Ellipse**. A **Viewport** property can be used to adjust the area being painted. If you do not use up all the space you can do tiling by setting the **TileMode** property. Stretch can be used to determine how the image will be resized to fit into the space drawn using the brush.

Creating a ImageBrush resource: Brushes4\MainWindow.xaml

```
<Window.Resources>
    <ImageBrush x:Key="brush1" Stretch="Fill"
        ImageSource="YouGame.ico" />
</Window.Resources>
```

Painting shape with ImageBrush

```
<Ellipse Width="64" Height="64"
    Fill="{StaticResource brush1}" Stroke="Black"/>
```

Setting Viewport and TileMode properties

```
<ImageBrush x:Key="brush1" Stretch="Fill"
    Viewport="0,0,0.5,0.5" TileMode="Tile"
    ImageSource="YouGame.ico" />
```

Now that you know about brushes and resources, you can improve the background of your application window by using brushes. Add the following LinearGradientBrush and ImageBrush to **Application.Resources**.

Adding brushes to resource dictionary: SymBank\MyApp.xaml

```
<Application.Resources>
        :
    <LinearGradientBrush x:Key="BackBrush1"
        StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="Black" Offset="0" />
        <GradientStop Color="DarkGray" Offset="0.5" />
        <GradientStop Color="Gray" Offset="1" />
    </LinearGradientBrush>
    <ImageBrush x:Key="BackBrush2"
        Stretch="Uniform" Viewport="0.1,0.1,0.8,0.8"
        ImageSource="{StaticResource SymBankIcon}" />
</Application.Resources>
```

Assign **BackBrush1** as **Background** of **shellLayout** DockPanel and **BackBrush2** to Background of the **shellWorkspace** grid. Remove the previous image inside the grid since we are now using ImageBrush instead to display the icon.

2.3 Using Styles

You may not only want many elements to share the same brush but having the same property settings as well. For example you may want a group of buttons to have the same font, size and margin. A **Style** is a resource that contains a collection of property setters. When you bind a style resource to **Style** property of an element the values in the setters will be used for the properties which has not been explicitly set. Bind **TargetType** property to the type of UI element to allow you to set all the properties for that element. If you bind to a higher-level type, you can then assign the style to more elements but you will not be able to set those properties that are specific to certain elements only. Setter values can be overridden by setting the properties directly. This allows you to still use the same style for multiple elements even though not of all them are completely the same.

Creating style resources: Styles1\MainWindow.xaml

```
<Window.Resources>
    <Style x:Key="style1" TargetType="Button">
        <Setter Property="FontFamily" Value="Georgia" />
        <Setter Property="FontStyle" Value="Italic" />
        <Setter Property="FontSize" Value="16" />
        <Setter Property="Width" Value="128" />
        <Setter Property="Height" Value="32" />
        <Setter Property="Margin" Value="4" />
        <Setter Property="BorderBrush" Value="SteelBlue" />
        <Setter Property="Foreground" Value="White" />
        <Setter Property="Background">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="SteelBlue" Offset="0" />
                    <GradientStop Color="LightSteelBlue" Offset="0.25" />
                    <GradientStop Color="LightSteelBlue" Offset="0.75" />
                    <GradientStop Color="SteelBlue" Offset="1" />
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Application.Resources>
```

Assigning style to multiple elements

```
<StackPanel Background="Black">
    <Button Style="{StaticResource style1}" Content="Button1" />
    <Button Style="{StaticResource style1}" Content="Button2" />
    <Button Style="{StaticResource style1}" Content="Button3" />
    <Button Style="{StaticResource style1}" Content="Button4" />
</StackPanel>
```

Overriding Style properties

```
<Button Style="{StaticResource style1}" Content="Button5"
    Margin="8" Height="48" BorderBrush="Red" />
```

Changing TargetType to support more element types

```
<Style x:Key="style1" TargetType="Control">
```

Style is now applicable to other element types

```
<Label Style="{StaticResource style1}" Content="Label1" />
<RadioButton Style="{StaticResource style1}" Content="Option1" IsChecked="True"/>
<RadioButton Style="{StaticResource style1}" Content="Option2" />
<CheckBox Style="{StaticResource style1}" Content="Check1" IsChecked="True"/>
```

In the previous example, you can see that you can also assign complex objects by expanding the **Setter.Value** property to assign a **LinearGradientBrush** to the Background of the button. However you can create the brush as a separate resource if you wish to use outside of the style or use it in more than one style. Order is important so if a style depends on the brush, then you must make sure that the brush is already added into the resource dictionary before the style.

Resource binding between resources

```
<LinearGradientBrush x:Key="brush1" StartPoint="0,0" EndPoint="0,1">
    <GradientStop Color="SteelBlue" Offset="0" />
    <GradientStop Color="LightSteelBlue" Offset="0.25" />
    <GradientStop Color="LightSteelBlue" Offset="0.75" />
    <GradientStop Color="SteelBlue" Offset="1" />
</LinearGradientBrush>
<Style x:Key="style1" TargetType="Control">
    <Setter Property="FontFamily" Value="Georgia" />
    <Setter Property="FontStyle" Value="Italic" />
    <Setter Property="FontSize" Value="16" />
    <Setter Property="Width" Value="128" />
    <Setter Property="Height" Value="32" />
    <Setter Property="Margin" Value="4" />
    <Setter Property="BorderBrush" Value="SteelBlue" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="Background" Value="{StaticResource brush1}" />
</Style>
```

You can also create a new style based on an existing style to inherit the existing setters and adding new setters. You can override existing setters. You can bind to an existing style using the **BasedOn** property. **TargetType** may be same or different as long as the properties do exist on the type.

Inherit setters using BasedOn style

```
<Style x:Key="style2" TargetType="Button"
   BasedOn="{StaticResource style1}">
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="FontStyle" Value="Normal"/>
    <Setter Property="FontWeight" Value="Bold"/>
    <Setter Property="FontSize" Value="24"/>
</Style>
```

2.4 Resource Dictionaries

XAML can be used to serialize any kind of object not only UI elements. Thus it is also possible to serialize resources or entire resource dictionaries by using XAML as well. A large XAML document will be difficult to edit so if you have too many resources, you should create the resource dictionary in a separate document. In this way not only do you reduce the size of your main XAML document but it will also make it easier for you to edit and reuse resources across projects. You can now add a *Resource Dictionary* named **Default.xaml** in the project under the **Themes** folder and move all the application resources into the new resource dictionary.

Separate resource dictionary: Themes\Default.xaml

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
                :
</ResourceDictionary>
```

To use an external resource dictionary, expand **Resources** property to assign a new **ResourceDictionary** instance and set the **Source** property to point to the external document. You can see that the following content is much reduced by moving resources into a separate document. Even though a Resources property can only be assigned one resource dictionary, a **MergedDictionaries** property is available on a ResourceDictionary to merge all resources from a collection of dictionaries.

Assigning resource dictionaries: MyApp.xaml

```
<Application.Resources>
    <ResourceDictionary Source="Themes/Default.xaml"/>
</Application.Resources>
```

Merging resources from a collection of dictionaries

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Themes/Default.xaml"/>
        <ResourceDictionary Source="Themes/MyResources1.xaml />
        <ResourceDictionary Source="Themes/MyResources2.xaml />
    </Resource.Dictionary.MergedDictionaries>
</ResourceDictionary>
```

2.5 Using Themes

Themes can be distributed in XAML files as resource dictionaries or precompiled into assemblies. Sometimes even though themes are distributed in XAML, it may require you to reference additional assemblies containing types used within the XAML file. There are a number of themes available on the WPF Toolkit site. WPF Toolkit provides additional controls that was not included in the last release of WPF but may be integrated in a future release. Even if you do not wish to use it, you have to install it if you intent to use themes downloaded from the site as the XAML references the WPF Toolkit assembly. You will be using the controls in the toolkit later on so you may as well install it so we can start using any of the themes provided. You can either download both the toolkit and the themes from the following site or use the files provided by the instructor. Basically a theme is a collection of styles that replaces the default **Template** of the control. Through **TemplateBinding** changes to a control will then affect elements within the template. Special elements like **ContentPresenter** and **ItemsPresenter** will be used in the templates to display any content for the controls that are either ContentControl or ItemsControl.

Official site of the WPF Toolkit

http://wpf.codeplex.com

Using theme dictionaries: MyApp.xaml

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Themes/ExpressionDark.xaml" />
                    :
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

WPF comes along with a basic set of themes distributed in pre-compiled assemblies. The following is a list of assemblies and the name of the XAML files embedded inside the assembly.

Name of theme assemblies and the XAML theme files

```
PresentationFramework.Aero      aero.normalcolor.xaml
PresentationFramework.Classic   classic.xaml
PresentationFramework.Luna      luna.homestead.xaml
                                luna.metallic.xaml
                                luna.normalcolor.xaml
PresentationFramework.Royale    royale.normalcolor.xaml
```

If you want to reference assemblies from the GAC rather than attaching the assembly with the application, you also need to know the version number and the public key token. You can then combine the following parts shown below to form source for the resource dictionary.

Required parts for referencing resources in a shared assembly

```
Assembly name    : PresentationFramework.Aero;
Version          : V4.0.0.0;
Public key token: 31bf3856ad364e35;
Relative URL     : component/themes/aero.normalcolor.xaml
```

Merging resource dictionaries from external assemblies

```
<ResourceDictionary>
     <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="/PresentationFramework.Aero;
V4.0.0.0;31bf3856ad364e35;component/themes/aero.normalcolor.xaml" />
                              :
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

You can fetch styles from resource dictionaries just like any resource and then assign them directly to the **Style** property of UI elements. For example we can add styles for each view host.

Adding styles for view hosts: Themes/Default.xaml

```
<Style x:Key="SideRegionItemStyle" TargetType="Expander"></Style>
<Style x:Key="MainRegionItemStyle" TargetType="TabItem" ></Style>
```

We can then declare fields to assign the styles so that they do not have to be fetched again each we need to use them. In the shell window constructor you can add code to fetch the styles and assign to the following fields. Then update the **Attach** method to assign the style to each view host that we create for the view.

Fields to maintain information for view hosts: Shell.xaml.cs

```
private Style _sideRegionItemStyle;
private Style _mainRegionItemStyle;
```

Fetching styles and assign to fields

```
public Shell() {
    _instance = this;
    InitializeComponent();
    _sideRegionViews = new Dictionary<BaseView, Expander>();
    _mainRegionViews = new Dictionary<BaseView, TabItem >();
    _sideRegionItemStyle = (Style)FindResource("SideRegionItemStyle");
    _mainRegionItemStyle = (Style)FindResource("MainRegionItemStyle");
}
```

Assign style to elements directly in code

```
public void Attach(BaseView view) {
    switch (view.Region) {
        case Region.SideRegion:
            var host1 = new Expander();
            _sideRegionViews.Add(view, host1);
            host1.Style = _sideRegionItemStyle;
                    :
            break;
        case Region.MainRegion:
            var host2 = new TabItem();
            _mainRegionViews.Add(view, host2);
            host2.Style = _mainRegionItemStyle;
                    :
            break;
    }
}
```

3 Media

3.1 MediaElement

You can use **MediaElement** to play audio and video files. By default the media will start playing immediately but you can set **LoadBehavior** to **Manual** to control the media by writing code. The following shows the UI for a simple media player.

Using MediaElement to play audio file: Media1\MainWindow.xaml

```
<DockPanel>
    <StackPanel DockPanel.Dock="Top"Orientation="Horizontal">
        <Button x:Name="btnPlay" Content="Play" Click="btnPlay_Click"/>
        <Button x:Name="btnPause" Content="Pause" Click="btnPause_Click"/>
        <Button x:Name="btnStop" Content="Stop" Click="btnStop_Click"/>
        <Button x:Name="btnMute" Content="Mute" Click="btnMute_Click"/>
    </StackPanel>
    <MediaElement x:Name="media1" Source="C:\WPFDEV\RSC\Wild Horses.mp3"
        MediaOpened="media1_MediaOpened" MediaFailed="media1_MediaFailed"
        MediaEnded="media1_MediaEnded" LoadedBehavior="Manual" />
</DockPanel>
```

Controlling media from code

```
private void btnPlay_Click(object sender, RoutedEventArgs e) { media1.Play(); }
private void btnStop_Click(object sender, RoutedEventArgs e) { media1.Stop(); }
private void btnPause_Click(object sender, RoutedEventArgs e) {
    if (media1.CanPause) media1.Pause(); }
private void btnMute_Click(object sender, RoutedEventArgs e) {
    media1.IsMuted = !media1.IsMuted; }
```

Using MediaElement to play video file

```
<MediaElement x:Name="media1" Source="C:\WPFDEV\RSC\Wild Horses.mp4"
    MediaOpened="media1_MediaOpened" MediaFailed="media1_MediaFailed"
    MediaEnded="media1_MediaEnded" LoadedBehavior="Manual"/>
```

There are additional **Volume** and **Balance** properties to control the volume and the balance across multiple speakers. Volume will be from 0 to 1 and Balance is from -1 to +1. You can allow a user to control the volume and balance without writing any code. Use **Slider** UI element that a user can adjust to set the value. You can then use an element-to-element binding expression to bind **Volume** or **Balance** property of the MediaElement to the **Value** property of the Slider.

Using Slider to control volume

```
<Slider x:Name="hslVolume" Width="200" Value="0.5"
    Minimum="0.0" Maximum="1.0" LargeChange="0.1" SmallChange="0.01"
    TickFrequency="0.1" TickPlacement="Both"/>
```

Binding expression to bind MediaElement.Volume to Slider.Value

```
<MediaElement x:Name="media1"
    Source="C:\WPFDEV\RSC\Wild Horses.mp4"
    Volume="{Binding Path=Value,ElementName=hslVolume}"
                :
```

3.2 VisualBrush

You can use MediaElement as a brush. In fact you can use any UI element as a brush by using **VisualBrush**. VisualBrush has a **Visual** property that to assign a single UI element to be used for painting.

Using VisualBrush to paint with video: Media2\MainWindow.xaml

```
<Window x:Class="Media2.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="MainWindow" MinWidth="300" MinHeight="300">
    <Grid Background="Black">
        <Ellipse Width="300" Height="300">
            <Ellipse.Fill>
                <VisualBrush>
                    <VisualBrush.Visual>
                        <MediaElement
                            Source="C:\WPFDEV\RSC\Wild Horses.mp4"/>
                    </VisualBrush.Visual>
                </VisualBrush>
            </Ellipse.Fill>
        </Ellipse>
        <Ellipse Fill="Black" Width="100" Height="100"/>
    </Grid>
</Window>
```

3.3 Transparency & Alpha-Blending

WPF uses a 32-bit color system where there are four channels; alpha, red, green and blue where alpha controls the transparency and the other three is for the color. Since SolidColorBrush, LinearGradientBrush and RadialGradientBrush uses colors, brushes can support transparency. Anything painted using a partial transparent brush will be blended with the background.

Using opaque and transparent colors in brush: Transparency1\MainWindow.xaml

```
<Grid Background="Blue">
    <MediaElement x:Name="VideoSource" IsMuted="True"
        Source="C:\WPFDEV\RSC\Wild Horses.mp4" />
    <Rectangle>
        <Rectangle.Fill>
            <RadialGradientBrush GradientOrigin="0.5,0.5">
                <GradientStop Color="#00FFFFFF" Offset="0" />
                <GradientStop Color="#80FFFFFF" Offset="0.2" />
                <GradientStop Color="#FFFFFFFF" Offset="0.6" />
                <GradientStop Color="#80000000" Offset="0.75" />
                <GradientStop Color="#FF000000" Offset="1" />
            </RadialGradientBrush>
        </Rectangle.Fill>
    </Rectangle>
</Grid>
```

Alpha-blending is already supported for many 32-bit image formats like TGA and PNG. Even if you are not using transparent colors or 32-bit images, every UI element has an **Opacity** property that defaults to 1 which means 100% opaque that is reducable to be partly transparent. The element is entirely invisible when the value is 0.

Using Opacity for transparency

```
<MediaElement x:Name="VideoSource" IsMuted="True"
    Source="C:\WPFDEV\RSC\Wild Horses.mp4"
    Opacity="0.5" />
```