module 15

wpf resources, styles, control templates & animation

INTRODUCTION

Building on a basic knowledge of Windows Presentation Foundation, this module explains how to build advanced graphical applications. It covers ideal application structure as well as graphical customization and animation.

OUTCOMES

By the end of this module, you should be able to do the following:

- ***Create a well-organised WPF application with customized visual components.***
- ***Explain how to define and use WPF Styles and the difference between WPF Styles and CSS Styles.***
- ***Explain the difference between StaticResource and DynamicResource.***
- ***Define a custom Control Template.***

module 15: resources, styles, control templates & animation

15.1 – Property values

Property values on WPF controls are similar to their Win Forms or Web Forms counterparts. A control has a number of properties which control the look and functionality of the controls (background colour, screen position, etc.). The property values of controls can be set either in the designer, or programmatically in code. The properties can also be altered with event triggers or over time via animations.

15.2 – Styles

XAML gives you the ability to theme and style your entire application easily. You can define application wide defaults for buttons, list boxes, etc. You could set the property element of each control individually, but if you want them to share similar properties you can create a style which can be applied to them with the properties set.

You can start off with everything a matte lime green, and then decide to make everything glossy peach. This is done easily by changing the XAML, no code changes are required. It is similar to CSS in how you can set up the style piece by piece, and use inheritance to build on top of styles already defined.

Extending Styles is similar to subclassing in C#/VB. You may want buttons and text boxes to have the same font size and colour, but would like the button to have some additional properties to specify a gradient across the background. This can be done by first specifying one style which has the Font properties, and then creating another style that is based on that one which then specifies the background property.

When defining a style, a key is always needed. If one is not specified, then the key name will be defaulted to the **TargetType** e.g. `TargetType="Button"` will implicitly set the key to be `x:Name=Typeof(Button)`. When this happens the style also becomes the default style for that control type. Explicitly setting the key is as simple as defining `x:Key="MyStyle"`. All key names must be unique.

15.3 – Control Templates

Styles allow for changing properties on a control: font size, colour, etc. Templates differ by allowing you to override how a control renders itself.

Typically Buttons have a **ContentPresenter** nested inside their container which is used to display the content of the button (the button text). However you may want to create a new style of button that has an image on the left hand side and its text on the right hand side. Traditionally this type of functionality required the creation of a brand new custom control class to be created.

In WPF most controls have an appearance and a behaviour which are kept separately. If the behaviour of a control is appropriate (say the clicking functionality of a button) but you want to change the appearance then you can use a control template to define how the control should display itself whilst keeping the functionality provided. XAML allows you to define a new template for the button and define what should be created *inside* of it, saving the need to get into code and writing a brand new class.

15.3.1 – Triggers / Event Triggers

Triggers happen when a property value changes on a control, this is commonly used for simple things like changing the look of a control when a user is interacting with it, For example, when the control is selected, change the background colour to a dark colour.

EventTriggers are akin to responding to events firing in Win Forms. However instead of writing the code to handle it in C#/VB we can now write XAML code to respond to this, which can allow us to do animate the UI in response to events.

15.3.2 – Animation

WPF provides a framework for animating elements of the screen. Traditionally this required the use of rendering loops and methods dedicated to tracking where all the screen objects were. Now with WPF it is simply a matter of describing what an element should do over a time period. These animations can be activated from triggers in XAML or in C#/VB.

A wide variety of animations can be created easily in the XAML code, allowing a designer to worry about how screen elements react to the user completely independently of the programmer who is writing the code behind the scenes.

A designer can animate screen elements to slide in and out of view when they are activated, to slowly change colour when a MouseOver event occurs, or just have an animated company logo on the screen.

There are many elements that can be used to animate properties, which one you need to use is based on the value type of the property. Some common ones are

- ByteAnimation
- ColorAnimation
- DecimalAnimation
- DoubleAnimation
- Int(16/32/64)Animation
- ThicknessAnimation

For example, animating a brush's colour would require ColourAnimation, but a Margin would require ThicknessAnimation.

15.3.3 – Storing resources

WPF stores resources directly in XAML. Resources can be defined in a Page or Window, in the App.xaml file, or in an external ResourceDictionary.xaml file. If resources are defined in an external Resource dictionary, then the page or window that wants to reference it simply needs to add it as a Resource location.

Styles can also be defined inline, and can be scoped at the element, window, page or application level. For example, you could define a style which is only accessible within a StackPanel, or just in one window, or can be accessed from the entire application.

When searching for styles to apply, WPF will take the most relevant one found. It will start at the control element, and then slowly move up and up until it finds the first one which can be applied (The control level, the parent control, the parent control, the page, the application)

- Resources can be defined as being **Static** or **Dynamic**.
  StaticResources are loaded once into memory at runtime and cannot be modified afterwards.
- **DymanicResources** can be modified after being loaded, however they take up more resources.

For 90% of purposes, using StaticResources will suffice.

module 15: hands on lab

Exercise 1

This exercise will demonstrate how to take an existing WPF application and refactor the styles into an external dictionary. We will also touch on control templates.

Styles

1. Open the Start project for Exercise 1 and then open Window1.Xaml in code. You can see that whoever built this app (aside from them having really bad taste) applied all the styling directly to the control's properties. The same properties have been set on every label. We are going to extract these and place them into a external dictionary which can be used on all the controls in this window and reused by controls in other windows.
2. Start by adding a new resource dictionary to the project. Right click the project in the Solution Explorer, add a new item, then select **ResourceDictionary**. Name it `ReadifyDictionary.Xaml`.
3. Open App.Xaml. Specify that the application should use the **ReadifyDictionary** when looking for styles and templates.

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

4. Begin by stripping the style information from the slider.

```
<Slider x:Name="slider1" Margin="5" Value="2"
Background="#FFE92D2D" />
```

5. Create a new style in ReadifyDictionary.Xaml and specify the background property (feel free to use a different colour if you want).
   (**NOTE**: The following code wraps because the text is too long – make sure you put all of the text for each namespace on a single line)

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

    <Style TargetType="Slider" x:Key="ReadifySlider">
        <Setter Property="Background" Value="#FFE92D2D"/>
    </Style>

</ResourceDictionary>
```

6. Change both the sliders to now use this style

```
<Slider x:Name="slider1" Margin="5" Value="2"
```

```
Style="{StaticResource ReadifySlider}" />
<Slider Grid.Row="1" x:Name="slider2" Margin="5"
    Value="6" Style="{StaticResource ReadifySlider}" />
```

7. Repeat the process with the **ProgressBar**. Strip out the style information from the control and create a new style in the Resource Dictionary.
8. Create a new style in the Resource Dictionary

```
<Style TargetType="ProgressBar" x:Key="ReadifyProgressBar">
    <Setter Property="Foreground" Value="#FFE43CED" />
    <Setter Property="Background" Value="#FFF9B5F3" />
    <Setter Property="Margin" Value="10" />
</Style>
```

9. Apply the style to the progressbar

```
<ProgressBar x:Name="progressBar1" Grid.Row="2" Value="75"
Style="{StaticResource ReadifyProgressBar}" />
```

10. Finally do the same for the Labels. Create a single style called **ReadifyLabel**. Much of this can be copy/pasted from the original label styling

```
<Style TargetType="Label" x:Key="ReadifyLabel">
    <Setter Property="FontFamily" Value="Segoe Script"/>
    <Setter Property="FontSize" Value="20"/>
    <Setter Property="Background">
        <Setter.Value>
            <LinearGradientBrush EndPoint="0.5,1"
                StartPoint="0.5,0">
                <GradientStop Color="#FFFFFFFF"
                    Offset="0"/>
                <GradientStop Color="#FF00FFDE"
                    Offset="1"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

11. Apply this style to each of the labels

```
<Label Grid.Column="0" Grid.Row="0" x:Name="lblName"
Content="Name" Style="{StaticResource ReadifyLabel}" />

<TextBox Grid.Column="1" Grid.Row="0" x:Name="txtName" />
<Label Grid.Column="0" Grid.Row="1" x:Name="lblAge"
Content="Age" Style="{StaticResource ReadifyLabel}" />

<TextBox Grid.Column="1" Grid.Row="1" x:Name="txtAge" />
<Label Grid.Column="0" Grid.Row="2" x:Name="lblHeight"
Content="Height" Style="{StaticResource ReadifyLabel}" />

<TextBox Grid.Column="1" Grid.Row="2" x:Name="txtHeight" />
<Label Grid.Column="0" Grid.Row="3" x:Name="lblEyeColour"
Content="Eye colour" Style="{StaticResource
ReadifyLabel}" />
```

```
<TextBox Grid.Column="1" Grid.Row="3" x:Name="txtEyeColour"
/>
```

Control Template

Now we will modify the Control Template of the Button, to alter its look. The new button style has already been predefined in **ReadifyButtonDictionary.Xaml**.

12. Merge **ReadifyButtonDictionary** into ReadifyDictionary.xaml. Inside ReadifyDictionary.xaml add this block of code to the top of the file.

```
<ResourceDictionary.MergedDictionaries>
    <ResourceDictionary
Source="ReadifyButtonDictionary.xaml"/>
</ResourceDictionary.MergedDictionaries>
```

13. Make the button use this new Template. You can change the Background colour to something that suits you.

```
<Button Grid.Column="1" Grid.Row="2" x:Name="button1"
   Content="Submit"
   Template="{StaticResource ReadifyButtonControlTemplate}"
   Background="#FF9AFFD1" />
```

14. Congratulations, you now have a FishButton. Something every business app needs! But this demonstrates how you can change a button to look like anything that you want.

Exercise 2

This exercise will demonstrate how we can add animation effects to our controls. We will demonstrate how this can be done by editing a resource dictionary, meaning that no changes to the main application window need to be made. We will then demonstrate how animation can be triggered through code.

Adding animation to the ProgressBar

We will start with a simple exercise of just animating the ProgressBar when the user places their mouse over it.

1. Open the Start project for Exercise 2. and edit **ReadifyDictionary.Xaml**.
2. Add a **trigger** to the ProgressBar style. Do this by uncommenting the **MouseEnter** block of XAML defined in the trigger block. The XAML code specifies that something should happen when the MouseEnter event happens. We then specify the actions which should happen, in this case it is to begin an animation. We change the thickness of the margin property to a value of 20 over half a second.
3. Run the program and check that the ProgressBar now contracts when the mouse is placed over it. There is one problem here, the control shrinks but

does not return to its original size! We need to add a **MouseLeave** event to revert the property back to the original value.

4. Add the MouseLeave event to the ProgressBar. Uncomment the MouseLeave XAML block inside the **<Style.Triggers>** element
   This animation targets the Margin property again, and will take place over the course of one second. Notice that there isn't a "To" value set. Since this was omitted, the animation will return the control to its original value.

Animating the Label Control

This time we will animate the Label control when a user places their cursor over it. We will change the gradient colour of one of the points.

5. In the ReadifyDictionary.Xaml file uncomment the **<Style.Triggers>** block inside the Label style. Here we are doing a similar set of tasks as last time; two event triggers (MouseEnter/Leave) and changing the value and then returning it to its original value. The only tricky thing is that this time we are modifying a colour so we need to use **ColorAnimation** this time. Also notice that the property we are modifying is one of the points of the gradient. Now when the mouse is moved over one of the labels, the colours will change and fade in and out of the transition.

Animating the FishButton

The FishButton will now be upgraded to Hypno fish!

6. Open ReadifyButtonDictionary.xaml (because the fishbutton is defined in there)
7. Uncomment the animation block defined in **<ControlTemplate.Triggers>**. The Fill colour of the ellipse is changed but is taken through a number of transitions.

Animating an entire window from code

This time we will invoke an animation from code. We will animate the base grid object which all the other controls are nested inside to shake wildly if there was a validation error.

8. Open Window1.Xaml.
9. Update the definition of the root grid. This specifies a **RenderTransform** which can be used to modify the control as we need. (in this case, we will be modifying the **RotateTransform**)

```
<Grid RenderTransformOrigin="0.5,0.5" x:Name="grid">
    <Grid.RenderTransform>
        <TransformGroup>
            <ScaleTransform ScaleX="1" ScaleY="1"/>
```

```
            <SkewTransform AngleX="0" AngleY="0"/>
            <RotateTransform Angle="0"/>
            <TranslateTransform X="0" Y="0"/>
        </TransformGroup>
    </Grid.RenderTransform>
```

10. Uncomment the **<Window.Resources>** code block to define the window animation. This animation works on the Rotate transform property of the grid to rotate it back and forward around that point
11. Double click on the fish button to define a new click event
12. In Window1.xaml.cs place a **using** statement at the top

```
using System.Windows.Media.Animation;
```

13. Change the click event handler method to activate the animation if there is a validation error (if there isn't a value entered for name), or to play the other animation to reset the screen.

```
private void button1_Click(object sender,
    RoutedEventArgs e)
{
    Storyboard validationAnimation;

    if (txtName.Text == "")
        validationAnimation = (Storyboard)
             FindResource("ValidationErrorStoryboard");
    else
        validationAnimation = (Storyboard)
             FindResource("ValidationPassStoryboard");

    validationAnimation.Begin(this);
}
```

14. Try changing the **ValidationPassAnimation** to change everything to a light shade of green when validation passes.
15. Extend the solution by modifying **ValidationErrorStoryboard** to do crazier things link shrinking, expanding and zipping around.