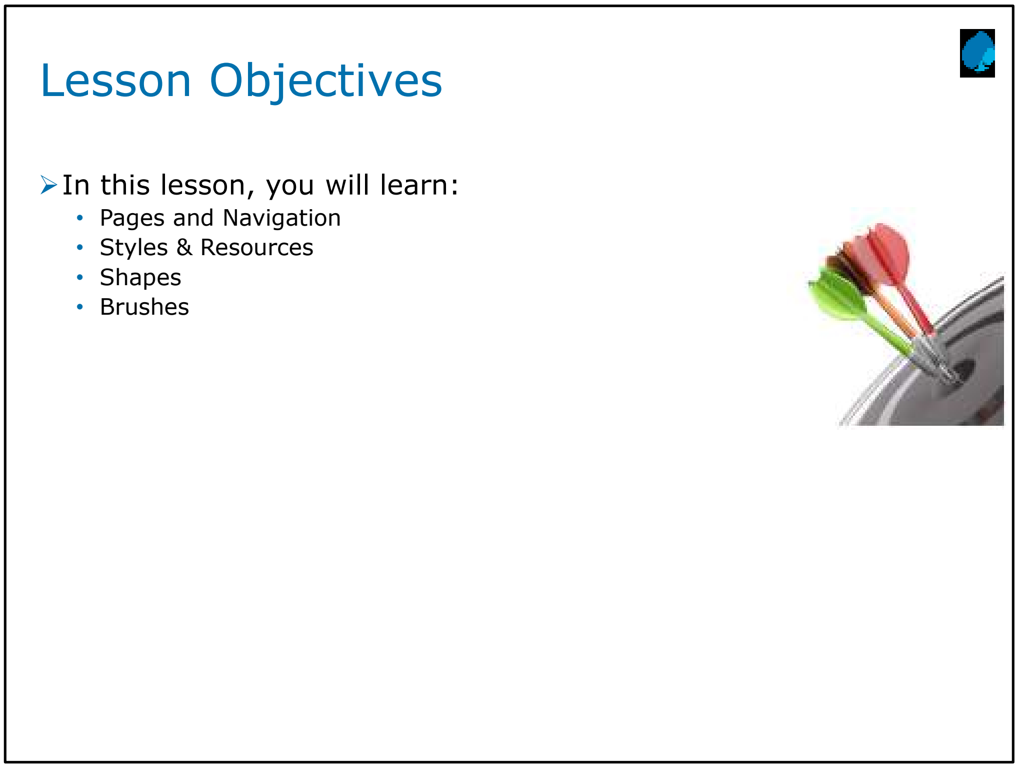
WPF 4.5

Lesson 4: Styles, Resources & Shapes

Capgemini



Lesson Objectives

- In this lesson, you will learn:
  - Pages and Navigation
  - Styles & Resources
  - Shapes
  - Brushes

4.1: Pages and Navigation

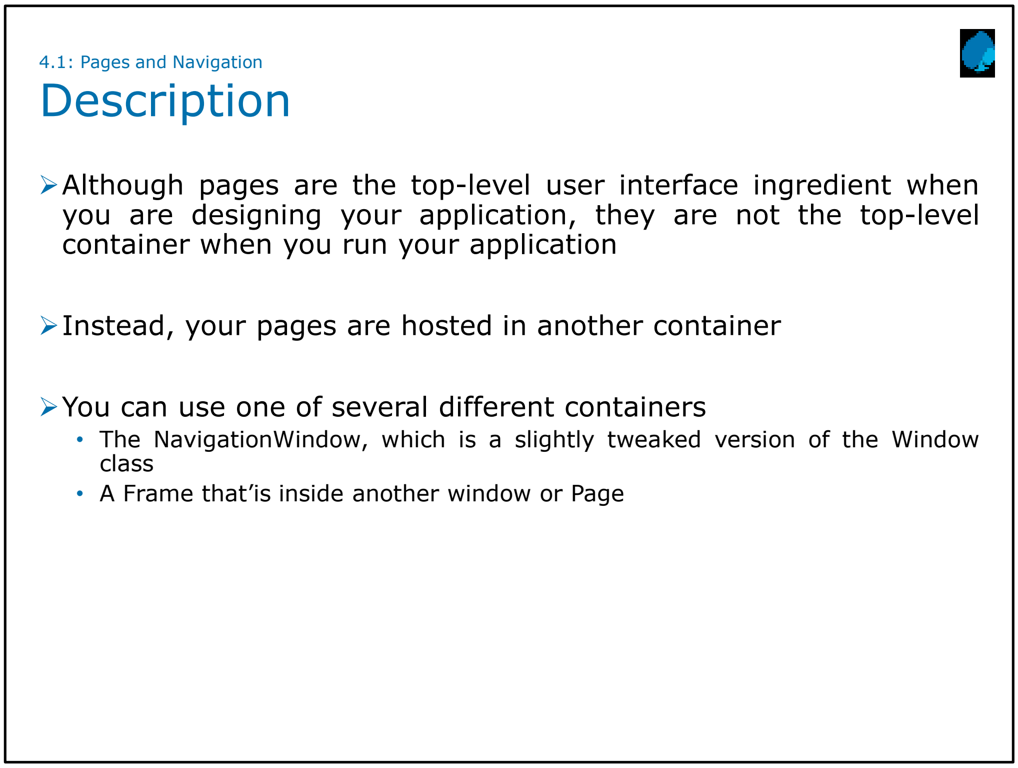
Description

- Most traditional Windows applications are arranged around a window that contains toolbars and menus
- The toolbars and menus drive the application—as the user clicks them, actions happen, and other windows appear
- In a bid to give desktop developers the ability to build web-like desktop applications, WPF includes its own page-based navigation system

4.1: Pages and Navigation

Description

- To create a page-based application in WPF, you need to stop using the Window class as your top-level container for user interfaces
- Instead, WPF provides a Page class available under System.Windows.Controls.Page
- You can add a page to any WPF project
- Just choose Project ➤ Add Page in Visual Studio



4.1: Pages and Navigation

Description

- Although pages are the top-level user interface ingredient when you are designing your application, they are not the top-level container when you run your application
- Instead, your pages are hosted in another container
- You can use one of several different containers
  - The NavigationWindow, which is a slightly tweaked version of the Window class
  - A Frame that'is inside another window or Page

4.1: Pages and Navigation

Code

```
<Page x:Class="NavigationApplication.Page1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presenta
tion"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
WindowTitle="Page1"
>
</Page>
```

4.1: Pages and Navigation

Description

- When you run this application, WPF is intelligent enough to realize that you are pointing it to a page rather than a window.
- It automatically creates a new NavigationWindow object to serve as a container and shows your page inside of it
- The NavigationWindow looks more or less like an ordinary window, aside from the back and forward navigation buttons that appear in the bar at the top

4.1: Pages and Navigation

Hyperlink

```
<TextBlock Margin="3" TextWrapping="Wrap">
This is a simple page.
Click <Hyperlink NavigateUri="Page2.xaml">here</Hyperlink> to go to
  Page2.
</TextBlock>
```

- The easiest way to allow the user to move from one page to another is using hyperlinks

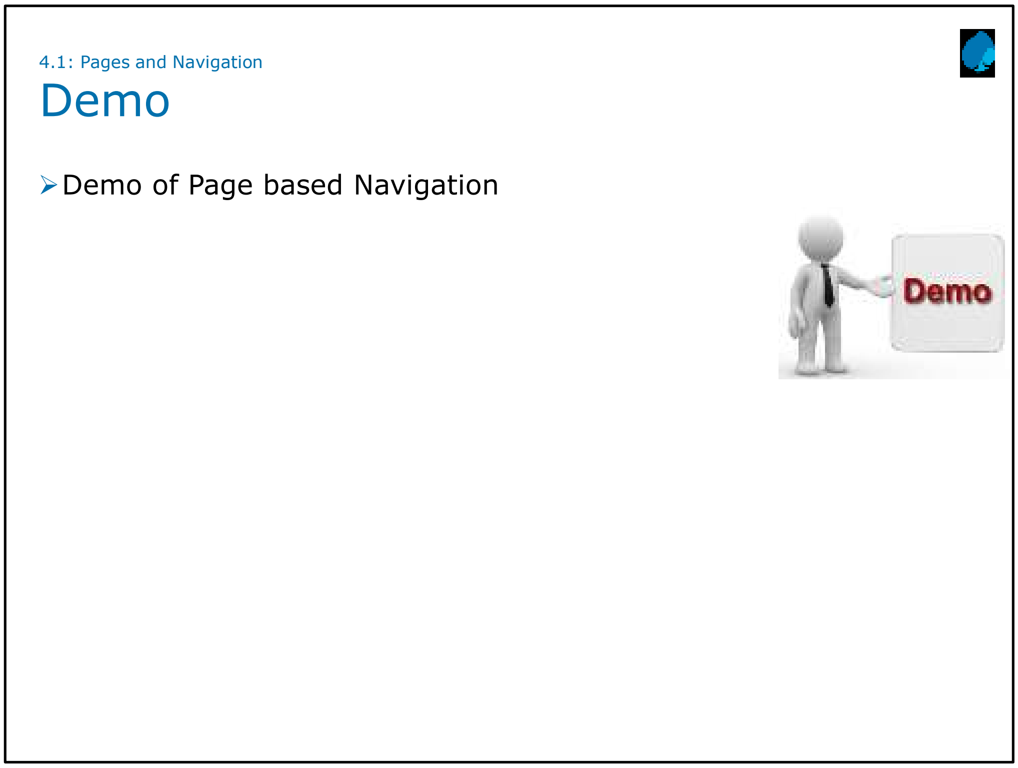
4.1: Pages and Navigation

The Navigation Service

- To allow programmatic navigation, the most flexible and powerful approach is to use the WPF navigation service

- You can access the navigation service through the static NavigationService.GetNavigationService() method

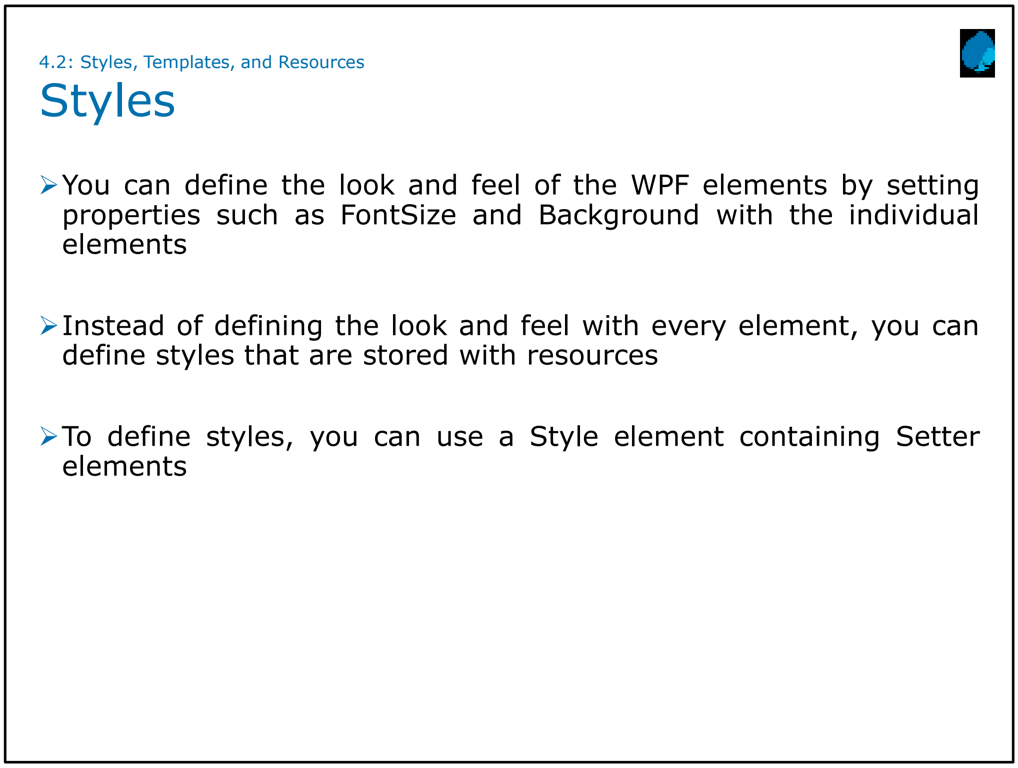
```
NavigationService nav;
nav = NavigationService.GetNavigationService(this);
Page2 nextPage = new Page2();
nav.Navigate(nextPage);
```



4.1: Pages and Navigation

Demo

- Demo of Page based Navigation



4.2: Styles, Templates, and Resources

Styles

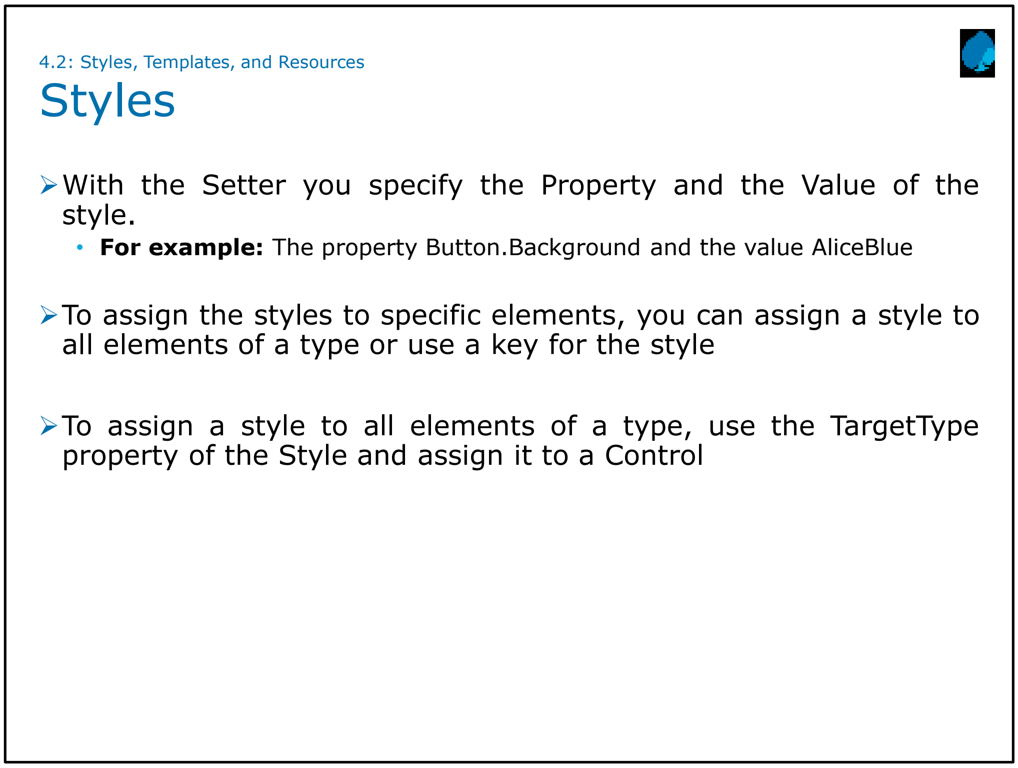
- You can define the look and feel of the WPF elements by setting properties such as FontSize and Background with the individual elements
- Instead of defining the look and feel with every element, you can define styles that are stored with resources
- To define styles, you can use a Style element containing Setter elements

WPF Design Principles: Styles, Templates, and Resources:
Styles:
In WPF, a style is also a set of properties applied to content used for visual rendering.

A style can be used to set properties on an existing visual element, such as setting the font weight of a Button control.
It can also be used to define the way an object looks, such as showing the name and age from a Person object.

In addition to the features in word processing styles, WPF styles have specific features for building applications, including the following:

The ability to associate different visual effects based on user events
Provide entirely new looks for existing controls
Designate rendering behavior for non-visual objects



4.2: Styles, Templates, and Resources

Styles

- With the Setter you specify the Property and the Value of the style.
  - **For example:** The property Button.Background and the value AliceBlue
- To assign the styles to specific elements, you can assign a style to all elements of a type or use a key for the style
- To assign a style to all elements of a type, use the TargetType property of the Style and assign it to a Control

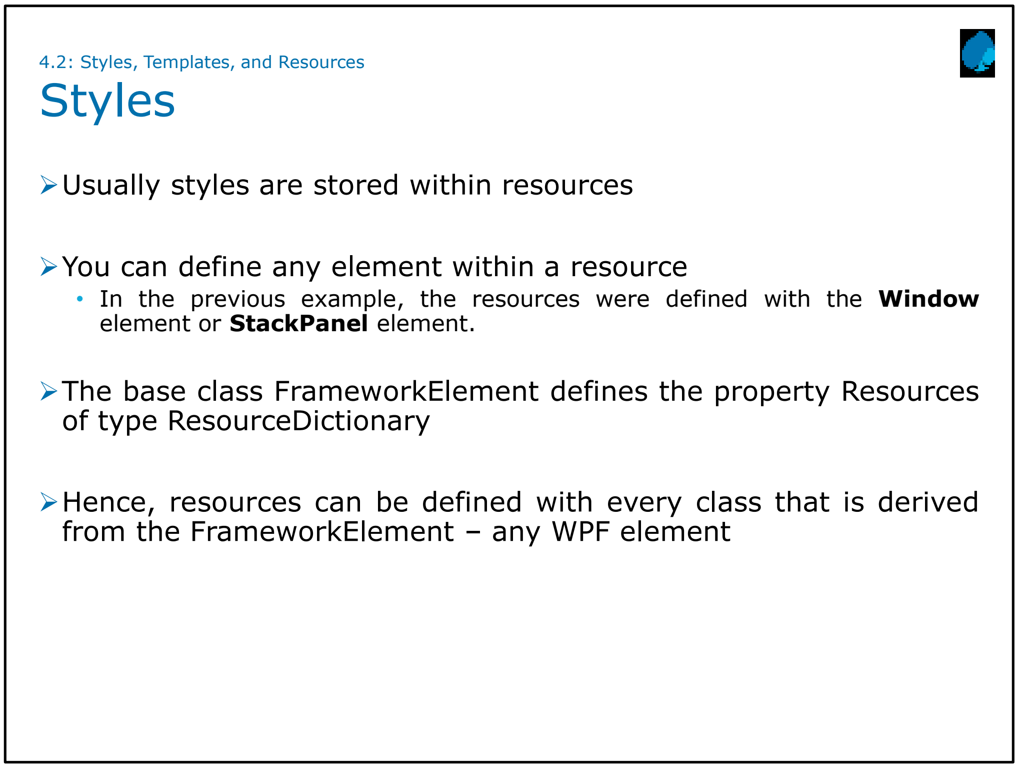
WPF Design Principles: Styles, Templates, and Resources:
Styles:
Let us see an example of setting a named style:

```
<Window ...>
 <Window.Resources>
   <Style x:Key="CellTextStyle">
        <Setter Property="Control.FontSize" Value="32" />
        <Setter Property="Control.FontWeight" Value="Bold" />
   </Style>
  </Window.Resources>
   ...
  <Button Style="{StaticResource CellTextStyle}" ...
x:Name="cell00" /> ... </Window>
```

4.2: Styles, Templates, and Resources

Demo

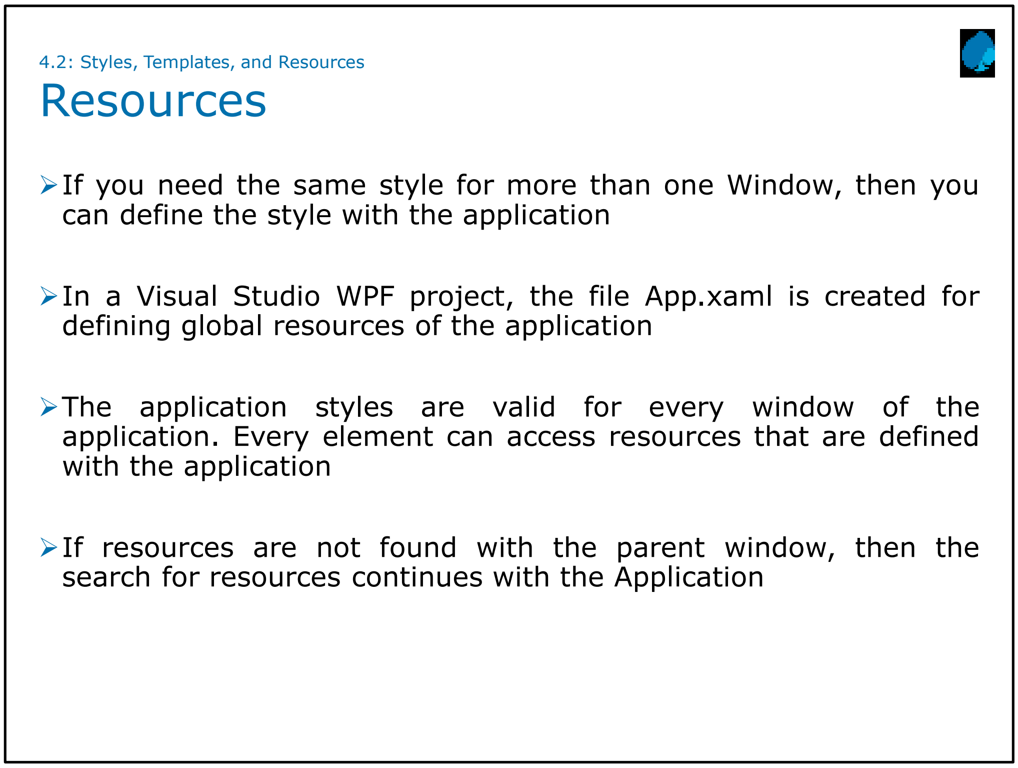
- Demo of styles



4.2: Styles, Templates, and Resources

Styles

- Usually styles are stored within resources
- You can define any element within a resource
  - In the previous example, the resources were defined with the **Window** element or **StackPanel** element.
- The base class FrameworkElement defines the property Resources of type ResourceDictionary
- Hence, resources can be defined with every class that is derived from the FrameworkElement – any WPF element



4.2: Styles, Templates, and Resources

Resources

- If you need the same style for more than one Window, then you can define the style with the application
- In a Visual Studio WPF project, the file App.xaml is created for defining global resources of the application
- The application styles are valid for every window of the application. Every element can access resources that are defined with the application
- If resources are not found with the parent window, then the search for resources continues with the Application

4.2: Styles, Templates, and Resources

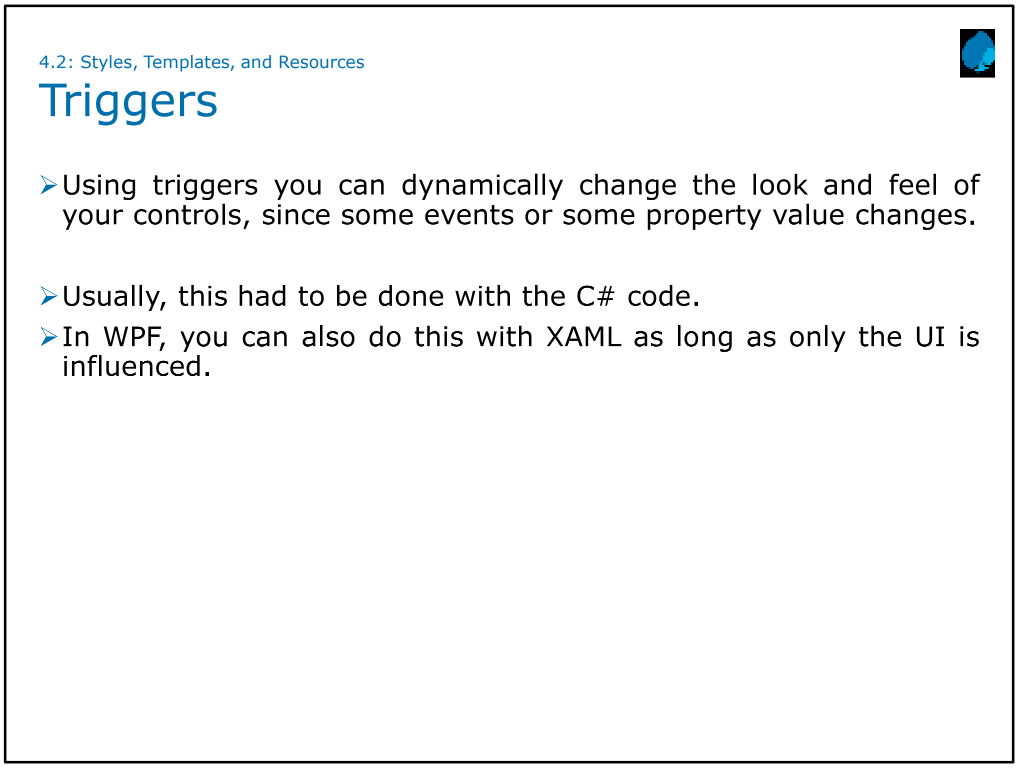
Dynamic Resources

- With the StaticResource markup extension, resources are searched at load time.
- If the resource changes while the program is running, then you should use the DynamicResource markup extension instead.

4.2: Styles, Templates, and Resources

Demo

- Demo of Resources

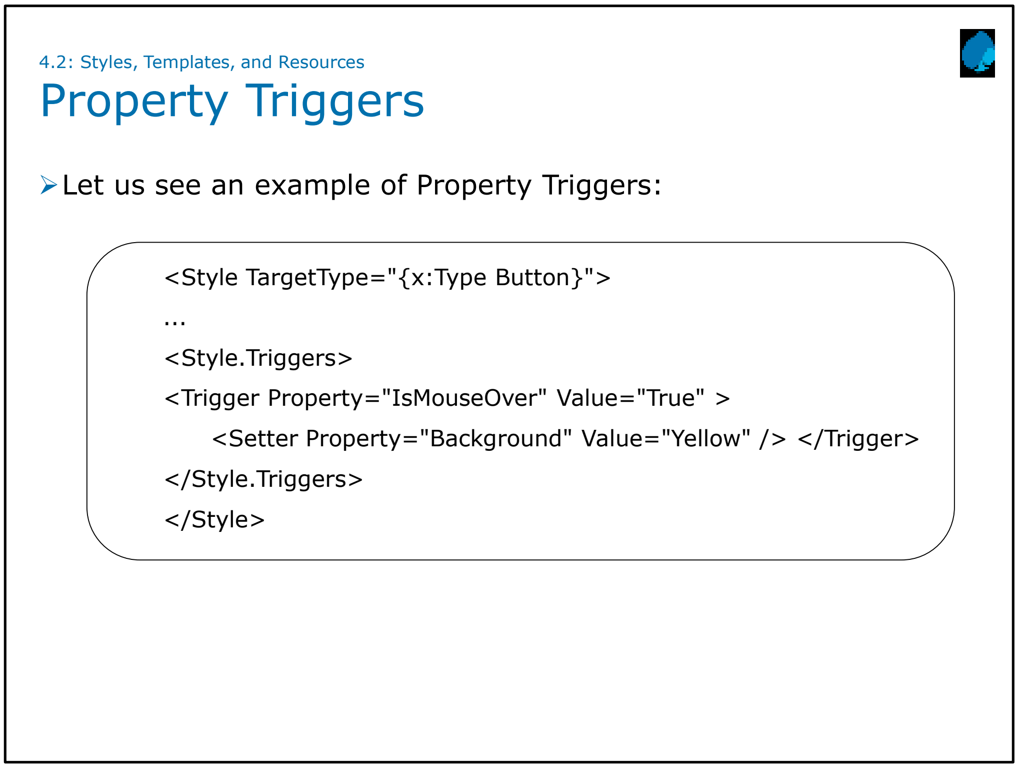


4.2: Styles, Templates, and Resources

Triggers

- Using triggers you can dynamically change the look and feel of your controls, since some events or some property value changes.
- Usually, this had to be done with the C# code.
- In WPF, you can also do this with XAML as long as only the UI is influenced.

WPF Design Principles: Styles, Templates, and Resources:
Triggers:
So far, we have seen styles as a collection of Setter elements. When a style is applied, the settings described in the Setter elements are applied unconditionally (unless overridden by per-instance settings).
On the other hand, triggers are a way to wrap one or more Setter elements in a condition so that, if the condition is true, the corresponding Setter elements are executed. Furthermore, when the condition becomes false, the property value reverts to its pre-trigger value.



4.2: Styles, Templates, and Resources

Property Triggers

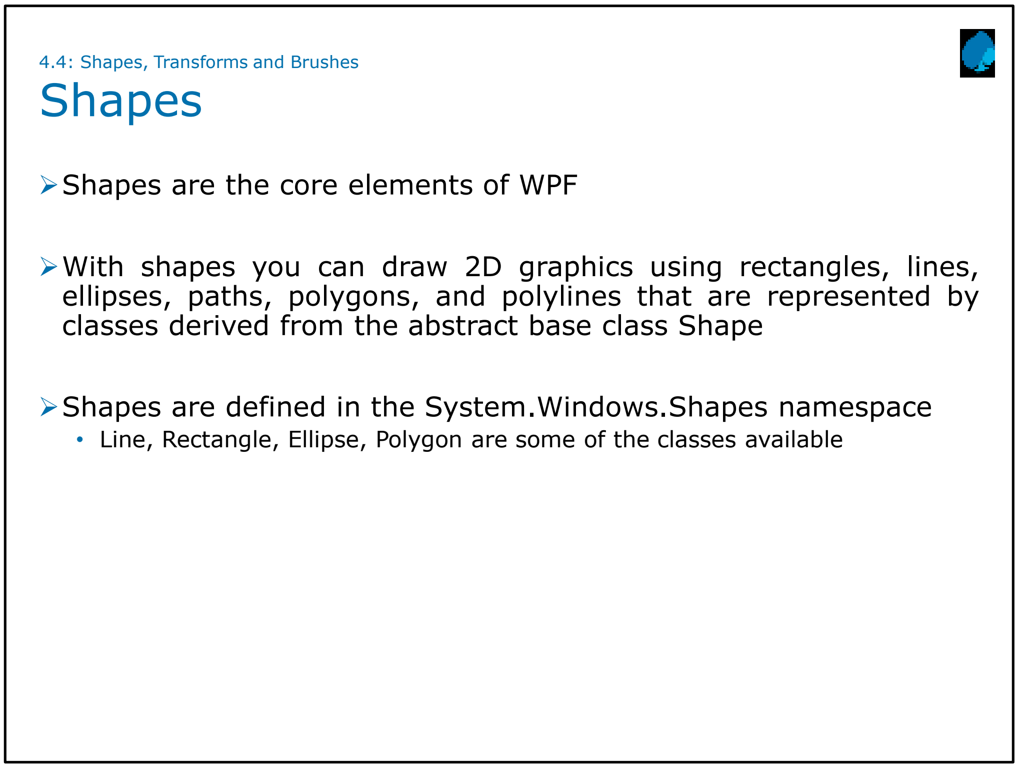
➢Let us see an example of Property Triggers:

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

WPF Design Principles: Styles, Templates, and Resources:
Property Triggers:
The simplest form of a trigger is a property trigger. It watches for a dependency property to have a certain value. For example, suppose we want to light up a button in yellow as the user moves the mouse over it. Then we can do so by watching for the IsMouseOver property to have a value of True, as shown in the example in the above slide.
Triggers are grouped together under the Style.Triggers element. In this case, we have added a Trigger element to the button style. When the IsMouseOver property of our button is true, the Background value of the button will be set to yellow.

4.2: Styles, Templates, and Resources

Demo

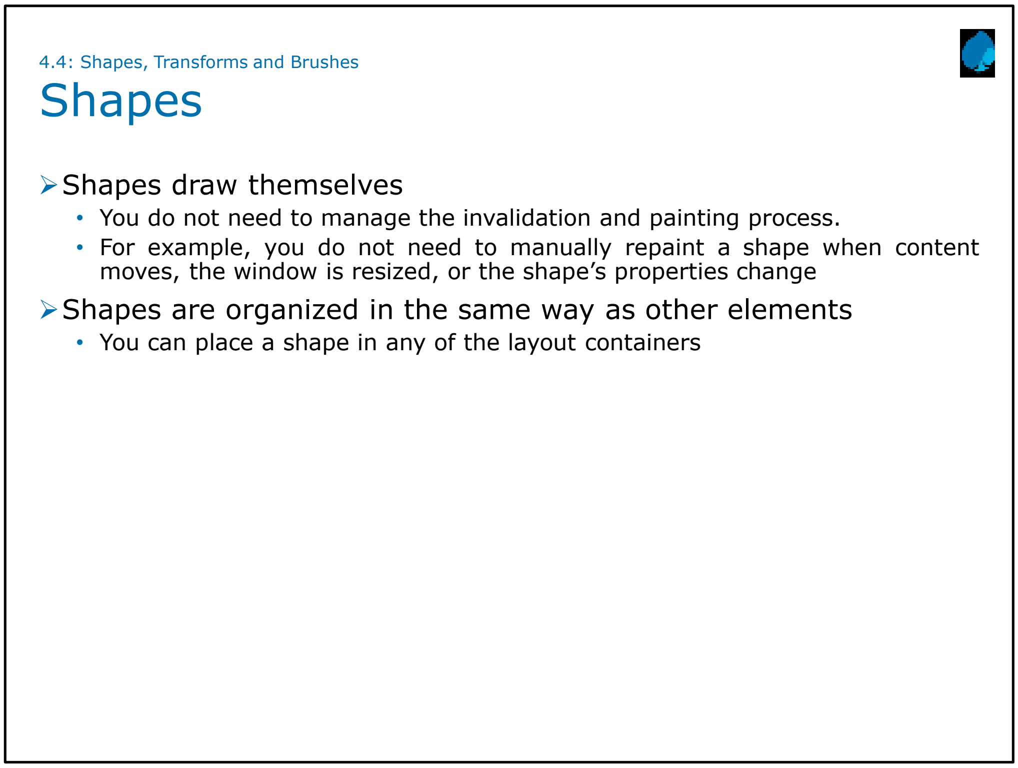


4.4: Shapes, Transforms and Brushes

Shapes

- Shapes are the core elements of WPF
- With shapes you can draw 2D graphics using rectangles, lines, ellipses, paths, polygons, and polylines that are represented by classes derived from the abstract base class Shape
- Shapes are defined in the System.Windows.Shapes namespace
  - Line, Rectangle, Ellipse, Polygon are some of the classes available

WPF Design Principles: Shapes, Controls, and Layouts:
Shapes:
Shapes are drawing primitives, represented as elements in the user interface tree. WPF supports a variety of different shapes and provides element types for each of them.
All of the elements listed in this slide derive from a common abstract base class, that is Shape. Although you cannot use this type directly, it is useful to know about it, because it defines a common set of features that you can use on all shapes. These common properties are all concerned with the way in which the interior and outline of the shape are painted.
The Fill property specifies the Brush that will be used to paint the interior. The Line and Polyline classes do not have interiors, so they will ignore this property. (This was simpler than complicating the inheritance hierarchy by having separate Shape and FilledShape base classes.) The Stroke property specifies the Brush that will be used to paint the outline of the shape.



4.4: Shapes, Transforms and Brushes

Shapes

- Shapes draw themselves
  - You do not need to manage the invalidation and painting process.
  - For example, you do not need to manually repaint a shape when content moves, the window is resized, or the shape's properties change
- Shapes are organized in the same way as other elements
  - You can place a shape in any of the layout containers

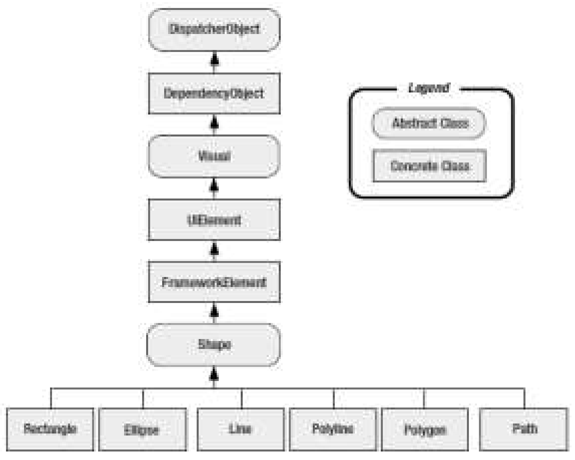
4.4: Shapes, Transforms and Brushes

The Shape Class

- Every shape derives from the abstract System.Windows.Shapes.Shape class
- Shape Class Properties
  - Fill: Sets the brush object that paints the surface of the shape (everything inside its borders)
  - Stroke: Sets the brush object that paints the edge of the shape (its border)
  - StrokeThickness: Sets the thickness of the border, in device-independent units

4.4: Shapes, Transforms and Brushes

The Shape Class



4.4: Shapes, Transforms and Brushes

Rectangle and Ellipse

- The Rectangle and Ellipse are the two simplest shapes
- To create either one, set the Height and Width properties to define the size of your shape, and then set the Fill or Stroke property (or both) to make the shape visible

```
<Ellipse Fill="Yellow" Stroke="Blue"
Height="50" Width="100" Margin="5" HorizontalAlignment="Left"></Ellipse>
<Rectangle Fill="Yellow" Stroke="Blue"
Height="50" Width="100" Margin="5" HorizontalAlignment="Left"></Rectangle>
```

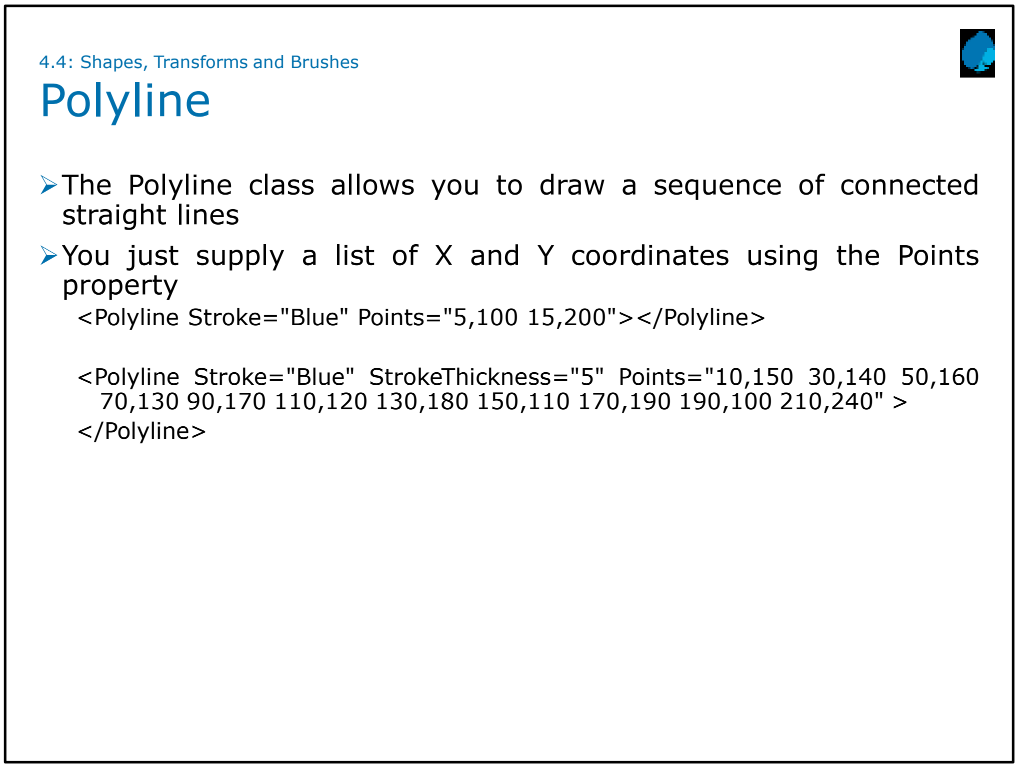
4.4: Shapes, Transforms and Brushes

Line

- The Line shape represents a straight line that connects one point to another
- The starting and ending points are set by four properties: X1 and Y1 (for the first point) and X2 and Y2 (for the second)

```
<Line Stroke="Blue" X1="0" Y1="0" X2="10" Y2="100"></Line>
```

  - The Fill property has no effect for a line. You must set the Stroke property

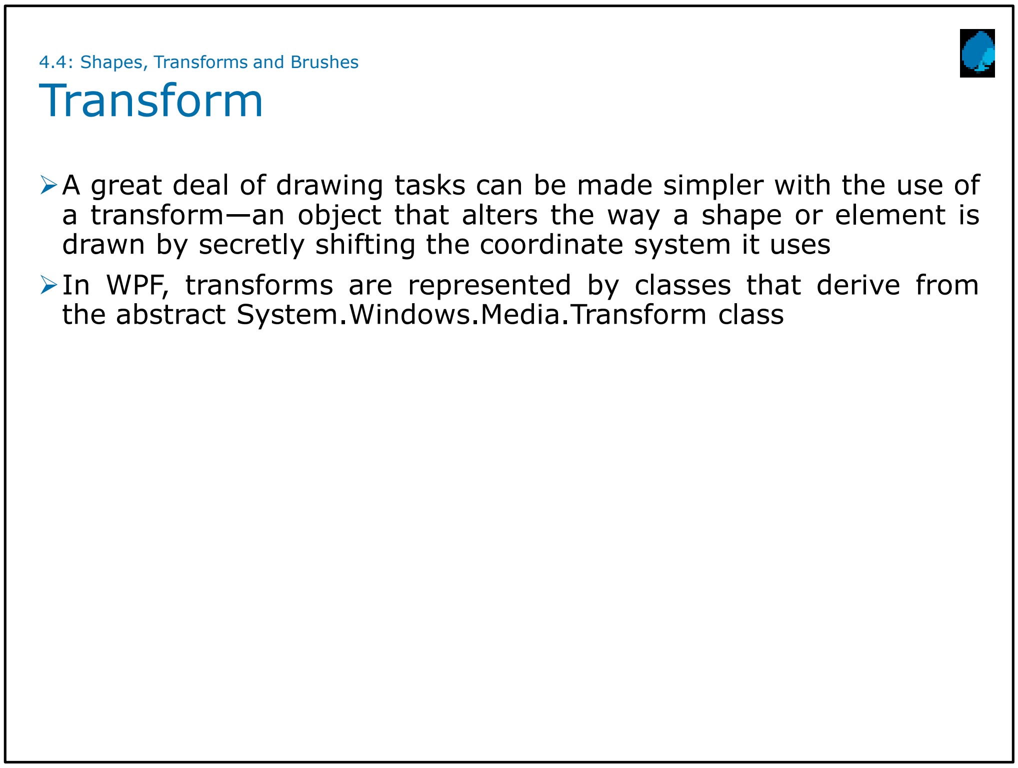


4.4: Shapes, Transforms and Brushes

Polyline

- The Polyline class allows you to draw a sequence of connected straight lines
- You just supply a list of X and Y coordinates using the Points property

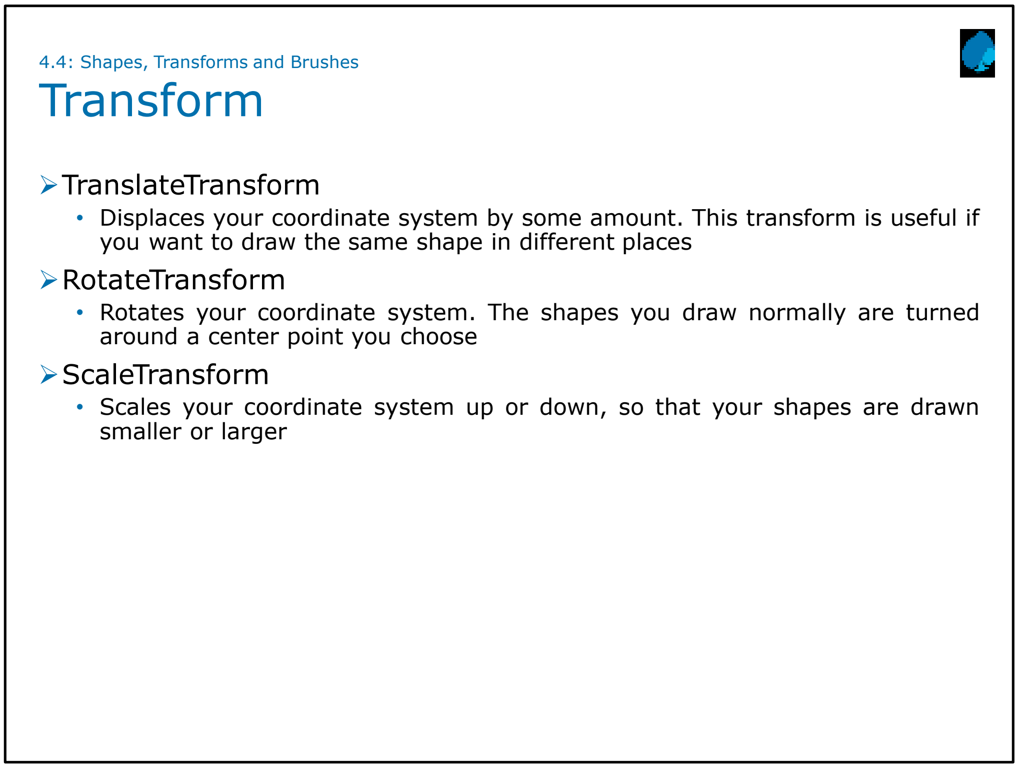
```
<Polyline Stroke="Blue" Points="5,100 15,200"></Polyline>

<Polyline Stroke="Blue" StrokeThickness="5" Points="10,150 30,140 50,160
  70,130 90,170 110,120 130,180 150,110 170,190 190,100 210,240" >
</Polyline>
```



4.4: Shapes, Transforms and Brushes

Transform

- A great deal of drawing tasks can be made simpler with the use of a transform—an object that alters the way a shape or element is drawn by secretly shifting the coordinate system it uses
- In WPF, transforms are represented by classes that derive from the abstract System.Windows.Media.Transform class



4.4: Shapes, Transforms and Brushes

Transform

- TranslateTransform
  - Displaces your coordinate system by some amount. This transform is useful if you want to draw the same shape in different places
- RotateTransform
  - Rotates your coordinate system. The shapes you draw normally are turned around a center point you choose
- ScaleTransform
  - Scales your coordinate system up or down, so that your shapes are drawn smaller or larger

4.4: Shapes, Transforms and Brushes

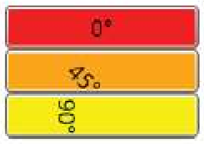
Transform

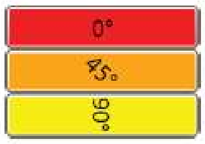
- SkewTransform
  - Warps your coordinate system by slanting it a number of degrees. For example, if you draw a square, it becomes a parallelogram

4.4: Shapes, Transforms and Brushes

Sample Code

```
<Button Background="Orange">
  <TextBlock RenderTransformOrigin="0.5,0.5">
  <TextBlock.RenderTransform>
    <RotateTransform Angle="45"/>
  </TextBlock.RenderTransform>
    45°
  </TextBlock>
</Button>
```

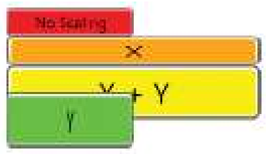


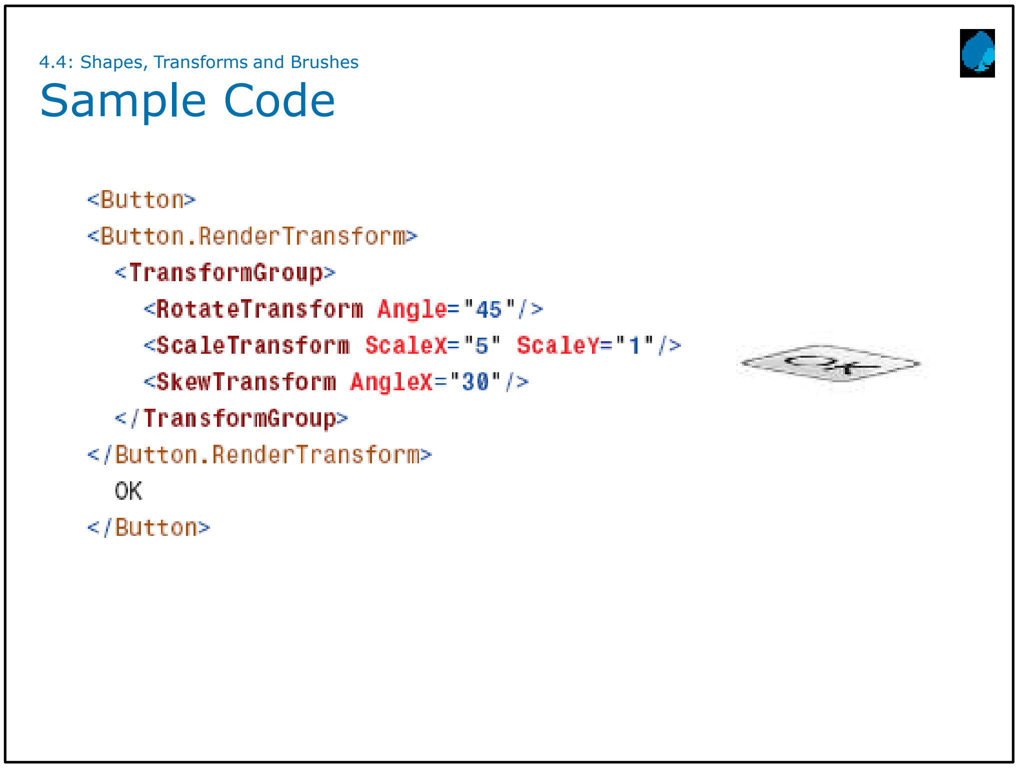


4.4: Shapes, Transforms and Brushes

Sample Code

```
<StackPanel Width="100">
  <Button Background="Red">No Scaling</Button>
  <Button Background="Orange">
  <Button.RenderTransform>
    <ScaleTransform ScaleX="2"/>
  </Button.RenderTransform>
    X</Button>
  <Button Background="Yellow">
  <Button.RenderTransform>
    <ScaleTransform ScaleX="2" ScaleY="2"/>
  </Button.RenderTransform>
    X + Y</Button>
  <Button Background="Lime">
  <Button.RenderTransform>
    <ScaleTransform ScaleY="2"/>
  </Button.RenderTransform>
    Y</Button>
</StackPanel>
```

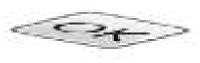




4.4: Shapes, Transforms and Brushes

Sample Code

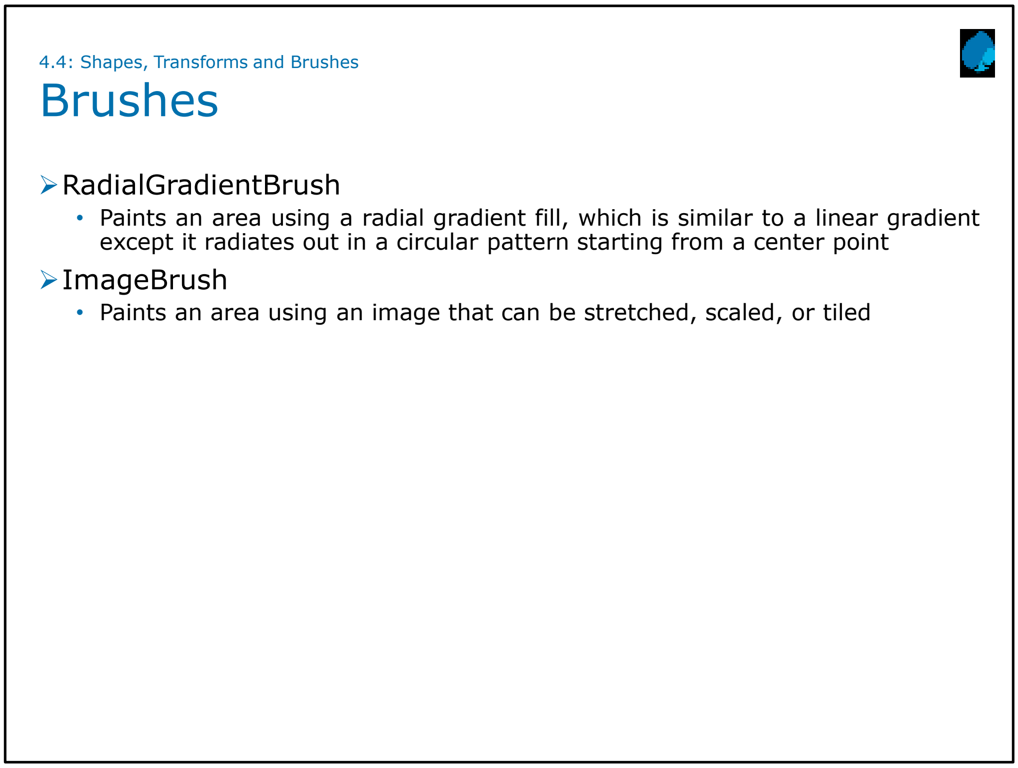
```
<Button>
<Button.RenderTransform>
  <TransformGroup>
    <RotateTransform Angle="45"/>
    <ScaleTransform ScaleX="5" ScaleY="1"/>
    <SkewTransform AngleX="30"/>
  </TransformGroup>
</Button.RenderTransform>
  OK
</Button>
```



4.4: Shapes, Transforms and Brushes

Brushes

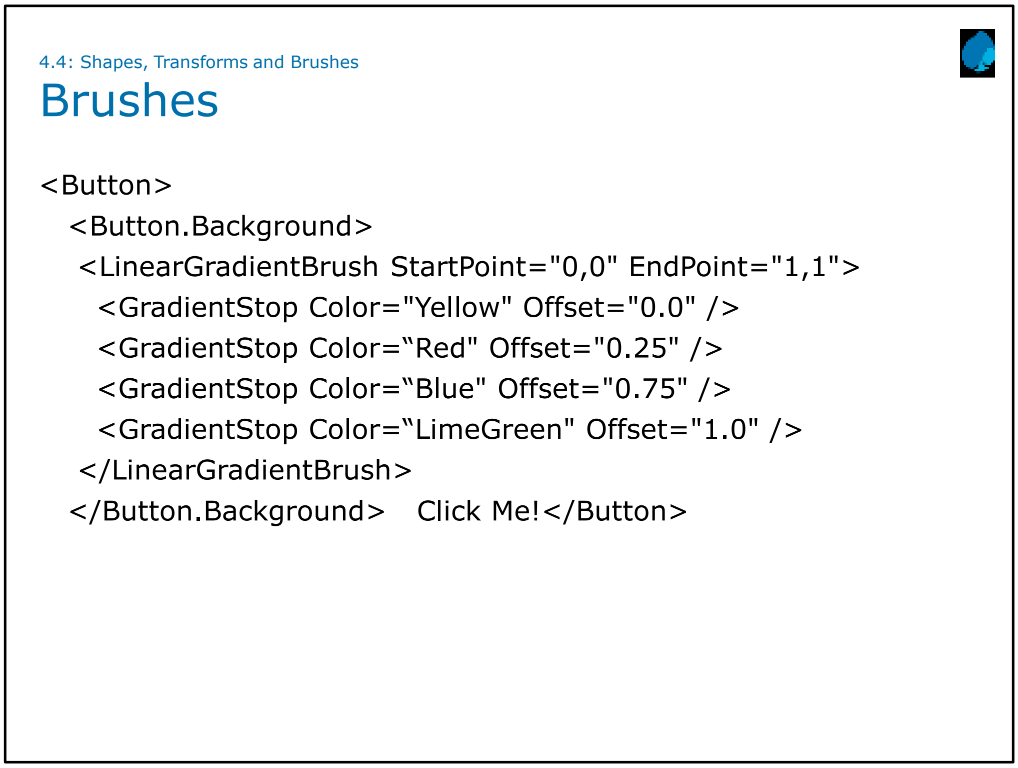
- Brushes fill an area, whether it is the background, foreground, or border of an element, or the fill or stroke of a shape
- Brush Classes:
- LinearGradientBrush
  - Paints an area using a gradient fill, a gradually shaded fill that changes from one color to another (and, optionally, to another and then another, and so on)



4.4: Shapes, Transforms and Brushes

Brushes

- RadialGradientBrush
  - Paints an area using a radial gradient fill, which is similar to a linear gradient except it radiates out in a circular pattern starting from a center point
- ImageBrush
  - Paints an area using an image that can be stretched, scaled, or tiled



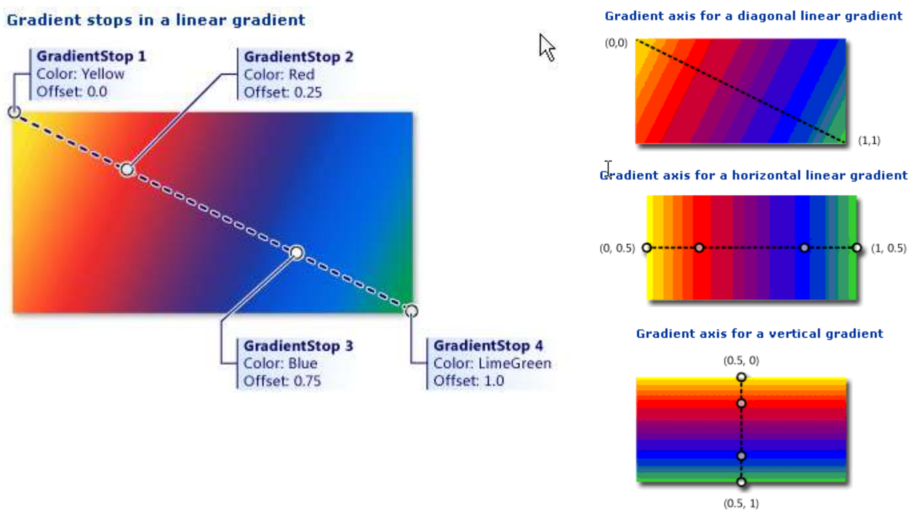
4.4: Shapes, Transforms and Brushes

Brushes

```
<Button>
  <Button.Background>
   <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
     <GradientStop Color="Yellow" Offset="0.0" />
     <GradientStop Color=“Red" Offset="0.25" />
     <GradientStop Color=“Blue" Offset="0.75" />
     <GradientStop Color=“LimeGreen" Offset="1.0" />
   </LinearGradientBrush>
  </Button.Background>   Click Me!</Button>
```

4.4: Shapes, Transforms and Brushes

Brushes



4.4: Shapes, Transforms and Brushes

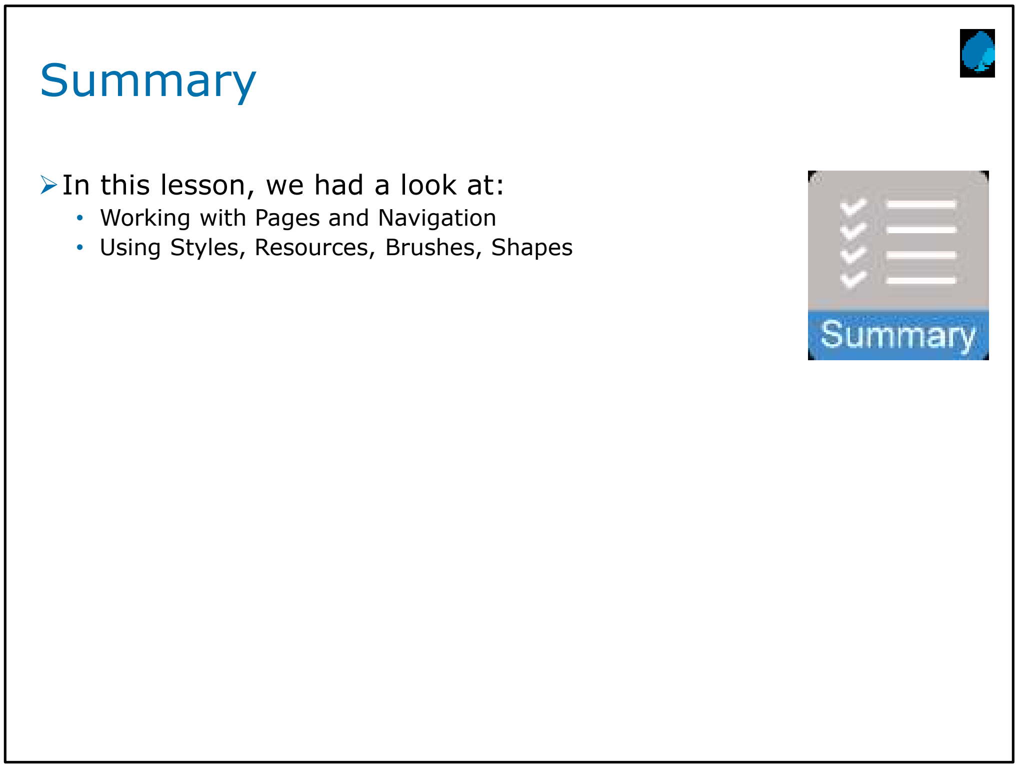
Brushes

- The GradientStop is the basic building block of a gradient brush. A gradient stop specifies a Color at an Offset along the gradient axis:
  - The gradient stop's Color property specifies the color of the gradient stop. You may set the color by using a predefined color or by specifying ScRGB or hexadecimal ARGB values
  - The gradient stop's Offset property specifies the position of the gradient stop's color on the gradient axis
  - The offset is a Double that ranges from 0 to 1
  - The closer a gradient stop's offset value is to 0, the closer the color is to the start of the gradient
  - The closer the gradient's offset value is to 1, the closer the color is to the end of the gradient

4.4: Shapes, Transforms and Brushes

Demo

- Drawing various shapes, Transforms, Brushes



Summary

- In this lesson, we had a look at:
  - Working with Pages and Navigation
  - Using Styles, Resources, Brushes, Shapes