Styles and Triggers

Agenda

- Style
  - Definition
  - Inline
  - Named
  - Reuse
  - Extending
  - Setting programmatically
- Triggers

STYLE

Definition of Style

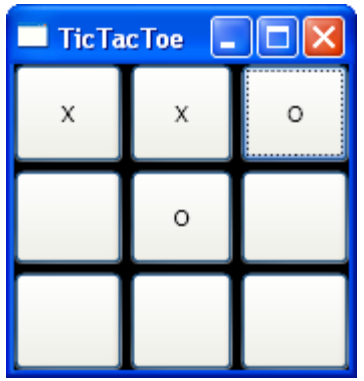
- A ***style*** is a set of properties applied to content used for visual rendering
  - like setting the font weight of a Button control

- Styles can be dynamic in WPF
  - In addition to the features in word-processing styles, **WPF** styles have specific features for building applications, including the ability to apply different visual effects based on user events

Without Styles

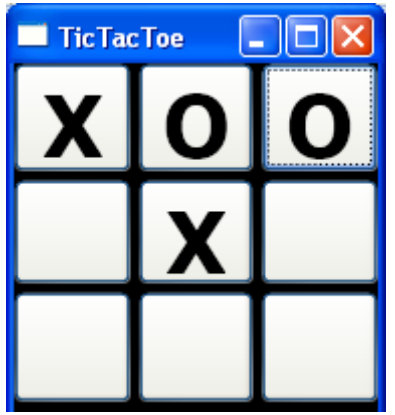
- A simple tic-tac-toe implementation

```
...
<Button Margin="0,0,2,2" Grid.Row="0" Grid.Column="0" Name="cell00" />
<Button Margin="2,0,2,2" Grid.Row="0" Grid.Column="1" Name="cell01" />
<Button Margin="2,0,0,2" Grid.Row="0" Grid.Column="2" Name="cell02" />
<Button Margin="0,2,2,2" Grid.Row="1" Grid.Column="0" Name="cell10" />
<Button Margin="2,2,2,2" Grid.Row="1" Grid.Column="1" Name="cell11" />
<Button Margin="2,2,0,2" Grid.Row="1" Grid.Column="2" Name="cell12" />
<Button Margin="0,2,2,0" Grid.Row="2" Grid.Column="0" Name="cell20" />
<Button Margin="2,2,2,0" Grid.Row="2" Grid.Column="1" Name="cell21" />
<Button Margin="2,2,0,0" Grid.Row="2" Grid.Column="2" Name="cell22" />
</Grid>
```



- Setting control properties individually

```
...
<Button FontSize="32pt" FontWeight="Bold" ... Name="cell00" />
<Button FontSize="32pt" FontWeight="Bold" ... Name="cell01" />
...
<Button FontSize="32pt" FontWeight="Bold" ... Name="cell22" />
</Grid>
```



- Though possible this is NOT smart
  - It requires a lot of effort to change the appearance later!

Inline Styles

- A style in WPF is expressed as zero or more Setter objects inside a Style object
- Every element in WPF that derives from either FrameworkElement or FrameworkContentElement has a Style property, which you can set inline using standard XAML property element syntax.

```
...
<Button ... Name="cell00">
  <Button.Style>
    <Style>
      <Setter Property="Button.FontSize" Value="32pt" />
      <Setter Property="Button.FontWeight" Value="Bold" />
    </Style>
  </Button.Style>
</Button>
...
</Grid>
```

- Due to the extra style syntax and because inline styles can't be shared across elements, inline styles actually involve more typing than just setting the properties
  - **Use named styles instead!**

  (An inline style is useful if you want to add property and data triggers to an individual element)

Named Styles

- Creating a named style

```
<Window ...>
  <Window.Resources>
    <Style x:Key="CellTextStyle">
      <Setter Property="Control.FontSize" Value="32pt" />
      <Setter Property="Control.FontWeight" Value="Bold" />
    </Style>
  </Window.Resources>
  ...
</Window>
```

- Applying a style to Button and TextBlock elements

```
<Button Style="{StaticResource CellTextStyle}" ... />
  ...
<TextBlock
    Style="{StaticResource CellTextStyle}"
    Foreground="White“
    Grid.Row="3“
    Grid.ColumnSpan="3“
    Name="statusTextBlock" />
```

Named Styles: Target Type Attribute

- If all of the properties can be set on a shared base class, you can promote the class prefix into the TargetType attribute and remove it from the name of the property:

```
<Window ...>
<Style x:Key="CellTextStyle" TargetType="{x:Type Control}">
  <Setter Property="FontSize" Value="32pt" />
  <Setter Property="FontWeight" Value="Bold" />
</Style>
...
</Window>
```

- You can only set properties available on that type.
  - If you'd like to expand to a greater set of properties down the inheritance tree, you can do so by using a more derived type

Reusing Styles

- If we'd like to define a style that contains properties not shared by every element to which we'd like to apply them, we can do that by dropping the TargetType and putting back the property prefix:

```
<Window ...>
<Style x:Key="CellTextStyle">
  <Setter Property="TextElement.FontSize" Value="32pt" />
  <Setter Property="Button.IsCancel" Value="False" />
</Style>
...
<!-- has an IsCancel property -->
<Button Style="{StaticResource CellTextStyle}" ... />
<!-- does *not* have an IsCancel property -->
<TextBlock Style="{StaticResource CellTextStyle}" ... />
...
</Window>
```

- At runtime, WPF will apply the dependency properties and the elements themselves will ignore those values that don't apply to them

Overriding Style Properties

- If we want to override a style property on a specific instance, we can do so by setting the property on the instance

```
<Window ...>
<Style x:Key="CellTextStyle">
  <Setter Property="TextElement.FontSize" Value="32pt" />
  <Setter Property="TextElement.FontWeight" Value="Bold" />
</Style>
...
<TextBlock
  Style="{StaticResource CellTextStyle}“
  FontWeight="Normal" ... />
...
</Window>
```

Extending Styles

- You can also extend a style, adding new properties or overriding existing ones:
  - The BasedOn style attribute is used to designate the style being extended

```
<Window ...>
<Style x:Key="CellTextStyle">
  <Setter Property="Control.FontSize" Value="32pt" />
  <Setter Property="Control.FontWeight" Value="Bold" />
</Style>
<Style x:Key="StatusTextStyle" BasedOn="{StaticResource CellTextStyle}">
  <Setter Property="TextBlock.FontWeight" Value="Normal" />
  <Setter Property="TextBlock.Foreground" Value="White" />
  <Setter Property="TextBlock.HorizontalAlignment" Value="Center" />
</Style>
...
</Window>
```

Setting Styles Programmatically

- Once a style has a name, it's easily available from our code

```
<Window ...>
void cell_Click(object sender, RoutedEventArgs e) {
    Button button = (Button)sender;
    ...
  // Set button content
    button.Content = this.CurrentPlayer;
    ...
    if( this.CurrentPlayer == "X" ) {
       button.Style = (Style)FindResource("XStyle");
       this.CurrentPlayer == "O";
    }
    else {
      button.Style = (Style)FindResource("OStyle");
       this.CurrentPlayer == "X";

    }
...
</Window>
```

Data triggers should be preferred to setting styles programmatically!

Implicit use of a Style

- You can define a style that is applied automatically to an element without the need for the explicit resource reference
- When you create a Style with a TargetType and do not specify the x:Key, the x:Key is implicitly set to be the same as the TargetType
  - This key is used to locate the style
- If a FrameworkElement does not have an explicitly specified Style, it will always look for a Style resource, using its own type as the key

```
<Window x:Class="ResourcesExample.StyleExplicitReference“
 Title="Resources"
 …
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Button.FontSize" Value="36" />
    </Style>
  </Window.Resources>

   <Grid>
    <Button>Hello</Button>
  </Grid>
</Window>
```

TRIGGERS

Automate Style Changes with Triggers

- Using triggers, you can automate simple style changes
- Triggers are linked to styles through the Style.Triggers collection
- Every style can have an unlimited number of triggers

The Different Triggers

Trigger	Watches for a change in a dependency property and then uses a setter to change the style.
MultiTrigger	All the conditions must be met before the trigger springs into action.
DataTrigger	watches for a change in any bound data.
MultiDataTrigger	Combines multiple data triggers.
EventTrigger	Applies an animation when an event occurs.

Simple Trigger aka. Property Trigger

- You can attach a simple trigger to any dependency property
  - For example, you can create mouseover and focus effects by responding to changes in the IsFocused, IsMouseOver, and IsPressed properties of the Control class
- Every simple trigger identifies the property you're watching and the value that you're waiting for
  - When this value occurs, the setters you've stored in the Trigger.Setters collection are applied

```
<Style x:Key="BigFontButton">
 <Style.Setters>
  <Setter Property="Control.FontFamily" Value="Times New Roman" />
  <Setter Property="Control.FontSize" Value="18" />
 </Style.Setters>
 <Style.Triggers>
   <Trigger Property="Control.IsFocused" Value="True">
     <Setter Property="Control.Foreground" Value="DarkRed" />
   </Trigger>
 </Style.Triggers>
</Style>
```