Custom Controls

Estimated time for completion: 60-120 minutes

Goals:

- To experiment with the different options of creating custom controls with WPF.
- Changing the behavior of existing controls through derivation.
- Creating composite controls by combining existing controls together.
- To create "lookless" controls which use templates and themes.

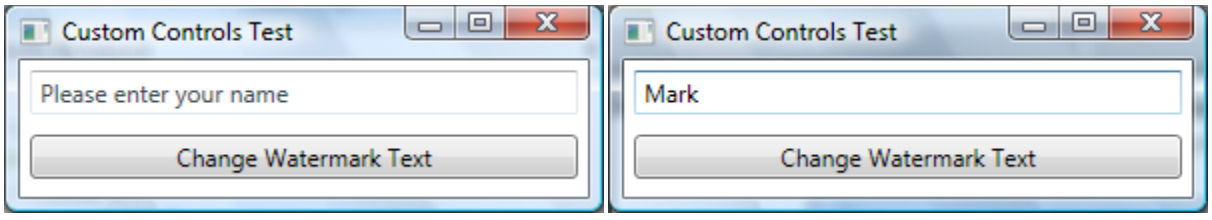
Overview:

This lab will walk you through various ways to create custom control content. We will start with overriding an existing control to change the behaviour and then write a full custom control which uses themes and control templates for styling.

Note that this is an exceptionally long lab due to the complexity of the topic. It is best reserved to the end of the day. You might consider doing a portion of the lab that you find more interesting – and then come back and go through the rest the next day.

Part 1 – Creating custom controls through derivation

In this part of the lab you will create a new TextBox style control which supports a watermark. When there is no text entry, a watermark will be shown; otherwise it will have the proper text:



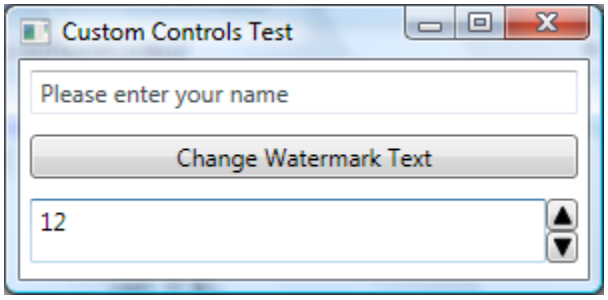
Steps:

1. Create a new WPF application using Visual Studio.
2. Add a new source file to the project called "**WatermarkTextBox.cs**"
    a. Derive the class from `TextBox`
    b. Add a `DependencyProperty` to hold the watermark text. The solution will call it `WatermarkTextProperty`.

3. Add a private method to set the watermark. It should set the foreground brush to a light gray color (the solution will use `SystemColors.InactiveCaptionTextBrushKey`) and set the `Text` property to the watermark text.
4. Override `OnLostFocus` and check the `Text` for data – if it is empty, call your `SetWatermark` method.
5. Override `OnGotFocus` and check if the watermark is in use. If so, clear the text field and reset the background color.
    a. Hint: use the `Clear()` and `ClearValue()` methods!
6. Finally, override the `OnInitialized` method and call your `SetWatermark` method if no initial text is provided.
7. Use the new control in your window.
8. As a bonus, see if you can detect if the watermark text changes. If it does, you should reset the watermark if it is being displayed.
    a. Hint: try adding a `PropertyChangedCallback` to your dependency property.

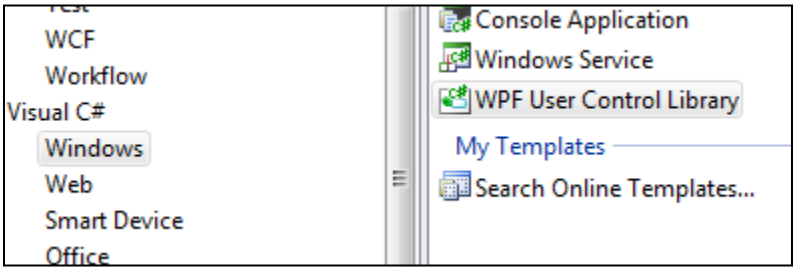
Part 2 – Creating a UserControl

In this part of the lab you will create a new style of control by combining a TextBox and a couple of RepeatButtons to generate the UpDownControl:



Steps:

1. Add a new WPF User Control to your solution using the Visual Studio project wizard.



2. Look at what Visual Studio generates – this is a UserControl which is essentially a block of reusable XAML. Our goal is to create a usable control with this.

3. Add two columns to the `Grid` – the second one should auto-size and the first should stretch.
4. Place a `TextBox` into the first column and give it a name so you can access it in code behind.
5. Place a child `Grid` with two rows into the second column and put two `RepeatButtons`, one on top of the other. Give each of them names as well. The solution will use "upBtn" and "dnBtn".
6. Draw some arrows as the content for the repeat buttons. The solution will use a Path, but you can use whatever graphics technique you are comfortable with. Feel free to steal the XAML from the presented code below.
7. When you are finished with the XAML, it should look something like:

```
<UserControl x:Class="UserControls.UpDownControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <TextBox x:Name="tb" />

        <Grid Grid.Column="1">
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <RepeatButton x:Name="upBtn" Width="16" Height="16">
                <Path Stroke="Black" Fill="Black" StrokeLineJoin="Round"
                      Data="M5,1 L1,9 L9,9Z" />
            </RepeatButton>
            <RepeatButton x:Name="dnBtn" Width="16" Height="16" Grid.Row="1">
                <Path Stroke="Black" Fill="Black" StrokeLineJoin="Round"
                      Data="M5,9 L1,1 L9,1Z" />
            </RepeatButton>
        </Grid>
    </Grid>
</UserControl>
```

8. Open the code behind for your user control. The thing to keep in mind when building these composite controls is the individual elements (the `TextBox` and `RepeatButtons`) are not generally visible to the consumer – instead you need to create public properties to expose the specific details you want consumers to see.

a. Add a `DependencyProperty` to expose an integer "Value". You should add a `PropertyChangedCallback` to this property so you can detect when code changes the value directly.

b. Add two normal properties to hold a Minimum and Maximum integer value. You could use `DependencyProperty` values here too (if you wanted to data bind to them for example).

9. Next, we want to restrict the input of the `TextBox` to numeric values. Can you think of an easy way to do that?

   a. Hint: it involves "eating" an event before the `TextBox` can see it.

10. In the `InitializeComponent` method, add event handlers to the `TextBox`.

    a. `TextChanged` and `PreviewKeyDown` for the `TextBox`.

11. In the `PreviewKeyDown`, check for keys you want to allow and then handle the event if it is not in that group. The solution allows arrow keys, backspace, delete, tab and any digits.

```
void tb_PreviewKeyDown(object sender, KeyEventArgs e)
{
    if ((e.Key >= Key.D0 && e.Key <= Key.D9) ||
        e.Key == Key.Delete || e.Key == Key.Back ||
        e.Key == Key.Left || e.Key == Key.Right ||
        e.Key == Key.Tab)
        return;

    e.Handled = true;
}
```

12. Next, code handlers for the repeat buttons `Click` event to increment and decrement the `Value` property appropriately.

13. Now, we'd like the `TextBox` to display the `Value` at all times. How can we achieve that? Is there a way to use data binding? See if you can get that implemented. Check the solution if you have trouble.

14. Finally, implement the `PropertyChangedCallback` for Value. It needs to validate the range for the new `Value` against `Minimum` and `Maximum`.

Note that it would be nice to use a validation callback – but those callbacks do not have access to the instance data we need in this case (the range) and so we must use the property changed callback instead.

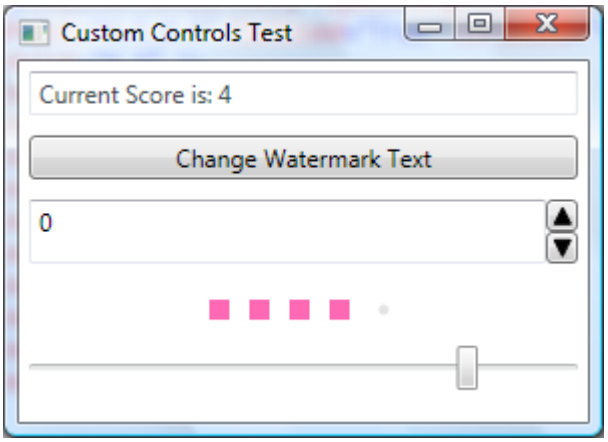
```
public static readonly DependencyProperty ValueProperty =
            DependencyProperty.Register("Value", typeof(int),
                        typeof(UpDownControl),
                        new UIPropertyMetadata(0, OnValueChanged));

static void OnValueChanged(DependencyObject d,
```

```
                                DependencyPropertyChangedEventArgs e)
{
   if (e.Property == ValueProperty)
   {
      UpDownControl upc = (UpDownControl)d;
      if (upc.Value < upc.Minimum)
          upc.Value = upc.Minimum;
      else if (upc.Value > upc.Maximum)
          upc.Value = upc.Maximum;
   }
}
```

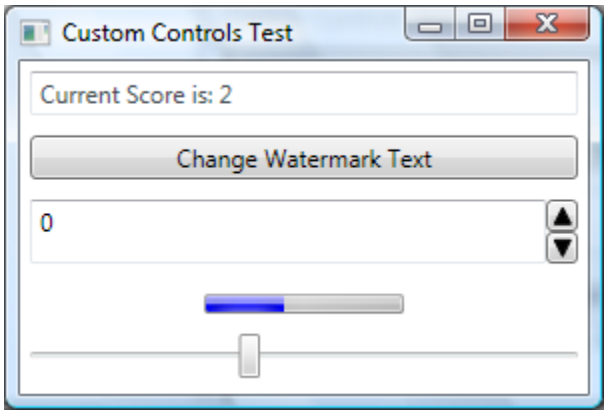
15. Test out your new control by adding an instance of it to your test program. Check out the solution if you have trouble.

Part 3 – Creating a Custom Control (Optional)

In this part of the lab you will create a full custom control – utilizing styles to give it a visual appearance. The following instructions will lead you through building a rating control – similar to that used in iTunes or Windows Media Player:

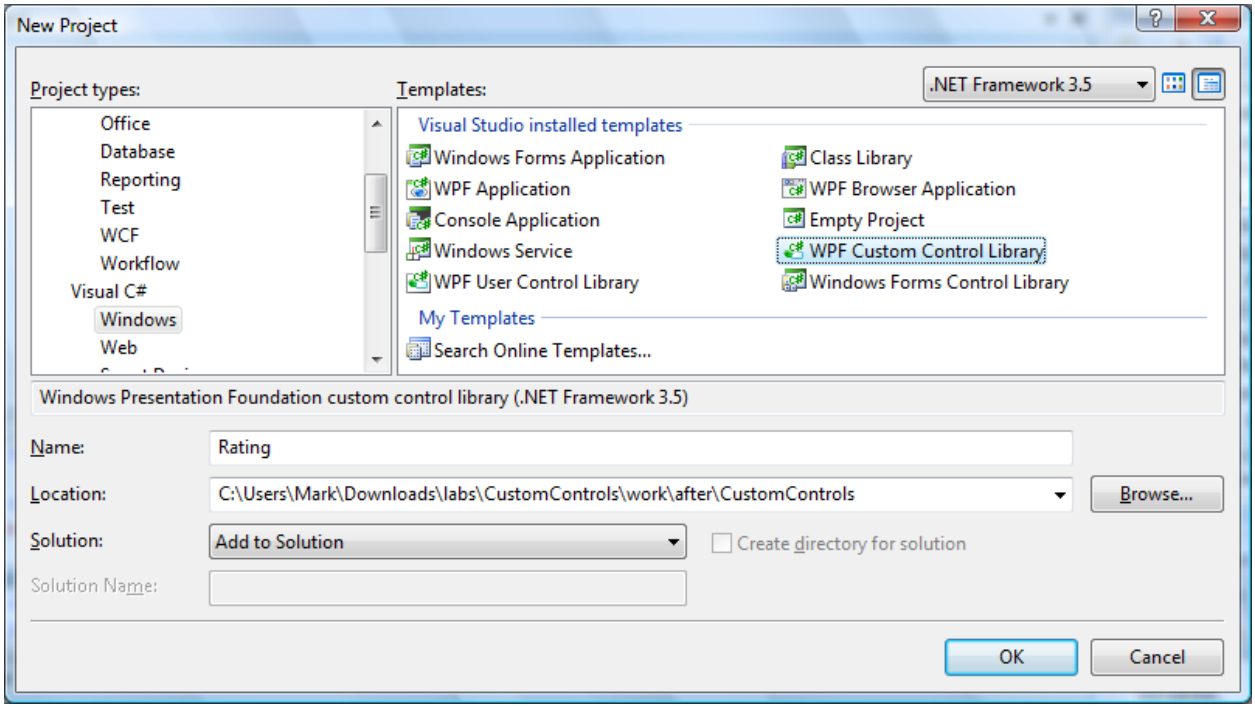


You can apply control templates in your testing program to change the visual look of the control:



Steps:

1. Add a new WPF Custom Control to your solution using the Visual Studio project wizard.
    a. Name it "RatingControl".



2. Change the name of the custom control (and source file **CustomControl1.cs**) to "Rating".
    a. You will also need to change the **generic.xaml** file – it references this type in the generated control template.
3. Open the **Rating.cs** file and examine the static constructor created by the wizard.
    a. This is the default style being applied for this control.
    b. Notice that it derives from `Control`. You could also derive from `ContentControl`, or `ItemsControl` if you wanted to have a piece of content or a list of items.
4. Now open the **generic.xaml** file in the Themes folder.
    a. This is the default `ControlTemplate` used to visually represent the control. It simply places a Border into the visual tree. We will replace this with our own content shortly.
5. Switch back to the **Rating.cs** file.
6. Create a new `DependencyProperty` to represent the score. Use an integer type and create both a `PropertyChangedCallback` and `CoerceValueCallback` handler for it.

```
public static readonly DependencyProperty ScoreProperty =
            DependencyProperty.Register("Score", typeof(int), typeof(Rating),
```

```
            new UIPropertyMetadata(0, OnScoreChanged, OnCoerceScore));

static void OnScoreChanged(DependencyObject d,
                           DependencyPropertyChangedEventArgs e)
{
}

static object OnCoerceScore(DependencyObject d, object value)
{
   return value;
}
```

7. For controls, it's useful to expose events when things are happening, as well as a virtual method for derived types to override. In this case, we'd like a `ScoreChanged` event to be raised when the score changes, and an `OnScoreChanged` virtual method to be called. Go ahead and provide an implementation for this – if you need help, follow these steps.
    a. Create a virtual method `OnScoreChanged` which takes two integers. Call that method from your `PropertyChangedCallback` method – the passed `DependencyObject` is the `Rating` control and the `EventArgs` has the previous and new values.
    b. Create a `RoutedEvent` called `ScoreChangedEvent` and a .NET event wrapper which adds/removes handlers from the routed event.
    c. Raise the new `ScoreChangeEvent` from your virtual `OnScoreChanged` method.
    d. When you are finished, the implementation should look something like:

```
public static RoutedEvent ScoreChangedEvent =
       EventManager.RegisterRoutedEvent("ScoreChanged",
                        RoutingStrategy.Direct,
                        typeof(RoutedEventHandler), typeof(Rating));

public event RoutedEventHandler ScoreChanged
{
   add
   {
      this.AddHandler(ScoreChangedEvent, value);
   }

   remove
   {
      this.RemoveHandler(ScoreChangedEvent, value);
   }
}
```

```
static void OnScoreChanged(DependencyObject d,
                           DependencyPropertyChangedEventArgs e)
{
   Rating r = (Rating)d;
   r.OnScoreChanged((int) e.OldValue, (int) e.NewValue);
}

public virtual void OnScoreChanged(int oldValue, int newValue)
{
   RaiseEvent(new RoutedEventArgs(ScoreChangedEvent));
}
```

8. Next, use the `CoerceValueCallback` to restrict the score to be within the range 0-5.
9. Let's now begin implementing the visual appearance of the control. Open the **generic.xaml** file and locate the `ControlTemplate`.
10. In the supplied Border, add a `Grid` that has 5 equal columns. Give the parent border a name of "PART_Track". We will reference this in the code behind.
11. Next, in each column, create a 5x5 `Ellipse` filled with some light color (the solution uses `SystemColors.ControlLightBrushKey`). Set some margin on each Ellipse as well. Give each one a name – the solution uses "d1" – "d5".
12. In each column (along with the Ellipse), create a `Path` with some shape to it – the solution will use a square (feel free to steal the below XAML). The important part is to set the `Visibility` to "Collapsed". Go ahead and set a margin on each one and fill it with the foreground color of the control using a template binding. Each path will need a name as well – the solution uses "e1" – "e5".
13. You can use the following definition to help you if necessary:

```
<ControlTemplate TargetType="{x:Type local:Rating}">
   <Border x:Name="PART_Track" Background="{TemplateBinding Background}"
           BorderBrush="{TemplateBinding BorderBrush}"
           BorderThickness="{TemplateBinding BorderThickness}">
      <Grid>
         <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
          </Grid.ColumnDefinitions>

          <Path Margin="3" x:Name="e1" Visibility="Collapsed"
                Fill="{TemplateBinding Foreground}"
                Data="M0,0 L10,0 10,10 0,10 Z"/>
          <Ellipse Margin="4" x:Name="d1" Width="5" Height="5"
```

```
                    Fill="{DynamicResource {x:Static
SystemColors.ControlLightBrushKey}}" />
       ... repeat 4 more times with different names in each cell ...
       </Grid>
   </Border>
</ControlTemplate>
```

14. Now, let's add the triggers which cause the ellipses and squares to work properly. We will use the named elements and key off our `Score` property. Add a Trigger element for each score value (0-5) and change the `Visibility` of the d1-d5 and e1-e5 shapes appropriately.
    a. For example, when the score is "1", the d1 shape should be collapsed and the e1 shape should be visible.
    b. Check your XAML against the following example:

```
<ControlTemplate.Triggers>
   <Trigger Property="Score" Value="1">
      <Setter TargetName="d1" Property="Visibility" Value="Collapsed" />
      <Setter TargetName="e1" Property="Visibility" Value="Visible" />
   </Trigger>
   <Trigger Property="Score" Value="2">
      <Setter TargetName="d1" Property="Visibility" Value="Collapsed" />
      <Setter TargetName="e1" Property="Visibility" Value="Visible" />
      <Setter TargetName="d2" Property="Visibility" Value="Collapsed" />
      <Setter TargetName="e2" Property="Visibility" Value="Visible" />
   </Trigger>
   <Trigger Property="Score" Value="3">
      <Setter TargetName="d1" Property="Visibility" Value="Collapsed" />
      <Setter TargetName="e1" Property="Visibility" Value="Visible" />
      <Setter TargetName="d2" Property="Visibility" Value="Collapsed" />
      <Setter TargetName="e2" Property="Visibility" Value="Visible" />
      <Setter TargetName="d3" Property="Visibility" Value="Collapsed" />
      <Setter TargetName="e3" Property="Visibility" Value="Visible" />
   </Trigger>
   ... repeat for values 4 and 5 ...
</ControlTemplate.Triggers>
```

15. Switch to your code behind file for the control. First, let's note that we have a required portion to our template. We do this with a `[TemplatePart]` attribute applied to the class definition. We will require that the template part is, at a minimum, a `FrameworkElement` (pretty safe bet).

```
    [TemplatePart(Name="PART_Track", Type=typeof(FrameworkElement))]
```

16. Next, let's connect the visuals to the code behind. Override the `OnApplyTemplate` method – this is the first opportunity where you can safely access the control template from code.
17. In this method, locate your "Part_Track" element.
    a. This is done by calling `this.Template.FindName()` or `this.GetTemplateChild()`.

> **Note:** MSDN documents `GetTemplateChild` as deprecated so you should now use `FindName`, they are equivalent in functionality for our use here, but `FindName` is capable of locating the proper element in more places than `GetTemplateChild`.

    b. You should not assume the element is the `Border` – instead, cast it to a `FrameworkElement`.
    c. If the template child is not found you should throw an exception so the consumer knows they've not supplied a required portion of the control template.
18. Add an event handler to the template part to handle the `MouseLeftButtonDown` event. You should handle the event *even if it has already been handled* so use the proper override of `AddHandler`.
19. Your code should look something like:

```
public override void OnApplyTemplate()
{
   base.OnApplyTemplate();

   FrameworkElement fe = this.Template.FindName("PART_Track", this) as
FrameworkElement;
   if (fe == null)
      throw new Exception("Missing PART_Track from Rating control template");

   fe.AddHandler(FrameworkElement.MouseLeftButtonDownEvent,
             new MouseButtonEventHandler(OnClickRating), true);
}

void OnClickRating(object sender, MouseButtonEventArgs e)
{
}
```

20. When the click event occurs, you need to set the `Score` property appropriately – we want it to change based on where the user clicked. The further to the right, the higher the score. Remember to bound it from 0 – 5.
21. If you need help, you can use the following code example:

```
   FrameworkElement fe = (FrameworkElement)sender;
   Debug.Assert(fe != null);
```

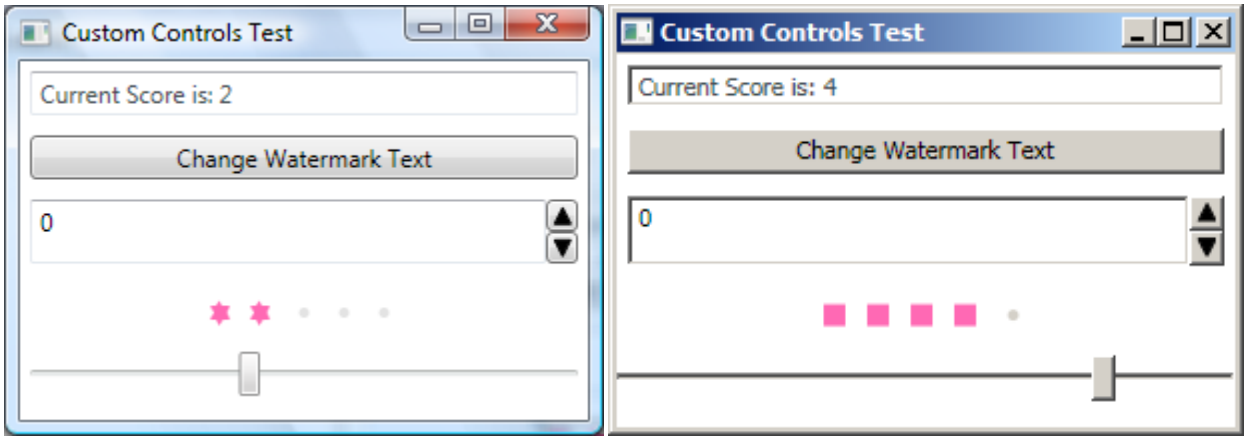
```
Point pt = e.GetPosition(fe);
double width = fe.ActualWidth;
if (width != Double.NaN)
   Score = (int) Math.Round(pt.X / (width / MAXIMUM_VALUE), 0,
                            MidpointRounding.AwayFromZero);
```

22. Add a Rating control to your sample window – set the Score to some value and run the application.
    a. Try adding a `Slider` and binding to the Score – sliding it should change the rating.
    b. Try changing the control template out in your application code. For example, use a `ProgressBar` and bind the `ProgressBar.Value` to the `Score`.

> Note that since these two values are different types you cannot use `{TemplateBinding}` – it will not coerce the integer to a double. Instead, you will need to use the full `{Binding RelativeSource}` syntax shown in the control templates slide deck notes.

Part 4 – Adding theme support to your custom control (Optional)

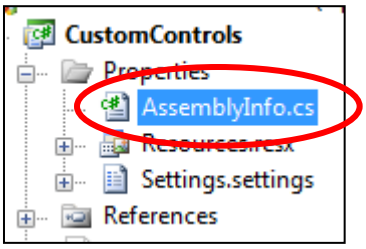
In this final part of the lab you will add theme support to your control so that it dynamically changes its own appearance based on the operating system theme being used.



The left side is running under Windows Vista with Aero, and the right side under Windows 2003, or under XP/Vista with the normal theme applied. The control is identical and no code changes are made between these two views – it is all behavior supplied by WPF!

Steps:

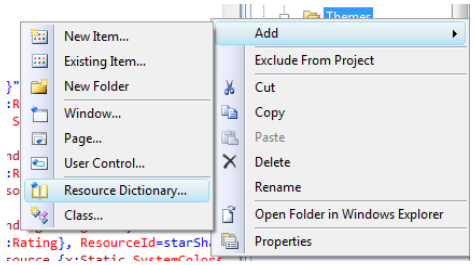
1. The first step is to tell WPF we want to use themes. We do this through a `[ThemeInfo]` attribute which is applied at the assembly level. It turns out that the WPF Custom Control project has already created one of these in your **assemblyinfo.cs** file. This file is "hidden" under the properties folder:



2. Open the file and comment out the existing `ThemeInfo` attribute you see there.
3. Open your **rating.cs** file and add the following attribute at the top of the file – just after the namespace declarations:

```
[assembly: ThemeInfo(ResourceDictionaryLocation.SourceAssembly,
ResourceDictionaryLocation.SourceAssembly)]
```

4. This tells WPF that our themes are located in the source assembly. We could just as easily place our theme definitions into separate resource-only assemblies and change the above definition to get WPF to look there.
5. Next, add a new Resource Dictionary to your themes folder – just right-click on the folder and select "Add" and then "Resource Dictionary":



6. Name the dictionary "Classic.xaml". This is the theme used when the "classic" theme is in place.
7. Add a namespace into the dictionary for the current assembly. The convention is to name it "x:local". You can copy it from the **generic.xaml** file if you like.
8. We want to add a `PathGeometry` to describe the square you created earlier and then reference that in our ControlTemplate. That way we can change the shape by simply changing the resource – much easier than replacing the control template, although we could do that too.
9. We will need to give it a key since it is in a dictionary. When using resources in themes, you need to supply not just the name for the key, but also the assembly it is in. We use the `ComponentResourceKey` markup extension to supply this.

a. It needs two things – the TypeInTargetAssembly which is a System.Type and a ResourceId which is a name (key).

b. Use {x:Type local:Rating} for the type and "ratingShape" for the key.

c. Here's the XAML if you don't want to think too hard about it:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:RatingControl">

    <!-- Simple square -->
    <PathGeometry x:Key="{ComponentResourceKey
            TypeInTargetAssembly={x:Type local:Rating},
            ResourceId=starShape}">
        <PathFigure StartPoint="0,0" IsClosed="True">
            <LineSegment Point="10,0" />
            <LineSegment Point="10,10" />
            <LineSegment Point="0,10" />
        </PathFigure>
    </PathGeometry>

</ResourceDictionary>
```

10. Next, let's modify our `ControlTemplate` to use the above `Geometry`. In all the `<Path>` objects, bind the `Path.Data` property to the created `Geometry`. Use a `DynamicResource` so it will see changes and use the `ComponentResourceKey` as the key. An example would be:

```
Data="{DynamicResource {ComponentResourceKey
   TypeInTargetAssembly={x:Type local:Rating}, ResourceId=ratingShape}}" />
```

11. Now add another `ResourceDictionary` to your themes folder – name this one **Aero.NormalColor.xaml**. This will be the theme resources if you are running Vista.

    a. Note: if you are running XP, then use the name **Luna.NormalColor.xaml** instead – this is the XP normal theme.

12. Repeat steps 3-5 in this new file and use the following `PathGeometry` definition:

```
<PathGeometry x:Key="{ComponentResourceKey
    TypeInTargetAssembly={x:Type local:Rating}, ResourceId=ratingShape}">
    <PathFigure StartPoint="4.5,0.5" IsClosed="True" IsFilled="True">
        <LineSegment Point="6,3" />
        <LineSegment Point="10,3" />
        <LineSegment Point="8.5,6.5" />
        <LineSegment Point="10,10" />
        <LineSegment Point="6.5,10" />
        <LineSegment Point="4.5,13" />
```

```
        <LineSegment Point="3,10" />
        <LineSegment Point="0,10" />
        <LineSegment Point="2,6.5" />
        <LineSegment Point="0,3" />
        <LineSegment Point="3,3" />
    </PathFigure>
</PathGeometry>
```

13. This creates a "star" shape. Feel free to replace it with one you create in Blend or by hand if you like.
14. Remove any `ControlTemplate` definition you created in your testing application so the default one is being used and run the application.
15. Try changing the theme while the application is running – it should dynamically adjust to the new theme by changing the geometry shape.

As you have seen, it is not terribly difficult to create a custom control – but it does require several steps. In our particular example here, we could also have replaced the control template for a Slider control and created a rating control out of that. You will find that in most cases it is unnecessary to really create a full-blown custom control but you can instead either create it through composition (UserControl) or via derivation or styles.

Solution

There is a full solution to this lab available in the **after** folder.