developmentor

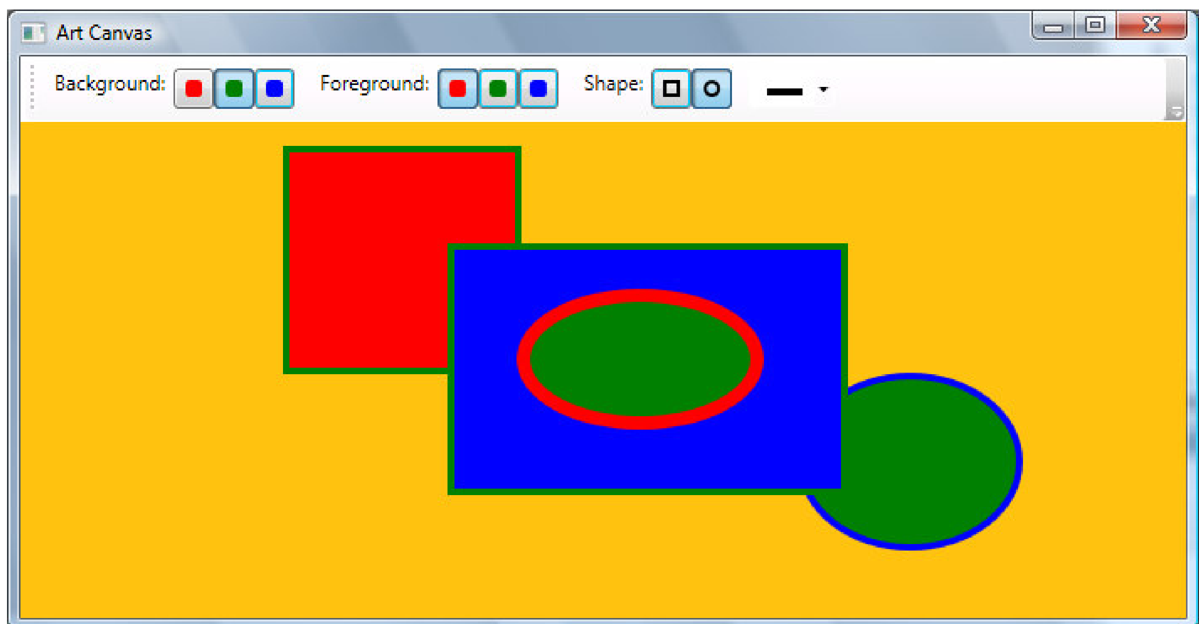
Organizing XAML

Estimated time for completion: **60 minutes**

Overview:

In this lab you will learn the various options of organizing XAML and the resources that make up the visual aspects of your application. During this lab, we will also reinforce some of the things you've already learned such as using controls and shapes and handling mouse events.

We'll create a simple art application which will look like the following:



Goals:

- Learn how to define reusable resources in XAML
- Supply application-wide reusable resources
- Separate resources into individual XAML files

© 2007 Education Experiences, Inc. All rights reserved

Part 1 – Setup the application

In this part, we are going to write the initial shell of the application which will be used for the remainder of the lab. You can follow along the detailed steps, or build it using your own experimentation and current level of WPF knowledge.

1. Start by creating a new Windows application using the WPF VS.NET template. Call it **ArtCanvas**. It should be composed of a `DockPanel` with a `ToolBar` docked to the top and a `Canvas` filling the remainder of the area.
    a. Change the root panel from `Grid` to a `DockPanel`.
    b. Add a `ToolBar` with the `DockPanel.Dock` property set to "Top".
    c. Add a `Canvas` after the `ToolBar` – give it a name so it is accessible to the code behind.
2. Next, add a set of `ToggleButtons` in the `ToolBar` to select foreground and background colors just as shown in the above picture. The content for each `ToggleButton` should be a 10x10 `Rectangle` filled with a red, green, or blue brush.

    a. Create a `StackPanel` in the horizontal orientation.
    b. Add a `TextBlock` as the header with the text "Background".
    c. Add a `ToggleButon` and add a `Rectangle` inside it.
    d. Set the `Width` and `Height` to be "10"
    e. Use a red brush to fill the `Rectangle` through the `Fill` property.
    f. Repeat steps [c-d] for the other two colors.
    g. Repeat steps [a-e] for the Foreground set of buttons.
3. If you are having trouble, use the following XAML as a guide:

```
<StackPanel Margin="5" Orientation="Horizontal">
   <TextBlock Margin="5,0">Background:</TextBlock>
   <ToggleButton Padding="5">
      <Rectangle Fill="Red" RadiusX="2.5" RadiusY="2.5"
                 Width="10" Height="10" />
   </ToggleButton>
    ... More toggle buttons here ...
</StackPanel>
```

4. Add another set of `ToggleButtons` for the shapes. Inside these, draw a `Rectangle` and an `Ellipse` – just as shown in the above picture. The size for both shapes is 10x10.

© 2007 Education Experiences, Inc. All rights reserved

a. Create a StackPanel in the horizontal orientation.

b. Add a TextBlock as the header with the text “Background”.

c. Add a ToggleButon and add a Rectangle inside it.

d. Set the Width and Height to “10”.

e. Set the Stroke to “Black”

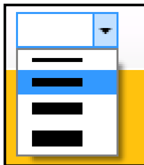
f. Set the StrokeThickness to “1”.

g. Repeat [c-f] and create an Ellipse instead.

Once you are finished, the XAML will look something like:

```
<StackPanel Margin="5" Orientation="Horizontal">
   <TextBlock Margin="5,0">Shape:</TextBlock>
   <ToggleButton Padding="5">
      <Rectangle Stroke="Black" StrokeThickness="1" Width="10" Height="10" />
   </ToggleButton>
   <ToggleButton Padding="5">
      <Ellipse Stroke="Black" StrokeThickness="1" Width="10" Height="10"/>
   </ToggleButton>
</StackPanel>
```

5. Finally, add a ComboBox to select the line width. Put (4) 25px-wide black-filled Rectangle objects of increasing height (2,4,6,8 is nice) into the combo box and select the 1st one in the combo box.



```
<ComboBox Margin="5" Padding="5" SelectedIndex="1">
   <Rectangle Margin="4" Fill="Black" Width="25" Height="2" />
   <Rectangle Margin="4" Fill="Black" Width="25" Height="4" />
   <Rectangle Margin="4" Fill="Black" Width="25" Height="6" />
   <Rectangle Margin="4" Fill="Black" Width="25" Height="8" />
</ComboBox>
```

6. The application should run and look something like the application picture. There’s no logic to it at this point so the buttons will not function properly – that will be the last thing we do. First, we are going to refactor some things to share resources and settings.

© 2007 Education Experiences, Inc. All rights reserved

Part 2 – Refactor out common settings and resources

In this part, we are going to reuse the common colors and settings by refactoring out the duplications into styles and resources within the window.

1. The first thing we are going to move out is brush colors. Notice that we are using the Black brush in quite a few places – if we wanted to change that color, we'd have to do it in every spot we used it. This is exactly what resources are for.
2. Create a `<Window.Resources>` section in your `<Window>` element. Inside that, create a `SolidColorBrush` with the color black and assign it a key of "blackBrush".
3. Next, replace all instances of the black brush within your current XAML with this new resource. Remember that to use an item in resources from XAML, we use either `{StaticResource}` or `{DynamicResource}`. Which do you think is more appropriate in this situation? Hint: think how often your resource will *change*.
4. Run the application and make sure that it has not changed visually. Your XAML should look something like this:

```
<Window ...>
   <Window.Resources>
      <SolidColorBrush x:Key="blackBrush" Color="Black" />
   </Window.Resources>
   ...
   <StackPanel>
      <ToggleButton ...>
         <Rectangle Stroke="{StaticResource blackBrush}" ... />
      </ToggleButton>
   </StackPanel>
</Window>
```

5. Next, create three brush resources and replace the colors for the foreground and background toggle buttons. You can name them color1, color2 and color3. Once this is done, you can change the colors in one place and affect both sets of buttons (Go ahead and try it!).
6. The next step takes a little explaining because we are going to utilize a feature of WPF you haven't seen yet called *Styles*. You're going to get a lot more detail very soon, but for now just know that Styles are a way of globally setting default properties for elements.
7. Creating a style is easy – you add a `<Style>` element into a resource dictionary – in this case the `Window.Resources` collection. The children of the style are `<Setter>` tags which have a `Property` and a `Value`. You then apply the style by name to the control you want to set the property values on. We'll start with the `Rectangle` objects contained in the `ComboBox` for stroke heights.

© 2007 Education Experiences, Inc. All rights reserved

8. Look at each of the rectangles and you'll see that they are almost all identical – with the exception of the `Height` property. We can remove all the identical properties and replace it with a style in the `ComboBox` resources. You will need to fully specify the property name – prefix each property with the base class of "Shape".

   For example, the `Width` property would be :

   ```
   <Setter Property="Shape.Width" Value="25" />
   ```

   Remember to give the style a `Key` because it is located in the resources!

```
<ComboBox.Resources>
   <Style x:Key="lineStyle">
      <Setter Property="Shape.Margin" Value="4" />
      <Setter Property="Shape.Fill" Value="{StaticResource blackBrush}" />
      <Setter Property="Shape.Width" Value="25" />
   </Style>
   <Rectangle Height="2" />
   <Rectangle Height="4" />
   <Rectangle Height="6" />
   <Rectangle Height="8" />
</ComboBox.Resources>
```

9. Next we need to tell each rectangle to use this new style. You do that by setting the `Style` property on the `Rectangle` to the resource you just defined.

```
<ComboBox Margin="5" Padding="5" SelectedIndex="1">
   ...
   <Rectangle Style="{StaticResource lineStyle}" Height="2" />
   <Rectangle Style="{StaticResource lineStyle}" Height="4" />
   <Rectangle Style="{StaticResource lineStyle}" Height="6" />
   <Rectangle Style="{StaticResource lineStyle}" Height="8" />
</ComboBox>
```

10. Do the same thing for the `Ellipse` and `Rectangle` shapes in the `ToggleButtons` – create a style and place it at the `StackPanel` level and remove all the common properties. Make sure to fully prefix the property name with the property type name (Shape in this case).
11. Run the application – it should still visually look the same, but we've consolidated a lot of the properties off to the resources.
12. Next, we are going to introduce yet another new feature in WPF which will be covered later in the course – Data Templates. Essentially, Data Templates allow you to provide visual information for non-visual objects. We're going to use them here to give a visual appearance to a brush.
13. What we want to do is set the `Content` property for the six `ToggleButtons` representing Foreground and Background colors to the brush you have set as a resource.

© 2007 Education Experiences, Inc. All rights reserved

Since the `Brush` will not have a visual representation it wouldn't look good if we just left it like that – so we will tell each of the `ToggleButtons` to use a different visual representation for thie content.

14. Set the `ToggleButton.Content` property of the six color buttons to the brush color. You can simply do this as a property attribute using the `{StaticResource}` you are already filling the `Rectangle` with:

```
<ToggleButton Content="{StaticResource color1}"> ...
```

15. Next, wrap the inner `Rectangle` with a `<DataTemplate>` tag so the entire "original" content for the button is now contained inside the `<DataTemplate>`
16. Change the `Rectangle.Fill` property (which is currently one of the three brush resources) to the value `"{TemplateBinding Content}"`. This will pull the Content property from the ToggleButton and use it here – don't worry about understanding that now, we'll cover this in a future session.
17. Next, wrap the `<DataTemplate>` within a `<ToggleButton.ContentTemplate>` element. This is the property you need to set the data template to in order for the `ToggleButton` to use it. When you are done, a single `ToggleButton` will look like:

```
<ToggleButton Content="{StaticResource color1}">
  <ToggleButton.ContentTemplate>
     <DataTemplate x:Key="ColorButtonTemplate">
         <Rectangle Fill="{TemplateBinding Content}"
               RadiusX="2.5" RadiusY="2.5" Width="10" Height="10" />
     </DataTemplate>
  </ToggleButton.ContentTemplate>
</ToggleButton>
```

18. Do the same steps for the remainder of the toggle buttons and run the application to verify it still looks the same.

Part 3 – Refactor resource into separate XAML files

Now we've got a lot of resources contained within the Window1 file. It works and is reusing resources but has bloated our XAML considerably and makes it hard to read through. In this step, we are going to separate out different pieces into separate files.

1. The first thing we are going to move out is brush colors. Notice that we are using the Black brush in quite a few places – if we wanted to change that color, we'd have to do it in every spot we used it. This is exactly what resources are for.
    a. Create a new Resource Dictionary named "Colors.xaml" in your project – use the "Add New" menu option available on the project.

© 2007 Education Experiences, Inc. All rights reserved

   b. Move all the brushes into this dictionary. Since they already have keys you will only need to move the location of the brush itself.

2. Next, create another dictionary called "Styles.xaml" and move all the created styles into that dictionary. Give each a key name and replace the usage within the main application window using the `{StaticResource}` markup extension.
3. Finally, create a dictionary called "DataTemplates.xaml" and move your data template into that dictionary, giving it a key name. You will need use the `{StaticResource}` markup extension to identify your named template.
4. Now we need to integrate these new dictionaries into the application. Open the App.xaml file and merge each new dictionary into the application resources.
   a. Create a `<ResourceDictionary>` in the application resources section
   b. Assign the `ResourceDictionary.MergedDictionaries` to a list of `ResourceDictionary` tags where the `Source` is set to each of your named files.
   c. Once you are finished, it will look something like:

```
<Application x:Class="ArtCanvas.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

      <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
          <ResourceDictionary Source="Colors.xaml" />
          <ResourceDictionary Source="DataTemplates.xaml" />
          <ResourceDictionary Source="Styles.xaml" />
        </ResourceDictionary.MergedDictionaries>
      </ResourceDictionary>

    </Application.Resources>
</Application>
```

5. Examine the structure of the application now – do you think this is easier to find the composite elements that make up the application?
6. Run the application and note that it still looks the same. We've simply reorganized where things are placed and provided some structure to the source code.

Part 4 – Using Dynamic Resources

In this final section, we are going to change the background color of our canvas to be the same color as the desktop.

© 2007 Education Experiences, Inc. All rights reserved

1. Set the Background property of the Canvas in the main window to be assigned the static property `SystemColors.DesktopBrushKey`
    a. Use the `StaticResource` markup extension.
2. Run the application. Note the new background color.
3. Open the desktop properties and change the background color. Does the application background change too?
4. Stop the application and change the code to use a `DynamicResource` static extension.
5. Run the application.
6. Open the desktop properties and change the background color – notice how it changes in the running application and is changed dynamically.

Solutions

The full lab solution is available at: work\after\ArtCanvasSolution.sln

© 2007 Education Experiences, Inc. All rights reserved