CHAPTER 7

Building the User Interface

Simplicity is the ultimate sophistication.
—Leonardo da Vinci

In the previous chapters, you learned about touch concepts, Windows design principles, Windows application types, and the many controls that are available to you in the Visual Studio Toolbox. In this chapter, you will learn about XAML styles and data templates, which provide more depth on the ListView, GridView, FlipView, and SemanticZoom controls that were introduced in Chapters 5 and 6. Once I have covered these more advanced topics, the foundation will have been laid for me to introduce the sample application that you will build upon through the remainder of the book. You will plan and build the skeleton of the user interface for this sample application.

Introducing Style

When building applications with XAML controls, very often you will find yourself applying custom sets of attributes to controls across the application. For example, you may decide you want all of your TextBlock controls to use the Helvetica font. One way to achieve this would be to place FontFamily = "Helvetica" in the XAML markup for each TextBlock control in the application. This would do the job, but you run the risk of forgetting a TextBlock and being left with an inconsistent interface or, even worse, having to go change each TextBlock later when you review the design and realize that Segoe UI would have been a more appropriate font. This is where styles come into play.

XAML styles provide a way for developers to create standard sets of property values for UI elements. In many ways, they are similar to the Cascading Style Sheets (CSS) used in HTML development, but because any property of the element being styled can be set within a XAML style, they go far beyond the capabilities of CSS. When creating XAML styles, the following properties, set as attributes within the XAML, are used:

- **TargetType**: This property is required and specifies the element type for which this Style will set properties. Only elements of the specified type can reference the Style.
- **BasedOn**: This property allows for one Style element to inherit the settings of another Style element and then provide further customization. The order in which Style elements are defined is important, and Style elements that are used in the BasedOn property of other Style elements must first be defined.
- **x:Key**: This property is not technically a property of the Style element but of an entry within a ResourceDictionary. Because Style elements are typically defined as entries in a ResourceDictionary, this property is used by elements that reference the Style, such as in the BasedOn property.
Inside the Style element is a collection of one or more Setter elements. These elements specify the name of the property to be set and the desired value of the property. Often these property values can be expressed using XML attributes, as in Value = "Helvetica", but sometimes the value itself requires additional XML, and in this case the value is specified using element syntax, as in the following:

```xml
<Setter Property="Background">
    <Setter.Value>
        <ImageBrush ImageSource="/Assets/Logo.png" />
    </Setter.Value>
</Setter>
```

### Where Styles Are Defined

When adding a Style to your application, the decision should be made whether the Style will be used with only a single page or more globally throughout the application. In this section, you will look at each of these methods.

#### Page-Level Style Definition

Style elements that will be used only within a single page are typically added to that page's Resources, as in the following example:

```xml
<Page ...>
    <Page.Resources>
        <Style x:Key="MyTextBlockStyle" TargetType="TextBlock">
            <Setter Property="FontFamily" Value="Helvetica" />
            <Setter Property="FontSize" Value="48" />
        </Style>
    </Page.Resources>
</Page>
```

The preceding sample created a Style that is available throughout the page, and referencing it from any TextBlock will cause the TextBlock to be rendered in 48-point Helvetica font as long as the property has not been overridden by the element itself. This concept of properties being overridden is an important piece of XAML styles. For example, I could define a TextBlock that uses the Helvetica face but an alternative size on the page using the following XAML:

```xml
<TextBlock Style="{StaticResource MyTextBlockStyle}" FontSize="24" Text="Smaller Text" />
```

I could also create another style based on the first but with a smaller size if I wanted to have multiple TextBlock elements sharing the same attributes. This Style would be defined in the Page.Resources element following the MyTextBlockStyle definition and would look like this:

```xml
<Style x:Key="MySmallerTextBlockStyle" TargetType="TextBlock" BasedOn="{StaticResource MyTextBlockStyle}"
    <Setter Property="FontSize" Value="24" />
</Style>
```