# Professional WPF Programming:
## .NET Development with the Windows® Presentation Foundation

### Chapter 7: Custom Controls

ISBN-10: 0-470-04180-3
ISBN-13: 978-0-470-04180-2

# 7

# Custom Controls

Windows Presentation Foundation controls are based on the concept of composition. Controls can contain other controls. For instance, a button may contain another button as its content, or it may contain an image, video, animation, or even a text box. The power of composition in WPF is that it provides a tremendous amount of flexibility right out of the box for customizing controls. In addition, WPF introduces styling, control templates, and triggers, which further enhance the extensibility of controls. Still there may be times when you need to create a custom control and, of course, WPF supports that as well.

The purpose for any application user interface is to present users with a set of visual components that coordinate user input and organize workflow. Controls are the individual visual elements that are logically grouped to accept, organize, and validate user input. In both Windows and web applications today, you commonly see these elements as text boxes, drop-down lists, radio buttons, and checkboxes.

WPF controls are referred to as *lookless controls*, which refers to the separation of visual appearance from behavior. WPF allows you to modify or extend the visual appearance of any control without affecting the control's behavior. Developers can modify the visual appearance of any pre-existing control, enhancing its visual impact while allowing the user to retain familiarity with the expected behavior. For example, with the media and animation capabilities provided in WPF, a button can exhibit animation features when the user interacts with it, but the button's behavior does not change. The button may "spin" or "glow" in reaction to a user clicking on it but the button still fires the expected click event.

Within WPF, the developer also has the ability to author custom controls tailored to the specific data input and validation needs of their application. The base classes provided by WPF for extending and creating controls provide the developer with great flexibility in the design of their control's visual appearance. Such flexibility in visual appearance is made available through the styling and control template features of WPF.

Excerpted from Professional WPF Programming: .NET Development with the Windows(r) Communication Foundation
Wrox Press, www.wrox.com

The following concepts are covered in this chapter:

- Choosing a control base class
- Custom control authoring via the `UserControl` base class
- Data binding to controls using both declarative XAML and procedural code
- Customizing a control's look and feel with styles and templates

# Overview

WPF offers the developer an extensive feature set for constructing dynamic controls that push the boundaries of what users expect from conventional Windows application development. For instance, WPF controls can now be animated quickly and easily and 3D graphics and video can be incorporated to give controls a new level of interactivity and dynamism.

In spite of the advances that WPF provides, the problem-solving process for controls remains unchanged. Controls are still intended to serve a purpose and should define behavior accordingly. When developing a control, the developer must ask himself basic design questions, such as the following:

- What are the requirements of my new control?
- What behavior or functionality should my control provide?
- Does a control already exist that I can customize using styling or control templates in order to get the behavior I desire?
- How flexible does the control need to be for stylizing and extension by the control consumer?
- What type of user will be interacting with the control?
- Does the functionality meet the specified business requirements?

The answers to these questions will define not only the path you take — customizing an existing control or creating a new control — but also will define your control's behavior, referred to as its *API*. Designing a streamlined, flexible, and well–thought out API is the goal of the custom control author.

In WPF, the choice of base class for your control is also dependent on the answers to these questions. The amount of customization required for your control will indicate the starting point for extending a new control.

> *Before heading down the path of creating a new control, it is important to note that many of the default controls within the WPF Framework allow for custom styling, triggers, and templating. If, for example, the need of your control is only to introduce an animation behavior, then creating a subclassed control would be overkill.*

# Control Base Classes

Once you have determined that creating a new control is the way to go, it's time to select a base class. In WPF, you can create custom controls based on a number of base classes, including `Control`, `UserControl`, and `FrameworkElement`. Selecting which base class to inherit from when creating a new control is contingent on the level of flexibility and customization you desire for your control. For example, the questions that you should ask yourself about the purpose of your control include the following:

- Will your control be composed of existing WPF elements?
- How flexible does your control need to be?
- Will you be doing custom low-level rendering in your control?
- What is the application context of the control? Will it be used by one application or many?
- How much visual customization will be required by consumers of your control? Should users be able to override the visual appearance of your control?

If you are simply composing existing elements in your control and consumers will not need to override its visual appearance, then most likely subclassing `UserControl` is the way to go. If separation of visual appearance from your control behavior is important in order that it may be visually changed by a consumer, then subclassing `Control` is the way to go. If you can't get the visuals you want out-of-the-box with WPF and you'll be doing custom rendering, then subclassing `FrameworkElement` is your best choice. There may be other factors that affect your choice of base class as well. Becoming familiar with these classes will go a long way in helping you make the right choice.

## The UserControl Class

Subclassing the `UserControl` class is the simplest way to create a new control and is the method you will explore in this chapter. A user control is composed of standard controls that together perform a particular interface function. Because a user control typically is meant to be used within a certain application context, its ability to be customized doesn't warrant as much attention as perhaps its reusability.

Within your user control you can apply styles to individual elements as well as handle specific events and raise custom events specific to the functionality of your control.

## Creating a User Control

In this example, you will create a new user control that inherits from the `System.Windows.Controls.UserControl` base class. The control will be one that I'm sure many developers have come across before: a pie graph chart. This control will be a good example of using the dynamic drawing features of WPF and container controls.

Before you get started in the code, let's expand on the process of defining the behavior (API) of this control, the logic necessary to generate the graph, and the WPF objects that you will use to draw the graph.

With WPF, a piece of constructed geometry that is drawn to screen is a visual element. Therefore, it is a targetable object that can be accessed directly in code rather than having to be redrawn as pixels to a screen as would have been the case prior to WPF. This means that it becomes much easier for you to detect collisions (hit-testing) as well as apply transformations and animation to the geometry object.

In order to draw to specific coordinate locations you will use a `Canvas` as the container for your pie drawing. A `Path` object will represent each pie slice. In WPF, a `Path` object can be made up of multiple geometry objects. Therefore, each pie section of your pie graph will be a `Path` object, which contains a `PathGeometry` object.

In WPF, the `PathGeometry` object represents a complex shape that is made up of any combination of arcs, curves, ellipses, lines, or rectangles. In WPF geometry, these arcs, curves, lines, and rectangles are represented by `PathFigure` and `PathSegment` objects. This will provide you with the flexibility to operate on each segment of the graph individually. For example, you can apply individual animations (if you so desire) to each piece rather than to the graph as a whole. You can also capture events on a piece-by-piece basis making it easier to respond to an input event as it pertains to each segment of the graph. Similarly, you can utilize the WPF routed event model to allow events fired by any pie piece to bubble up to the pie graph's parent container.

In order to create the `PathGeometry` object, you must first create a `PathFigure` object for each side of the pie graph piece. The `PathFigure` class represents a subsection of the `PathGeometry` object you will create. The `PathFigure` is itself made up of one or more `PathSegment` objects, which are specific types of geometric segments, such as the `LineSegment` and `ArcSegment`. You will use two `LineSegment` classes to create the initial and terminal sides of the piece. An instance of the `ArcSegment` class will construct the circular arc that attaches the two line segments together. Figure 7-1 illustrates how each pie piece will be constructed using the `Segment` classes.

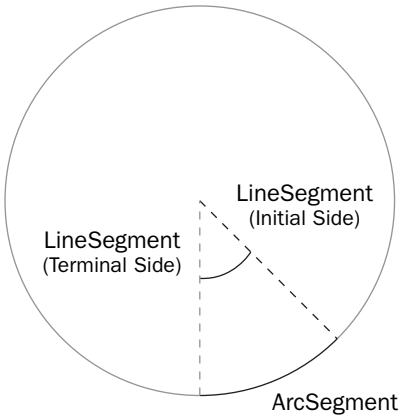


**Figure 7-1**

## *The ArcSegment Class*

The `ArcSegment` constructs an elliptical arc based on the initial starting point of the `PathFigure` object and a terminal point. Alternatively, you can construct the elliptical arc from the sibling, preceding it within the collection of `PathSegment` objects in its `Segments` property. The following table outlines the argument list used to construct an `ArcSegment`.

| Argument | Specification |
|---|---|
| `Point` | The terminal point for the arc segment. |
| `Size` | Specifies the x and y radius of the arc. The more circular the arc desired, the closer the x and y radius will be. |
| `RotationAngle` | The x-axis rotation angle. |
| `IsLargeArc` | Flags if the arc to be drawn is greater than 180 degrees. |
| `SweepDirection` | Enumeration value that specifies whether the arc sweeps clockwise or counter-clockwise. |

The following XAML defines an `ArcSegment` that will start from the initial point based on the `PathFigure` to which it belongs.

```
<Path Stroke="Black" Fill="Gray" StrokeThickness="2" Width="Auto">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigureCollection>
          <PathFigure StartPoint="200,200">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <ArcSegment
                    Point="300,200"
                    Size="200,50"
                    RotationAngle="90"
                    IsLargeArc="False"
                    SweepDirection="Clockwise"
                    />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>
        </PathFigureCollection>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

The following procedural code generates the equivalent to the declarative XAML:

```
PathGeometry pathGeometry = new PathGeometry();

PathFigure figure = new PathFigure();
figure.StartPoint = new Point(200,200);

figure.Segments.Add(
new ArcSegment(
  new Point(300,200),
```

```
    new Size(200,50),
    90,
    false,
    SweepDirection.Clockwise,
    true
    )
  );

  pathGeometry.Figures.Add(figure);

  Path path = new Path();
  path.Data = pathGeometry;
  path.Fill = Brushes.Gray;
  path.Stroke = Brushes.Black;

  myContainer.Children.Add(path);
```

Figure 7-2 illustrates the ArcSegment created in the preceding code examples.

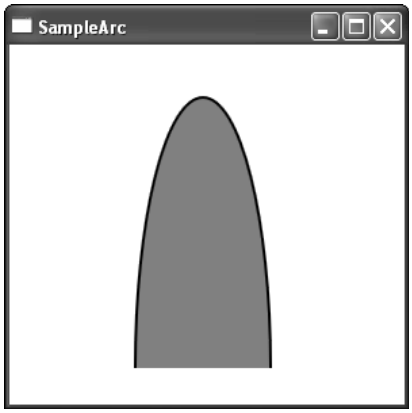


Figure 7-2

Each subsequent arc segment that is drawn for each piece in the pie graph will have to start at the last terminal point of the former segment. To calculate this you'll need to use some trigonometry to determine each pie piece's initial and terminal points relative to the angle of each pie piece. The pie piece angle will be determined by the percentage it represents of the underlying data. Figure 7-3 illustrates the incremental process.

To keep track of the next starting point you'll create a private local variable that you'll increment by the angle of the current pie piece.

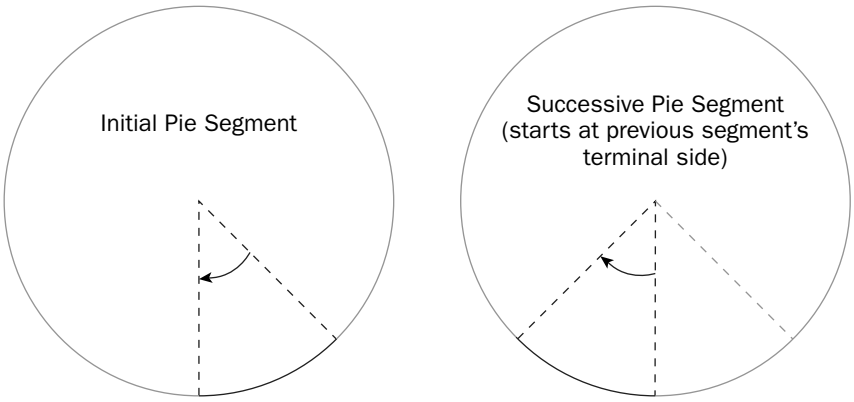


Figure 7-3

Now that you have established a basis for your control, you can build it in Visual Studio. The steps are as follows:

1. In Visual Studio, select File ➪ New Project from the menu. From the Project Types tree view, select Visual C# ➪ Windows (.NET Framework 3.0) Node, and then click on the Windows Application (WPF) icon from the list. Name your project **WPFWindowsApplication**, and click OK. This new project should create a `Window1.xaml` file, and a supporting partial class code-behind file.
2. Right-click on the project node and select Add ➪ New Item. Select User Control (WPF) and rename the file to **PieGraphControl**.
3. Open the PieGraphControl.xaml.cs file and add the following `using` directives, if they don't exist:

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Navigation;
using System.Windows.Shapes;
```

4. Within the pie graph control, you will create a private class named `PiePieceData` that will hold the data required to generate each section of the pie graph. Specifically, it will contain properties pertinent to the construction of each `ArcSegment` for a `PathFigure`.

   In the PieGraphControl.xaml.cs file, add the following private class declaration:

```
private class PiePieceData
{
  private double percentage;
```

```
    private string label;
    private bool isGreaterThan180;
    private Point initialSide;
    private Point terminalSide;
    private Brush color;

    public Brush Color
    {
      get { return color; }
      set { color = value; }
    }

    public double Percentage
    {
      get { return percentage; }
      set { percentage = value; }
    }

    public string Label
    {
      get { return label; }
      set { label = value; }
    }

    public bool IsGreaterThan180
    {
      get { return isGreaterThan180; }
      set { isGreaterThan180 = value; }
    }

    public Point InitialSide
    {
      get { return initialSide; }
      set { initialSide = value; }
    }

    public Point TerminalSide
    {
      get { return terminalSide; }
      set { terminalSide = value; }
    }

    public PiePieceData()
    {

    }

    public PiePieceData(double percentage, string label)
    {
      this.percentage = percentage;
      this.label = label;
    }
  }
```

5. Shifting focus back to our `PieGraphControl` class, you will need to perform a little math to determine where each segment will be placed within the area of the pie graph.

   - The `ConvertToRadians` method will take an angle measured in degrees and convert it to its radian measure equivalent so that you can use the `Math.Cos()` and `Math.Sin()` methods to find the point positions.
   - The `CalcAngleFromPercentage` will take the percentage value that the pie segment represents and calculate the angle that represents the given percentage.
   - The `CreatePointFromAngle` will take the radian angle and produce a point that intersects the pie graph circle for the given angle. This will help you determine the initial and terminal sides.

   Add the following private methods to your `PieGraphControl` class definition:

```
private Point CreatePointFromAngle(double angleInRadians)
{
  Point point = new Point();

  point.X = radius * Math.Cos(angleInRadians) + origin.X;
  point.Y = radius * Math.Sin(angleInRadians) + origin.Y;

  return point;
}

private double CalcAngleFromPercentage(double percentage)
{
  return 360 * percentage;
}

private double ConvertToRadians(double theta)
{
  return (Math.PI / 180) * theta;
}
```

6. You now need to include some private members that will aid in calculations and hold the collection of data from which you'll want to generate the graph. You'll also include a list of colors for each piece. For the sake of this example, you'll create a finite number of colors that will be used with the pie pieces. For a more flexible control, you would probably include a more dynamic color creation mechanism.

   Include the following code in the `PieGraphControl` class definition:

```
private Point origin = new Point(100, 100);
private int radius = 100;
private double percentageTotal = 0;
private double initialAngle = 0;
private List<PiePieceData> piePieces = new List<PiePieceData>();

private Brush[] brushArray = new Brush[]
{
  Brushes.Aquamarine,
  Brushes.Azure,
  Brushes.Blue,
  Brushes.Chocolate,
```

```
    Brushes.Crimson,
    Brushes.DarkGreen,
    Brushes.DarkGray,
    Brushes.DarkSlateBlue,
    Brushes.Maroon,
    Brushes.Teal,
    Brushes.Violet
  };
```

7. The following code constructs a `PathFigure` based on the `PiePieceData`, which is passed into the method.

   Include the following code in the `PieGraphControl` class definition:

```
  private PathFigure CreatePiePiece(PiePieceData pieceData)
  {
    PathFigure piePiece = new PathFigure();
    piePiece.StartPoint = origin;

    // Create initial side
    piePiece.Segments.Add(new LineSegment(pieceData.InitialSide, true));

    // Add arc
    Size size = new Size(radius,radius);

    piePiece.Segments.Add(
      new ArcSegment(
        pieceData.TerminalSide,
        size,
        0,
        pieceData.IsGreaterThan180,
        SweepDirection.Clockwise,
        true
        )
    );

    // Complete the terminal side line
    piePiece.Segments.Add(new LineSegment(new Point(origin.X,origin.Y), true));

    return piePiece;
  }
```

8. The next method definition will be a public method to allow developers to add a new pie percentage value. This method also checks to make sure that the total percentage doesn't exceed the value of 100 so that there is no overlap in the pie segments.

   Add the following to the `PieGraphControl` class definition:

```
  public void AddPiePiece(double percentage, string label)
  {
    if (percentageTotal + percentage > 1.00)
      throw new Exception("Cannot add percentage. Will make total greater than
        100%.");

    PiePieceData pieceData = new PiePieceData();
```

```
    pieceData.Percentage = percentage;
    pieceData.Label = label;

    // Calculate initial and terminal sides
    double angle = CalcAngleFromPercentage(percentage);
    double endAngle = initialAngle + angle;
    double thetaInit = ConvertToRadians(initialAngle);
    double thetaEnd = ConvertToRadians(endAngle);

    pieceData.InitialSide = CreatePointFromAngle(thetaInit);
    pieceData.TerminalSide = CreatePointFromAngle(thetaEnd);
    pieceData.IsGreaterThan180 = (angle > 180);

    // Update the start angle
    initialAngle = endAngle;

    piePieces.Add(pieceData);
  }
```

9. Once the values have been added to the control, it is now ready to proceed in rendering the data. The following method creates a new `PathGeometry` object for each pie piece figure so that it can be drawn to screen. Add the following `RenderGraph` method to your `PieGraphControl` class definition:

```
public void RenderGraph()
{
  int i = 0;

  foreach (PiePieceData piePiece in piePieces)
  {
    PathGeometry pieGeometry = new PathGeometry();
    pieGeometry.Figures.Add(CreatePiePiece(piePiece));

    Path path = new Path();
    path.Data = pieGeometry;
    path.Fill = brushArray[i++ % brushArray.Length];
    piePiece.Color = (Brush)brushArray[i++ % brushArray.Length];
    canvas1.Children.Add(path);
  }
}
```

10. In the default Window1.cs code-behind file that was generated when the project was created, include the following code in the constructor of the `PieGraphControl` class:

```
public partial class Window1 : Window
{
  public Window1()
  {
    InitializeComponent();

    PieGraphControl ctrl = new PieGraphControl();

    ctrl.AddPiePiece(.20, "Latino");
    ctrl.AddPiePiece(.20, "Asian");
    ctrl.AddPiePiece(.30, "African-American");
```

```
      ctrl.AddPiePiece(.30, "Caucasian");

      ctrl.RenderGraph();
      this.myGrid.Children.Add(ctrl);
    }
  }
```

**11.** The following XAML code contains the drawing canvas to which the drawing output will be directed.

Copy the following XAML code into the PieGraphControl.xaml file:

```
<UserControl x:Class="WPFWindowsApplication.PieGraphControl"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Width="300" Height="400">
  <Grid>
    <Canvas Margin="50,38,51,162" MinHeight="50" MinWidth="50" Name="canvas1" />
  </Grid>
</UserControl>
```

**12.** Modify the Window1.xaml code generated by the New Project Wizard so that it looks like this:

```
<Window x:Class="WPFWindowsApplication.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="WPFWindowsApplication" Height="300" Width="300"
  >
  <Grid Name="myGrid"/>
</Window>
```

**13.** Build the project in Debug to view the custom user control in WPF. Figure 7-4 shows the compiled result.

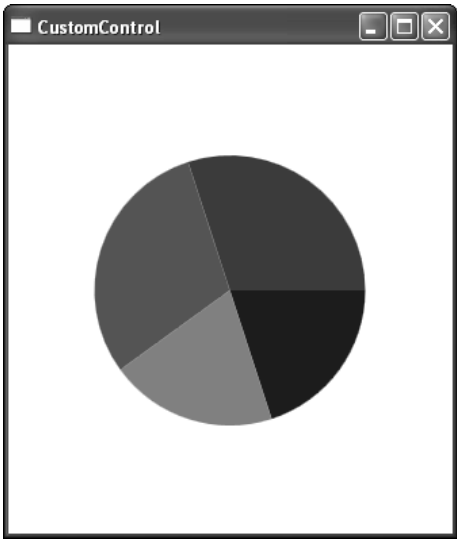


Figure 7-4

# Data Binding in WPF

In WPF, data binding is an integral part of the platform. Data binding creates a connection between the properties of two objects. When the property of one object (the source) changes, the property of the other object (the target) is updated. In the most common scenario, you will use data binding to connect application interface controls to the underlying source data that populates them. A great example of data binding is using a `Slider` control to change a property of another control. The `Value` property of the `Slider` is bound to a specified property of some control such as the width of a button. When the slider is moved, its `Value` property changes and through data binding the `Width` property of the button is updated. The change is reflected at runtime, so as you might imagine, data binding is a key concept in animation. WPF introduces new methods for implementing data binding, but the concept remains the same: a source object property is bound to a target object property.

Prior to WPF, developers created data bindings by setting the `DataSource` property of a binding target object and calling its `DataBind` method. In the case of binding to a collection of objects, sometimes event handling methods would perform additional logic on an item-by-item basis to produce the output result. In WPF, the process for creating a binding between object and data source can be done procedurally through code or declaratively using XAML markup syntax.

## Binding Markup Extensions

WPF introduces a new XAML markup extension syntax used to create binding associations in XAML. A markup extension signals that the value of a property is a reference to something else. This could be a resource declared in XAML or a dynamic resource determined at runtime. Markup extensions are placed within a XAML element attribute value between a pair of curly braces. You can also set properties within markup extensions. The following code example illustrates a markup extension for a binding:

```
<TextBlock
  Text="{Binding Source={StaticResource resourceKey},Path= Property}"
  Margin="2,2,2,2"
  FontSize="12"
  />
```

In the preceding XAML code, you designate that the `Text` property of the `TextBox` control is data bound by using a `Binding` extension. Within the `Binding` markup extension you specify two properties: the source object to bind from and the property of the source object to bind to. The source object is defined using another extension that denotes the source is a static resource defined within the XAML document. As you can see from this example, markup extensions can be nested within other markup extensions as well.

## Binding Modes

Binding modes allow you to specify the binding relationship between the target and source objects. In WPF, a binding can have one of four modes, as the following table illustrates.

| Mode | Description |
|---|---|
| `OneWay` | The target object is updated to reflect changes to the source object. |
| `TwoWay` | The target and source objects are updated to reflect changes on either end. |
| `OneWayToSource` | The converse of OneWay binding where target object changes are propagated back to the source object. |
| `OneTime` | The target object is populated to the source data once and changes between the target and source aren't reflected upon one another. |

To operate within a `OneWay` or `TwoWay` binding mode, the source data object needs to be able to notify the target object of source data changes. The source object must implement the `INotifyPropertyChanged` interface to allow the target object to subscribe to the `PropertyChanged` event to refresh the target binding.

## *Data Binding to a List<T>*

To pull all of these concepts together, you will modify the pie graph control you created previously to include some data bound elements. In this example, you will add a `ListBox` control to your user control that will display the percentage and description for each segment of the pie chart. You will bind the ListBox items to the pie pieces.

Because the source data object is a private list within the custom user control, you will need to set the context of the binding to the private list. To do this you will add a line of code to the constructor of the `PieGraphControl` that initializes the `DataContext` property of the user control to the list of pie pieces. The `DataContext` property allows child elements to bind to the same source as its parent element.

```
public PieGraphControl()
{
  InitializeComponent();
  this.DataContext = piePieces;
}
```

Next, in the XAML code for the PieGraphControl, you'll replace the current Grid with the following markup to introduce a `ListBox` control into the element tree. You will need to add a `Binding` markup extension to the `ItemSource` property of the `ListBox` to designate that it is data bound. Because you set the `DataContext` of the control to the list of pie pieces, you don't need to specify the source in the binding extension. You only need to include the binding extension itself within the curly braces. The following code segment illustrates the XAML markup required for this.

```
<Grid>
  <Canvas
    Margin="50,38,51,162"
    MinHeight="50"
    MinWidth="50"
    Name="canvas1"
    />
  <ListBox
    Height="138"
    Margin="49,0,49,13"
```

```
      Name="listBox1"
      VerticalAlignment="Bottom"
      ItemsSource="{Binding}">
    </ListBox>
  </Grid>
```

At this point, the process creates the binding but gives you no visualization of the data elements with which you would like to populate the `ListBox`. If you were to compile this project and view the resulting list box, you would see that all of the list items would just be `ToString()` representations of each `PiePieceData` object. To customize the result of each `ListBox` item and to display the data you want you'll need to use a data template.

## Data Templates

Data templates provide you with a great deal of flexibility in controlling the visual appearance of data in a binding scenario. A data template can include its own unique set of controls, elements, and styles, resulting in a customized look and feel.

To customize the visual appearance of each data bound item that appears in the ListBox you will define a `DataTemplate` as a resource of the PieGraphControl. Defining our data template as a PieGraphControl resource will allow child elements within the custom control to use the same template if so desired. The following code segment defines a data template that you will apply to each list item in our PieGraphControl.

```
  <UserControl.Resources>
    <DataTemplate x:Key="ListTemplate">
      <Grid>
        <Border
          BorderBrush="{Binding Path=Color}"
          BorderThickness="1"
          CornerRadius="3"
          Width="150">
          <StackPanel Orientation="Horizontal">
            <TextBlock
              Text="{Binding Path=Percentage}"
              Margin="2,2,2,2"
              FontSize="12"
              />
            <TextBlock
              Text="{Binding Path=Label}"
              Margin="2,2,2,2"
              FontSize="12"
              />
          </StackPanel>
        </Border>
      </Grid>
    </DataTemplate>
  </UserControl.Resources>
```

Within the data template, you will notice that Binding Extensions have been added to those elements that will display the source data. The template includes a `Border` whose border brush will be specified by the color of the pie piece that is bound to it. Within the `BorderBrush` attribute of the `Border` element, you include a binding extension to specify that it binds to the `Color` property of the pie piece.

Now that the template has been created, you need to add an attribute to the `ListBox` element to specify that it should use the new template. Within the `ItemTemplate` attribute of the `ListBox` element, you add a binding extension to define the template as a `StaticResource` that you've defined in XAML. The resource key serves as the name of the template — in this case the name is `ListTemplate`.

```
<Grid>
  <Canvas
    Margin="50,38,51,162"
    MinHeight="50"
    MinWidth="50"
    Name="canvas1"
    />
  <ListBox
    Height="138"
    Margin="49,0,49,13"
    Name="listBox1"
    VerticalAlignment="Bottom"
    ItemsSource="{Binding}"
    ItemTemplate="{StaticResource ListTemplate}">
  </ListBox>
</Grid>
```

Figure 7-5 shows the newly formatted data bound ListBox as is defined in the template.

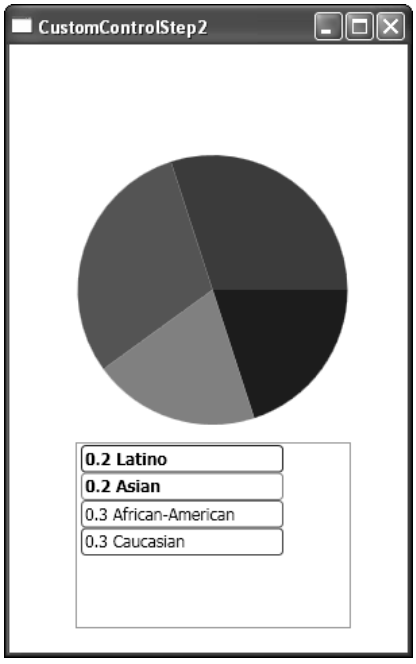


Figure 7-5

## Data Conversions

Within a binding, you can also specify if any type conversions need to take place before data is bound to the target control. In order to support a custom conversion, you need to create a class that implements the `IValueConverter` interface. `IValueConverter` contains two methods that need to be implemented: `Convert` and `ConvertBack`.

In the pie chart, recall that each pie piece on the graph represents a fraction of the total pie. This fraction is stored within each `PiePieceData` object as a double. While it is common knowledge that decimals can be thought of in terms of percentages, it would be nice to display to the user the double value as a percentage to make the data easier to read.

In your PieGraphControl, you will now define a new class that converts the decimal value to a string value for its percentage representation. Copy the following code sample, which contains the class definition for the converter into the PieGraphControl.xaml.cs file:

```
[ValueConversion(typeof(double), typeof(string))]
public class DecimalToPercentageConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
  {
    double decimalValue = (double)value;
    return String.Format("{0:0%}", decimalValue);
  }

  public object ConvertBack(object value, Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
  {
    throw new InvalidOperationException("Not expected");
  }

  public DecimalToPercentageConverter()
  {
  }
}
```

In order to use the type converter within XAML, you will need to create a new XML namespace that maps to the CLR namespace. To do so, add the following `xmlns` attribute to the `UserControl` element of the PieGraphControl.xaml file.

```
    xmlns:local="clr-namespace:WPFWindowsApplication"
```

Next you must add an additional property to the Binding markup extension to denote that you would like to use a type converter during the binding. In the following code example, you add the `Converter` property to the binding and set it to the `DecimalToPercentageConverter` resource you've defined in the control resources and assign it the given key:

```
<UserControl.Resources>
  <local:DecimalToPercentageConverter x:Key="decimalConverter" />
  <DataTemplate x:Key="ListTemplate">
    ...
```

```
      <StackPanel Orientation="Horizontal">
      <TextBlock
        Text="{Binding Path=Percentage, Converter={StaticResource decimalConverter}}"
        Margin="2,2,2,2"
        FontSize="12"
        />
      <TextBlock
      ...
    </DataTemplate>
  </UserControl.Resources>
```

Figure 7-6 shows the results of the modified PieGraphControl that now includes the data conversion for each percentage within the `ListBox` item list.

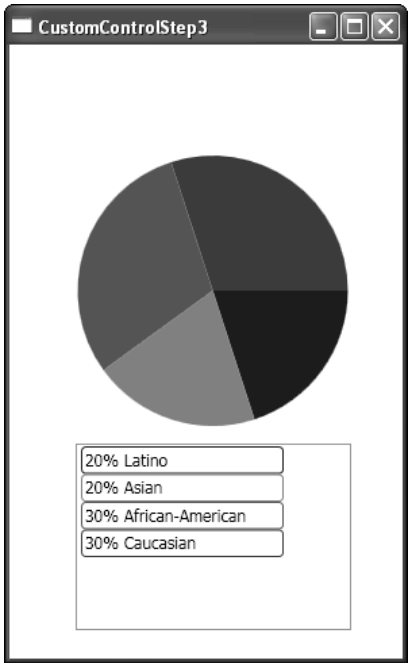


Figure 7-6

# Creating and Editing Styles

WPF styles allow you to uniformly apply a specific group of property settings to controls in order to define a desired look and feel. WPF has a styling mechanism similar to CSS and its application to HTML elements. Styles provide a centralized location where you define the visual appearance for your controls. In addition, styles propagate those attributes to any element that inherits the style. Styles also allow the definition of custom triggers that can change the visual behavior of a control.

Before you begin applying styles to controls, you will cover the basics of styles and how to apply them to standard controls.

Styles are declared in the `Resources` property of a window, container, or application. Where you place your style definition depends on the scope of your style. For styles that have a global context within your application, you can add your style to the `Resources` property of the `App.xaml` file. The following code segment illustrates a global application-level style defined in the `<Application>` element of an App.xaml file:

```
<Application x:Class="WinFxBrowserApplication1.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="Window1.xaml"
  >
  <Application.Resources>
    <Style x:Key="GlobalTextStyle">
      <Setter Property="TextElement.FontSize" Value="14"></Setter>
    </Style>
  </Application.Resources>
</Application>
```

To apply this style to an element, you simply set the `Style` attribute of the control to the name of the defined style:

```
<Label Style="{StaticResource GlobalTextStyle}" Name="label1" >My Label
  Text</Label>
```

Figure 7-7 illustrates your label with an application-level style applied.


Figure 7-7

## Specifying a Style's Target Type

The previous example illustrated a generic style definition that could be applied to any WPF element in your application, provided that it had a `FontSize` attribute to set. The style is generic because it does not specify a `TargetType` attribute. You use the `TargetType` attribute to indicate a specific control type to which the style will apply. If no `x:Key` attribute is specified along with the `TargetType` attribute, then the style will be applied to all elements of the specified target type.

The following code segment illustrates the use of the `TargetType` attribute. This style will be applied to elements of type `Label`. Additionally, the code defines an `x:Key` attribute so that only `Label` elements that specify the style property `LabelStyle` will inherit the style.

```
<Grid.Resources>
  <Style x:Key="LabelStyle" TargetType="{x:Type Label}">
    <Setter Property="Background" Value="BlueViolet" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="FontSize" Value="20" />
    <Setter Property="Width" Value="50" />
    <Setter Property="Height" Value="30" />
  </Style>...
</Grid.Resources>
```

As mentioned, specifying a target type for a style can also create a default style that is to be added to any element of the specified type by default. By removing the `x:Key` attribute, the style will be applied by default to elements of the specified target type — in this case, labels.

```
<Application.Resources>
  <Style TargetType="{x:Type Label}">
    <Setter Property="Background" Value="LightGray" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="FontSize" Value="14" />
    <Setter Property="Width" Value="100" />
    <Setter Property="Height" Value="50" />
  </Style>...
</Application.Resources>
```

The preceding code snippet declares a style that will be applied to all elements of type `Label`. Figure 7-8 illustrates the results of a simple label without a Style property defined. The style is picked up simply because the label is of type `Label`.


Figure 7-8

## *Inheriting and Overriding Styles*

A style is a class just like any other in WPF, so it can be inherited from and extended easily. Inheriting from a style class comes in handy if you want to define a global style to apply to all of your elements. For instance, maybe you want all of your text elements to conform to one font face. Of course, you may also want a small subset of text elements to use other text styles. In this case you adjust the weight and decoration by overriding the global style, either with an additional style, or by setting the element attributes directly.

To inherit from a style in XAML, you simply specify the `BasedOn` attribute of the `Style` element and create a markup extension that points to the base style. The following code segment defines a base style that is applied to a button, as well as a derived style that inherits from the base style.

```
<Window.Resources>
  <Style x:Key="BaseStyle">
    <Setter Property="TextElement.FontSize" Value="12"></Setter>
  </Style>
  <Style x:Key="BoldStyle" BasedOn="{StaticResource BaseStyle}">
    <Setter Property="TextElement.FontWeight" Value="Bold"></Setter>
  </Style>
  <Style x:Key="ItalicStyle" BasedOn="{StaticResource BoldStyle}">
    <Setter Property="TextElement.FontStyle" Value="Italic"></Setter>
  </Style>
</Window.Resources>
<Grid Name="myGrid">
  <Label Style="{StaticResource BaseStyle}" Height="29" VerticalAlignment="Top"
    Margin="91,31,100,0">This is my base styled text.</Label>
  <Label Style="{StaticResource BoldStyle}" Margin="55,79,49,92"  >This is my base
    derived bold styled text.</Label>
  <Label Style="{StaticResource ItalicStyle}" Margin="57,0,47,41" Height="26"
    VerticalAlignment="Bottom">This is my bold derived italic styled text.</Label>
</Grid>
```

Figure 7-9 illustrates the results.


Figure 7-9

Style properties are overridden by in-line styles in WPF. In the following code segment, you adjust the italicized styled label in the previous code block but override the `FontSize` inline.

```
<Label Style="{StaticResource ItalicStyle}" FontSize="16" Margin="17,61,22,58">This
  is my bold derived italic styled text at 16pt.</Label>
```

The results are illustrated in Figure 7-10.

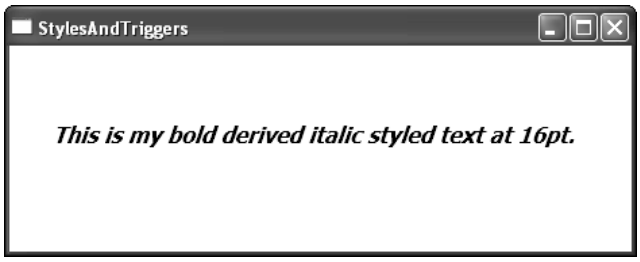


Figure 7-10

## Style Triggers

Triggers enable you to develop interaction logic based on conditions or events. They allow you to alter the behavior of controls depending on which conditions of the control are true. Triggers can apply a new set of style changes or can initiate more complex actions, such as animating properties of the control. Triggers fall into three categories: properties, events, and data.

Property triggers watch specific properties and fire when their values change. In the following example, you define a style for a button that contains a property trigger that fires when the `IsMouseOver` property change occurs. The trigger alters the background and foreground properties of the button.

```
<Style x:Key="ButtonStyle" TargetType="{x:Type Button}">
  <Setter Property="Background" Value="LightBlue" />
  <Setter Property="Foreground" Value="Blue" />
  <Setter Property="FontSize" Value="14" />
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value="50" />
  <Setter Property="TextElement.FontSize" Value="12"></Setter>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Background" Value="Orange" />
      <Setter Property="Foreground" Value="Black"></Setter>
    </Trigger>
  </Style.Triggers>
</Style>
```

The important thing to note about triggers is that they are designed to handle the return to original state automatically. Therefore, there is no need to apply an additional trigger to return the button back to its original state once the property condition isn't true.

Triggers can also have multiple conditions, which must hold true before new property changes are applied. Triggers with multiple conditions are modeled with a `MultiTrigger` element. The `MultiTrigger` element contains a collection of `Condition` child elements for each logical condition that must be applied before the trigger is fired. The following code segment illustrates a sample `MultiTrigger` for a button where the `IsMouseOver` and `IsEnabled` properties must be true before applying new property changes:

```
<Style.Triggers>
  <MultiTrigger>
    <MultiTrigger.Conditions>
```

```
        <Condition Property="IsMouseOver" Value="True"/>
        <Condition Property="IsEnabled" Value="True" />
      </MultiTrigger.Conditions>
      <Setter Property="Background" Value="Orange" />
      <Setter Property="Foreground" Value="Black"></Setter>
    </MultiTrigger>
  </Style.Triggers>
```

Unlike property triggers, event triggers listen for specific events rather than watching for property changes.

A data trigger can be used when an element participates in data binding. A data trigger will fire when a value that is bound to an element is equal to the value specified in the data trigger's `Value` attribute. Using data triggers allows you to define styles or behaviors based on runtime data. For example, you may have a list box containing items that represent tasks in a queue that are color-coded based on a priority level.

To illustrate the application of a data trigger, add the following code representing a new style to the resources collection of the PieGraphControl. The data trigger specifies the binding path that maps to the `PiePieceData` object's `Percentage` property. It also specifies a target value of .2 which, if matched during the binding, will apply a bold font weight to the text element of the list box item. Figure 7-11 illustrates the compiled result.

```
<Style TargetType="ListBoxItem">
  <Style.Triggers>
    <DataTrigger Binding="{Binding Path=Percentage}" Value=".2">
      <Setter Property="TextElement.FontWeight" Value="Bold" />
    </DataTrigger>
  </Style.Triggers>
</Style>
```

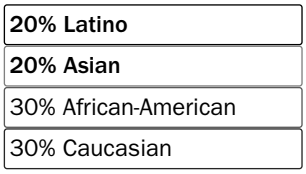


**Figure 7-11**

An important thing to remember about all of the trigger types is that they can coexist with one another within a style definition. Each type of trigger can also have multiple declarations for multiple properties or multiple events. If your control requires the use of event, property, and data triggers, they can simply be placed as sibling elements within the `Style.Triggers` XAML declaration.

```
<Style TargetType="ListBoxItem">
  <Style.Triggers>
    <EventTrigger ...="">
    </EventTrigger>
    <DataTrigger ...>
    </DataTrigger>
    <Trigger ...>
    </Trigger>
  </Style.Triggers>
</Style>
```

# Customizing Existing Controls with Templates

The composition design model of WPF offers such an unprecedented level of control customization that creating and implementing a custom control is often unnecessary. Using templates, the entire look and behavior of a control can be customized easily within XAML.

To customize a control in WPF, you can define and apply a control template. A control template allows you to change the composition of a control to any degree you like. As mentioned earlier, in WPF this is possible because controls are *lookless*, meaning their visual appearance is separate from the behavior. Control templates differ from styles. Styles allow you to customize a control's visual appearance based on setting properties that exist on the elements that make up the control (the default control template). A control template, however, can extend and even completely replace the elements that make up the control.

The one thing to remember when customizing an existing control with templates is that to offer a complete implementation of a new template, it is good practice to make sure you include overrides for all of the common triggers for events and property changes. For example, when creating a template for a button, it would be a good idea to create property and event triggers for mouse property changes and events.

The following example walks you through the basics of creating a custom control template for a button and then applying that template to a button.

1. In Visual Studio, add a new WPF window to the application you created for the custom user control. Leave the default name intact, as it is of no importance to this example.
2. Create a set of brushes as static resources, which you will use to define the mouse over, mouse out, and mouse pressed look and feel of the button. Place the following code within the `Window.Resources` element of the window XAML file.

```
<RadialGradientBrush x:Key="ButtonOffStateColor">
  <GradientStop Color="sc#1, 0, 0.05307113, 0.8078084" Offset="1"/>
  <GradientStop Color="#FFFFFFFF" Offset="0"/>
  <GradientStop Color="#FF5D87F0" Offset="0.64595660749506867"/>
</RadialGradientBrush>

<RadialGradientBrush x:Key="ButtonOnStateColor">
  <GradientStop Color="sc#1, 0.0145082464, 0.0145082464, 0.0145082464" Offset="1"/>
  <GradientStop Color="sc#1, 0, 0.07862101, 0.451001883"
    Offset="0.64595660749506867"/>
  <GradientStop Color="#FF7DA5D8" Offset="0"/>
</RadialGradientBrush>

<RadialGradientBrush x:Key="ButtonPressedStateColor">
  <GradientStop Color="sc#1, 0, 0.07862101, 0.451001883"
    Offset="0.63872452333990792"/>
  <GradientStop Color="#FF7DA5D8" Offset="0.85568704799474"/>
  <GradientStop Color="#FF074492" Offset="1"/>
  <GradientStop Color="#FF12345F" Offset="0"/>
  <GradientStop Color="#FF0A3F83" Offset="0"/>
</RadialGradientBrush>
```

3. You now need to define the style and control template for the button. The following XAML code defines a new control template for a button and includes property triggers for the `IsMouseOver` and `IsPressed` properties. It applies the appropriate brush you previously defined to each trigger respectively. Include the following code in the `Windows.Resources` declaration:

```
<Style x:Key="RadialButton" TargetType="Button">
  <Setter Property="SnapsToDevicePixels" Value="true"/>
  <Setter Property="OverridesDefaultStyle" Value="true"/>
  <Setter Property="MinHeight" Value="60"/>
  <Setter Property="MinWidth" Value="60"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="Button">
        <Ellipse
          Stroke="{x:Null}"
          Fill="{StaticResource ButtonOffStateColor}"
          HorizontalAlignment="Center"
          VerticalAlignment="Center"
          Width="60"
          Height="60"
          x:Name="Ellipse">
        </Ellipse>
        <ControlTemplate.Triggers>
          <Trigger Property="IsMouseOver" Value="true">
            <Setter TargetName="Ellipse" Property="Fill" Value="{StaticResource
              ButtonOnStateColor}" />
          </Trigger>
          <Trigger Property="IsPressed" Value="true">
            <Setter TargetName="Ellipse" Property="Fill" Value="{StaticResource
              ButtonPressedStateColor}" />
            <Setter TargetName="Ellipse" Property="Stroke" Value="Black" />
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

4. Add a Button declaration to the default window Grid and apply the `RadialButton` style as follows:

```
<Grid HorizontalAlignment="Center" VerticalAlignment="Center">
  <Button Style="{StaticResource RadialButton}" />
</Grid>
```

5. Compile the application and run the new Window file. The compile result is illustrated in Figures 9-12 through 9-14.

Figure 7-12 illustrates the default state of your button.

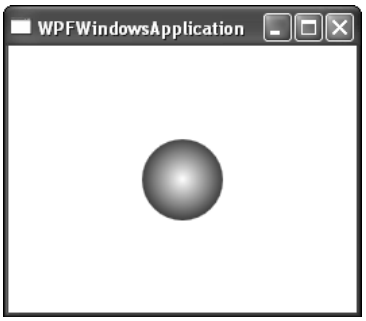


Figure 7-12

Figure 7-13 illustrates the `IsMouseOver` state of your button.

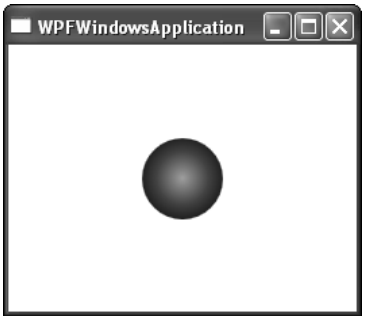


Figure 7-13

Figure 7-14 illustrates the `IsPressed` state of your button.

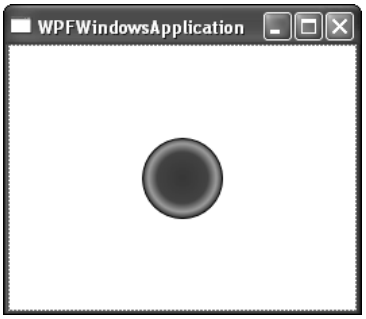


Figure 7-14