Professional WPF with C# and .NET 4.5

Christian Nagel

ABOUT THE AUTHORS

CHRISTIAN NAGEL is a Microsoft Regional Director and Microsoft MVP, an associate of thinktecture, and founder of CN innovation. A software architect and developer, he offers training and consulting on how to develop solutions using the Microsoft platform. He draws on more than 25 years of software development experience. Christian started his computing career with PDP 11 and VAX/VMS systems, covering a variety of languages and platforms. Since 2000, when .NET was just a technology preview, he has been working with various .NET technologies to build .NET solutions. Currently, he mainly coaches the development of Windows Store apps accessing Windows Azure services. With his profound knowledge of Microsoft technologies, he has written numerous books, and is certified as a Microsoft Certified Trainer and Professional Developer. Christian speaks at international conferences such as TechEd, Basta!, and TechDays, and he founded INETA Europe to support .NET user groups. You can contact Christian via his websites, `www.cninnovation.com` and `www.thinktecture.com`, and follow his tweets at @christiannagel.

JAY GLYNN started writing software more than 20 years ago, writing applications for the PICK operating system using PICK basic. Since then, he has created software using Paradox PAL and Object PAL, Delphi, VBA, Visual Basic, C, Java, and of course C#. He currently works for UL PureSafety as a senior software engineer writing web-based software.

MORGAN SKINNER began his computing career at a young age on the Sinclair ZX80 at school, where he was underwhelmed by some code a teacher had written and so began programming in assembly language. Since then he has used a wide variety of languages and platforms, including VAX Macro Assembler, Pascal, Modula2, Smalltalk, X86 assembly language, PowerBuilder, C/C++, VB, and currently C#. He's been programming in .NET since the PDC release in 2000, and liked it so much he joined Microsoft in 2001. He's now an independent consultant.

CONTENTS

PART II: VISUAL STUDIO

35

Core WPF

WHAT'S IN THIS CHAPTER?

- Shapes and geometry as the base drawing elements
- Scaling, rotating, and skewing with transformations
- Brushes to fill backgrounds
- WPF controls and their features
- Defining a layout with WPF panels
- Styles, templates, and resources
- Triggers and the Visual State Manager
- Animations
- 3-D

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `http://www.wrox.com/remtitle.cgi?isbn=1118314425` on the Download Code tab. The code for this chapter is divided into the following major examples:

- Shapes Demo
- Geometry Demo
- Transformation Demo
- Brushes Demo
- Decorations Demo
- Layout Demo
- Styles and Resources
- Trigger Demo
- Template Demo
- Animation Demo
- Visual State Demo
- 3D Demo

UNDERSTANDING WPF

Windows Presentation Foundation (*WPF*) is a library to create the UI for smart client applications. This chapter gives you broad information on the important concepts of WPF. It covers a large number of different controls and their categories, including how to arrange the controls with panels, customize the appearance using styles, resources, and templates, add some dynamic behavior with triggers and animations, and create 3-D with WPF.

One of the main advantages of WPF is that work can be easily separated between designers and developers. The outcome from the designer's work can directly be used by the developer. To make this possible, you need to understand *eXtensible Application Markup Language*, or *XAML*. Readers unfamiliar with XAML can read Chapter 29, "Core XAML," for information about its syntax.

The first topic of this chapter provides an overview of the class hierarchy and categories of classes that are used with WPF, including additional information to understand the principles of XAML. WPF consists of several assemblies containing thousands of classes. To help you navigate within this vast number of classes and find what you need, this section explains the class hierarchy and namespaces in WPF.

Namespaces

Classes from Windows Forms and WPF can easily be confused. The Windows Forms classes are located in the `System.Windows.Forms` namespace, and the WPF classes are located inside the namespace `System.Windows` and subnamespaces thereof, with the exception of `System.Windows.Forms`. For example, the `Button` class for Windows Forms has the full name `System.Windows.Forms.Button`, and the `Button` class for WPF has the full name `System.Windows.Controls.Button`.

Namespaces and their functionality within WPF are described in the following table.

NAMESPACE	DESCRIPTION
`System.Windows`	The core namespace of WPF. Here you can find core classes from WPF such as the `Application` class; classes for dependency objects, `DependencyObject` and `DependencyProperty`; and the base class for all WPF elements, `FrameworkElement`.
`System.Windows.Annotations`	Classes from this namespace are used for user-created annotations and notes on application data that are stored separately from the document. The namespace `System.Windows.Annotations.Storage` contains classes for storing annotations.
`System.Windows.Automation`	This namespace can be used for automation of WPF applications. Several sub-namespaces are available. `System.Windows.Automation.Peers` exposes WPF elements to automation — for example, `ButtonAutomationPeer` and `CheckBoxAutomationPeer`. The namespace `System.Windows.Automation.Provider` is needed if you create a custom automation provider.
`System.Windows.Baml2006`	This namespace contains the `Baml2006Reader` class, which is used to read binary markup language and produces XAML.
`System.Windows.Controls`	This namespace contains all the WPF controls, such as `Button`, `Border`, `Canvas`, `ComboBox`, `Expander`, `Slider`, `ToolTip`, `TreeView`, and the like. In the namespace `System.Windows.Controls.Primitives`, you can find classes to be used within complex controls, such as `Popup`, `ScrollBar`, `StatusBar`, `TabPanel`, and so on.

System.Windows.Converters	This namespace contains classes for data conversion. Don't expect to find all converter classes in this namespace; core converter classes are defined in the namespace System.Windows.
System.Windows.Data	This namespace is used by WPF data binding. An important class in this namespace is the Binding class, which is used to define the binding between a WPF target element and a CLR source. Data binding is covered in Chapter 36, "Business Applications with WPF."
System.Windows.Documents	When working with documents, you can find many helpful classes in this namespace. FixedDocument and FlowDocument are content elements that can contain other elements from this namespace. With classes from the namespace System.Windows.Documents.Serialization you can write documents to disk. The classes from this namespace are explained in Chapter 37, "Creating Documents with WPF."
System.Windows.Ink	With the increasingly popular Windows Tablet PC and Ultra Mobile PCs, ink can be used for user input. The namespace System.Windows.Ink contains classes to deal with ink input.
System.Windows.Input	Contains several classes for command handling, keyboard inputs, working with a stylus, and so on
System.Windows.Interop	For integration of WPF with native Window handles from the Windows API and Windows Forms, you can find classes in this namespace.
System.Windows.Markup	Helper classes for XAML markup code are located in this namespace.
System.Windows.Media	To work with images, audio, and video content, you can use classes in this namespace.
System.Windows.Navigation	Contains classes for navigation between windows
System.Windows.Resources	Contains supporting classes for resources
System.Windows.Shapes	Core classes for the UI are located in this namespace: Line, Ellipse, Rectangle, and the like.
System.Windows.Threading	WPF elements are bound to a single thread. In this namespace, you can find classes to deal with multiple threads—for example, the Dispatcher class belongs to this namespace.
System.Windows.Xps	XML Paper Specification (XPS) is a document specification that is also supported by Microsoft Word. In the namespaces System .Windows.Xps, System .Windows.Xps.Packaging and System.Windows.Xps.Serialization, you can find classes to create and stream XPS documents.

Class Hierarchy

WPF consists of thousands of classes within a deep hierarchy. For an overview of the relationships between the classes, see Figure 35-1. Some classes and their functionality are described in the following table.

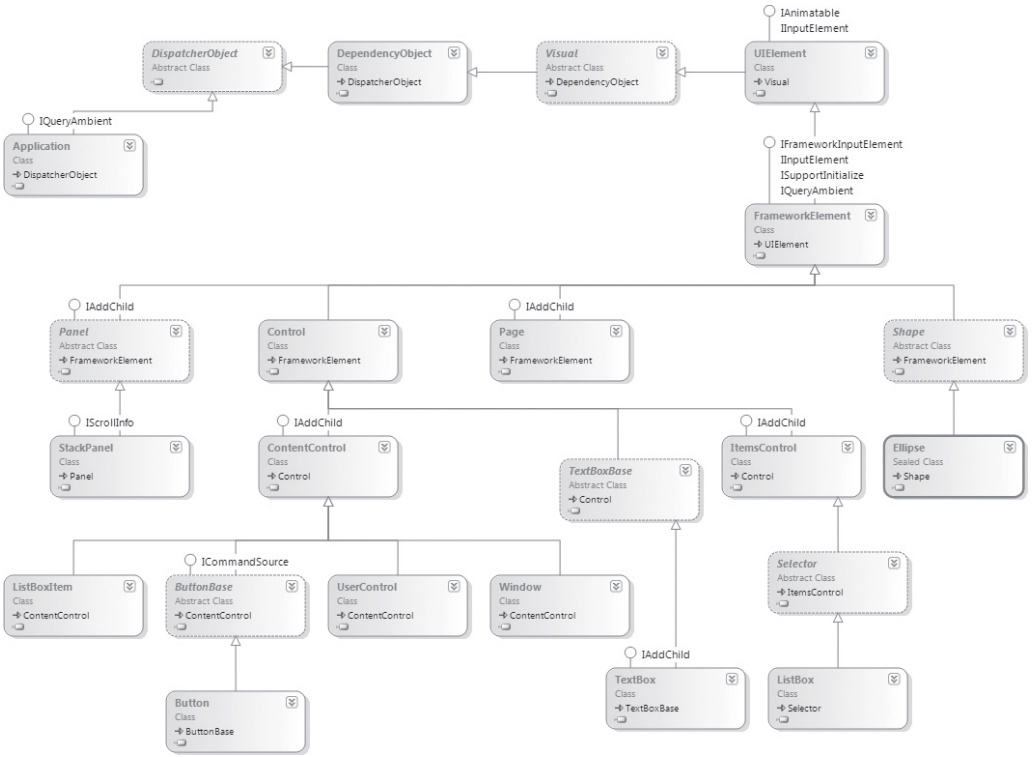


FIGURE 35-1

CLASS	DESCRIPTION
`DispatcherObject`	An abstract base class for classes that are bound to one thread. WPF controls require that methods and properties be invoked only from the creator thread. Classes derived from `DispatcherObject` have an associated `Dispatcher` object that can be used to switch the thread.
`Application`	In a WPF application, one instance of the `Application` class is created. This class implements a singleton pattern for access to the application windows, resources, and properties.
`DependencyObject`	This is the base class for all classes that support dependency properties. Dependency properties are discussed in Chapter 29, "Core XAML."
`Visual`	The base class for all visual elements. This class includes features for hit testing and transformation.
`UIElement`	The abstract base class for all WPF elements that need basic presentation features. This class provides tunneling and bubbling events for mouse moves, drag and drop, and key clicks. It exposes virtual methods for rendering that can be overridden by derived classes, and it provides methods for layout. As WPF does not use Window handles, you can consider this class equivalent to Window handles.

CLASS	DESCRIPTION
FrameworkElement	FrameworkElement is derived from the base class UIElement and implements the default behavior of the methods defined by the base class.
Shape	Base class for all shape elements, such as Line, Ellipse, Polygon, and Rectangle
Control	Control derives from FrameworkElement and is the base class for all user-interactive elements.
ContentControl	Base class for all controls that have a single content (for example, Label, Button). The default style of a content control may be limited, but it is possible to change the look by using templates.
ItemsControl	Base class for all controls that contain a collection of items as content (for example, ListBox, ComboBox)
Panel	This class derives from FrameworkElement and is the abstract base class for all panels. Panel has a Children property for all UI elements within the panel and defines methods for arranging the child controls. Classes derived from Panel define different behavior regarding how the children are organized—for example, WrapPanel, StackPanel, Canvas, and Grid.

As this brief introduction demonstrates, WPF classes have a deep hierarchy. This chapter and the next few chapters cover their core functionality, but it is not possible to provide comprehensive coverage all the WPF features in this book.

SHAPES

Shapes are the core elements of WPF. With shapes you can draw two-dimensional graphics using rectangles, lines, ellipses, paths, polygons, and polylines that are represented by classes derived from the abstract base class Shape. Shapes are defined in the namespace System.Windows.Shapes.

The following XAML example (code file ShapesDemo/MainWindow.xaml) draws a yellow face consisting of an ellipse for the face, two ellipses for the eyes, two ellipses for the pupils in the eyes, and a path for the mouth:

```
<Window x:Class="ShapesDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="300" Width="300">
  <Canvas>
    <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
        Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
    <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
      <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
          <GradientStop Offset="0.1" Color="DarkGreen" />
          <GradientStop Offset="0.7" Color="Transparent" />
        </LinearGradientBrush>
      </Ellipse.Fill>
    </Ellipse>
    <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20" Stroke="Blue"
        StrokeThickness="3" Fill="White" />
    <Ellipse Canvas.Left="40" Canvas.Top="43" Width="6" Height="5" Fill="Black" />
```

```
    <Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20" Stroke="Blue"
        StrokeThickness="3" Fill="White" />
    <Ellipse Canvas.Left="75" Canvas.Top="43" Width="6" Height="5" Fill="Black" />
    <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
        Data="M 40,74 Q 57,95 80,74 " />
  </Canvas>

</Window>
```

FIGURE 35-2

Figure 35-2 shows the result of the XAML code.

All these WPF elements can be accessed programmatically, even if they are buttons or shapes, such as lines or rectangles. Setting the `Name` or `x:Name` property with the `Path` element to `mouth` enables you to access this element programmatically with the variable name `mouth`:

```
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
    Data="M 40,74 Q 57,95 80,74 " />
```

FIGURE 35-3

In the code-behind `Data` property of the `Path` element (code file `ShapesDemo/MainWindow.xaml.cs`), `mouth` is set to a new geometry. For setting the path, the `Path` class supports `PathGeometry` with path markup syntax. The letter `M` defines the starting point for the path; the letter `Q` specifies a control point and an endpoint for a quadratic Bézier curve. Running the application results in the image shown in Figure 35-3.

```
public MainWindow()
{
  InitializeComponent();
  mouth.Data = Geometry.Parse("M 40,92 Q 57,75 80,92");
}
```

The following table describes the shapes available in the namespace `System.Windows.Shapes`.

SHAPE CLASS	DESCRIPTION
`Line`	You can draw a line from the coordinates `X1.Y1` to `X2.Y2`.
`Rectangle`	Enables drawing a rectangle by specifying `Width` and `Height`
`Ellipse`	With the `Ellipse` class, you can draw an ellipse.
`Path`	You can use the `Path` class to draw a series of lines and curves. The `Data` property is a `Geometry` type. You can do the drawing by using classes that derive from the base class `Geometry`, or you can use the path markup syntax to define geometry.
`Polygon`	Enables drawing a closed shape formed by connected lines with the `Polygon` class. The polygon is defined by a series of `Point` objects assigned to the `Points` property.
`Polyline`	Similar to the `Polygon` class, you can draw connected lines with `Polyline`. The difference is that the polyline does not need to be a closed shape.

GEOMETRY

One of the shapes, `Path`, uses `Geometry` for its drawing. `Geometry` elements can also be used in other places, such as with a `DrawingBrush`.

In some ways, geometry elements are very similar to shapes. Just as there are `Line`, `Ellipse`, and `Rectangle` shapes, there are also geometry elements for these drawings: `LineGeometry`, `EllipseGeometry`, and `RectangleGeometry`. There are also big differences between shapes and geometries. A `Shape` is a `FrameworkElement` and can be used with any class that supports `UIElement` as its children. `FrameworkElement` derives from `UIElement`. Shapes participate with the layout system and render themselves. The `Geometry` class can't render itself and has fewer features and less overhead than `Shape`. The `Geometry` class derives from the `Freezable` base class and can be shared from multiple threads.

The `Path` class uses `Geometry` for its drawing. The geometry can be set with the `Data` property of the `Path`. Simple geometry elements that can be set are `EllipseGeometry` for drawing an ellipse, `LineGeometry` for drawing a line, and `RectangleGeometry` for drawing a rectangle. Combining multiple geometries, as demonstrated in the next example, can be done with `CombinedGeometry`.

`CombinedGeometry` has the properties `Geometry1` and `Geometry2` and allows them to combine with `GeometryCombineMode` to form a `Union`, `Intersect`, `Xor`, and `Exclude`. `Union` merges the two geometries. With `Intersect`, only the area that is covered with both geometries is visible. `Xor` contrasts with `Intersect` by showing the area that is covered by one of the geometries but not showing the area covered by both. `Exclude` shows the area of the first geometry minus the area of the second geometry.

The following example (code file `GeometryDemo/MainWindow.xaml`) combines an `EllipseGeometry` and a `RectangleGeometry` to form a union, as shown in Figure 35-4.

FIGURE 35-4

```
<Path Canvas.Top="0" Canvas.Left="250" Fill="Blue" Stroke="Black" >
  <Path.Data>
    <CombinedGeometry GeometryCombineMode="Union">
      <CombinedGeometry.Geometry1>
        <EllipseGeometry Center="80,60" RadiusX="80" RadiusY="40" />
      </CombinedGeometry.Geometry1>
      <CombinedGeometry.Geometry2>
        <RectangleGeometry Rect="30,60 105 50" />
      </CombinedGeometry.Geometry2>
    </CombinedGeometry>
  </Path.Data>
</Path>
```

Geometries can also be created by using segments. The geometry class `PathGeometry` uses segments for its drawing. The following code segment uses the `BezierSegment` and `LineSegment` elements to build one red and one green figure, as shown in Figure 35-5. The first `BezierSegment` draws a Bézier curve between the points 70,40, which is the starting point of the figure, and 150,63 with control points 90,37 and 130,46. The following `LineSegment` uses the ending point of the Bézier curve and draws a line to 120,110:

```
<Path Canvas.Left="0" Canvas.Top="0" Fill="Red" Stroke="Blue"
      StrokeThickness="2.5">
  <Path.Data>
    <GeometryGroup>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="70,40" IsClosed="True">
            <PathFigure.Segments>
              <BezierSegment Point1="90,37" Point2="130,46" Point3="150,63" />
              <LineSegment Point="120,110" />
              <BezierSegment Point1="100,95" Point2="70,90" Point3="45,91" />
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
```

```
        </PathGeometry>
      </GeometryGroup>
    </Path.Data>
  </Path>

  <Path Canvas.Left="0" Canvas.Top="0" Fill="Green" Stroke="Blue"
        StrokeThickness="2.5">
    <Path.Data>
      <GeometryGroup>
        <PathGeometry>
          <PathGeometry.Figures>
            <PathFigure StartPoint="160,70">
              <PathFigure.Segments>
                <BezierSegment Point1="175,85" Point2="200,99"
                               Point3="215,100" />
                <LineSegment Point="195,148" />
                <BezierSegment Point1="174,150" Point2="142,140"
                               Point3="129,115" />
                <LineSegment Point="160,70" />
              </PathFigure.Segments>
            </PathFigure>
          </PathGeometry.Figures>
        </PathGeometry>
      </GeometryGroup>
    </Path.Data>
  </Path>
```

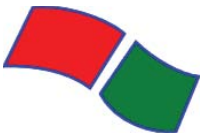


FIGURE 35-5

Other than the `BezierSegment` and `LineSegment` elements, you can use `ArcSegment` to draw an elliptical arc between two points. With `PolyLineSegment` you can define a set of lines: `PolyBezierSegment` consists of multiple Bézier curves, `QuadraticBezierSegment` creates a quadratic Bézier curve, and `PolyQuadraticBezierSegment` consists of multiple quadratic Bézier curves.

A speedy drawing can be created with `StreamGeometry`. Programmatically, the figure can be defined by creating lines, Bézier curves, and arcs with members of the `StreamGeometryContext` class. With XAML, path markup syntax can be used. You can use path markup syntax with the `Data` property of the `Path` class to define `StreamGeometry`. Special characters define how the points are connected. In the following example, `M` marks the start point, `L` is a line command to the point specified, and `Z` is the Close command to close the figure. Figure 35-6 shows the result. The path markup syntax allows more commands such as horizontal lines (`H`), vertical lines (`V`), cubic Bézier curves (`C`), quadratic Bézier curves (`Q`), smooth cubic Bézier curves (`S`), smooth quadratic Bézier curves (`T`), and elliptical arcs (`A`):

FIGURE 35-6

```
<Path Canvas.Left="0" Canvas.Top="200" Fill="Yellow" Stroke="Blue"
      StrokeThickness="2.5"
      Data="M 120,5 L 128,80 L 220,50 L 160,130 L 190,220 L 100,150
            L 80,230 L 60,140 L0,110 L70,80 Z" StrokeLineJoin="Round">
</Path>
```

TRANSFORMATION

Because WPF is vector-based, you can resize every element. In the next example, the vector-based graphics are now scaled, rotated, and skewed. Hit testing (for example, with mouse moves and mouse clicks) still works but without the need for manual position calculation.

Adding the `ScaleTransform` element to the `LayoutTransform` property of the `Canvas` element, as shown here (code file `TransformationDemo/MainWindow.xaml`), resizes the content of the complete canvas by 1.5 in the x and *y* axes:

```
<Canvas.LayoutTransform>
  <ScaleTransform ScaleX="1.5" ScaleY="1.5" />
</Canvas.LayoutTransform>
```

Rotation can be done in a similar way as scaling. Using the `RotateTransform` element you can define the `Angle` for the rotation:

```
<Canvas.LayoutTransform>
  <RotateTransform Angle="40" />
</Canvas.LayoutTransform>
```

For skewing, you can use the `SkewTransform` element. With skewing you can assign angles for the *x* and *y* axes:

```
<Canvas.LayoutTransform>
  <SkewTransform AngleX="20" AngleY="25" />
</Canvas.LayoutTransform>
```

To rotate and skew together, it is possible to define a `TransformGroup` that contains both `RotateTransform` and `SkewTransform`. You can also define a `MatrixTransform` whereby the `Matrix` element specifies the properties `M11` and `M22` for stretch and `M12` and `M21` for skew:

```
<Canvas.LayoutTransform>
  <MatrixTransform>
    <MatrixTransform.Matrix>
      <Matrix M11="0.8" M22="1.6" M12="1.3" M21="0.4" />
    </MatrixTransform.Matrix>
  </MatrixTransform>
</Canvas.LayoutTransform>
```

Figure 35-7 shows the result of all these transformations. The figures are placed inside a `StackPanel`. Starting from the left, the first image is resized, the second image is rotated, the third image is skewed, and the fourth image uses a matrix for its transformation. To highlight the differences between these four images, the `Background` property of the `Canvas` elements is set to different colors.

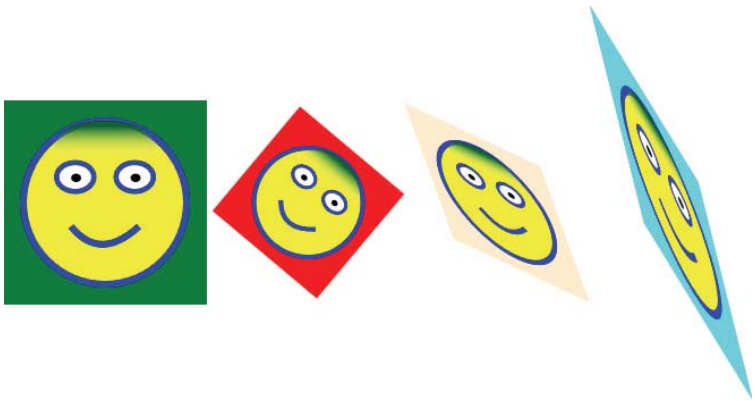


FIGURE 35-7

> **NOTE** *In addition to* `LayerTransform` *there's also a* `RenderTransform`. `LayerTransform` *happens before the layout phase and* `RenderTransform` *happens after.*

BRUSHES

This section demonstrates how to use the brushes that WPF offers for drawing backgrounds and foregrounds. The examples in this section reference Figure 35-8, which shows the effects of using various brushes within a `Path` and the `Background` of `Button` elements.

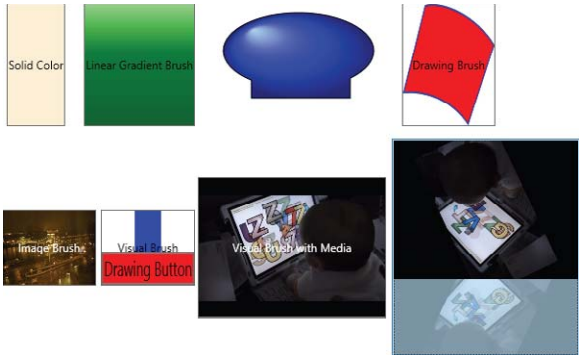


FIGURE 35-8

SolidColorBrush

The first button in Figure 35-8 uses the `SolidColorBrush`, which, as the name suggests, uses a solid color. The complete area is drawn with the same color.

You can define a solid color just by setting the `Background` attribute to a string that defines a solid color. The string is converted to a `SolidColorBrush` element with the help of the `BrushValueSerializer`:

```
<Button Height="30" Background="PapayaWhip">Solid Color</Button>
```

Of course, you will get the same effect by setting the `Background` child element and adding a `SolidColorBrush` element as its content (code file `BrushesDemo/MainWindow.xaml`). The first button in the application is using `PapayaWhip` as the solid background color:

```
<Button Content="Solid Color" Margin="10">
  <Button.Background>
    <SolidColorBrush Color="PapayaWhip" />
  </Button.Background>
</Button>
```

LinearGradientBrush

For a smooth color change, you can use the `LinearGradientBrush`, as the second button shows. This brush defines the `StartPoint` and `EndPoint` properties. With this, you can assign two-dimensional coordinates for the linear gradient. The default gradient is diagonal linear from `0,0` to `1,1`. By defining different values, the gradient can take different directions. For example, with a `StartPoint` of `0,0` and an `EndPoint` of `0,1`, you get a vertical gradient. The `StartPoint` and `EndPoint` value of `1,0` creates a horizontal gradient.

With the content of this brush, you can define the color values at the specified offsets with the `GradientStop` element. Between the stops, the colors are smoothed (code file `BrushesDemo/MainWindow.xaml`):

```
<Button Content="Linear Gradient Brush" Margin="10">
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <GradientStop Offset="0" Color="LightGreen" />
      <GradientStop Offset="0.4" Color="Green" />
      <GradientStop Offset="1" Color="DarkGreen" />
```

```
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

RadialGradientBrush

With the `RadialGradientBrush` you can smooth the color in a radial way. In Figure 35-8, the third element is a `Path` that uses `RadialGradientBrush`. This brush defines the color start with the `GradientOrigin` point (code file `BrushesDemo/MainWindow.xaml`):

```
<Canvas Width="200" Height="150">
  <Path Canvas.Top="0" Canvas.Left="20" Stroke="Black" >
    <Path.Fill>
      <RadialGradientBrush GradientOrigin="0.2,0.2">
        <GradientStop Offset="0" Color="LightBlue" />
        <GradientStop Offset="0.6" Color="Blue" />
        <GradientStop Offset="1.0" Color="DarkBlue" />
      </RadialGradientBrush>
    </Path.Fill>
    <Path.Data>
      <CombinedGeometry GeometryCombineMode="Union">
        <CombinedGeometry.Geometry1>
          <EllipseGeometry Center="80,60" RadiusX="80" RadiusY="40" />
        </CombinedGeometry.Geometry1>
        <CombinedGeometry.Geometry2>
          <RectangleGeometry Rect="30,60 105 50" />
        </CombinedGeometry.Geometry2>
      </CombinedGeometry>
    </Path.Data>
  </Path>
</Canvas>
```

DrawingBrush

The `DrawingBrush` enables you to define a drawing that is created with the brush. The drawing that is shown with the brush is defined within a `GeometryDrawing` element. The `GeometryGroup`, which you can see within the `Geometry` property, consists of the `Geometry` elements discussed earlier in this chapter (code file `BrushesDemo/MainWindow.xaml`):

```
<Button Content="Drawing Brush" Margin="10" Padding="10">
  <Button.Background>
    <DrawingBrush>
      <DrawingBrush.Drawing>
        <GeometryDrawing Brush="Red">
          <GeometryDrawing.Pen>
            <Pen>
              <Pen.Brush>
                <SolidColorBrush>Blue</SolidColorBrush>
              </Pen.Brush>
            </Pen>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <PathGeometry>
              <PathGeometry.Figures>
                <PathFigure StartPoint="70,40">
                  <PathFigure.Segments>
                    <BezierSegment Point1="90,37" Point2="130,46"
                                   Point3="150,63" />
                    <LineSegment Point="120,110" />
                    <BezierSegment Point1="100,95" Point2="70,90"
                                   Point3="45,91" />
                    <LineSegment Point="70,40" />
```

```
                    </PathFigure.Segments>
                  </PathFigure>
                </PathGeometry.Figures>
              </PathGeometry>
            </GeometryDrawing.Geometry>
          </GeometryDrawing>
        </DrawingBrush.Drawing>
      </DrawingBrush>
    </Button.Background>
  </Button>
```

ImageBrush

To load an image into a brush, you can use the `ImageBrush` element. With this element, the image defined by the `ImageSource` property is displayed. The image can be accessed from the file system or from a resource within the assembly. In the example (code file `BrushesDemo/MainWindow.xaml`), the image is added as a resource to the assembly and referenced with the assembly and resource names:

```
<Button Content="Image Brush" Width="100" Height="80" Margin="5"
        Foreground="White">
  <Button.Background>
    <ImageBrush ImageSource="/BrushesDemo;component/Budapest.jpg" />
  </Button.Background>
</Button>
```

VisualBrush

The `VisualBrush` enables you to use other WPF elements in a brush. The following example (code file `BrushesDemo/MainWindow.xaml`) adds a WPF element to the `Visual` property. The sixth element in Figure 35-8 contains a `Rectangle` and a `Button`:

```
<Button Content="Visual Brush" Width="100" Height="80">
  <Button.Background>
    <VisualBrush>
      <VisualBrush.Visual>
        <StackPanel Background="White">
          <Rectangle Width="25" Height="25" Fill="Blue" />
          <Button Content="Drawing Button" Background="Red" />
        </StackPanel>
      </VisualBrush.Visual>
    </VisualBrush>
  </Button.Background>
</Button>
```

You can add any `UIElement` to the `VisualBrush`. For example, you can play a video by using the `MediaElement`:

```
<Button Content="Visual Brush with Media" Width="200" Height="150"
    Foreground="White">
  <Button.Background>
    <VisualBrush>
      <VisualBrush.Visual>
        <MediaElement Source="./Stephanie.wmv" />
      </VisualBrush.Visual>
    </VisualBrush>
  </Button.Background>
</Button>
```

You can also use the `VisualBrush` to create interesting effects such as reflection. The button coded in the following example contains a `StackPanel` that itself contains a `MediaElement` playing a video and a `Border`. The `Border` contains a `Rectangle` that is filled with a `VisualBrush`. This brush defines an opacity

value and a transformation. The `Visual` property is bound to the `Border` element. The transformation is achieved by setting the `RelativeTransform` property of the `VisualBrush`. This transformation uses relative coordinates. By setting `ScaleY` to `-1`, a reflection in the *y* axis is done. `TranslateTransform` moves the transformation in the y axis so that the reflection is below the original object. You can see the result in the eighth element in Figure 35-8.

> **NOTE** *Data binding and the* `Binding` *element used here are explained in detail in Chapter 36, "Business Applications with WPF."*

```
<Button Width="200" Height="200" Foreground="White">
  <StackPanel>
    <MediaElement x:Name="reflected" Source="./Stephanie.wmv" />
    <Border Height="100">
      <Rectangle>
        <Rectangle.Fill>
          <VisualBrush Opacity="0.35" Stretch="None"
              Visual="{Binding ElementName=reflected}">
            <VisualBrush.RelativeTransform>
              <TransformGroup>
                <ScaleTransform ScaleX="1" ScaleY="-1" />
                <TranslateTransform Y="1" />
              </TransformGroup>
            </VisualBrush.RelativeTransform>
          </VisualBrush>
        </Rectangle.Fill>
      </Rectangle>
    </Border>
  </StackPanel>
</Button>
```

CONTROLS

Because you can use hundreds of controls with WPF, they are categorized into the following groups, each of which is described in the following sections.

Simple Controls

Simple controls are controls that don't have a `Content` property. With the `Button` class, you have seen that the `Button` can contain any shape, or any element you like. This is not possible with simple controls. The following table describes the simple controls.

SIMPLE CONTROL	DESCRIPTION
`PasswordBox`	This control is used to enter a password and has specific properties for password input, such as `PasswordChar`, to define the character that should be displayed as the user enters the password, or `Password`, to access the password entered. The `PasswordChanged` event is invoked as soon as the password is changed.
`ScrollBar`	This control contains a `Thumb` that enables the user to select a value. A scrollbar can be used, for example, if a document doesn't fit on the screen. Some controls contain scrollbars that are displayed if the content is too big.

continues

continued

`ProgressBar`	Indicates the progress of a lengthy operation.
`Slider`	Enables users to select a range of values by moving a `Thumb`. `ScrollBar`, `ProgressBar`, and `Slider` are derived from the same base class, `RangeBase`.
`TextBox`	Used to display simple, unformatted text
`RichTextBox`	Supports rich text with the help of the `FlowDocument` class. `RichTextBox` and `TextBox` are derived from the same base class, `TextBoxBase`.
`Calendar`	Displays a month, year, or decade. The user can select a date or range of dates.
`DatePicker`	Opens a calendar onscreen for date selection by the user

> **NOTE** *Although simple controls do not have a* `Content` *property, you can completely customize the look of a control by defining a template. Templates are discussed later in this chapter in the section Templates.*

Content Controls

A `ContentControl` has a `Content` property, with which you can add any content to the control. The `Button` class derives from the base class `ContentControl`, so you can add any content to this control. In a previous example, you saw a `Canvas` control within the `Button`. Content controls are described in the following table.

CONTENTCONTROL CONTROLS	DESCRIPTION

`ButtonRepeat`
`ButtonToggle`
`ButtonCheckBox`
`RadioButton`
	The classes `Button`, `RepeatButton`, `ToggleButton`, and `GridViewColumnHeader` are derived from the same base class, `ButtonBase`. All buttons react to the `Click` event. The `RepeatButton` raises the `Click` event repeatedly until the button is released. `ToggleButton` is the base class for `CheckBox` and `RadioButton`. These buttons have an on and off state. The `CheckBox` can be selected and cleared by the user; the `RadioButton` can be selected by the user. Clearing the `RadioButton` must be done programmatically.

`Label`	The `Label` class represents the text label for a control. This class also has support for access keys—for example, a menu command.
`Frame`	The `Frame` control supports navigation. You can navigate to a page's content with the `Navigate` method. If the content is a web page, then the `WebBrowser` control is used for display.
`ListBoxItem`	An item inside a `ListBox` control
`StatusBarItem`	An item inside a `StatusBar` control
`ScrollViewer`	A content control that includes scrollbars. You can put any content in this control; the scrollbars are displayed as needed.
`ToolTip`	Creates a pop-up window to display additional information for a control.

CONTENTCONTROL CONTROLS	DESCRIPTION
`UserControl`	Using this class as a base class provides a simple way to create custom controls. However, the `UserControl` base class does not support templates.
`Window`	This class enables you to create windows and dialogs. It includes a frame with minimize/maximize/close buttons and a system menu. When showing a dialog, you can use the `ShowDialog` method; the `Show` method opens a window.
`NavigationWindow`	This class derives from the `Window` class and supports content navigation.

Only a `Frame` control is contained within the `Window` of the following XAML code (code file `FrameDemo/MainWindow.xaml`). The `Source` property is set to `http://www.cninnovation.com`, so the `Frame` control navigates to this website, as shown in Figure 35-9.

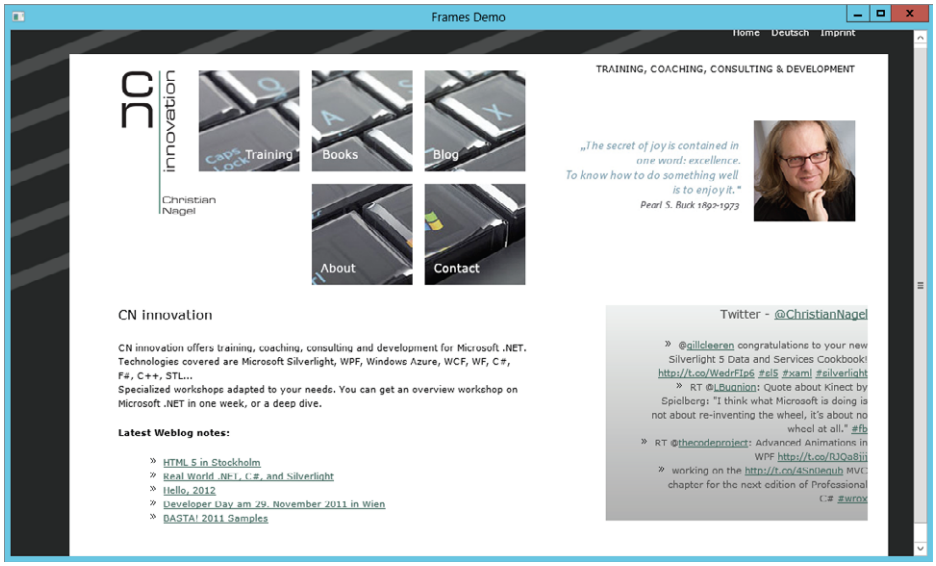


FIGURE 35-9

```
<Window x:Class="FrameDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Frames Demo" Height="240" Width="500">
  <Frame Source="http://www.cninnovation.com" />
</Window>
```

Headered Content Controls

Content controls with a header are derived from the base class `HeaderedContentControl`, which itself is derived from the base class `ContentControl`. The `HeaderedContentControl` class has a property `Header` to define the content of the header and `HeaderTemplate` for complete customization of the header. The controls derived from the base class `HeaderedContentControl` are listed in the following table.

HEADEREDCONTENTCONTROL	DESCRIPTION
`Expander`	This control enables you to create an "advanced" mode with a dialog that, by default, does not show all information but can be expanded by the user for additional details. In the unexpanded mode, header information is shown. In expanded mode, the content is visible.
`GroupBox`	Provides a border and a header to group controls
`TabItem`	These controls are items within the class `TabControl`. The `Header` property of the `TabItem` defines the content of the header shown with the tabs of the `TabControl`.

A simple use of the `Expander` control is shown in the next example. The `Expander` control has the property `Header` set to `Click for more`. This text is displayed for expansion. The content of this control is shown only if the control is expanded. Figure 35-10 shows the application with a collapsed `Expander` control, and Figure 35-11 shows the same application with an expanded `Expander` control. The code (code file `ExpanderDemo/MainWindow.xaml`) is as follows:

```
<Window x:Class="ExpanderDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Expander Demo" Height="240" Width="500">
  <StackPanel>
    <TextBlock>Short information</TextBlock>
    <Expander Header="Additional Information">
      <Border Height="200" Width="200" Background="Yellow">
        <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center">
          More information here!
        </TextBlock>
      </Border>
    </Expander>
  </StackPanel>

</Window>
```

> **NOTE** *To make the header text of the* `Expander` *control change when the control is expanded, you can create a trigger. Triggers are explained later in this chapter in the section Triggers.*

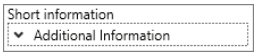


FIGURE 35-10

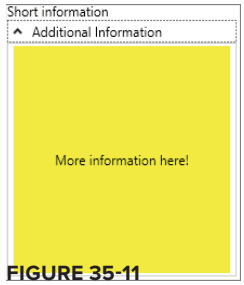


FIGURE 35-11

Items Controls

The `ItemsControl` class contains a list of items that can be accessed with the `Items` property. Classes derived from `ItemsControl` are shown in the following table.

ITEMSCONTROL	DESCRIPTION
`Menu` and `ContextMenu`	These classes are derived from the abstract base class `MenuBase`. You can offer menus to the user by placing `MenuItem` elements in the items list and associating commands.
`StatusBar`	This control is usually shown at the bottom of an application to give status information to the user. You can put `StatusBarItem` elements inside a `StatusBar` list.
`TreeView`	Use this control for a hierarchical display of items.
`ListBox` `ComboBox` `TabControl`	These have the same abstract base class, `Selector`. This base class makes it possible to select items from a list. The `ListBox` displays the items from a list. The `ComboBox` has an additional `Button` control to display the items only if the button is clicked. With `TabControl`, content can be arranged in tabular form.
`DataGrid`	This control is a customizable grid that displays data. It is discussed in detail in the next chapter.

Headered Items Controls

`HeaderedItemsControl` is the base class of controls that include items but also have a header. The class `HeaderedItemsControl` is derived from `ItemsControl`.

Classes derived from `HeaderedItemsControl` are listed in the following table.

HEADEREDITEMSCONTROL	DESCRIPTION
`MenuItem`	The menu classes `Menu` and `ContextMenu` include items of the `MenuItem` type. Menu items can be connected to commands, as the `MenuItem` class implements the interface `ICommandSource`.
`TreeViewItem`	This class can include items of type `TreeViewItem`.
`ToolBar`	This control is a container for a group of controls, usually `Button` and `Separator` elements. You can place the `ToolBar` inside a `ToolBarTray` that handles the rearranging of `ToolBar` controls.

Decoration

You can add decorations to a single element with the `Decorator` class. `Decorator` is a base class that has derivations such as `Border`, `Viewbox`, and `BulletDecorator`. Theme elements such as `ButtonChrome` and `ListBoxChrome` are also decorators.

The following example (code file `DecorationsDemo/MainWindow.xaml`) demonstrates a `Border`, `Viewbox`, and `BulletDecorator`, as shown in Figure 35-12. The `Border` class decorates the `Children` element by adding a border around it. You can define a brush and the thickness of the border, the background, the radius of the corner, and the padding of its children:

```
<Border BorderBrush="Violet" BorderThickness="5.5">
  <Label>Label with a border</Label>
</Border>
```

FIGURE 35-12

The Viewbox stretches and scales its child to the available space. The StretchDirection and Stretch properties are specific to the functionality of the Viewbox. These properties enable specifying whether the child is stretched in both directions, and whether the aspect ratio is preserved:

```
<Viewbox StretchDirection="Both" Stretch="Uniform">
  <Label>Label with a viewbox</Label>
</Viewbox>
```

The BulletDecorator class decorates its child with a bullet. The child can be any element (in this example, a TextBlock). Similarly, the bullet can also be any element. The example uses an Image, but you can use any UIElement:

```
<BulletDecorator>
  <BulletDecorator.Bullet>
    <Image Width="25" Height="25" Margin="5" HorizontalAlignment="Center"
           VerticalAlignment="Center"
           Source="/DecorationsDemo;component/images/apple1.jpg" />
  </BulletDecorator.Bullet>
  <BulletDecorator.Child>
    <TextBlock VerticalAlignment="Center" Padding="8">Granny Smith</TextBlock>
  </BulletDecorator.Child>
</BulletDecorator>
```

LAYOUT

To define the layout of the application, you can use a class that derives from the Panel base class. A layout container needs to do two main tasks: measure and arrange. With *measuring*, the container asks its children for the preferred sizes. Because the full size requested by the controls might not be available, the container determines the available sizes and *arranges* the positions of its children accordingly. This section discusses several available layout containers.

StackPanel

The Window can contain just a single element as content, but if you want more than one element inside it, you can use a StackPanel as a child of the Window, and add elements to the content of the StackPanel. The StackPanel is a simple container control that just shows one element after the other. The orientation of the StackPanel can be horizontal or vertical. The class ToolBarPanel is derived from StackPanel (code file LayoutDemo/StackPanelWindow.xaml):

```
<Window x:Class="LayoutDemo.StackPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StackPanelWindow" Height="300" Width="300">
  <StackPanel Orientation="Vertical">
    <Label>Label</Label>
    <TextBox>TextBox</TextBox>
    <CheckBox>CheckBox</CheckBox>
    <CheckBox>CheckBox</CheckBox>
    <ListBox>
      <ListBoxItem>ListBoxItem One</ListBoxItem>
      <ListBoxItem>ListBoxItem Two</ListBoxItem>
    </ListBox>
    <Button>Button</Button>
  </StackPanel>
</Window>
```

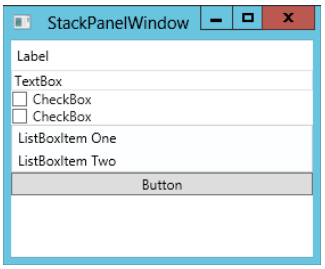


FIGURE 35-13

Figure 35-13 shows the child controls of the StackPanel organized vertically.

WrapPanel

The `WrapPanel` positions the children from left to right, one after the other, as long as they fit into the line, and then continues with the next line. The panel's orientation can be horizontal or vertical (code file `LayoutDemo/WrapPanelWindow.xaml`):

```
<Window x:Class="LayoutDemo.WrapPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WrapPanelWindow" Height="300" Width="300">
  <WrapPanel>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
    <Button Width="100" Margin="5">Button</Button>
  </WrapPanel>
</Window>
```

Figure 35-14 shows the output of the panel. If you resize the application, the buttons will be rearranged accordingly so that they fit into a line.

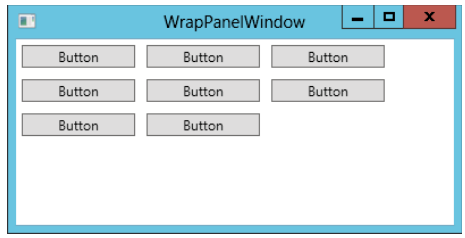


FIGURE 35-14

Canvas

`Canvas` is a panel that enables you to explicitly position controls. `Canvas` defines the attached properties `Left`, `Right`, `Top`, and `Bottom` that can be used by the children for positioning within the panel (code file `LayoutDemo/CanvasWindow.xaml`):

```
<Window x:Class="LayoutDemo.CanvasWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CanvasWindow" Height="300" Width="300">
  <Canvas Background="LightBlue">
    <Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>
    <TextBox Canvas.Top="30" Canvas.Left="120" Width="100" />
    <Button Canvas.Top="70" Canvas.Left="130" Content="Click Me!" Padding="5" />
  </Canvas>
</Window>
```

Figure 35-15 shows the output of the `Canvas` panel with the positioned children `Label`, `TextBox`, and `Button`.

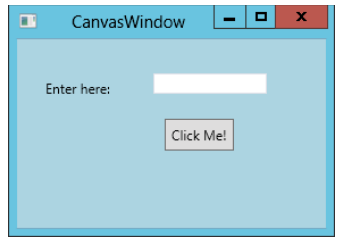


FIGURE 35-15

DockPanel

The `DockPanel` is very similar to the Windows Forms docking functionality. Here, you can specify the area in which child controls should be arranged. `DockPanel` defines the attached property `Dock`, which you can set in the children of the controls to the values `Left`, `Right`, `Top`, and `Bottom`. Figure 35-16 shows the outcome of text blocks with borders that are arranged in the dock

panel. For easier differentiation, different colors are specified for the various areas (code file LayoutDemo/DockPanelWindow.xaml):

```
<Window x:Class="LayoutDemo.DockPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DockPanelWindow" Height="300" Width="300">
  <DockPanel>
    <Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">
      <TextBlock>Menu</TextBlock>
    </Border>
    <Border Height="25" Background="Aqua" DockPanel.Dock="Top">
      <TextBlock>Ribbon</TextBlock>
    </Border>
    <Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">
      <TextBlock>Status</TextBlock>
    </Border>
    <Border Height="80" Background="Azure" DockPanel.Dock="Left">
      <TextBlock>Left Side</TextBlock>
    </Border>
    <Border Background="HotPink">
      <TextBlock>Remaining Part</TextBlock>
    </Border>
  </DockPanel>
</Window>
```

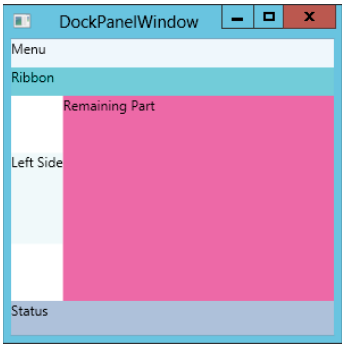


FIGURE 35-16

Grid

Using the Grid, you can arrange your controls with rows and columns. For every column, you can specify a ColumnDefinition. For every row, you can specify a RowDefinition. The following example code (code file LayoutDemo/GridWindow.xaml) lists two columns and three rows. With each column and row, you can specify the width or height. ColumnDefinition has a Width dependency property; RowDefinition has a Height dependency property. You can define the height and width in pixels, centimeters, inches, or points, or set it to Auto to determine the size depending on the content. The grid also allows *star sizing*, whereby the space for the rows and columns is calculated according to the available space and relative to other rows and columns. When providing the available space for a column, you can set the Width property to *. To have the size doubled for another column, you specify 2*. The sample code, which defines two columns and three rows, doesn't define additional settings with the column and row definitions; the default is the star sizing.

The grid contains several Label and TextBox controls. Because the parent of these controls is a grid, you can set the attached properties Column, ColumnSpan, Row, and RowSpan:

```
<Window x:Class="LayoutDemo.GridWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="GridWindow" Height="300" Width="300">
  <Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
```

```
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0"
           VerticalAlignment="Center" HorizontalAlignment="Center" Content="Title"
    />
    <Label Grid.Column="0" Grid.Row="1" VerticalAlignment="Center"
           Content="Firstname:" Margin="10" />
    <TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30" />
    <Label Grid.Column="0" Grid.Row="2" VerticalAlignment="Center"
           Content="Lastname:" Margin="10" />
    <TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30" />
  </Grid>
</Window>
```

The outcome of arranging controls in a grid is shown in Figure 35-17. For easier viewing of the columns and rows, the property `ShowGridLines` is set to `true`.

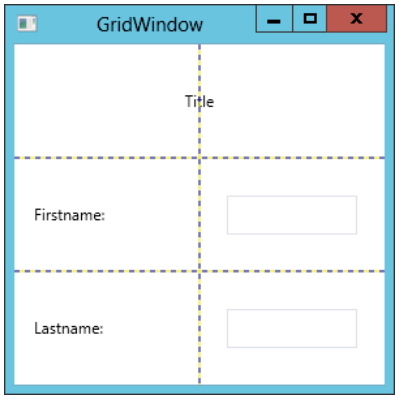


FIGURE 35-17

> **NOTE** *For a grid in which every cell is the same size, you can use the* `UniformGrid` *class.*

STYLES AND RESOURCES

You can define the look and feel of the WPF elements by setting properties, such as `FontSize` and `Background`, with the `Button` element (code file `StylesAndResources/MainWindow.xaml`):

```
<Button Width="150" FontSize="12" Background="AliceBlue" Content="Click Me!" />
```

Instead of defining the look and feel with every element, you can define styles that are stored with resources. To completely customize the look of controls, you can use templates and add them to resources.

Styles

The `Style` property of a control can be assigned to a `Style` element that has setters associated with it. A `Setter` element defines the `Property` and `Value` properties and sets a specified property to a value. In the following example (code file `StylesAndResources/MainWindow.xaml`), the `Background`, `FontSize`, and `FontWeight` properties are set. The `Style` is set to the `TargetType` `Button`, so that the properties of the `Button` can be directly accessed. If the `TargetType` of the style is not set, the properties can be accessed via `Button.Background`, `Button.FontSize`. This is especially important if you need to set properties of different element types:

```
<Button Width="150" Content="Click Me!">
  <Button.Style>
    <Style TargetType="Button">
      <Setter Property="Background" Value="Yellow" />
      <Setter Property="FontSize" Value="14" />
      <Setter Property="FontWeight" Value="Bold" />
    </Style>
  </Button.Style>
</Button>
```

Setting the `Style` directly with the `Button` element doesn't really help a lot in regard to style sharing. Styles can be put into resources. Within the resources you can assign styles to specific elements, assign a style to all elements of a type, or use a key for the style. To assign a style to all elements of a type, use the `TargetType` property of the `Style` and assign it to a `Button` by specifying the `x:Type` markup extension `{x:Type Button}`. To define a style that needs to be referenced, `x:Key` must be set:

```
<Window.Resources>
  <Style TargetType="{x:Type Button}">
    <Setter Property="Background" Value="LemonChiffon" />
    <Setter Property="FontSize" Value="18" />
  </Style>
  <Style x:Key="ButtonStyle">
    <Setter Property="Button.Background" Value="Red" />
    <Setter Property="Button.Foreground" Value="White" />
    <Setter Property="Button.FontSize" Value="18" />
  </Style>
</Window.Resources>
```

In the following XAML code the first button — which doesn't have a style defined with the element properties — gets the style that is defined for the `Button` type. With the next button, the `Style` property is set with the `StaticResource` markup extension to `{StaticResource ButtonStyle}`, whereas `ButtonStyle` specifies the key value of the style resource defined earlier, so this button has a red background and a white foreground:

```
<Button Width="200" Content="Uses named style"
        Style="{StaticResource ButtonStyle}" Margin="3" />
```

Rather than set the `Background` of a button to just a single value, you can also do more. You can set the `Background` property to a `LinearGradientBrush` with a gradient color definition:

```
<Style x:Key="FancyButtonStyle">
  <Setter Property="Button.FontSize" Value="22" />
  <Setter Property="Button.Foreground" Value="White" />
  <Setter Property="Button.Background">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0.0" Color="LightCyan" />
          <GradientStop Offset="0.14" Color="Cyan" />
```

```
            <GradientStop Offset="0.7" Color="DarkCyan" />
          </LinearGradientBrush>
        </Setter.Value>
      </Setter>
    </Style>
```

The next button in this example has a fancy style with cyan applied as the linear gradient:

```
<Button Width="200" Content="Fancy button style"
    Style="{StaticResource FancyButtonStyle}" Margin="3" />
```

Styles offer a kind of inheritance. One style can be based on another one. The style `AnotherButtonStyle` is based on the style `FancyButtonStyle`. It uses all the settings defined by the base style (referenced by the `BasedOn` property), except the `Foreground` property—which is set to `LinearGradientBrush`:

```
<Style x:Key="AnotherButtonStyle" BasedOn="{StaticResource FancyButtonStyle}"
    TargetType="Button">
  <Setter Property="Foreground">
    <Setter.Value>
      <LinearGradientBrush>
        <GradientStop Offset="0.2" Color="White" />
        <GradientStop Offset="0.5" Color="LightYellow" />
        <GradientStop Offset="0.9" Color="Orange" />
      </LinearGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

The last button has the `AnotherButtonStyle` applied:

```
<Button Width="200" Content="Style inheritance"
    Style="{StaticResource AnotherButtonStyle}" Margin="3" />
```

The result of all these buttons after styling is shown in Figure 35-18.

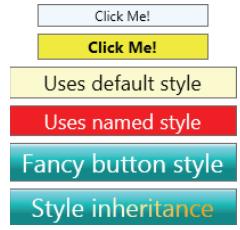


FIGURE 35-18

Resources

As you have seen with the styles sample, usually styles are stored within resources. You can define any freezable element within a resource. For example, the brush created earlier for the background style of the button can itself be defined as a resource, so you can use it everywhere a brush is required.

The following example (code file `StylesAndResources/ResourceDemo.xaml`) defines a `LinearGradientBrush` with the key name `MyGradientBrush` inside the `StackPanel` resources. `button1` assigns the `Background` property by using a `StaticResource` markup extension to the resource `MyGradientBrush`. Figure 35-19 shows the output from this XAML code:

```
<StackPanel x:Name="myContainer">
  <StackPanel.Resources>
    <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0,0"
        EndPoint="0.3,1">
      <GradientStop Offset="0.0" Color="LightCyan" />
      <GradientStop Offset="0.14" Color="Cyan" />
      <GradientStop Offset="0.7" Color="DarkCyan" />
    </LinearGradientBrush>
  </StackPanel.Resources>
  <Button Width="200" Height="50" Foreground="White" Margin="5"
      Background="{StaticResource MyGradientBrush}" Content="Click Me!" />
</StackPanel>
```

FIGURE 35-19

Here, the resources have been defined with the `StackPanel`. In the previous example, the resources were defined with the `Window` element. The base class `FrameworkElement` defines the property `Resources` of type `ResourceDictionary`. That's why resources can be defined with every class that is derived from the `FrameworkElement` — any WPF element.

Resources are searched hierarchically. If you define the resource with the `Window`, it applies to every child element of the `Window`. If the `Window` contains a `Grid`, and the `Grid` contains a `StackPanel`, and you define the resource with the `StackPanel`, then the resource applies to every control within the `StackPanel`. If the `StackPanel` contains a `Button`, and you define the resource just with the `Button`, then this style is valid only for the `Button`.

> **NOTE** *In regard to hierarchies, you need to pay attention if you use the* `TargetType` *without a* `Key` *for styles. If you define a resource with the* `Canvas` *element and set the* `TargetType` *for the style to apply to* `TextBox` *elements, then the style applies to all* `TextBox` *elements within the* `Canvas`*. The style even applies to* `TextBox` *elements that are contained in a* `ListBox` *when the* `ListBox` *is in the* `Canvas`*.*

If you need the same style for more than one window, then you can define the style with the application. In a Visual Studio WPF project, the file `App.xaml` is created for defining global resources of the application. The application styles are valid for every window of the application. Every element can access resources that are defined with the application. If resources are not found with the parent window, then the search for resources continues with the `Application` (code file `StylesAndResources/App.xaml`):

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

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

System Resources

Some system-wide resources for colors and fonts are available for all applications. These resources are defined with the classes `SystemColors`, `SystemFonts`, and `SystemParameters`:

- `SystemColors` — Provides the color settings for borders, controls, the desktop, and windows, such as `ActiveBorderColor`, `ControlBrush`, `DesktopColor`, `WindowColor`, `WindowBrush`, and so on.
- `SystemFonts` — Returns the settings for the fonts of the menu, status bar, and message box. These include `CaptionFont`, `DialogFont`, `MenuFont`, `MessageBoxFont`, `StatusFont`, and so on.
- `SystemParameters` — Provides settings for sizes of menu buttons, cursors, icons, borders, captions, timing information, and keyboard settings, such as `BorderWidth`, `CaptionHeight`, `CaptionWidth`, `MenuButtonWidth`, `MenuPopupAnimation`, `MenuShowDelay`, `SmallIconHeight`, `SmallIconWidth`, and so on.

Accessing Resources from Code

To access resources from code-behind, the base class `FrameworkElement` implements the method `FindResource`, so you can invoke this method with every WPF object. To do this, `button1` doesn't have a background specified, but the `Click` event is assigned to the method `button1_Click` (code file `StylesAndResources/ResourceDemo.xaml`):

```
<Button Name="button1" Width="220" Height="50" Margin="5"
    Click="button1_Click" Content="Apply Resource Programmatically" />
```

With the implementation of `button1_Click`, the `FindResource` method is used on the `Button` that was clicked. Then a search for the resource `MyGradientBrush` happens hierarchically, and the brush is applied to the `Background` property of the control. The resource `MyGradientBrush` was created previously in the resources of the `StackPanel` (code file `StylesAndResources/ResourceDemo.xaml.cs`):

```
public void button1_Click(object sender, RoutedEventArgs e)
{
   Control ctrl = sender as Control;
   ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;
}
```

> **NOTE** *If* `FindResource` *does not find the resource key, then an exception is thrown. If you aren't certain whether the resource is available, then you can instead use the method* `TryFindResource`*, which returns null if the resource is not found.*

Dynamic Resources

With the `StaticResource` markup extension, resources are searched at load time. If the resource changes while the program is running, then you should use the `DynamicResource` markup extension instead.

The next example (code file `StylesAndResources/ResourceDemo.xaml`) is using the same resource defined previously. The earlier example used `StaticResource`. This button uses `DynamicResource` with the `DynamicResource` markup extension. The event handler of this button changes the resource programmatically. The handler method `button2_Click` is assigned to the `Click` event handler:

```
<Button Name="button2" Width="200" Height="50" Foreground="White" Margin="5"
    Background="{DynamicResource MyGradientBrush}" Content="Change Resource"
    Click="button2_Click" />
```

The implementation of `button2_Click` clears the resources of the `StackPanel` and adds a new resource with the same name, `MyGradientBrush`. This new resource is very similar to the resource defined in XAML code; it just defines different colors (code file `StylesAndResources/ResourceDemo.xaml.cs`):

```
private void button2_Click(object sender, RoutedEventArgs e)
{
  myContainer.Resources.Clear();
  var brush = new LinearGradientBrush
  {
    StartPoint = new Point(0, 0),
    EndPoint = new Point(0, 1)
  };

  brush.GradientStops = new GradientStopCollection()
  {
    new GradientStop(Colors.White, 0.0),
    new GradientStop(Colors.Yellow, 0.14),
    new GradientStop(Colors.YellowGreen, 0.7)
  };
  myContainer.Resources.Add("MyGradientBrush", brush);
}
```

When running the application, the resource changes dynamically by clicking the Change Resource button. Using the button with `DynamicResource` gets the dynamically created resource; the button with `StaticResource` looks the same as before.

Resource Dictionaries

If the same resources are used with different applications, it's useful to put the resource in a resource dictionary. Using resource dictionaries, the files can be shared between multiple applications, or the resource dictionary can be put into an assembly and shared by the applications.

To share a resource dictionary in an assembly, create a library. A resource dictionary file, here `Dictionary1.xaml`, can be added to the assembly. The build action for this file must be set to `Resource` so that it is added as a resource to the assembly.

`Dictionary1.xaml` defines two resources: `LinearGradientBrush` with the `CyanGradientBrush` key, and a style for a `Button` that can be referenced with the `PinkButtonStyle` key (code file download `ResourcesLib/Dictionary1.xaml`):

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <LinearGradientBrush x:Key="CyanGradientBrush" StartPoint="0,0"
      EndPoint="0.3,1">
    <GradientStop Offset="0.0" Color="LightCyan" />
    <GradientStop Offset="0.14" Color="Cyan" />
    <GradientStop Offset="0.7" Color="DarkCyan" />
  </LinearGradientBrush>

  <Style x:Key="PinkButtonStyle" TargetType="Button">
    <Setter Property="FontSize" Value="22" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="Background">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
          <GradientStop Offset="0.0" Color="Pink" />
          <GradientStop Offset="0.3" Color="DeepPink" />
          <GradientStop Offset="0.9" Color="DarkOrchid" />
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

With the target project, the library needs to be referenced, and the resource dictionary added to the dictionaries. You can use multiple resource dictionary files that can be added with the `Merged Dictionaries` property of the `ResourceDictionary`. A list of resource dictionaries can be added to the merged dictionaries. With the `Source` property of `ResourceDictionary`, a dictionary can be referenced. For the reference, the pack URI syntax is used. The pack URI can be assigned as *absolute*, whereby the URI begins with `pack://`, or as *relative,* as it is used in this example. With relative syntax, the referenced assembly `ResourceLib`, which includes the dictionary, is first after the `/` followed by `;component`. `Component` means that the dictionary is included as a resource in the assembly. After that, the name of the dictionary file `Dictionary1.xaml` is added. If the dictionary is added into a subfolder, the folder name must be declared as well (code file `StylesAndResources/App.xaml`):

```
<Application x:Class="StylesAndResources.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="/ResourceLib;component/Dictionary1.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Now it is possible to use the resources from the referenced assembly in the same way as local resources (code file `StylesAndResources/ResourceDemo.xaml`):

```
    <Button Width="300" Height="50" Style="{StaticResource PinkButtonStyle}"
        Content="Referenced Resource" />
```

TRIGGERS

With triggers you can change the look and feel of your controls dynamically based on certain events or property value changes. For example, when the user moves the mouse over a button, the button can change its look. Usually, you need to do this with the C# code. With WPF, you can also do this with XAML, as long as only the UI is influenced.

There are several triggers with XAML. Property triggers are activated as soon as a property value changes. Multi-triggers are based on multiple property values. Event triggers fire when an event occurs. Data triggers happen when data that is bound is changed. This section discusses property triggers, multi-triggers, and data triggers. Event triggers are explained later with animations.

Property Triggers

The `Style` class has a `Triggers` property whereby you can assign property triggers. The following example (code file `TriggerDemo/PropertyTriggerWindow.xaml`) includes a `Button` element inside a `Grid` panel. With the `Window` resources, a default style for `Button` elements is defined. This style specifies that the `Background` is set to `LightBlue` and the `FontSize` to `17`. This is the style of the `Button` elements when the application is started. Using triggers, the style of the controls change. The triggers are defined within the `Style.Triggers` element, using the `Trigger` element. One trigger is assigned to the property `IsMouseOver`; the other trigger is assigned to the property `IsPressed`. Both of these properties are defined with the `Button` class to which the style applies. If `IsMouseOver` has a value of `true`, then the trigger fires and sets the `Foreground` property to `Red` and the `FontSize` property to `22`. If the `Button` is pressed, then the property `IsPressed` is `true`, and the second trigger fires and sets the `Foreground` property of the `TextBox` to `Yellow`:

> **NOTE** *If the* `IsPressed` *property is set to* `true`*, the* `IsMouseOver` *property will be* `true` *as well. Pressing the button also requires the mouse to be over the button. Pressing the button triggers it to fire and changes the properties accordingly. Here, the order of triggers is important. If the* `IsPressed` *property trigger is moved before the* `IsMouseOver` *property trigger, the* `IsMouseOver` *property trigger overwrites the values that the first trigger set.*

```
<Window x:Class="TriggerDemo.PropertyTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PropertyTriggerWindow" Height="300" Width="300">
  <Window.Resources>
```

```
    <Style TargetType="Button">
      <Setter Property="Background" Value="LightBlue" />
      <Setter Property="FontSize" Value="17" />
      <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
          <Setter Property="Foreground" Value="Red" />
          <Setter Property="FontSize" Value="22" />
        </Trigger>
        <Trigger Property="IsPressed" Value="True">
          <Setter Property="Foreground" Value="Yellow" />
          <Setter Property="FontSize" Value="22" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <Grid>
    <Button Width="200" Height="30" Content="Click me!" />
  </Grid>
</Window>
```

You don't need to reset the property values to the original values when the reason for the trigger is not valid anymore. For example, you don't need to define a trigger for `IsMouseOver=true` and `IsMouseOver=false`. As soon as the reason for the trigger is no longer valid, the changes made by the trigger action are reset to the original values automatically.

Figure 35-20 shows the trigger sample application in which the foreground and font size of the button are changed from their original values when the button has the focus.

FIGURE 35-20

> **NOTE** *When using property triggers, it is extremely easy to change the look of controls, fonts, colors, opacity, and the like. When the mouse moves over them, the keyboard sets the focus—not a single line of programming code is required.*

The `Trigger` class defines the following properties to specify the trigger action.

TRIGGER PROPERTY	DESCRIPTION
`Property` `Value`	With property triggers, the `Property` and `Value` properties are used to specify when the trigger should fire—for example, `Property="IsMouseOver" Value="True"`.
`Setters`	As soon as the trigger fires, you can use `Setters` to define a collection of `Setter` elements to change values for properties. The `Setter` class defines the properties `Property`, `TargetName`, and `Value` for the object properties to change.
`EnterActions` `ExitActions`	Instead of defining setters, you can define `EnterActions` and `ExitActions`. With both of these properties, you can define a collection of `TriggerAction` elements. `EnterActions` fires when the trigger starts (with a property trigger, when the `Property/Value` combination applies); `ExitActions` fires before it ends (just at the moment when the `Property/Value` combination no longer applies).Trigger actions that you can specify with these actions are derived from the base class `TriggerAction`, such as, `SoundPlayerAction` and `BeginStoryboard`. With `SoundPlayerAction`, you can start the playing of sound. `BeginStoryboard` is used with animation, discussed later in this chapter.

MultiTrigger

A property trigger fires when a value of a property changes. If you need to set a trigger because two or more properties have a specific value, you can use `MultiTrigger`.

`MultiTrigger` has a `Conditions` property whereby valid values of properties can be specified. It also has a `Setters` property that enables you to specify the properties that need to be set. In the following example (code file `TriggerDemo/MultiTriggerWindow.xaml`), a style is defined for `TextBox` elements such that the trigger applies if the `IsEnabled` property is `True` and the `Text` property has the `value Test`. If both apply, the `Foreground` property of the `TextBox` is set to `Red`:

```
<Window x:Class="TriggerDemo.MultiTriggerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MultiTriggerWindow" Height="300" Width="300">
  <Window.Resources>
    <Style TargetType="TextBox">
      <Style.Triggers>
        <MultiTrigger>
          <MultiTrigger.Conditions>
            <Condition Property="IsEnabled" Value="True" />
            <Condition Property="Text" Value="Test" />
          </MultiTrigger.Conditions>
          <MultiTrigger.Setters>
            <Setter Property="Foreground" Value="Red" />
          </MultiTrigger.Setters>
        </MultiTrigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <Grid>
    <TextBox />
  </Grid>
</Window>
```

Data Triggers

Data triggers fire if bound data to a control fulfills specific conditions. In the following example (code file `TriggerDemo/Book.cs`), a `Book` class is used that has different displays depending on the publisher of the book.

The `Book` class defines the properties `Title` and `Publisher` and has an overload of the `ToString` method:

```
public class Book
{
  public string Title { get; set; }
  public string Publisher { get; set; }

  public override string ToString()
  {
    return Title;
  }
}
```

In the XAML code, a style is defined for `ListBoxItem` elements. The style contains `DataTrigger` elements that are bound to the `Publisher` property of the class that is used with the items. If the value of the `Publisher` property is Wrox Press, the `Background` is set to `Red`. With the publishers Dummies and Wiley, the `Background` is set to `Yellow` and `DarkGray`, respectively (code file `TriggerDemo/DataTriggerWindow.xaml`):

```
<Window x:Class="TriggerDemo.DataTriggerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Data Trigger Window" Height="300" Width="300">
  <Window.Resources>
    <Style TargetType="ListBoxItem">
      <Style.Triggers>
        <DataTrigger Binding="{Binding Publisher}" Value="Wrox Press">
          <Setter Property="Background" Value="Red" />
        </DataTrigger>
        <DataTrigger Binding="{Binding Publisher}" Value="Dummies">
          <Setter Property="Background" Value="Yellow" />
        </DataTrigger>
        <DataTrigger Binding="{Binding Publisher}" Value="Wiley">
          <Setter Property="Background" Value="DarkGray" />
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <Grid>
    <ListBox x:Name="list1" />
  </Grid>
</Window>
```

In the code-behind (code file `TriggerDemo/DataTriggerWindow.xaml.cs`), the list with the name `list1` is initialized to contain several `Book` objects:

```
public DataTriggerWindow()
{
  InitializeComponent();
  list1.Items.Add(new Book
  {
    Title = "Professional C# 4.0 and .NET 4",
    Publisher = "Wrox Press"
  });
  list1.Items.Add(new Book
  {
    Title = "C# 2010 for Dummies",
    Publisher = "For Dummies"
  });
  list1.Items.Add(new Book
  {
    Title = "HTML and CSS: Design and Build Websites",
    Publisher = "Wiley"
  });
}
```

Running the application, you can see in Figure 35-21 the `ListBoxItem` elements that are formatted according to the publisher value.

With `DataTrigger`, multiple properties must be set for `MultiDataTrigger` (similar to `Trigger` and `MultiTrigger`).

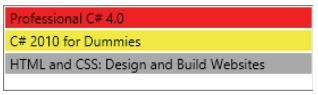


FIGURE 35-21

TEMPLATES

In this chapter, you have already seen that a `Button` control can contain any content. The content can be simple text, but you can also add a `Canvas` element, which can contain shapes; a `Grid`; or a video. In fact, you can do even more than that with a button!

In WPF, the functionality of controls is completely separate from their look and feel. A button has a default look, but you can completely customize that look as you like with templates.

WPF provides several template types that derive from the base class `FrameworkTemplate`.

TEMPLATE TYPE	DESCRIPTION
`ControlTemplate`	Enables you to specify the visual structure of a control and override its look
`ItemsPanelTemplate`	For an `ItemsControl` you can specify the layout of its items by assigning an `ItemsPanelTemplate`. Each `ItemsControl` has a default `ItemsPanelTemplate`. For the `MenuItem`, it is a `WrapPanel`. The `StatusBar` uses a `DockPanel`, and the `ListBox` uses a `VirtualizingStackPanel`.
`DataTemplate`	These are very useful for graphical representations of objects. When styling a `ListBox`, by default the items of the `ListBox` are shown according to the output of the `ToString` method. By applying a `DataTemplate` you can override this behavior and define a custom presentation of the items.
`HierarchicalDataTemplate`	Used for arranging a tree of objects. This control supports `HeaderedItemsControls`, such as `TreeViewItem` and `MenuItem`.

Control Templates

Previously in this chapter you've seen how the properties of a control can be styled. If setting simple properties of the controls doesn't give you the look you want, you can change the `Template` property. With the `Template` property, you can customize the complete look of the control. The next example demonstrates customizing buttons; and later in the following sections ("Data Templates," "Styling a ListBox," "ItemTemplate," and "Control Templates for ListBox Elements"), list boxes are customized step by step, so you can see the intermediate results of the changes.

You customize the `Button` type in a separate resource dictionary file, `Styles.xaml`. Here, a style with the key name `RoundedGelButton` is defined. The style `GelButton` sets the properties `Background`, `Height`, `Foreground`, and `Margin`, and the `Template`. The `Template` is the most interesting aspect with this style. The `Template` specifies a `Grid` with just one row and one column.

Inside this cell, you can find an ellipse with the name `GelBackground`. This ellipse has a linear gradient brush for the stroke. The stroke that surrounds the rectangle is very thin because the `StrokeThickness` is set to 0.5.

The second ellipse, `GelShine`, is a small ellipse whose size is defined by the `Margin` property and so is visible within the first ellipse. The stroke is transparent, so there is no line surrounding the ellipse. This ellipse uses a linear gradient fill brush, which transitions from a light, partly transparent color to full transparency. This gives the ellipse a shimmering effect (code file `TemplateDemo/Styles.xaml`):

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Style x:Key="RoundedGelButton" TargetType="Button">
    <Setter Property="Width" Value="100" />
    <Setter Property="Height" Value="100" />
    <Setter Property="Foreground" Value="White" />
```

```
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}">
          <Grid>
            <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
              <Ellipse.Stroke>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="#ff7e7e7e" />
                  <GradientStop Offset="1" Color="Black" />
                </LinearGradientBrush>
              </Ellipse.Stroke>
            </Ellipse>
            <Ellipse Margin="15,5,15,50">
              <Ellipse.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="#aaffffff" />
                  <GradientStop Offset="1" Color="Transparent" />
                </LinearGradientBrush>
              </Ellipse.Fill>
            </Ellipse>
          </Grid>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

From the `app.xaml` file, the resource dictionary is referenced as shown here (code file `TemplateDemo/App.xaml`):

```
<Application x:Class="TemplateDemo.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 Source="Styles.xaml" />
  </Application.Resources
</Application>
```

Now a `Button` control can be associated with the style. The new look of the button is shown in Figure 35-22 and uses code file `TemplateDemo/StyledButtonWindow.xaml`:

FIGURE 35-22

```
    <Button Style="{StaticResource RoundedGelButton}" Content="Click Me!" />
```

The button now has a completely different look. However, the content that is defined with the button itself is missing. The template created previously must be extended to get the content of the `Button` into the new look. What needs to be added is a `ContentPresenter`. The `ContentPresenter` is the placeholder for the control's content, and it defines the place where the content should be positioned. In the code that follows (code file `TemplateDemo/StyledButtonWindow.xaml`), the content is placed in the first row of the `Grid`, as are the `Ellipse` elements. The `Content` property of the `ContentPresenter` defines what the content should be. The content is set to a `TemplateBinding` markup expression. `TemplateBinding` binds the template parent, which is the `Button` element in this case. `{TemplateBinding Content}` specifies that the value of the `Content` property of the `Button` control should be placed inside the placeholder as content. Figure 35-23 shows the result with the content shown in the here:

```
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}">
```

```
    <Grid>
      <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
        <Ellipse.Stroke>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="#ff7e7e7e" />
            <GradientStop Offset="1" Color="Black" />
          </LinearGradientBrush>
        </Ellipse.Stroke>
      </Ellipse>
      <Ellipse Margin="15,5,15,50">
        <Ellipse.Fill>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="#aaffffff" />
            <GradientStop Offset="1" Color="Transparent" />
          </LinearGradientBrush>
        </Ellipse.Fill>
      </Ellipse>
      <ContentPresenter Name="GelButtonContent"
                        VerticalAlignment="Center"
                        HorizontalAlignment="Center"
                        Content="{TemplateBinding Content}" />
    </Grid>
  </ControlTemplate>
</Setter.Value>
```

FIGURE 35-23

Such a styled button now looks very fancy on the screen, but there's still a problem: There is no action if the mouse is clicked or the mouse moves over the button. This isn't the typical experience a user has with a button. This can be solved, however. With a template-styled button, you must have triggers that enable the button to look differently in response to mouse moves and mouse clicks.

Using property triggers (discussed previously), this can be done easily. The triggers just need to be added to the `Triggers` collection of the `ControlTemplate` as shown next. Here, two triggers are defined. One property trigger is active when the `IsMouseOver` property of the button is `true`. Then the `Fill` property of the `Ellipse` with the name `GelBackground` is changed to a `RadialGradientBrush` with values from `Lime` to `DarkGreen`. With the `IsPressed` property, other colors are specified for the `RadialGradientBrush`:

```
<ControlTemplate.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
    <Setter Property="Ellipse.Fill" TargetName="GelBackground">
      <Setter.Value>
        <RadialGradientBrush>
          <GradientStop Offset="0" Color="Lime" />
          <GradientStop Offset="1" Color="DarkGreen" />
        </RadialGradientBrush>
      </Setter.Value>
    </Setter>
  </Trigger>
  <Trigger Property="IsPressed" Value="True">
    <Setter Property="Ellipse.Fill" TargetName="GelBackground">
      <Setter.Value>
        <RadialGradientBrush>
          <GradientStop Offset="0" Color="#ffcc34" />
          <GradientStop Offset="1" Color="#cc9900" />
        </RadialGradientBrush>
      </Setter.Value>
    </Setter>
  </Trigger>

</ControlTemplate.Triggers>
```

Now run the application and you should see visual feedback from the button as soon as the mouse hovers over it or the mouse is clicked.

Data Templates

The content of `ContentControl` elements can be any content—not only WPF elements but also .NET objects. For example, an object of the `Country` type can be assigned to the content of a `Button` class. In the following example (code file `TemplateDemo/Country.cs`), the `Country` class is created to represent the name and flag with a path to an image. This class defines the `Name` and `ImagePath` properties, and it has an overridden `ToString` method for a default string representation:

```
public class Country
{
  public string Name { get; set; }
  public string ImagePath { get; set; }

  public override string ToString()
  {
    return Name;
  }
}
```

How does this content look within a `Button` or any other `ContentControl`? By default, the `ToString` method is invoked, and the string representation of the object is shown. For a custom look you can also create a `DataTemplate` for the `Country` type.

Here, within the resources of the `Window`, a `DataTemplate` is created. This `DataTemplate` doesn't have a key assigned and thus is a default for the `Country`. `src` type—it is also the alias of the XML namespace referencing the .NET assembly and .NET namespace. Within the `DataTemplate` the main elements are a `TextBox` with the `Text` property bound to the `Name` property of the `Country`, and an `Image` with the `Source` property bound to the `ImagePath` property of the `Country`. The `Grid`, `Border`, and `Rectangle` elements define the layout and visual appearance (code file `TemplateDemo/StyledButtonWindow.xaml`):

```
<Window.Resources>
  <DataTemplate DataType="{x:Type src:Country}">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="60" />
    </Grid.RowDefinitions>
    <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
        Text="{Binding Name}" FontWeight="Bold" Grid.Column="0" />
      <Border Margin="4,0" Grid.Column="1" BorderThickness="2"
          CornerRadius="4">
        <Border.BorderBrush>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="#aaa" />
            <GradientStop Offset="1" Color="#222" />
          </LinearGradientBrush>
        </Border.BorderBrush>
        <Grid>
          <Rectangle>
            <Rectangle.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#444" />
                <GradientStop Offset="1" Color="#fff" />
```

```
            </LinearGradientBrush>
          </Rectangle.Fill>
        </Rectangle>
        <Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
      </Grid>
    </Border>
  </Grid>
</DataTemplate>
</Window.Resources>
```

With the XAML code, a simple `Button` element with the name `button1` is defined:

```
<Button Grid.Row="1" x:Name="button1" Margin="10" />
```

Within the code-behind (code file `TemplateDemo/StyledButtonWindow.xaml.cs`), a new `Country` object is instantiated that is assigned to the `Content` property of `button1`:

```
public StyledButtonWindow()
{
  InitializeComponent();
  button1.Content = new Country
  {
    Name = "Austria",
    ImagePath = "images/Austria.bmp"
  };
}
```

After running the application, you can see that the `DataTemplate` is applied to the `Button` because the `Country` data type has a default template, shown in Figure 35-24.

FIGURE 35-24

Of course, you can also create a control template and use a data template from within.

Styling a ListBox

Changing a style of a button or a label is a simple task, such as changing the style of an element that contains a list of elements. For example, how about changing a `ListBox`? Again, a list box has behavior and a look. It can display a list of elements, and you can select one or more elements from the list. For the behavior, the `ListBox` class defines methods, properties, and events. The look of the `ListBox` is separate from its behavior. It has a default look, but you can change this look by creating a template.

With a `ListBox`, the `ControlTemplate` defines how the complete control looks, an `ItemTemplate` defines how an item looks, and a `DataTemplate` defines the type that might be within an item.To fill a `ListBox` with some items, the static class `Countries` returns a list of a few countries that will be displayed (code file `TemplateDemo/Countries.cs`):

```
public class Countries
{
  public static IEnumerable<Country> GetCountries()
  {
    return new List<Country>
    {
      new Country { Name = "Austria", ImagePath = "Images/Austria.bmp" },
      new Country { Name = "Germany", ImagePath = "Images/Germany.bmp" },
      new Country { Name = "Norway", ImagePath = "Images/Norway.bmp" },
      new Country { Name = "USA", ImagePath = "Images/USA.bmp" }
    };
  }
}
```

Inside the code-behind file (code file TemplateDemo/StyledListBoxWindow1.xaml.cs) in the constructor of the StyledListBoxWindow1 class, the DataContext property of the StyledListBoxWindow1 instance is set to the list of countries returned from the method Countries.GetCountries. (The DataContext property is a data binding feature discussed in the next chapter.)

```
public partial class StyledListBoxWindow1 : Window
{
  public StyledListBoxWindow1()
  {
    InitializeComponent();
    this.DataContext = Countries.GetCountries();
  }
}
```

Within the XAML code (code file TemplateDemo/StyledListBoxWindow.xaml), the ListBox named countryList1 is defined. countryList1 doesn't have a different style. It uses the default look from the ListBox element. The property ItemsSource is set to the Binding markup extension, which is used by data binding. From the code-behind, you have seen that the binding is done to an array of Country objects. Figure 35-25 shows the default look of the ListBox. By default, only the names of the countries returned by the ToString method are displayed in a simple list:

```
<Window x:Class="TemplateDemo.StyledListBoxWindow1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:TemplateDemo"
    Title="StyledListBoxWindow1" Height="300" Width="300">
  <Grid>
    <ListBox ItemsSource="{Binding}" Margin="10" />
  </Grid>
</Window>
```

FIGURE 35-25

ItemTemplate

The Country objects contain both the name and the flag. Of course, you can display both values in the list box. To do this, you need to define a template.

The ListBox element contains ListBoxItem elements. You can define the content for an item with the ItemTemplate. The style ListBoxStyle1 defines an ItemTemplate with a value of a DataTemplate. A DataTemplate is used to bind data to elements. You can use the Binding markup extension with DataTemplate elements.

The DataTemplate contains a grid with three columns. The first column contains the string Country: The second column contains the name of the country. The third column contains the flag for the country. Because the country names are of different lengths but the view should be the same size for every country name, the SharedSizeGroup property is set with the second column definition. This shared size information for the column is used only because the property Grid.IsSharedSizeScope is also set.

After the column and row definitions, you can see two TextBlock elements. The first TextBlock element contains the text Country:. The second TextBlock element binds to the Name property defined in the Country class.

The content for the third column is a Border element containing a Grid. The Grid contains a Rectangle with a linear gradient brush and an Image element that is bound to the ImagePath property of the Country class. Figure 35-26 shows the countries in a ListBox with completely different output than before (code file TemplateDemo/Styles.xaml):

FIGURE 35-26

```
<Style x:Key="ListBoxStyle1" TargetType="{x:Type ListBox}" >
  <Setter Property="ItemTemplate">
    <Setter.Value>
      <DataTemplate>
        <Grid>
         <Grid.ColumnDefinitions>
           <ColumnDefinition Width="Auto" />
           <ColumnDefinition Width="*" SharedSizeGroup="MiddleColumn" />
           <ColumnDefinition Width="Auto" />
         </Grid.ColumnDefinitions>
         <Grid.RowDefinitions>
           <RowDefinition Height="60" />
         </Grid.RowDefinitions>
         <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                    FontStyle="Italic" Grid.Column="0" Text="Country:" />
         <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                    Text="{Binding Name}" FontWeight="Bold" Grid.Column="1" />
         <Border Margin="4,0" Grid.Column="2" BorderThickness="2"
                 CornerRadius="4">
           <Border.BorderBrush>
             <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
               <GradientStop Offset="0" Color="#aaa" />
               <GradientStop Offset="1" Color="#222" />
             </LinearGradientBrush>
           </Border.BorderBrush>
           <Grid>
             <Rectangle>
               <Rectangle.Fill>
                 <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                   <GradientStop Offset="0" Color="#444" />
                   <GradientStop Offset="1" Color="#fff" />
                 </LinearGradientBrush>
               </Rectangle.Fill>
             </Rectangle>
             <Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
           </Grid>
         </Border>
        </Grid>
      </DataTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="Grid.IsSharedSizeScope" Value="True" />
</Style>
```

Control Templates for ListBox Elements

It is not necessary for a `ListBox` to have items that follow vertically, one after the other. You can give the user a different view with the same functionality. The next style, `ListBoxStyle2`, defines a template in which the items are shown horizontally with a scrollbar.

In the previous example, only an `ItemTemplate` was created to define how the items should look in the default `ListBox`. In the following code (code file `TemplateDemo/Styles.xaml`), a template is created to define a different `ListBox`. The template contains a `ControlTemplate` element to define the elements of the `ListBox`. The element is now a `ScrollViewer`—a view with a scrollbar—that contains a `StackPanel`. Because the items should now be listed horizontally, the `Orientation` of the `StackPanel` is set to `Horizontal`. The stack panel will contain the items defined with the `ItemsTemplate`. As a result, the `IsItemsHost` of the `StackPanel` element is set to `true`. `IsItemsHost` is a property that is available with every `Panel` element that can contain a list of items.

The `ItemTemplate` that defines the look for the items in the stack panel is taken from the style `ListBoxStyle1` where `ListBoxStyle2` is based.

Figure 35-27 shows the `ListBox` styled with `ListBoxStyle2`, whereby the scrollbar appears automatically when the view is too small to display all items in the list:

```
<Style x:Key="ListBoxStyle2" TargetType="{x:Type ListBox}"
    BasedOn="{StaticResource ListBoxStyle1}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <ScrollViewer HorizontalScrollBarVisibility="Auto">
          <StackPanel Name="StackPanel1" IsItemsHost="True"
              Orientation="Horizontal" />
        </ScrollViewer>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="VerticalAlignment" Value="Center" />
</Style>
```

FIGURE 35-27

Certainly you see the advantages of separating the look of the controls from their behavior. You may already have many ideas about how you can display your items in a list that best fits the requirements of your application. Perhaps you just want to display as many items as will fit in the window, position them horizontally, and then continue to the next line vertically. That's where a `WrapPanel` comes in; and, of course, you can have a `WrapPanel` inside a template for a `ListBox`, as shown in `ListBoxStyle3`. Figure 35-28 shows the result of using the `WrapPanel`:

```
<Style x:Key="ListBoxStyle3" TargetType="{x:Type ListBox}">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type ListBox}">
        <ScrollViewer VerticalScrollBarVisibility="Auto"
            HorizontalScrollBarVisibility="Disabled">
          <WrapPanel IsItemsHost="True" />
        </ScrollViewer>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  <Setter Property="ItemTemplate">
    <Setter.Value>
      <DataTemplate>
        <Grid>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="140" />
          </Grid.ColumnDefinitions>
          <Grid.RowDefinitions>
            <RowDefinition Height="60" />
            <RowDefinition Height="30" />
          </Grid.RowDefinitions>
          <Image Grid.Row="0" Width="48" Margin="2,2,2,1"
              Source="{Binding ImagePath}" />
          <TextBlock Grid.Row="1" FontSize="14"
              HorizontalAlignment="Center" Margin="5" Text="{Binding Name}" />
```

```
        </Grid>
      </DataTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

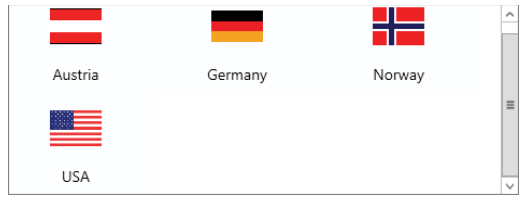


FIGURE 35-28

ANIMATIONS

Using animations you can make a smooth transition between images by using moving elements, color changes, transforms, and so on. WPF makes it easy to create animations. You can animate the value of any dependency property. Different animation classes exist to animate the values of different properties, depending on their type.

The major elements of animations are as follows:

- **Timeline**—Defines how a value changes over time. Different kinds of timelines are available for changing different types of values. The base class for all timelines is `Timeline`. To animate a `double`, the class `DoubleAnimation` can be used. `Int32Animation` is the animation class for `int` values. `PointAnimation` is used to animate points, and `ColorAnimation` is used to animate colors.
- **Storyboard**—Used to combine animations. The `Storyboard` class itself is derived from the base class `TimelineGroup`, which derives from `Timeline`. With `DoubleAnimation` you can animate a double value; with `Storyboard` you combine all the animations that belong together.
- **Triggers**—Used to start and stop animations. You've seen property triggers previously, which fire when a property value changes. You can also create an event trigger. An event trigger fires when an event occurs.

> **NOTE** *The namespace for animation classes is* `System.Windows.Media.Animation`.

Timeline

A `Timeline` defines how a value changes over time. The following example animates the size of an ellipse. In the code that follows (code file `AnimationDemo/EllipseWindow.xaml`), a `DoubleAnimation` timeline changes to a double value. The `Triggers` property of the `Ellipse` class is set to an `EventTrigger`. The event trigger is fired when the ellipse is loaded as defined with the `RoutedEvent` property of the `EventTrigger`. `BeginStoryboard` is a trigger action that begins the storyboard. With the storyboard, a `DoubleAnimation` element is used to animate the `Width` property of the `Ellipse` class. The animation changes the width of the ellipse from 100 to 300 within three seconds, and reverses the animation after three seconds. The animation `ColorAnimation` animates the color from the `ellipseBrush` which is used to fill the ellipse:

```
<Ellipse Height="50" Width="100">
  <Ellipse.Fill>
    <SolidColorBrush x:Name="ellipseBrush" Color="Yellow" />
  </Ellipse.Fill>
  <Ellipse.Triggers>
    <EventTrigger RoutedEvent="Ellipse.Loaded" >
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard Duration="00:00:06" RepeatBehavior="Forever">
            <DoubleAnimation Storyboard.TargetProperty="(Ellipse.Width)"
```

```
                Duration="0:0:3" AutoReverse="True" FillBehavior="Stop"
                RepeatBehavior="Forever" AccelerationRatio="0.9"
                DecelerationRatio="0.1" From="100" To="300" />
            <ColorAnimation Storyboard.TargetName="ellipseBrush"
                Storyboard.TargetProperty="(SolidColorBrush.Color)"
                Duration="0:0:3" AutoReverse="True"
                FillBehavior="Stop" RepeatBehavior="Forever"
                From="Yellow" To="Red" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Ellipse.Triggers>
</Ellipse>
```


FIGURE 35-29

Figures 35-29 and 35-30 show two states from the animated ellipse.

FIGURE 35-30

Animations are far more than typical window-dressing animation that appears onscreen constantly and immediately. You can add animation to business applications that make the user interface more responsive.

The following example (code file `AnimationDemo/ButtonAnimationWindow.xaml`) demonstrates a decent animation and shows how the animation can be defined in a style. Within the `Window` resources you can see the style `AnimatedButtonStyle` for buttons. In the template, a rectangle-named outline is defined. This template has a thin stroke with the thickness set to 0.4.

The template defines a property trigger for the `IsMouseOver` property. The `EnterActions` property of this trigger applies as soon as the mouse is moved over the button. The action to start is `BeginStoryboard`, which is a trigger action that can contain and thus start `Storyboard` elements. The `Storyboard` element defines a `DoubleAnimation` to animate a double value. The property value that is changed in this animation is the `Rectangle.StrokeThickness` of the `Rectangle` element with the name `outline`. The value is changed in a smooth way by 1.2, as the `By` property specifies, for a time length of 0.3 seconds as specified by the `Duration` property. At the end of the animation, the stroke thickness is reset to its original value because `AutoReverse="True"`. To summarize: As soon as the mouse moves over the button, the thickness of the outline is incremented by 1.2 for 0.3 seconds. Figure 35-31 shows the button without animation, and Figure 35-32 shows the button 0.3 seconds after the mouse moved over it. (Unfortunately, it's not possible to show the intermediate appearance of the smooth animation in a print medium.)

FIGURE 35-31

FIGURE 35-32

```
<Window x:Class="AnimationDemo.ButtonAnimationWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ButtonAnimationWindow" Height="300" Width="300">
  <Window.Resources>
    <Style x:Key="AnimatedButtonStyle" TargetType="{x:Type Button}">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="{x:Type Button}">
            <Grid>
              <Rectangle Name="outline" RadiusX="9" RadiusY="9"
                  Stroke="Black" Fill="{TemplateBinding Background}"
                  StrokeThickness="1.6">
              </Rectangle>
              <ContentPresenter VerticalAlignment="Center"
                  HorizontalAlignment="Center" />
            </Grid>
            <ControlTemplate.Triggers>
```

```
            <Trigger Property="IsMouseOver" Value="True">
              <Trigger.EnterActions>
                <BeginStoryboard>
                  <Storyboard>
                    <DoubleAnimation Duration="0:0:0.3" AutoReverse="True"
                        Storyboard.TargetProperty="(Rectangle.StrokeThickness)"
                        Storyboard.TargetName="outline" By="1.2" />
                  </Storyboard>
                </BeginStoryboard>
              </Trigger.EnterActions>
            </Trigger>
          </ControlTemplate.Triggers>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
 </Window.Resources>
 <Grid>
   <Button Style="{StaticResource AnimatedButtonStyle}" Width="200"
       Height="100" Content="Click Me!" />
 </Grid>
</Window>
```

The following table describes what you can do with a timeline.

TIMELINE PROPERTIES	DESCRIPTION
`AutoReverse`	Use this property to specify whether the value that is animated should return to its original value after the animation.
`SpeedRatio`	Use this property to transform the speed at which an animation moves. You can define the relation in regard to the parent. The default value is 1; setting the ratio to a smaller value makes the animation move slower; setting the value greater than 1 makes it move faster.
`BeginTime`	Use this to specify the time span from the start of the trigger event until the moment the animation starts. You can specify days, hours, minutes, seconds, and fractions of seconds. This might not be real time, depending on the speed ratio. For example, if the speed ratio is set to 2, and the beginning time is set to six seconds, the animation will start after three seconds.
`AccelerationRatio` `DecelerationRatio`	An animation's values need not be changed in a linear way. You can specify an `AccelerationRatio` and `DecelerationRatio` to define the impact of acceleration and deceleration. The sum of both values must not be greater than 1.
`Duration`	Use this property to specify the length of time for one iteration of the animation.
`RepeatBehavior`	Assigning a `RepeatBehavior` struct to the `RepeatBehavior` property enables you to define how many times or for how long the animation should be repeated.
`FillBehavior`	This property is important if the parent timeline has a different duration. For example, if the parent timeline is shorter than the duration of the actual animation, setting `FillBehavior` to `Stop` means that the actual animation stops. If the parent timeline is longer than the duration of the actual animation, `HoldEnd` keeps the actual animation active before resetting it to its original value (if `AutoReverse` is set).

Depending on the type of the `Timeline` class, more properties may be available. For example, with `DoubleAnimation` you can specify `From` and `To` properties for the start and end of the animation. An alternative is to specify the `By` property, whereby the animation starts with the current value of the `Bound` property and is incremented by the value specified by `By`.

Nonlinear Animations

One way to define nonlinear animations is by setting the speed of `AccelerationRatio` and `DecelerationRatio` animation at the beginning and at the end. .NET 4.5 has more flexible possibilities than that.

Several animation classes have an `EasingFunction` property. This property accepts an object that implements the interface `IEasingFunction`. With this interface, an easing function object can define how the value should be animated over time. Several easing functions are available to create a nonlinear animation. Examples include `ExponentialEase`, which uses an exponential formula for animations; `QuadraticEase`, `CubicEase`, `QuarticEase`, and `QuinticEase`, with powers of 2, 3, 4, or 5; and `PowerEase`, with a power level that is configurable. Of special interest are `SineEase`, which uses a sinusoid curve, `BounceEase`, which creates a bouncing effect, and `ElasticEase`, which resembles animation values of a spring oscillating back and forth.

Such an ease can be specified in XAML by adding the ease to the `EasingFunction` property of the animation as shown in the following code (code file `AnimationDemo/EllipseWindow.xaml`). Adding different ease functions results in very interesting animation effects:

```
<DoubleAnimation Storyboard.TargetProperty="(Ellipse.Width)"
                 Duration="0:0:3" AutoReverse="True"
                 FillBehavior=" RepeatBehavior="Forever"
                 From="100" To="300">
  <DoubleAnimation.EasingFunction>
    <BounceEase EasingMode="EaseInOut" />
  </DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

Event Triggers

Instead of having a property trigger, you can define an event trigger to start the animation. The property trigger fires when a property changes its value; the event trigger fires when an event occurs. Examples of such events are the `Load` event from a control, the `Click` event from a `Button`, and the `MouseMove` event.

The next example creates an animation for the face that was created earlier with shapes. It is now animated so that the eye moves as soon as a `Click` event from a button is fired.

Inside the `Window` element, a `DockPanel` element is defined to arrange the face and buttons to control the animation. A `StackPanel` that contains three buttons is docked at the top. The `Canvas` element that contains the face gets the remaining part of the `DockPanel`.

The first button is used to start the animation of the eye; the second button stops the animation. A third button is used to start another animation to resize the face.

The animation is defined within the `DockPanel.Triggers` section. Instead of a property trigger, an event trigger is used. The first event trigger is fired as soon as the `Click` event occurs with the `buttonBeginMove-Eyes` button defined by the `RoutedEvent` and `SourceName` properties. The trigger action is defined by the `BeginStoryboard` element that starts the containing `Storyboard`. `BeginStoryboard` has a name defined because a name is needed to control the storyboard with pause, continue, and stop actions. The `Storyboard` element contains four animations. The first two animate the left eye; the last two animate the right eye. The first and third animation change the `Canvas.Left` position for the eyes, and the second and fourth animation change `Canvas.Top`. The animations in the *x* and *y* axes have different time values that make the eye movement very interesting using the defined repeated behavior.

The second event trigger is fired as soon as the `Click` event of the `buttonStopMoveEyes` button occurs. Here, the storyboard is stopped with the `StopStoryboard` element, which references the started storyboard `beginMoveEye`.

The third event trigger is fired by clicking the `buttonResize` button. With this animation, the transformation of the `Canvas` element is changed. Because this animation doesn't run endlessly, there's no stop. This storyboard also makes use of the `EaseFunction` explained previously (code file `AnimationDemo/EventTriggerWindow.xaml`):

```
<Window x:Class="AnimationDemo.EventTriggerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="EventTriggerWindow" Height="300" Width="300">
  <DockPanel>
    <DockPanel.Triggers>
      <EventTrigger RoutedEvent="Button.Click" SourceName="buttonBeginMoveEyes">
        <BeginStoryboard x:Name="beginMoveEyes">
          <Storyboard>
            <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                AutoReverse="True" By="6" Duration="0:0:1"
                Storyboard.TargetName="eyeLeft"
                Storyboard.TargetProperty="(Canvas.Left)" />
            <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                By="6" Duration="0:0:5"
                Storyboard.TargetName="eyeLeft"
                Storyboard.TargetProperty="(Canvas.Top)" />
            <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                AutoReverse="True" By="-6" Duration="0:0:3"
                Storyboard.TargetName="eyeRight"
                Storyboard.TargetProperty="(Canvas.Left)" />
            <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                By="6" Duration="0:0:6"
                Storyboard.TargetName="eyeRight"
                Storyboard.TargetProperty="(Canvas.Top)" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
      <EventTrigger RoutedEvent="Button.Click" SourceName="buttonStopMoveEyes">
        <StopStoryboard BeginStoryboardName="beginMoveEyes" />
      </EventTrigger>
      <EventTrigger RoutedEvent="Button.Click" SourceName="buttonResize">
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
                Storyboard.TargetName="scale1"
                Storyboard.TargetProperty="(ScaleTransform.ScaleX)"
                From="0.1" To="3" Duration="0:0:5">
              <DoubleAnimation.EasingFunction>
                <ElasticEase />
              </DoubleAnimation.EasingFunction>
            </DoubleAnimation>
            <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
                Storyboard.TargetName="scale1"
                Storyboard.TargetProperty="(ScaleTransform.ScaleY)"
                From="0.1" To="3" Duration="0:0:5">
              <DoubleAnimation.EasingFunction>
                <BounceEase />
              </DoubleAnimation.EasingFunction>
            </DoubleAnimation>
          </Storyboard>
        </BeginStoryboard>
```

```
      </EventTrigger>
    </DockPanel.Triggers>
    <StackPanel Orientation="Vertical" DockPanel.Dock="Top">
      <Button x:Name="buttonBeginMoveEyes" Content="Start Move Eyes" Margin="5" />
      <Button x:Name="buttonStopMoveEyes" Content="Stop Move Eyes" Margin="5" />
      <Button x:Name="buttonResize" Content="Resize" Margin="5" />
    </StackPanel>
    <Canvas>
      <Canvas.LayoutTransform>
        <ScaleTransform x:Name="scale1" ScaleX="1" ScaleY="1" />
      </Canvas.LayoutTransform>
      <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
          Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
      <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
        <Ellipse.Fill>
          <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
            <GradientStop Offset="0.1" Color="DarkGreen" />
            <GradientStop Offset="0.7" Color="Transparent" />
          </LinearGradientBrush>
        </Ellipse.Fill>
      </Ellipse>
      <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20"
          Stroke="Blue" StrokeThickness="3" Fill="White" />
      <Ellipse x:Name="eyeLeft" Canvas.Left="40" Canvas.Top="43" Width="6"
          Height="5" Fill="Black" />
      <Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20"
          Stroke="Blue" StrokeThickness="3" Fill="White" />
      <Ellipse x:Name="eyeRight" Canvas.Left="75" Canvas.Top="43" Width="6"
          Height="5" Fill="Black" />
      <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
          Data="M 40,74 Q 57,95 80,74 " />
    </Canvas>
  </DockPanel>
</Window>
```

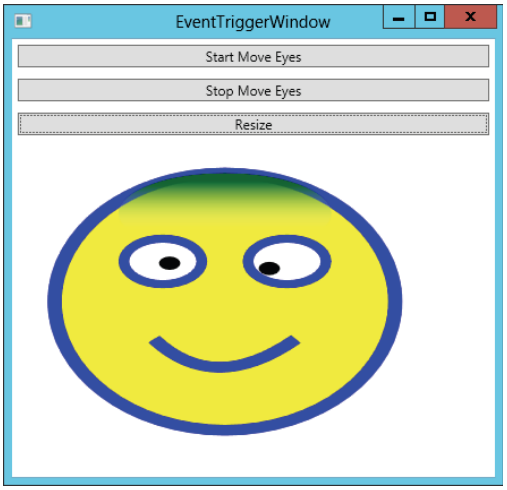


FIGURE 35-33

Figure 35-33 shows the output after running the application.

Rather than start and stop the animation directly from event triggers in XAML, you can easily control the animation from code-behind. You just need to assign a name to the `Storyboard` and invoke the `Begin`, `Stop`, `Pause`, and `Resume` methods.

Keyframe Animations

With acceleration and deceleration ratio as well as the ease functions, you've seen how animations can be built in a nonlinear fashion. If you need to specify several values for an animation, you can use *keyframe animations*. Like normal animations, keyframe animations are various animation types that exist to animate properties of different types.

`DoubleAnimationUsingKeyFrames` is the keyframe animation for double types. Other keyframe animation types are `Int32AnimationUsingKeyFrames`, `PointAnimationUsingKeyFrames`, `ColorAnimationUsingKeyFrames`, `SizeAnimationUsingKeyFrames`, and `ObjectAnimationUsingKeyFrames`.

The following example XAML code (code file AnimationDemo/KeyFrameWindow.xaml) animates the position of an ellipse by animating the X and Y values of a TranslateTransform element. The animation starts when the ellipse is loaded by defining an EventTrigger to the RoutedEvent Ellipse.Loaded. The event trigger starts a Storyboard with the BeginStoryboard element. The Storyboard contains two keyframe animations of type DoubleAnimationUsingKeyFrame. A keyframe animation consists of frame elements. The first keyframe animation uses a LinearKeyFrame, a DiscreteDoubleKeyFrame, and a SplineDoubleKeyFrame; the second animation is an EasingDoubleKeyFrame. The LinearDoubleKeyFrame makes a linear change of the value. The KeyTime property defines when in the animation the value of the Value property should be reached.

Here, the LinearDoubleKeyFrame has three seconds to move the property X to the value 30. DiscreteDoubleKeyFrame makes an immediate change to the new value after four seconds. SplineDoubleKeyFrame uses a Bézier curve whereby two control points are specified by the KeySpline property. EasingDoubleKeyFrame is a frame class that supports setting an easing function such as BounceEase to control the animation value:

```
<Canvas>
  <Ellipse Fill="Red" Canvas.Left="20" Canvas.Top="20" Width="25" Height="25">
    <Ellipse.RenderTransform>
      <TranslateTransform X="50" Y="50" x:Name="ellipseMove" />
    </Ellipse.RenderTransform>
      <Ellipse.Triggers>
        <EventTrigger RoutedEvent="Ellipse.Loaded">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="X"
                  Storyboard.TargetName="ellipseMove">
                <LinearDoubleKeyFrame KeyTime="0:0:2" Value="30" />
                <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="80" />
                <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                    KeyTime="0:0:10" Value="300" />
                <LinearDoubleKeyFrame KeyTime="0:0:20" Value="150" />
              </DoubleAnimationUsingKeyFrames>
              <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Y"
                  Storyboard.TargetName="ellipseMove">
                <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                    KeyTime="0:0:2" Value="50" />
                <EasingDoubleKeyFrame KeyTime="0:0:20" Value="300">
                  <EasingDoubleKeyFrame.EasingFunction>
                    <BounceEase />
                  </EasingDoubleKeyFrame.EasingFunction>
                </EasingDoubleKeyFrame>
              </DoubleAnimationUsingKeyFrames>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Ellipse.Triggers>
    </Ellipse>
  </Canvas>
```

VISUAL STATE MANAGER

Beginning with .NET 4, Visual State Manager offers an alternative way to control animations. Controls can have specific states. The *state* defines a look that is applied to controls when the state is reached. A *state transition* defines what happens when one state changes to another one.

With a data grid you can use Read, Selected, and Edit states to define different looks for a row, depending on user selection. MouseOver and IsPressed are states that replace the triggers, which have been discussed earlier.

The following example (code file VisualStateDemo/Style.xaml) creates a custom template for the Button type whereby visual states are used instead of the triggers used earlier. The XAML code in this snippet defines a template for the Button type that consists of Ellipse elements with gradient brushes. As the code stands here, nothing happens when the user moves the mouse over a button or clicks it. This is going to be changed using visual states.

```
<Style TargetType="Button">
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value="100" />
  <Setter Property="Foreground" Value="White" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Button}">
        <Grid>
          <Ellipse Name="GelBackground" StrokeThickness="0.5">
            <Ellipse.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="Black" />
                <GradientStop Offset="1" Color="Black" />
              </LinearGradientBrush>
            </Ellipse.Fill>
            <Ellipse.Stroke>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#ff7e7e7e" />
                <GradientStop Offset="1" Color="Black" />
              </LinearGradientBrush>
            </Ellipse.Stroke>
          </Ellipse>
          <Ellipse Margin="15,5,15,50">
            <Ellipse.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0" Color="#aaffffff" />
                <GradientStop Offset="1" Color="Transparent" />
              </LinearGradientBrush>
            </Ellipse.Fill>
          </Ellipse>
          <ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
              HorizontalAlignment="Center"
              Content="{TemplateBinding Content}" />
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Visual States

The Button type defines several state groups and states. The state group CommonStates defines the states Normal, MouseOver, and Pressed. The state group FocusedStates defines Focused and Unfocused. As shown in the following example (code file VisualStateDemo/Style.xaml), the implementation of the Button class changes the states using the VisualStateManager—you just have to define a look for these states.

For defining a different appearance for the controls using visual states, the attached property VisualStateManager.VisualStateGroups is defined within the template. The first group defined is CommonStates. Within this group, looks for the MouseOver and Pressed states are defined. Within the MouseOver state, a key frame color animation changes the fill color of the ellipse to a gradient color from lime to dark green. The Pressed state has a similar implementation: the fill color changes to a new range from ffcc34 to cc9900:

```
<ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Content="{TemplateBinding Content}" />
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="CommonStates">
    <VisualState Name="Normal" />
    <VisualState Name="MouseOver">
      <Storyboard>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[0].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="Lime"/>
        </ColorAnimationUsingKeyFrames>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[1].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="DarkGreen"/>
        </ColorAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
    <VisualState Name="Pressed">
      <Storyboard>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[0].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="#ffcc34"/>
        </ColorAnimationUsingKeyFrames>
        <ColorAnimationUsingKeyFrames
            Storyboard.TargetProperty=
            "(Shape.Fill).(GradientBrush.GradientStops)[1].
            (GradientStop.Color)"
            Storyboard.TargetName="GelBackground">
          <EasingColorKeyFrame KeyTime="0" Value="#cc9900"/>
        </ColorAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
  </VisualStateGroup>
  <VisualStateGroup Name="FocusedStates">
    <VisualState Name="Focused" />
    <VisualState Name="Unfocused" />
  </VisualStateGroup>

</VisualStateManager.VisualStateGroups>
```

The state change is already evident. Moving the mouse over a `Button` or clicking the `Button` changes its user interface. Next, an animation between state transitions is added.

Transitions

With state transitions you can define what should happen when a change into a state occurs. Transitions are added by using `VisualStateGroup.Transitions`. In the following example (code file `VisualStateDemo/Style.xaml`), the first transition is a global transition specifying that the state change should take 0.2 seconds, and a `QuadraticEase` function should be used for the animation. The second defined transition is specified if the state changes into the MouseOver state. With the implementation of this state transition, the

thickness of the ellipse Gelbackground is changed by adding 2 within 0.5 seconds, and after the animation is completed it reverts to its original value:

```
<ContentPresenter Name="GelButtonContent" VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Content="{TemplateBinding Content}" />
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup Name="CommonStates">
    <!-- ... -->
    <VisualStateGroup.Transitions>
      <VisualTransition GeneratedDuration="0:0:0.2" >
        <VisualTransition.GeneratedEasingFunction>
          <QuadraticEase EasingMode="EaseOut" />
        </VisualTransition.GeneratedEasingFunction>
      </VisualTransition>
      <VisualTransition GeneratedDuration="0:0:0.5" To="MouseOver">
        <Storyboard>
          <DoubleAnimation By="2" Duration="0:0:0.5"
              AutoReverse="True"
              Storyboard.TargetProperty="(Shape.StrokeThickness)"
              Storyboard.TargetName="GelBackground" />
        </Storyboard>
      </VisualTransition>
    </VisualStateGroup.Transitions>
  </VisualStateGroup>
  <VisualStateGroup Name="FocusedStates">
    <VisualState Name="Focused" />
    <VisualState Name="Unfocused" />
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

> **NOTE** *Using custom states, you can easily change the state with the* `VisualStateManager` *class, invoking the method* `GoToElementState`.

3-D

This last section of a long chapter introduces the 3-D features of WPF. Here you'll find the information you need to get started.

> **NOTE** *The namespace for 3-D with WPF is* `System.Windows.Media.Media3D`.

To understand 3-D with WPF it is important to know the difference between the coordinate systems. Figure 35-34 shows the WPF 3-D coordinate system. The origin is placed in the center. The *x*-axis has positive values to the right and negative values to the left. The *y*-axis is vertical with positive values up and negative values down. The *z*-axis defines positive values in direction to the viewer of the scene.

The most important concepts to understand in order to understand 3-D with WPF are that of model, camera, and lights. The model defines what is shown using triangles. The camera defines the point at which and how we look at the model, and without light the model is dark. The light defines how the complete scene is illuminated. The following sections provide details about how to define the model, camera, and light with WPF and what different options are available. Also covered is how the scene can be animated.

Model

This section creates a model that has the 3-D look of a book. A 3-D model is made up of triangles, so the simplest model is just one triangle. More complex models are made from multiple triangles. Rectangles can be made from two triangles, and balls are made from a multiplicity of triangles. The more triangles used, the rounder the ball.

With the book model, each side is a rectangle, which could be made from only two triangles. However, because the front cover has three different materials, six triangles are used.

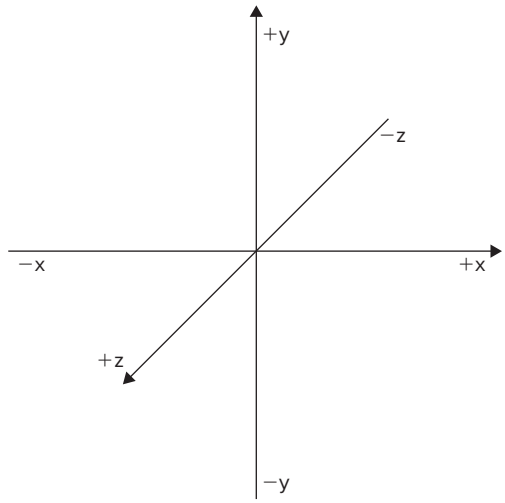


FIGURE 35-34

A triangle is defined by the `Positions` property of the `MeshGeometry3D`. This example uses just a part of the front side of the book. The `MeshGeometry3D` defines two triangles. You can count five coordinates for the points because the third point of the first triangle is also the first point of the second triangle. This can be done for optimization to reduce the size of the model. All the points use the same *z* coordinate, 0, and *x/y* coordinates 0 0, 10 0, 0 10, 10 10, and 10 0. The property `TriangleIndices` indicates the order of the positions. The first triangle is defined clockwise, the second triangle counterclockwise. With this property you define which side of the triangle is visible. One side of the triangle shows the color defined with the `Material` property of the `GeometryModel3D` class, and the other side shows the `BackMaterial` property.

The rendering surface for 3-D is `ModelVisual3D`, which surrounds the models as shown (code file `3DDemo/MainWindow.xaml`):

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <Model3DGroup>

      <!-- front -->
      <GeometryModel3D>
        <GeometryModel3D.Geometry>
          <MeshGeometry3D
              Positions="0 0 0, 10 0 0, 0 10 0, 10 10 0, 10 0 0"
              TriangleIndices="0, 1, 2, 2, 4, 3" />
        </GeometryModel3D.Geometry>
```

The `Material` property of the `GeometryModel` defines what material is used by the model. Depending on the viewpoint, the `Material` or `BackMaterial` property is important.

WPF offers different material types: `DiffuseMaterial`, `EmissiveMaterial`, and `SpecularMaterial`. The material influences the look of the model, together with the light that is used to illuminate the scene. `EmmisiveMaterial` and the color applied to the brush of the material are part of the calculations to define the light to show the model. `SpecularMaterial` adds illuminated highlight reflections when specular highlight reflections occur. The example code makes use of `DiffuseMaterial` and references a brush from the resource named `mainCover`:

```
        <GeometryModel3D.Material>
          <DiffuseMaterial Brush="{StaticResource mainCover}" />
        </GeometryModel3D.Material>
      </GeometryModel3D>
```

The brush for the main cover is a `VisualBrush`. The `VisualBrush` has a `Border` with a `Grid` that consists of two `Label` elements. One `Label` element defines the text "Professional C# 4" and is written to the cover:

```
<VisualBrush x:Key="mainCover">
  <VisualBrush.Visual>
    <Border Background="Red">
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="30" />
          <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Label Grid.Row="0" HorizontalAlignment="Center">
            Professional C# 5</Label>
        <Label Grid.Row="1"></Label>
      </Grid>
    </Border>
  </VisualBrush.Visual>
</VisualBrush>
```

Because a brush is defined by a 2-D coordinate system and the model has a 3-D coordinate system, a translation between them needs to be done. This translation is done by the `TextureCoordinates` property of the `MeshGeometry3D`. This property specifies every point of the triangle and shows how it maps to 2-D. The first point, 0 0 0, maps to 0 1, the second point, 10 0 0, maps to 1 1, and so on. Be aware that *y* has a different direction in the 3-D and 2-D coordinate systems. Figure 35-35 shows the coordinate system for 2-D:

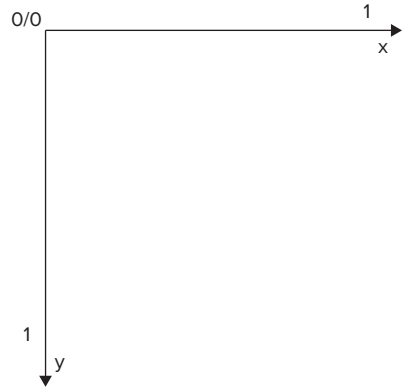


FIGURE 35-35

```
<MeshGeometry3D Positions="0 0 0, 10 0 0, 0 10 0, 10 10 0, 10 0 0"
    TriangleIndices="0, 1, 2, 2, 4, 3"
    TextureCoordinates="0 1, 1 1, 0 0, 1 0, 1 1" />
```

Cameras

A camera is needed with a 3-D model in order to see something. The following example (code file `3DDemo/MainWindow.xaml`) uses the `PerspectiveCamera`, which has a position and a direction. Changing the camera position to the left moves the model to the right and vice versa. Changing the *y* position of the camera, the model appears larger or smaller. With this camera, the further away the model is, the smaller it becomes:

```
<Viewport3D.Camera>
  <PerspectiveCamera Position="0,0,25" LookDirection="15,6,-50" />
</Viewport3D.Camera>
```

WPF also has an `OrtographicCamera` that doesn't have a horizon on the scene, so the size of the element doesn't change if it is further away. With `MatrixCamera`, the behavior of the camera can be exactly specified.

Lights

Without any light specified it is dark. A 3-D scene requires a light source to make the model visible. Different lights can be used. The `AmbientLight` lights the scene uniformly. `DirectionalLight` is a light that shines in one direction, similar to sunlight. `PointLight` has a position in space and lights in all directions. `SpotLight` has a position as well but uses a cone for its lighting.

The following example code uses a `SpotLight` with a position, a direction, and cone angles:

```
<ModelVisual3D>
  <ModelVisual3D.Content>
    <SpotLight Color="White" InnerConeAngle="20" OuterConeAngle="60"
               Direction="15,6,-50" Position="0,0,25" />
  </ModelVisual3D.Content>
</ModelVisual3D>
```

Rotation

To get a 3-D look from the model, it should be able to be rotated. For rotation, the `RotateTransform3D` element is used to define the center of the rotation and the rotation angle:

```
<Model3DGroup.Transform>
  <RotateTransform3D CenterX="0" CenterY="0" CenterZ="0">
    <RotateTransform3D.Rotation>
      <AxisAngleRotation3D x:Name="angle" Axis="-1,-1,-1" Angle="70" />
    </RotateTransform3D.Rotation>
  </RotateTransform3D>
</Model3DGroup.Transform>
```

To run a rotation from the completed model, an animation is started by an event trigger. The animation changes the `Angle` property of the `AxisAngleRotation3D` element continuously:

```
<Window.Triggers>
  <EventTrigger RoutedEvent=f"Window.Loaded">
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation From="0" To="360" Duration="00:00:10"
            Storyboard.TargetName="angle"
            Storyboard.TargetProperty="Angle"
            RepeatBehavior="Forever" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Window.Triggers>
```

Running the application results in the output shown in Figure 35-36.

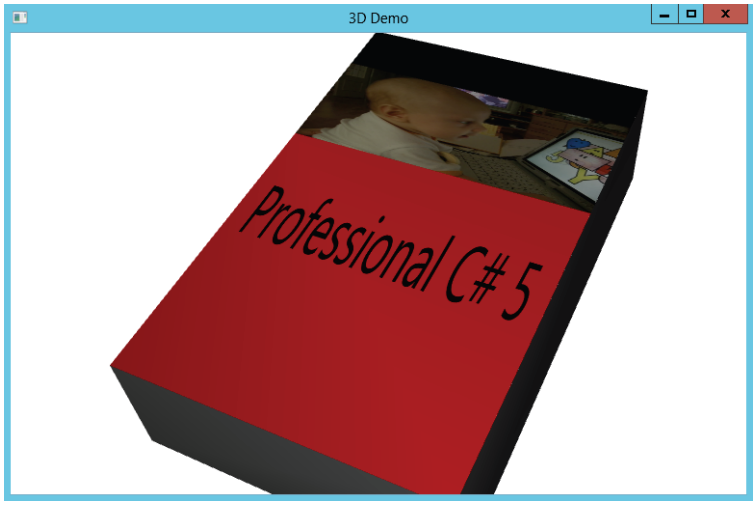


FIGURE 35-36

SUMMARY

In this chapter you have taken a brief tour through many of the features of WPF. WPF makes it easy to separate the work of developers and designers. All UI features can be created with XAML, and the functionality can be created by using code-behind.

You have seen many controls and containers, all of which are based on vector-based graphics. Vector-based graphics enable WPF elements to be scaled, sheared, and rotated. Because content controls offer great content flexibility, the event-handling mechanism is based on bubbling and tunneling events.

Different kinds of brushes are available to paint the background and foreground of elements. You can use not only solid brushes, and linear or radial gradient brushes, but also visual brushes to do reflections or show videos.

Styling and templates enable you to customize the look of controls; and triggers enable you to change properties of WPF elements dynamically. Animations can be done easily by animating a property value from a WPF control. The next chapter continues with WPF, covering data binding, commands, navigation, and several more features.

36

Business Applications with WPF

WHAT'S IN THIS CHAPTER?

- Menu and ribbon controls
- Using commanding for input handling
- Data binding to elements, objects, lists, and XML
- Value conversions and validation
- Using the `TreeView` to display hierarchical data
- Displaying and grouping data with the `DataGrid`
- Live shaping with the Collection View Source

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `http://www.wrox.com/remtitle.cgi?isbn=1118314425` on the Download Code tab. The code for this chapter is divided into the following major examples:

- Books Demo
- Multi Binding Demo
- Priority Binding Demo
- XML Binding Demo
- Validation Demo
- Formula 1 Demo
- Live Shaping

INTRODUCTION

In the previous chapter you read about some of the core functionality of WPF. This chapter continues the journey through WPF. Here you read about important aspects for creating complete applications, such as data binding and command handling, and about the `DataGrid` control. Data binding is an important concept for bringing data from .NET classes into the user interface, and allowing the user to change data. WPF not only allows binding to simple entities or lists, but also offers binding of one

UI property to multiple properties of possible different types with multi binding and priority binding that you'll learn here as well. Along with data binding it is also important to validate data entered by a user. Here, you can read about different ways for validation including the interface `INotifyDataErrorInfo` that is new with .NET 4.5. Also covered in this chapter is commanding, which enables mapping events from the UI to code. In contrast to the event model, this provides a better separation between XAML and code. You will learn about using predefined commands and creating custom commands.

The `TreeView` and `DataGrid` controls are UI controls to display bound data. You will see the `TreeView` control to display data in the tree where data is loaded dynamically depending on the selection of the user. With the `DataGrid` control you will learn how to using filtering, sorting, and grouping, as well as one new .NET 4.5 feature named *live shaping* that allows changing sorting or filtering options to change in real time.

To begin let's start with the `Menu` and the `Ribbon` controls. The `Ribbon` control made it into the release of .NET 4.5.

MENU AND RIBBON CONTROLS

Many data-driven applications contain menus and toolbars or ribbon controls to enable users to control actions. With WPF 4.5, ribbon controls are now available as well, so both menu and ribbon controls are covered here.

In this section, you create a new WPF application named BooksDemo to use throughout this chapter—not only with menu and ribbon controls but also with commanding and data binding. This application displays a single book, a list of books, and a grid of books. Actions are started from menu or ribbon controls to which commands associated.

Menu Controls

Menus can easily be created with WPF using the `Menu` and `MenuItem` elements, as shown in the following code snippet containing two main menu items, File and Edit, and a list of submenu entries. The _ in front of the characters marks the special character that can be used to access the menu item easily without using the mouse. Using the Alt key makes these characters visible and enables access to the menu with this character. Some of these menu items have a command assigned, as discussed in the next section (XAML file `BooksDemo/MainWindow.xaml`):

```
<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
        Title="Books Demo App" Height="400" Width="600">
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="_File">
        <MenuItem Header="Show _Book" />
        <MenuItem Header="Show Book_s" />
        <Separator />
        <MenuItem Header="E_xit" />
      </MenuItem>
      <MenuItem Header="_Edit">
        <MenuItem Header="Undo" Command="Undo" />
        <Separator />
        <MenuItem Header="Cut" Command="Cut" />
        <MenuItem Header="Copy" Command="Copy" />
        <MenuItem Header="Paste" Command="Paste" />
      </MenuItem>
    </Menu>
  </DockPanel>
</Window>
```

Running the application results in the menus shown in Figure 36-1. The menus are not active yet because commands are not active.

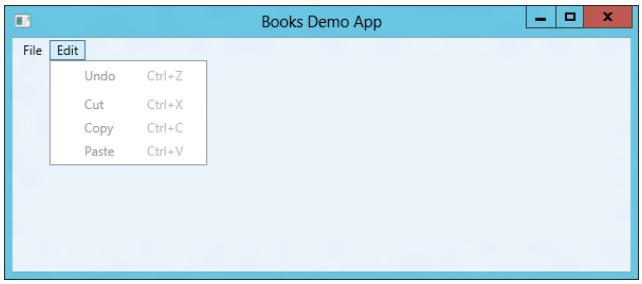


FIGURE 36-1

Ribbon Controls

Microsoft Office was the first application released with Microsoft's newly invented ribbon control. Shortly after its introduction, many users of previous versions of Office complained that they could not find the actions they wanted with the new UI. New Office users who had no experience with the previous user interface had a better experience with the new UI, easily finding actions that users of previous versions found hard to detect.

Of course, nowadays the ribbon control is very common in many applications. With Windows 8, the ribbon can be found in tools delivered with the operating system, e.g., Windows Explorer, Paint, and WordPad.

The WPF ribbon control is in the namespace `System.Windows.Controls.Ribbon` and requires referencing the assembly `system.windows.controls.ribbon`.

Figure 36-2 shows the ribbon control of the sample application. In the topmost line left of the title is the quick access toolbar. The leftmost item in the second line is the application menu, followed by two ribbon tabs: Home and Ribbon Controls. The Home tab, which is selected, shows two groups: Clipboard and Show. Both of these groups contain some button controls.

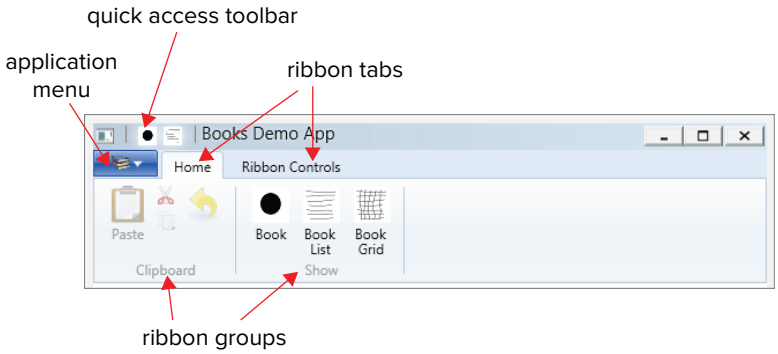


FIGURE 36-2

The `Ribbon` control is defined in the following code snippet. The first children of the `Ribbon` element are defined by the `QuickAccessToolBar` property. This toolbar contains two `RibbonButton`

controls with small images referenced. These buttons provide users with direct access to quickly and easily fulfill actions:

```
<Ribbon DockPanel.Dock="Top">
  <Ribbon.QuickAccessToolBar>
    <RibbonQuickAccessToolBar>
      <RibbonButton SmallImageSource="Images/one.png" />
      <RibbonButton SmallImageSource="Images/list.png" />
    </RibbonQuickAccessToolBar>
  </Ribbon.QuickAccessToolBar>
```

To get these buttons from the quick access toolbar directly to the chrome of the Window, the base class needs to be changed to the `RibbonWindow` class instead of the `Window` class (code file `BooksDemo/MainWindow.xaml.cs`):

```
public partial class MainWindow : RibbonWindow
{
```

Changing the base class with the code-behind also requires a change in the XAML code to use the `RibbonWindow` element:

```
<RibbonWindow x:Class="Wrox.ProCSharp.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
        Title="Books Demo App" Height="400" Width="600">
```

The application menu is defined by using the `ApplicationMenu` property. The application menu defines two menu entries—the first one to show a book, the second one to close the application:

```
<Ribbon.ApplicationMenu>
  <RibbonApplicationMenu SmallImageSource="Images/books.png" >
    <RibbonApplicationMenuItem Header="Show _Book" />
    <RibbonSeparator />
    <RibbonApplicationMenuItem Header="Exit" Command="Close" />
  </RibbonApplicationMenu>
</Ribbon.ApplicationMenu>
```

After the application menu, the content of the `Ribbon` control is defined by using `RibbonTab` elements. The title of the tab is defined with the `Header` property. The `RibbonTab` contains two `RibbonGroup` elements. Each of the `RibbonGroup` elements contains `RibbonButton` elements. With the buttons, a `Label` can be set to display a text and either `SmallImageSource` or `LargeImageSource` properties for displaying an image:

```
<RibbonTab Header="Home">
  <RibbonGroup Header="Clipboard">
    <RibbonButton Command="Paste" Label="Paste"
        LargeImageSource="Images/paste.png" />
    <RibbonButton Command="Cut" SmallImageSource="Images/cut.png" />
    <RibbonButton Command="Copy" SmallImageSource="Images/copy.png" />
    <RibbonButton Command="Undo" LargeImageSource="Images/undo.png" />
  </RibbonGroup>
  <RibbonGroup Header="Show">
    <RibbonButton LargeImageSource="Images/one.png" Label="Book" />
    <RibbonButton LargeImageSource="Images/list.png" Label="Book List" />
    <RibbonButton LargeImageSource="Images/grid.png" Label="Book Grid" />
  </RibbonGroup>
</RibbonTab>
```

The second `RibbonTab` is just used to demonstrate different controls that can be used within a ribbon control, for example, text box, check box, combo box, split button, and gallery elements. Figure 36-3 shows this tab open.

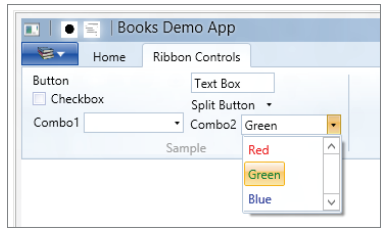


FIGURE 36-3

```
<RibbonTab Header="Ribbon Controls">
  <RibbonGroup Header="Sample">
    <RibbonButton Label="Button" />
    <RibbonCheckBox Label="Checkbox" />
    <RibbonComboBox Label="Combo1">
      <Label>One</Label>
      <Label>Two</Label>
    </RibbonComboBox>
    <RibbonTextBox>Text Box </RibbonTextBox>
    <RibbonSplitButton Label="Split Button">
      <RibbonMenuItem Header="One" />
      <RibbonMenuItem Header="Two" />
    </RibbonSplitButton>
    <RibbonComboBox Label="Combo2" IsEditable="False">
      <RibbonGallery SelectedValuePath="Content" MaxColumnCount="1"
          SelectedValue="Green">
        <RibbonGalleryCategory>
          <RibbonGalleryItem Content="Red" Foreground="Red" />
          <RibbonGalleryItem Content="Green" Foreground="Green" />
          <RibbonGalleryItem Content="Blue" Foreground="Blue" />
        </RibbonGalleryCategory>
      </RibbonGallery>
    </RibbonComboBox>
  </RibbonGroup>
</RibbonTab>
```

> **NOTE** *For additional information about the* `Ribbon` *control, read Chapter 30, "Managed Extensibility Framework," in which ribbon items are built dynamically.*

COMMANDING

Commanding is a WPF concept that creates a loose coupling between the source of an action (for example, a button) and the target that does the work (for example, a handler method). This concept is based on the *Command* pattern from the Gang of Four. Events are strongly coupled (at least with XAML 2006). Compiling the XAML code that includes references to events requires that the code-behind have a handler implemented and available at compile time. With commands, the coupling is loose.

The action that is executed is defined by a command object. Commands implement the interface `ICommand`. Command classes that are used by WPF are `RoutedCommand` and a class that derives from it, `RoutedUICommand`. `RoutedUICommand` defines an additional `Text` property that is not defined by `ICommand`. This property can be used as textual information in the UI. `ICommand` defines the methods `Execute` and `CanExecute`, which are executed on a target object.

The *command source* is an object that invokes the command. Command sources implement the interface `ICommandSource`. Examples of such command sources are button classes that derive from `ButtonBase`, `Hyperlink`, and `InputBinding`. `KeyBinding` and `MouseBinding` are examples of `InputBinding` derived classes. Command sources have a `Command` property whereby a command object implementing `ICommand` can be assigned. This fires the command when the control is used, such as with the click of a button.

The *command target* is an object that implements a handler to perform the action. With command binding, a mapping is defined to map the handler to a command. Command bindings define what handler is invoked on a command. Command bindings are defined by the `CommandBinding` property that is implemented in the `UIElement` class. Thus, every class that derives from `UIElement` has the `CommandBinding` property.

This makes finding the mapped handler a hierarchical process. For example, a button that is defined within a `StackPanel` that is inside a `ListBox`—which itself is inside a `Grid`—can fire a command. The handler is specified with command bindings somewhere up the tree—such as with command bindings of a `Window`. The next section changes the implementation of the BooksDemo project to use commands.

Defining Commands

.NET gives you classes that return predefined commands. The `ApplicationCommands` class defines the static properties `New`, `Open`, `Close`, `Print`, `Cut`, `Copy`, `Paste`, and others. These properties return `RoutedUICommand` objects that can be used for a specific purpose. Other classes offering commands are `NavigationCommands` and `MediaCommands`. `NavigationCommands` is self-explanatory, providing commands that are common for navigation such as `GoToPage`, `NextPage`, and `PreviousPage`. `MediaCommands` are useful for running a media player, with `Play`, `Pause`, `Stop`, `Rewind`, and `Record`.

It's not hard to define custom commands that fulfill application domain–specific actions. For this, the `BooksCommands` class is created, which returns `RoutedUICommand`s with the `ShowBook` and `ShowBooksList` properties. You can also assign an input gesture to a command, such as `KeyGesture` or `MouseGesture`. In the following example, a `KeyGesture` is assigned that defines the key `B` with the `Alt` modifier. An input gesture is a command source, so clicking the Alt+B combination invokes the command (code file `BooksDemo/BooksCommands.cs`):

```
public static class BooksCommands
{
  private static RoutedUICommand showBook;
  public static ICommand ShowBook
  {
    get
    {
      return showBook ?? (showBook = new RoutedUICommand("Show Book",
          "ShowBook", typeof(BooksCommands)));
    }
  }

  private static RoutedUICommand showBooksList;
  public static ICommand ShowBooksList
  {
    get
    {
      if (showBooksList == null)
      {
        showBooksList = new RoutedUICommand("Show Books", "ShowBooks",
            typeof(BooksCommands));
        showBook.InputGestures.Add(new KeyGesture(Key.B, ModifierKeys.Alt));
      }
      return showBooksList;
    }
  }
}
```

Defining Command Sources

Every class that implements the `ICommandSource` interface can be a source of commands, such as `Button` and `MenuItem`. Inside the `Ribbon` control created earlier, the `Command` property is assigned to several `RibbonButton` elements, e.g., in the quick access toolbar, as shown in the following code snippet (XAML file `BooksDemo/MainWindow.xaml`):

```
<Ribbon.QuickAccessToolBar>
  <RibbonQuickAccessToolBar>
    <RibbonButton SmallImageSource="Images/one.png"
```

```
            Command="local:BooksCommands.ShowBook" />
        <RibbonButton SmallImageSource="Images/list.png"
            Command="local:BooksCommands.ShowBooksList" />
      </RibbonQuickAccessToolBar>
    </Ribbon.QuickAccessToolBar>
```

Predefined commands such as `ApplicationCommands.Cut`, `Copy`, and `Paste` are assigned to the `Command` property of `RibbonButton` elements as well. With the predefined commands the shorthand notation is used:

```
<RibbonGroup Header="Clipboard">
  <RibbonButton Command="Paste" Label="Paste"
      LargeImageSource="Images/paste.png" />
  <RibbonButton Command="Cut" SmallImageSource="Images/cut.png" />
  <RibbonButton Command="Copy" SmallImageSource="Images/copy.png" />
  <RibbonButton Command="Undo" LargeImageSource="Images/undo.png" />
</RibbonGroup>
```

Command Bindings

Command bindings need to be added to connect them to handler methods. In the following example, the command bindings are defined within the `Window` element so these bindings are available to all elements within the window. When the command `ApplicationCommands.Close` is executed, the `OnClose` method is invoked. When the command `BooksCommands.ShowBooks` is executed, the `OnShowBooks` method is called:

```
<Window.CommandBindings>
  <CommandBinding Command="Close" Executed="OnClose" />
  <CommandBinding Command="local:BooksCommands.ShowBooksList"
      Executed="OnShowBooksList" />
</Window.CommandBindings>
```

With command binding you can also specify the `CanExecute` property, whereby a method is invoked to verify whether the command is available. For example, if a file is not changed, the `ApplicationCommands.Save` command could be unavailable.

The handler needs to be defined with an object parameter, for the sender, and `ExecutedRoutedEventArgs`, where information about the command can be accessed (code file `BooksDemo/MainWindow.xaml.cs`):

```
private void OnClose(object sender, ExecutedRoutedEventArgs e)
{
  Application.Current.Shutdown();
}
```

> **NOTE** *You can also pass parameters with a command. You can do this by specifying the* `CommandParameter` *property with a command source, such as the* `MenuItem`*. To access the parameter, use the* `Parameter` *property of* `ExecutedRoutedEventArgs`.

Command bindings can also be defined by controls. The `TextBox` control defines bindings for `ApplicationCommands.Cut`, `ApplicationCommands.Copy`, `ApplicationCommands.Paste`, and `ApplicationCommands.Undo`. This way, you only need to specify the command source and use the existing functionality within the `TextBox` control.

DATA BINDING

WPF data binding takes another huge step forward compared with previous technologies. Data binding gets data from .NET objects for the UI or the other way around. Simple objects can be bound to UI elements, lists of objects, and XAML elements themselves. With WPF data binding, the target can be any dependency

property of a WPF element, and every property of a CLR object can be the source. Because a WPF element is implemented as a .NET class, every WPF element can be the source as well. Figure 36-4 shows the connection between the source and the target. The `Binding` object defines the connection.

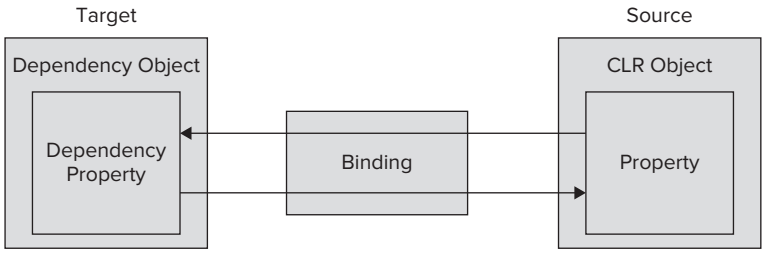


FIGURE 36-4

Binding supports several binding modes between the target and source. With *one-way* binding, the source information goes to the target but if the user changes information in the user interface, the source is not updated. For updates to the source, *two-way* binding is required.

The following table shows the binding modes and their requirements.

BINDING MODE	DESCRIPTION
One-time	Binding goes from the source to the target and occurs only once when the application is started or the data context changes. Here, you get a snapshot of the data.
One-way	Binding goes from the source to the target. This is useful for read-only data, because it is not possible to change the data from the user interface. To get updates to the user interface, the source must implement the interface `INotifyPropertyChanged`.
Two-way	With two-way binding, the user can make changes to the data from the UI. Binding occurs in both directions—from the source to the target and from the target to the source. The source needs to implement read/write properties so that changes can be updated from the UI to the source.
One-way-to-source	With one-way-to-source binding, if the target property changes, the source object is updated.

WPF data binding involves many facets besides the binding modes. This section provides details on binding to XAML elements, binding to simple .NET objects, and binding to lists. Using change notifications, the UI is updated with changes in the bound objects. The material presented here discusses getting the data from object data providers and directly from the code. Multibinding and priority binding demonstrate different binding possibilities other than the default binding. This section also describes dynamically selecting data templates, and validation of binding values.

Let's start with the BooksDemo sample application.

BooksDemo Application Content

In the previous sections, a ribbon and commands have been defined with the BooksDemo application. Now content is added. Change the XAML file `MainWindow.xaml` by adding a `ListBox`, a `Hyperlink`, and a `TabControl` (XAML file `BooksDemo/MainWindow.xaml`):

```
<ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
  <Hyperlink Click="OnShowBook">Show Book</Hyperlink>
</ListBox>
<TabControl Margin="5" x:Name="tabControl1">
</TabControl>
```

Now add a WPF user control named `BookUC`. This user control contains a `DockPanel`, a `Grid` with several rows and columns, a `Label`, and `TextBox` controls (XAML file `BooksDemo/BookUC.xaml`):

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <DockPanel>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
          HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Publisher" Grid.Row="1" Grid.Column="0"
          Margin="10,0,5,0" HorizontalAlignment="Left"
          VerticalAlignment="Center" />
      <Label Content="Isbn" Grid.Row="2" Grid.Column="0"
          Margin="10,0,5,0" HorizontalAlignment="Left"
          VerticalAlignment="Center" />
      <TextBox Grid.Row="0" Grid.Column="1" Margin="5" />
      <TextBox Grid.Row="1" Grid.Column="1" Margin="5" />
      <TextBox Grid.Row="2" Grid.Column="1" Margin="5" />
      <StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2">
        <Button Content="Show Book" Margin="5" Click="OnShowBook" />
      </StackPanel>
    </Grid>
  </DockPanel>
</UserControl>
```

Within the `OnShowBook` handler in the `MainWindow.xaml.cs`, create a new instance of the user control `BookUC` and add a new `TabItem` to the `TabControl`. Then change the `SelectedIndex` property of the `TabControl` to open the new tab (code file `BooksDemo/MainWindow.xaml.cs`):

```
private void OnShowBook(object sender, ExecutedRoutedEventArgs e)
{
  var bookUI = new BookUC();
  this.tabControl1.SelectedIndex = this.tabControl1.Items.Add(
      new TabItem { Header = "Book", Content = bookUI });
}
```

After building the project you can start the application and open the user control within the `TabControl` by clicking the hyperlink.

Binding with XAML

In addition to being the target for data binding, a WPF element can also be the source. You can bind the source property of one WPF element to the target of another WPF element.

In the following code example, data binding is used to resize the controls within the user control with a slider. You add a `StackPanel` control to the user control `BookUC`, which contains a `Label` and a `Slider`

control. The `Slider` control defines `Minimum` and `Maximum` values that define the scale, and an initial value of 1 is assigned to the `Value` property (XAML file `BooksDemo/BooksUC.xaml`):

```
<DockPanel>
  <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
      HorizontalAlignment="Right">
    <Label Content="Resize" />
    <Slider x:Name="slider1" Value="1" Minimum="0.4" Maximum="3"
        Width="150" HorizontalAlignment="Right" />
  </StackPanel>
```

Now you set the `LayoutTransform` property of the `Grid` control and add a `ScaleTransform` element. With the `ScaleTransform` element, the `ScaleX` and `ScaleY` properties are data bound. Both properties are set with the `Binding` markup extension. In the `Binding` markup extension, the `ElementName` is set to `slider1` to reference the previously created `Slider` control. The `Path` property is set to the `Value` property to get the value of the slider:

```
<Grid>
  <Grid.LayoutTransform>
    <ScaleTransform x:Name="scale1"
        ScaleX="{Binding Path=Value, ElementName=slider1}"
        ScaleY="{Binding Path=Value, ElementName=slider1}" />
  </Grid.LayoutTransform>
```

When running the application, you can move the slider and thus resize the controls within the `Grid`, as shown in Figures 36-5 and 36-6.

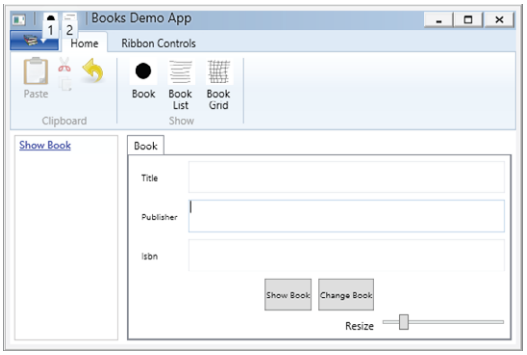


FIGURE 36-5

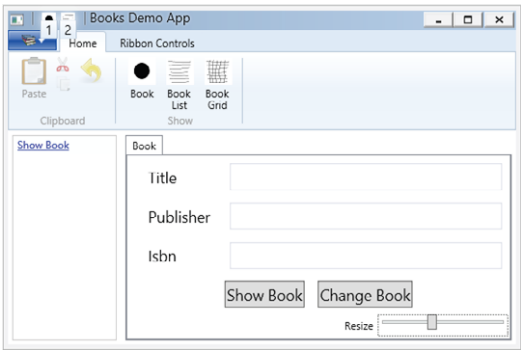


FIGURE 36-6

Rather than define the binding information with XAML code, as shown in the preceding code with the `Binding` metadata extension, you can do it with code-behind. With code-behind you have to create a new `Binding` object and set the `Path` and `Source` properties. The `Source` property must be set to the source object; here, it is the WPF object `slider1`. The `Path` is set to a `PropertyPath` instance that is initialized with the name of the property of the source object, `Value`. With controls that derive from `FrameworkElement`, you can invoke the method `SetBinding` to define the binding. However, `ScaleTransform` does not derive from `FrameworkElement` but from the `Freezable` base class instead. Use the helper class `BindingOperations` to bind such controls. The `SetBinding` method of the `BindingOperations` class requires a `DependencyObject`—which is the `ScaleTransform` instance in the example. With the second and third argument, the `SetBinding` method requires the `dependency` property of the target (which should be bound), and the `Binding` object:

```
var binding = new Binding
{
  Path = new PropertyPath("Value"),
  Source = slider1
```

```
};
BindingOperations.SetBinding(scale1, ScaleTransform.ScaleXProperty,
    binding);
BindingOperations.SetBinding(scale1, ScaleTransform.ScaleYProperty,
    binding);
```

> **NOTE** *Remember that all classes that derive from* `DependencyObject` *can have dependency properties. You can learn more about dependency properties in Chapter 29, "Core XAML."*

You can configure a number of binding options with the `Binding` class, as described in the following table:

BINDING CLASS MEMBERS	DESCRIPTION
`Source`	Use this property to define the source object for data binding.
`RelativeSource`	Specify the source in relation to the target object. This is useful to display error messages when the source of the error comes from the same control.
`ElementName`	If the source is a WPF element, you can specify the source with the `ElementName` property.
`Path`	Use this property to specify the path to the source object. This can be the property of the source object, but indexers and properties of child elements are also supported.
`XPath`	With an XML data source, you can define an XPath query expression to get the data for binding.
`Mode`	The mode defines the direction for the binding. The `Mode` property is of type `BindingMode`. `BindingMode` is an enumeration with the following values: `Default`, `OneTime`, `OneWay`, `TwoWay`, and `OneWayToSource`. The default mode depends on the target: with a `TextBox`, two-way binding is the default; with a `Label` that is read-only, the default is one-way. `OneTime` means that the data is only `init` loaded from the source; `OneWay` updates from the source to the target. With `TwoWay` binding, changes from the WPF elements are written back to the source. `OneWayToSource` means that the data is never read but always written from the target to the source.
`Converter`	Use this property to specify a converter class that converts the data for the UI and back. The converter class must implement the interface `IValueConverter`, which defines the methods `Convert` and `ConvertBack`. You can pass parameters to the converter methods with the `ConverterParameter` property. The converter can be culture-sensitive; and the culture can be set with the `ConverterCulture` property.
`FallbackValue`	Use this property to define a default value that is used if binding doesn't return a value.
`ValidationRules`	Using this property, you can define a collection of `ValidationRule` objects that are checked before the source is updated from the WPF target elements. The class `ExceptionValidationRule` is derived from the class `ValidationRule` and checks for exceptions.
`Delay`	This property is new with WPF 4.5. It enables you to specify an amount of time to wait before the binding source is updated. This can be used in scenarios where you want to give the user some time to enter more characters before starting a validation.

Simple Object Binding

To bind to CLR objects, with the .NET classes you just have to define properties, as shown in the `Book` class example and the properties `Title`, `Publisher`, `Isbn`, and `Authors`. This class is in the `Data` folder of the BooksDemo project (code file `BooksDemo/Data/Book.cs`).

```
using System.Collections.Generic;

namespace Wrox.ProCSharp.WPF.Data
{
  public class Book
  {
    public Book(string title, string publisher, string isbn,
                params string[] authors)
    {
      this.Title = title;
      this.Publisher = publisher;
      this.Isbn = isbn;
      this.authors.AddRange(authors);
    }
    public Book()
      : this("unknown", "unknown", "unknown")
    {
    }
    public string Title { get; set; }
    public string Publisher { get; set; }
    public string Isbn { get; set; }

    private readonly List<string> authors = new List<string>();
    public string[] Authors
    {
      get
      {
        return authors.ToArray();
      }
    }

    public override string ToString()
    {
      return Title;
    }
  }
}
```

In the XAML code of the user control `BookUC`, several labels and `TextBox` controls are defined to display book information. Using `Binding` markup extensions, the `TextBox` controls are bound to the properties of the `Book` class. With the `Binding` markup extension, nothing more than the `Path` property is defined to bind it to the property of the `Book` class. There's no need to define a source because the source is defined by assigning the `DataContext`, as shown in the code-behind that follows. The mode is defined by its default with the `TextBox` element, and this is two-way binding (XAML file `BooksDemo/BookUC.xaml`):

```
<TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
```

With the code-behind, a new `Book` object is created, and the book is assigned to the `DataContext` property of the user control. `DataContext` is a dependency property that is defined with the base class `FrameworkElement`. Assigning the `DataContext` with the user control means that every element in the user control has a default binding to the same data context (code file `BooksDemo/MainWindow.xaml.cs`):

```
private void OnShowBook(object sender, ExecutedRoutedEventArgs e)
{
  var bookUI = new BookUC();
  bookUI.DataContext = new Book
  {
    Title = "Professional C# 4 and .NET 4",
    Publisher = "Wrox Press",
    Isbn = "978-0-470-50225-9"
  };
  this.tabControl1.SelectedIndex =
    this.tabControl1.Items.Add(
      new TabItem { Header = "Book", Content = bookUI });
}
```

After starting the application, you can see the bound data, as shown in Figure 36-7.

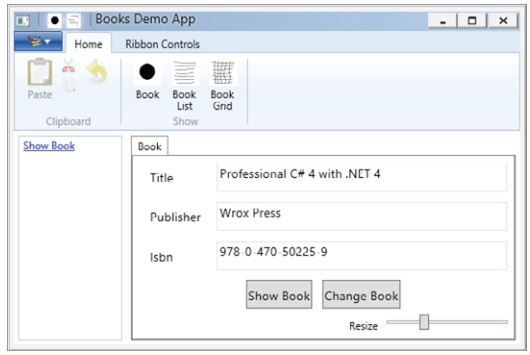


FIGURE 36-7

To see two-way binding in action (changes to the input of the WPF element are reflected inside the CLR object), the `Click` event handler of the button in the user control, the `OnShowBook` method, is implemented. When implemented, a message box pops up to show the current title and ISBN number of the `book1` object. Figure 36-8 shows the output from the message box after a change to the input was made during runtime (code file `BooksDemo/BookUC.xaml.cs`):

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
  Book theBook = this.DataContext as Book;
  if (theBook != null)
    MessageBox.Show(theBook.Title, theBook.Isbn);
}
```

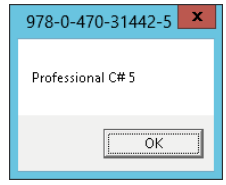


FIGURE 36-8

Change Notification

With the current two-way binding, the data is read from the object and written back. However, if data is not changed by the user, but is instead changed directly from the code, the UI does not receive the change information. You can easily verify this by adding a button to the user control and implementing the `Click` event handler `OnChangeBook` (XAML file `BooksDemo/BookUC.xaml`):

```
<StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2"
            Orientation="Horizontal" HorizontalAlignment="Center">
  <Button Content="Show Book" Margin="5" Click="OnShowBook" />
  <Button Content="Change Book" Margin="5" Click="OnChangeBook" />
</StackPanel>
```

Within the implementation of the handler, the book inside the data context is changed but the user interface doesn't show the change (code file `BooksDemo/BookUC.xaml.cs`):

```
private void OnChangeBook(object sender, RoutedEventArgs e)
{
  Book theBook = this.DataContext as Book;
  if (theBook != null)
  {
    theBook.Title = "Professional C# 5";
    theBook.Isbn = "978-0-470-31442-5";
  }
}
```

To get change information to the user interface, the entity class must implement the interface `INotifyPropertyChanged`. Instead of having an implementation with every class that needs this interface, the abstract base class `BindableObject` is created. This base class implements the interface `INotifyPropertyChanged`. The interface defines the event `PropertyChanged`, which is fired from the `OnPropertyChanged` method. As a convenience for firing the event from the property setters from the derived classes, the method `SetProperty` makes the change of the property and invokes the method `OnPropertyChanged` to fire the event. This method makes use of the caller information feature from C# using the attribute `CallerMemberName`. Defining the parameter `propertyName` as an optional parameter with this attribute, the C# compiler passes the name of the property with this parameter, so it's not necessary to add a hard-coded string to the code (code file `BooksDemo/Data/BindableObject.cs`):

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Wrox.ProCSharp.WPF.Data
{
  public abstract class BindableObject : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string propertyName)
    {
      var propertyChanged = PropertyChanged;
      if (propertyChanged != null)
      {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
      }
    }

    protected void SetProperty<T>(ref T item, T value,
        [CallerMemberName] string propertyName = null)
    {
      if (!EqualityComparer<T>.Default.Equals(item, value))
      {
        item = value;
        OnPropertyChanged(propertyName);
      }
    }
  }
}
```

> **NOTE** *Caller information is covered in Chapter 16.*

The class `Book` is now changed to derive from the base class `BindableObject` in order to inherit the implementation of the interface `INotifyPropertyChanged`. The property setters are changed to invoke the `SetProperty` method, as shown here (code file `BooksDemo/Data/Book.cs`):

```
using System.ComponentModel;
using System.Collections.Generic;

namespace Wrox.ProCSharp.WPF.Data
{
  public class Book : BindableObject
  {
    public Book(string title, string publisher, string isbn,
                params string[] authors)
```

```
    {
      this.title = title;
      this.publisher = publisher;
      this.isbn = isbn;
      this.authors.AddRange(authors);
    }
    public Book()
      : this("unknown", "unknown", "unknown")
    {
    }

    private string title;
    public string Title {
      get
      {
        return title;
      }
      set
      {
        SetProperty(ref title, value);
      }
    }

    private string publisher;
    public string Publisher
    {
      get
      {
        return publisher;
      }
      set
      {
        SetProperty(ref publisher, value);
      }
    }
    private string isbn;
    public string Isbn
    {
      get
      {
        return isbn;
      }
      set
      {
        SetProperty(ref isbn, value);
      }
    }

    private readonly List<string> authors = new List<string>();
    public string[] Authors
    {
      get
      {
        return authors.ToArray();
      }
    }

    public override string ToString()
    {
      return this.title;
    }
  }
}
```

With this change, the application can be started again to verify that the user interface is updated following a change notification in the event handler.

Object Data Provider

Instead of instantiating the object in code-behind, you can do this with XAML. To reference a class from code-behind within XAML, you have to reference the namespace with the namespace declarations in the XML root element. The XML attribute `xmlns:local="clr-namespace:Wrox.ProCsharp.WPF"` assigns the .NET namespace `Wrox.ProCSharp.WPF` to the XML namespace alias `local`.

One object of the `Book` class is now defined with the `Book` element inside the `DockPanel` resources. By assigning values to the XML attributes `Title`, `Publisher`, and `Isbn`, you set the values of the properties from the `Book` class. `x:Key="theBook"` defines the identifier for the resource so that you can reference the book object (XAML file `BooksDemo/BookUC.xaml`):

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <DockPanel>
    <DockPanel.Resources>
      <local:Book x:Key="theBook" Title="Professional C# 4 and .NET 4"
          Publisher="Wrox Press" Isbn="978-0-470-50225-9" />
    </DockPanel.Resources>
```

> **NOTE** *If the .NET namespace to reference is in a different assembly, you have to add the assembly to the XML declaration:*
>
> `xmlsn:sys="clr-namespace:System;assembly=mscorlib"`

In the `TextBox` element, the `Source` is defined with the `Binding` markup extension that references the `theBook` resource:

```
    <TextBox Text="{Binding Path=Title, Source={StaticResource theBook}}"
        Grid.Row="0" Grid.Column="1" Margin="5" />
    <TextBox Text="{Binding Path=Publisher, Source={StaticResource theBook}}"
        Grid.Row="1" Grid.Column="1" Margin="5" />
    <TextBox Text="{Binding Path=Isbn, Source={StaticResource theBook}}"
        Grid.Row="2" Grid.Column="1" Margin="5" />
```

Because all these `TextBox` elements are contained within the same control, it is possible to assign the `DataContext` property with a parent control and set the `Path` property with the `TextBox` binding elements. Because the `Path` property is a default, you can also reduce the `Binding` markup extension to the following code:

```
  <Grid x:Name="grid1" DataContext="{StaticResource theBook}">
    <!-- ... -->
    <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1"
        Margin="5" />
    <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
        Margin="5" />
    <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1"
        Margin="5" />
```

Instead of defining the object instance directly within XAML code, you can define an object data provider that references a class to invoke a method. For use by the `ObjectDataProvider`, it's best to create a factory class that returns the object to display, as shown with the `BookFactory` class (code file `BooksDemo/Data/BookFactory.cs`):

```
using System.Collections.Generic;

namespace Wrox.ProCSharp.WPF.Data
{
  public class BookFactory
  {
    private List<Book> books = new List<Book>();

    public BookFactory()
    {
      books.Add(new Book
      {
        Title = "Professional C# 4 and .NET 4",
        Publisher = "Wrox Press",
        Isbn = "978-0-470-50225-9"
      });
    }

    public Book GetTheBook()
    {
      return books[0];
    }
  }
}
```

The `ObjectDataProvider` element can be defined in the resources section. The XML attribute `ObjectType` defines the name of the class; with `MethodName` you specify the name of the method that is invoked to get the book object (XAML file `BooksDemo/BookUC.xaml`):

```
<DockPanel.Resources>
  <ObjectDataProvider x:Key="theBook" ObjectType="local:BookFactory"
      MethodName="GetTheBook" />
</DockPanel.Resources>
```

The properties you can specify with the `ObjectDataProvider` class are listed in the following table:

OBJECTDATAPROVIDER PROPERTY	DESCRIPTION
`ObjectType`	Defines the type to create an instance.
`ConstructorParameters`	Using the `ConstructorParameters` collection, you can add parameters to the class to create an instance.
`MethodName`	Defines the name of the method that is invoked by the object data provider.
`MethodParameters`	Using this property, you can assign parameters to the method defined with the `MethodName` property.
`ObjectInstance`	Using this property, you can get and set the object that is used by the `ObjectDataProvider` class. For example, you can assign an existing object programmatically rather than define the `ObjectType` so that an object is instantiated by `ObjectDataProvider`.
`Data`	Enables you to access the underlying object that is used for data binding. If the `MethodName` is defined, with the `Data` property you can access the object that is returned from the method defined.

List Binding

Binding to a list is more frequently done than binding to simple objects. Binding to a list is very similar to binding to a simple object. You can assign the complete list to the `DataContext` from code-behind, or you can use an `ObjectDataProvider` that accesses an object factory that returns a list. With elements that support binding to a list (for example, a `ListBox`), the complete list is bound. With elements that support binding to just one object (for example, a `TextBox`), the current item is bound.

With the `BookFactory` class, now a list of `Book` objects is returned (code file `BooksDemo/Data/BookFactory.cs`):

```
public class BookFactory
{
  private List<Book> books = new List<Book>();

  public BookFactory()
  {
    books.Add(new Book("Professional C# 4 with .NET 4", "Wrox Press",
                       "978-0-470-50225-9", "Christian Nagel", "Bill Evjen",
                       "Jay Glynn", "Karli Watson", "Morgan Skinner"));
    books.Add(new Book("Professional C# 2008", "Wrox Press",
                       "978-0-470-19137-8", "Christian Nagel", "Bill Evjen",
                       "Jay Glynn", "Karli Watson", "Morgan Skinner"));
    books.Add(new Book("Beginning Visual C# 2010", "Wrox Press",
                       "978-0-470-50226-6", "Karli Watson", "Christian Nagel",
                       "Jacob Hammer Pedersen", "Jon D. Reid",
                       "Morgan Skinner", "Eric White"));
    books.Add(new Book("Windows 7 Secrets", "Wiley", "978-0-470-50841-1",
                       "Paul Thurrott", "Rafael Rivera"));
    books.Add(new Book("C# 2008 for Dummies", "For Dummies",
                       "978-0-470-19109-5", "Stephen Randy Davis",
                       "Chuck Sphar"));
  }

  public IEnumerable<Book> GetBooks()
  {
    return books;
  }
}
```

To use the list, create a new `BooksUC` user control. The XAML code for this control contains `Label` and `TextBox` controls that display the values of a single book, as well as a `ListBox` control that displays a book list. The `ObjectDataProvider` invokes the `GetBooks` method of the `BookFactory`, and this provider is used to assign the `DataContext` of the `DockPanel`. The `DockPanel` has the bound `ListBox` and `TextBox` as its children (XAML file `BooksDemo/BooksUC.xaml`):

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <UserControl.Resources>
    <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
                        MethodName="GetBooks" />
  </UserControl.Resources>
  <DockPanel DataContext="{StaticResource books}">
    <ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
        MinWidth="120" />
```

```
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
          HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Publisher" Grid.Row="1" Grid.Column="0" Margin="10,0,5,0"
          HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Isbn" Grid.Row="2" Grid.Column="0" Margin="10,0,5,0"
          HorizontalAlignment="Left" VerticalAlignment="Center" />
      <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
      <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
          Margin="5" />
      <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
    </Grid>
  </DockPanel>
</UserControl>
```

The new user control is started by adding a `Hyperlink` to `MainWindow.xaml`. It uses the `Command` property to assign the `ShowBooks` command. The command binding must be specified as well to invoke the event handler `OnShowBooksList`. (XAML file `BooksDemo/BooksUC.xaml`):

```
<ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
  <ListBoxItem>
    <Hyperlink Command="local:BooksCommands.ShowBook">Show Book</Hyperlink>
  </ListBoxItem>
  <ListBoxItem>
    <Hyperlink Command="local:ShowCommands.ShowBooksList">
        Show Books List</Hyperlink>
  </ListBoxItem>
</ListBox>
```

The implementation of the event handler adds a new `TabItem` control to the `TabControl`, assigns the `Content` to the user control `BooksUC` and sets the selection of the `TabControl` to the newly created `TabItem` (code file `BooksDemo/BooksUC.xaml.cs`):

```
private void OnShowBooks(object sender, ExecutedRoutedEventArgs e)
{
  var booksUI = new BooksUC();
  this.tabControl1.SelectedIndex =
    this.tabControl1.Items.Add(
      new TabItem { Header="Books List", Content=booksUI});
}
```

Because the `DockPanel` has the `Book` array assigned to the `DataContext`, and the `ListBox` is placed within the `DockPanel`, the `ListBox` shows all books with the default template, as illustrated in Figure 36-9.

For a more flexible layout of the `ListBox`, you have to define a template, as discussed in the previous chapter for `ListBox` styling. The `ItemTemplate` of the `ListBox` defines a `DataTemplate` with a `Label` element. The content of the label is bound to the `Title`. The item

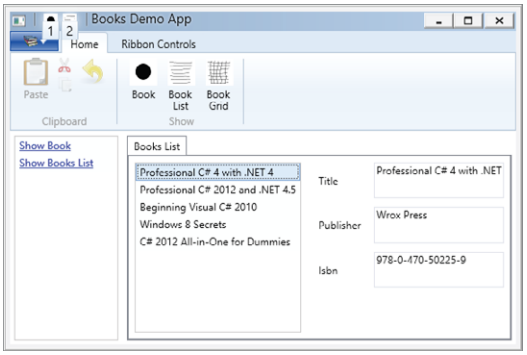


FIGURE 36-9

template is repeated for every item in the list. Of course, you can also add the item template to a style within resources (XAML file BooksDemo/BooksUC.xaml):

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
    MinWidth="120">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Label Content="{Binding Title}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Master Details Binding

Instead of just showing all the elements inside a list, you might want or need to show detail information about the selected item. It doesn't require a lot of work to do this. The Label and TextBox controls are already defined; currently, they only show the first element in the list.

There's one important change you have to make to the ListBox. By default, the labels are bound to just the first element of the list. By setting the ListBox property IsSynchronizedWithCurrentItem="True", the selection of the list box is set to the current item (XAML file BooksDemo/BooksUC.xaml):

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
         MinWidth="120" IsSynchronizedWithCurrentItem="True">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Label Content="{Binding Title}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Figure 36-10 shows the result; details about the selected item are shown on the right.

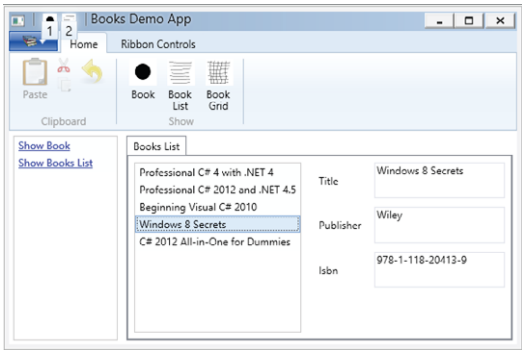


FIGURE 36-10

MultiBinding

Binding is one of the classes that can be used for data binding. BindingBase is the abstract base class of all bindings and has different concrete implementations. Besides Binding, there's also MultiBinding and PriorityBinding. MultiBinding enables you to bind one WPF element to multiple sources. For example, with a Person class that has LastName and FirstName properties, it is interesting to bind both properties to a single WPF element (code file MultiBindingDemo/Person.cs):

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

For MultiBinding, a markup extension is not available—therefore, the binding must be specified with XAML element syntax. The child elements of MultiBinding are Binding elements that specify the binding to the various properties. In the following example, the FirstName and LastName properties are used. The data context is set with the Grid element to reference the person1 resource.

To connect the properties, MultiBinding uses a Converter to convert multiple values to one. This converter uses a parameter that allows for different conversions based on the parameter (XAML file MultiBindingDemo/MainWindow.xaml):

```
<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
        Title="MainWindow" Height="240" Width="500">
  <Window.Resources>
    <local:Person x:Key="person1" FirstName="Tom" LastName="Turbo" />
    <local:PersonNameConverter x:Key="personNameConverter" />
  </Window.Resources>
  <Grid DataContext="{StaticResource person1}">
    <TextBox>
      <TextBox.Text>
        <MultiBinding Converter="{StaticResource personNameConverter}" >
          <MultiBinding.ConverterParameter>
            <system:String>FirstLast</system:String>
          </MultiBinding.ConverterParameter>
          <Binding Path="FirstName" />
          <Binding Path="LastName" />
        </MultiBinding>
      </TextBox.Text>
    </TextBox>
  </Grid>
</Window>
```

The multi-value converter implements the interface `IMultiValueConverter`. This interface defines two methods, `Convert` and `ConvertBack`. `Convert` receives multiple values with the first argument from the data source and returns one value to the target. With the implementation, depending on whether the parameter has a value of `FirstLast` or `LastFirst`, the result varies (code file `MultiBindingDemo/PersonNameConverter.cs`):

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace Wrox.ProCSharp.WPF
{
  public class PersonNameConverter : IMultiValueConverter
  {
    public object Convert(object[] values, Type targetType, object parameter,
                          CultureInfo culture)
    {
      switch (parameter as string)
      {
        case "FirstLast":
          return values[0] + " " + values[1];
        case "LastFirst":
          return values[1] + ", " + values[0];
        default:
          throw new ArgumentException(String.Format(
              "invalid argument {0}", parameter));
      }
    }

    public object[] ConvertBack(object value, Type[] targetTypes,
                                object parameter, CultureInfo culture)
    {
      throw new NotSupportedException();
    }
  }
}
```

In such simple scenarios, just combining some strings with a `MultiBinding` doesn't require an implementation of `IMultiValueConverter`. Instead, a definition for a format string is adequate,

as shown in the following XAML code snippet. The string format defined with the `MultiBinding` first needs a `{}` prefix. With XAML the curly brackets usually define a markup expression. Using {} as a prefix escapes this and defines that no markup expression, but instead a normal string, follows. The sample specifies that both Binding elements are separated by a comma and a blank (XAML file `MultiBindingDemo/MainWindow.xaml`):

```
<TextBox>
    <TextBox.Text>
      <MultiBinding StringFormat="{}{0}, {1}">
        <Binding Path="LastName" />
        <Binding Path="FirstName" />
      </MultiBinding>
    </TextBox.Text>
  </TextBox>
```

Priority Binding

`PriorityBinding` makes it easy to bind to data that is not readily available. If you need time to get the result with `PriorityBinding`, you can inform users about the progress so they are aware of the wait.

To illustrate priority binding, use the `PriorityBindingDemo` project to create the `Data` class. Accessing the `ProcessSomeData` property requires some time, which is simulated by calling the `Thread.Sleep` method (code file `PriorityBindingDemo/Data.cs`):

```
public class Data
{
  public string ProcessSomeData
  {
    get
    {
      Thread.Sleep(8000);
      return "the final result is here";
    }
  }
}
```

The `Information` class provides information to the user. The information from property `Info1` is returned immediately, whereas `Info2` returns information after five seconds. With a real implementation, this class could be associated with the processing class to get an estimated time frame for the user (code file `PriorityBindingDemo/Information.cs`):

```
public class Information
{
  public string Info1
  {
    get
    {
      return "please wait...";
    }
  }
  public string Info2
  {
    get
    {
      Thread.Sleep(5000);
      return "please wait a little more";
    }
  }
}
```

In the MainWindow.xaml file, the Data and Information classes are referenced and initiated within the resources of the Window (XAML file PriorityBindingDemo/MainWindow.xaml):

```
<Window.Resources>
    <local:Data x:Key="data1" />
    <local:Information x:Key="info" />
</Window.Resources>
```

PriorityBinding is done in place of normal binding within the Content property of a Label. It consists of multiple Binding elements whereby all but the last one have the IsAsync property set to True. Because of this, if the first binding expression result is not immediately available, the binding process chooses the next one. The first binding references the ProcessSomeData property of the Data class, which needs some time. Because of this, the next binding comes into play and references the Info2 property of the Information class. Info2 does not return a result immediately; and because IsAsync is set, the binding process does not wait but continues to the next binding. The last binding uses the Info1 property. If it doesn't immediately return a result, you would wait for the result because IsAsync is set to the default, False:

```
<Label>
  <Label.Content>
    <PriorityBinding>
      <Binding Path="ProcessSomeData" Source="{StaticResource data1}"
          IsAsync="True" />
      <Binding Path="Info2" Source="{StaticResource info}"
          IsAsync="True" />
      <Binding Path="Info1" Source="{StaticResource info}"
          IsAsync="False" />
    </PriorityBinding>
  </Label.Content>
</Label>
```

When the application starts, you can see the message "please wait…" in the user interface. After a few seconds the result from the Info2 property is returned as "please wait a little more." It replaces the output from Info1. Finally, the result from ProcessSomeData replaces the output again.

Value Conversion

Returning to the BooksDemo application, the authors of the book are still missing in the user interface. If you bind the Authors property to a Label element, the ToString method of the Array class is invoked, which returns the name of the type. One solution to this is to bind the Authors property to a ListBox. For the ListBox, you can define a template for a specific view. Another solution is to convert the string array returned by the Authors property to a string and use the string for binding.

The class StringArrayConverter converts a string array to a string. WPF converter classes must implement the interface IValueConverter from the namespace System.Windows.Data. This interface defines the methods Convert and ConvertBack. With the StringArrayConverter, the Convert method converts the string array from the variable value to a string by using the String.Join method. The separator parameter of the Join is taken from the variable parameter received with the Convert method (code file BooksDemo/Utilities/StringArrayConverter.cs):

```
using System;
using System.Diagnostics.Contracts;
using System.Globalization;
using System.Windows.Data;

namespace Wrox.ProCSharp.WPF.Utilities

{
  [ValueConversion(typeof(string[]), typeof(string))]
  class StringArrayConverter : IValueConverter
```

```
  {
    public object Convert(object value, Type targetType, object parameter,
                          CultureInfo culture)
    {
      if (value == null) return null;

      string[] stringCollection = (string[])value;
      string separator = parameter == null;

      return String.Join(separator, stringCollection);
    }

    public object ConvertBack(object value, Type targetType, object parameter,
                              CultureInfo culture)
    {
      throw new NotImplementedException();
    }
  }
}
```

> **NOTE** *You can read more about the methods of the* `String` *classes in Chapter 9, "Strings and Regular Expressions."*

In the XAML code, the `StringArrayConverter` class can be declared as a resource.This resource can be referenced from the `Binding` markup extension (XAML file `BooksDemo/BooksUC.xaml`):

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
    xmlns:utils="clr-namespace:Wrox.ProCSharp.WPF.Utilities"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <UserControl.Resources>
    <utils:StringArrayConverter x:Key="stringArrayConverter" />
    <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
                        MethodName="GetBooks" />
  </UserControl.Resources>
  <!-- -->
```

For multiline output, a `TextBlock` element is declared with the `TextWrapping` property set to `Wrap` to make it possible to display multiple authors. In the `Binding` markup extension, the `Path` is set to `Authors`, which is defined as a property returning a string array. The string array is converted from the resource `stringArrayConverter` as defined by the `Converter` property. The `Convert` method of the converter implementation receives the `ConverterParameter=', '` as input to separate the authors:

```
    <TextBlock Text="{Binding Authors,
               Converter={StaticResource stringArrayConverter},
               ConverterParameter=', '}"
               Grid.Row="3" Grid.Column="1" Margin="5"
                     VerticalAlignment="Center" TextWrapping="Wrap" />
```

Figure 36-11 shows the book details, including authors.

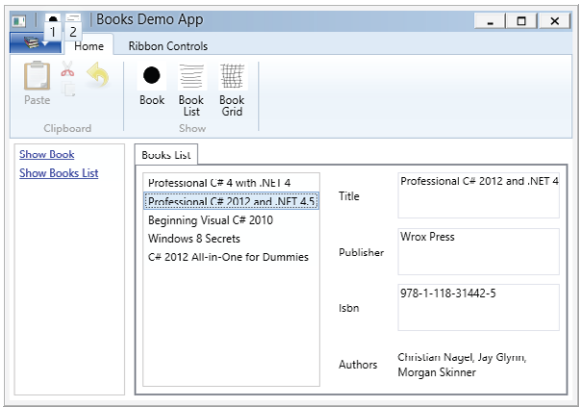


FIGURE 36-11

Adding List Items Dynamically

If list items are added dynamically, the WPF element must be notified of elements added to the list.

In the XAML code of the WPF application, a `Button` element is added inside a `StackPanel`. The `Click` event is assigned to the method `OnAddBook` (XAML file `BooksDemo/BooksUC.xaml`):

```
<StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom"
            HorizontalAlignment="Center">
  <Button Margin="5" Padding="4" Content="Add Book" Click="OnAddBook" />
</StackPanel>
```

In the method `OnAddBook`, a new `Book` object is added to the list. If you test the application with the `BookFactory` as it is implemented now, there's no notification to the WPF elements that a new object has been added to the list (code file `BooksDemo/BooksUC.xaml.cs`):

```
private void OnAddBook(object sender, RoutedEventArgs e)
{
  ((this.FindResource("books") as ObjectDataProvider).Data as IList<Book>).
      Add(new Book("HTML and CSS: Design and Build Websites",
          "Wiley", "978-1118-00818-8"));
}
```

The object that is assigned to the `DataContext` must implement the interface `INotifyCollectionChanged`. This interface defines the `CollectionChanged` event that is used by the WPF application. Instead of implementing this interface on your own with a custom collection class, you can use the generic collection class `ObservableCollection<T>` that is defined with the namespace `System.Collections.ObjectModel` in the assembly `WindowsBase`. Now, as a new item is added to the collection, the new item immediately appears in the `ListBox` (code file `BooksDemo/Data/BookFactory.cs`):

```
public class BookFactory
{
  private ObservableCollection<Book> books = new ObservableCollection<Book>();
  // ...


  public IEnumerable<Book> GetBooks()
  {
    return books;
  }
}
```

Adding Tab Items Dynamically

Adding items dynamically to a list is in principle the same scenario as adding user controls to the tab control dynamically. Until now, the tab items have been added dynamically using the `Add` method of the `Items` property from the `TabControl` class. In the following example, the `TabControl` is directly referenced from code-behind. Using data binding instead, information about the tab item can be added to an `ObservableCollection<T>`.

The code from the BookSample application is now changed to use data binding with the `TabControl`. First, the class `UIControlInfo` is defined. This class contains properties that are used with data binding within the `TabControl`. The `Title` property is used to show heading information within tab items, and the `Content` property is used for the content of the tab items:

```
using System.Windows.Controls;

namespace Wrox.ProCSharp.WPF
{
  public class UIControlInfo
  {
    public string Title { get; set; }
    public UserControl Content { get; set; }
  }
}
```

Now an observable collection is needed to allow the tab control to refresh the information of its tab items. `userControls` is a member variable of the `MainWindow` class. The property `Controls`—used for data binding—returns the collection (code file `BooksDemo/MainWindow.xaml.cs`):

```
private ObservableCollection<UIControlInfo> userControls =
    new ObservableCollection<UIControlInfo>();
public IEnumerable<UIControlInfo> Controls
{
  get { return userControls; }
}
```

With the XAML code the `TabControl` is changed. The `ItemsSource` property is bound to the `Controls` property. Now, two templates need to be specified. One template, `ItemTemplate`, defines the heading of the item controls. The `DataTemplate` specified with the `ItemTemplate` just uses a `TextBlock` element to display the value from the `Text` property in the heading of the tab item. The other template is `ContentTemplate`. This template specifies using the `ContentPresenter` that binds to the `Content` property of the bound items (XAML file `BooksDemo/MainWindow.xaml`):

```
<TabControl Margin="5" x:Name="tabControl1" ItemsSource="{Binding Controls}">
  <TabControl.ContentTemplate>
    <DataTemplate>
      <ContentPresenter Content="{Binding Content}" />
    </DataTemplate>
  </TabControl.ContentTemplate>
  <TabControl.ItemTemplate>
    <DataTemplate>
      <StackPanel Margin="0">
        <TextBlock Text="{Binding Title}" Margin="0" />
      </StackPanel>
    </DataTemplate>
  </TabControl.ItemTemplate>
</TabControl>
```

Now the event handlers can be modified to create new `UIControlInfo` objects and add them to the observable collection instead of creating `TabItem` controls. Changing the item and content templates is a much easier way to customize the look, instead of doing this with code-behind.

```
private void OnShowBooksList(object sender, ExecutedRoutedEventArgs e)
{
  var booksUI = new BooksUC();
  userControls.Add(new UIControlInfo
  {
    Title = "Books List",
    Content = booksUI
  });
}
```

Data Template Selector

The previous chapter described how you can customize controls with templates. You also saw how to create a data template that defines a display for specific data types. A *data template selector* can create different data templates dynamically for the same data type. It is implemented in a class that derives from the base class `DataTemplateSelector`.

The following example implements a data template selector by selecting a different template based on the publisher. These templates are defined within the user control resources. One template can be accessed by the key name `wroxTemplate`; the other template has the key name `dummiesTemplate`, and the third one is `bookTemplate` (XAML file `BooksDemo/BooksUC.xaml`):

```
<DataTemplate x:Key="wroxTemplate" DataType="{x:Type local:Book}">
  <Border Background="Red" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>

<DataTemplate x:Key="dummiesTemplate" DataType="{x:Type local:Book}">
  <Border Background="Yellow" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>

<DataTemplate x:Key="bookTemplate" DataType="{x:Type local:Book}">
  <Border Background="LightBlue" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>
```

For selecting the template, the class `BookDataTemplateSelector` overrides the method `SelectTemplate` from the base class `DataTemplateSelector`. The implementation selects the template based on the `Publisher` property from the `Book` class (code file `BooksDemo/Utilities/BookTemplateSelector.cs`):

```
using System.Windows;
using System.Windows.Controls;
using Wrox.ProCSharp.WPF.Data;

namespace Wrox.ProCSharp.WPF.Utilities
{
  public class BookTemplateSelector : DataTemplateSelector
  {
    public override DataTemplate SelectTemplate(object item,
```

```
            DependencyObject container)
        {
          if (item != null && item is Book)
          {
            var book = item as Book;
            switch (book.Publisher)
            {
              case "Wrox Press":
                return (container as FrameworkElement).FindResource(
                    "wroxTemplate") as DataTemplate;
              case "For Dummies":
                return (container as FrameworkElement).FindResource(
                    "dummiesTemplate") as DataTemplate;
              default:
                return (container as FrameworkElement).FindResource(
                    "bookTemplate") as DataTemplate;
            }
          }
          return null;
        }
      }
    }
```

For accessing the class `BookDataTemplateSelector` from XAML code, the class is defined within the `Window` resources (XAML file `BooksDemo/BooksUC.xaml`):

```
<src:BookDataTemplateSelector x:Key="bookTemplateSelector" />
```

Now the selector class can be assigned to the `ItemTemplateSelector` property of the `ListBox`:

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
         MinWidth="120" IsSynchronizedWithCurrentItem="True"
         ItemTemplateSelector="{StaticResource bookTemplateSelector}">
```

Running the application, you can see different data templates based on the publisher, as shown in Figure 36-12.

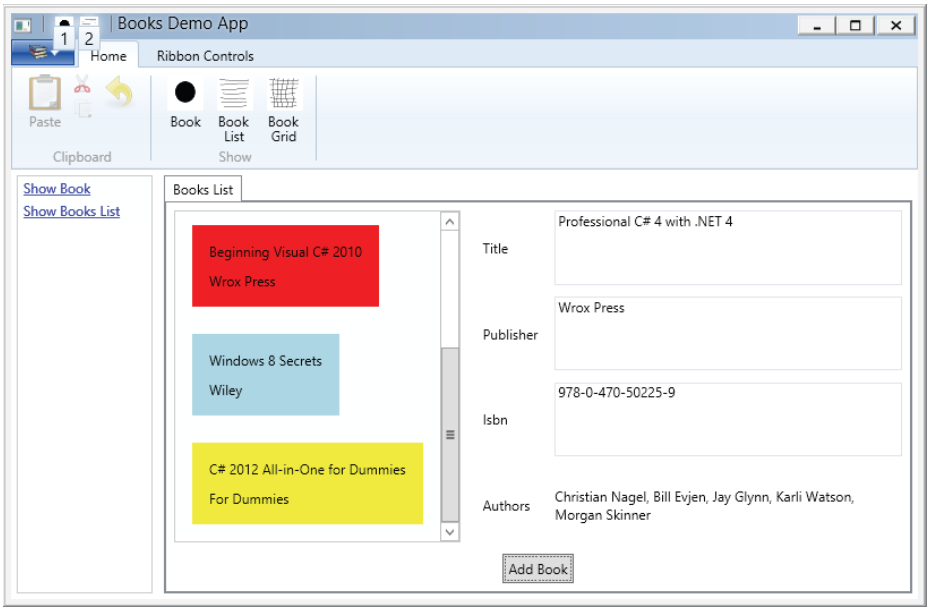


FIGURE 36-12

Binding to XML

WPF data binding has special support for binding to XML data. You can use `XmlDataProvider` as a data source and bind the elements by using XPath expressions. For a hierarchical display, you can use the `TreeView` control and create the view for the items by using the `HierarchicalDataTemplate`.

The following XML file containing `Book` elements is used as a source in the next examples (XML file `XmlBindingDemo/Books.xml`):

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book isbn="978-1-118-31442-5">
    <Title>Professional C# 2012</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Christian Nagel</Author>
    <Author>Jay Glynn</Author>
    <Author>Morgan Skinner</Author>
  </Book>
  <Book isbn="978-0-470-50226-6">
    <Title>Beginning Visual C# 2010</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Karli Watson</Author>
    <Author>Christian Nagel</Author>
    <Author>Jacob Hammer Pedersen</Author>
    <Author>John D. Reid</Author>
    <Author>Morgan Skinner</Author>
  </Book>
</Books>
```

Similarly to defining an object data provider, you can define an XML data provider. Both `ObjectDataProvider` and `XmlDataProvider` are derived from the same base class, `DataSourceProvider`. With the `XmlDataProvider` in the example, the `Source` property is set to reference the XML file `books.xml`. The `XPath` property defines an XPath expression to reference the XML root element `Books`. The `Grid` element references the XML data source with the `DataContext` property. With the data context for the grid, all `Book` elements are required for a list binding, so the XPath expression is set to `Book`. Inside the grid, you can find the `ListBox` element that binds to the default data context and uses the `DataTemplate` to include the title in `TextBlock` elements as items of the `ListBox`. You can also see three `Label` elements with data binding set to XPath expressions to display the title, publisher, and ISBN numbers:

```
<Window x:Class="XmlBindingDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Main Window" Height="240" Width="500">
  <Window.Resources>
    <XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />
    <DataTemplate x:Key="listTemplate">
      <TextBlock Text="{Binding XPath=Title}" />
    </DataTemplate>

    <Style x:Key="labelStyle" TargetType="{x:Type Label}">
      <Setter Property="Width" Value="190" />
      <Setter Property="Height" Value="40" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Window.Resources>

  <Grid DataContext="{Binding Source={StaticResource books}, XPath=Book}">
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
```

```
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <ListBox IsSynchronizedWithCurrentItem="True" Margin="5"
        Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
        ItemTemplate="{StaticResource listTemplate}" />

      <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Title}"
          Grid.Row="0" Grid.Column="1" />
      <Label Style="{StaticResource labelStyle}"
          Content="{Binding XPath=Publisher}" Grid.Row="1" Grid.Column="1" />
      <Label Style="{StaticResource labelStyle}"
          Content="{Binding XPath=@isbn}" Grid.Row="2" Grid.Column="1" />
    </Grid>
</Window>
```

Figure 36-13 shows the result of the XML binding.

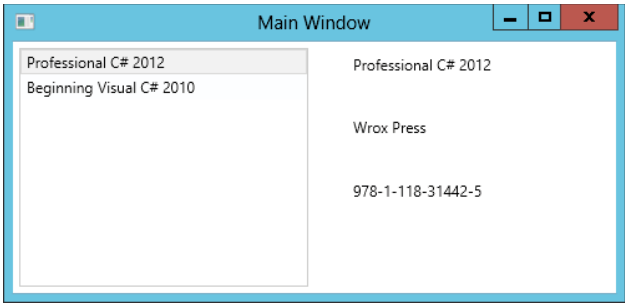


FIGURE 36-13

> **NOTE** *If XML data should be shown hierarchically, you can use the* `TreeView` *control.*

Binding Validation and Error Handling

Several options are available to validate data from the user before it is used with the .NET objects:

- Handling exceptions
- Handling data error information errors
- Handling notify data error information errors
- Defining custom validation rules

Handling Exceptions

The first option demonstrated here reflects the fact that the .NET class throws an exception if an invalid value is set, as shown in the class `SomeData`. The property `Value1` accepts values only larger than or equal to 5 and smaller than 12 (code file `ValidationDemo/SomeData.cs`):

```
public class SomeData
{
  private int value1;
  public int Value1 {
    get { return value1;  }
    set
```

```
    {
      if (value < 5 || value > 12)
      {
        throw new ArgumentException(
            "value must not be less than 5 or greater than 12");
      }
      value1 = value;
    }
  }
}
```

In the constructor of the `MainWindow` class, a new object of the class `SomeData` is initialized and passed to the `DataContext` for data binding (code file `ValidationDemo/MainWindow.xaml.cs`):

```
public partial class MainWindow: Window
{
  private SomeData p1 = new SomeData { Value1 = 11 };

  public MainWindow()
  {
    InitializeComponent();
    this.DataContext = p1;

  }
```

The event handler method `OnShowValue` displays a message box to show the actual value of the `SomeData` instance:

```
  private void OnShowValue(object sender, RoutedEventArgs e)
  {
    MessageBox.Show(p1.Value1.ToString());
  }
}
```

With simple data binding, the following shows the `Text` property of a `TextBox` bound to the `Value1` property. If you run the application now and try to change the value to an invalid one, you can verify that the value never changed by clicking the Submit button. WPF catches and ignores the exception thrown by the set accessor of the property `Value1` (XAML file `ValidationDemo/MainWindow.xaml`):

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1}" />
```

To display an error as soon as the context of the input field changes, you can set the `ValidatesOnException` property of the `Binding` markup extension to `True`. With an invalid value (as soon as the exception is thrown when the value should be set), the `TextBox` is surrounded by a red line. The application showing the error rectangle is shown in Figure 36-14.

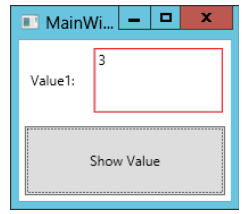


FIGURE 36-14

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}" />
```

To return the error information in a different way to the user, you can assign the attached property `ErrorTemplate` that is defined by the `Validation` class to a template defining the UI for errors. The new template to mark the error is shown as follows with the key `validationTemplate`. The `ControlTemplate` puts a red exclamation point in front of the existing control content:

```
<ControlTemplate x:Key="validationTemplate">
  <DockPanel>
    <TextBlock Foreground="Red" FontSize="40">!</TextBlock>
    <AdornedElementPlaceholder/>
  </DockPanel>
</ControlTemplate>
```

Setting the `validationTemplate` with the `Validation.ErrorTemplate` attached property activates the template with the `TextBox`:

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}"
    Validation.ErrorTemplate="{StaticResource validationTemplate}" />
```

The new look of the application is shown in Figure 36-15.

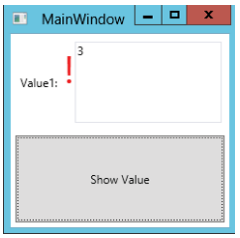


FIGURE 36-15

> **NOTE** *Another option for a custom error message is to register to the* `Error` *event of the* `Validation` *class. In this case, the property* `NotifyOnValidationError` *must be set to* `true`.

The error information itself can be accessed from the `Errors` collection of the `Validation` class. To display the error information in the `ToolTip` of the `TextBox` you can create a property trigger as shown next. The trigger is activated as soon as the `HasError` property of the `Validation` class is set to `True`. The trigger sets the `ToolTip` property of the `TextBox`:

```
<Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="ToolTip"
          Value="{Binding RelativeSource={x:Static RelativeSource.Self},
          Path=(Validation.Errors)[0].ErrorContent}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Data Error Information

Another way to deal with errors is when the .NET object implements the interface `IDataErrorInfo`. The class `SomeData` is now changed to implement this interface, which defines the property `Error` and an indexer with a string argument. With WPF validation during data binding, the indexer is called and the name of the property to validate is passed as the `columnName` argument. With the implementation, the value is verified as valid; if it isn't, an error string is passed. Here, the validation is done on the property `Value2`, which is implemented by using the C# automatic property notation (code file `ValiationDemo/SomeData.cs`):

```
public class SomeData: IDataErrorInfo
{
  //...

  public int Value2 { get; set; }

  string IDataErrorInfo.Error
  {
    get
    {
      return null;
    }
  }

  string IDataErrorInfo.this[string columnName]
```

```
    {
      get
      {
        if (columnName == "Value2")
        {
          if (this.Value2 < 0 || this.Value2 > 80)
              return "age must not be less than 0 or greater than 80";

        }
        return null;
      }
    }
}
```

> **NOTE** *With a .NET object, it would not be clear what an indexer would return; for example, what would you expect from an object of type* `Person` *calling an indexer? That's why it is best to do an explicit implementation of the interface* `IDataErrorInfo`*. This way, the indexer can be accessed only by using the interface, and the .NET class could use a different implementation for other purposes.*

If you set the property `ValidatesOnDataErrors` of the `Binding` class to `true`, the interface `IDataErrorInfo` is used during binding. In the following code, when the `TextBox` is changed the binding mechanism invokes the indexer of the interface and passes `Value2` to the `columnName` variable (XAML file `ValidationDemo/MainWindow.xaml`):

```
<Label Margin="5" Grid.Row="1" Grid.Column="0" >Value2:</Label>
<TextBox Margin="5" Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=Value2, ValidatesOnDataErrors=True}" />
```

Notify Data Error Info

Besides supporting validation with exceptions and the `IDataErrorInfo` interface, WPF with .NET 4.5 supports validation with the interface `INotifyDataErrorInfo` as well. Unlike the interface `IDataErrorInfo`, whereby the indexer to a property can return one error, with `INotifyDataErrorInfo` multiple errors can be associated with a single property. These errors can be accessed using the `GetErrors` method. The `HasErrors` property returns true if the entity has any error. Another great feature of this interface is the notification of errors with the event `ErrorsChanged`. This way, errors can be retrieved asynchronously on the client—for example, a Web service can be invoked to verify the input from the user. In this case, the user can continue working with the input form while the result is retrieved, and can be informed asynchronously about any mismatch.

Let's get into an example in which validation is done using `INotifyDataErrorInfo`. The base class `NotifyDataErrorInfoBase` is defined, which implements the interface `INotifyDataErrorInfo`. This class derives from the base class `BindableObject` to get an implementation for the interface `INotifyPropertyChanged` that you've seen earlier in this chapter. `NotifyDataErrorInfoBase` uses a dictionary named `errors` that contains a list for every property to store error information. The property `HasErrors` returns `true` if any property has an error; the method `GetErrors` returns the error list for a single property; and the event `ErrorsChanged` is fired every time error information is changed. In addition to the members of the interface `INotifyDataErrorInfo`, the base class implements the methods `SetError`, `ClearErrors`, and `ClearAllErrors` to make it easier to deal with setting errors (code file `ValidationDemo/NotifyDataErrorInfoBase.cs`):

```
using System;
using System.Collections;
using System.Collections.Generic;
```

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace ValidationDemo
{
  public abstract class NotifyDataErrorInfoBase : BindableObject,
      INotifyDataErrorInfo
  {
    public void SetError(string errorMessage,
        [CallerMemberName] string propertyName = null)
    {
      List<string> errorList;
      if (errors.TryGetValue(propertyName, out errorList))
      {
        errorList.Add(errorMessage);
      }
      else
      {
        errorList = new List<string> { errorMessage };
        errors.Add(propertyName, errorList);
      }
      HasErrors = true;
      OnErrorsChanged(propertyName);
    }

    public void ClearErrors([CallerMemberName] string propertyName = null)
    {
      if (hasErrors)
      {
        List<string> errorList;
        if (errors.TryGetValue(propertyName, out errorList))
        {
          errors.Remove(propertyName);
        }
        if (errors.Count == 0)
        {
          HasErrors = false;
        }
        OnErrorsChanged(propertyName);
      }
    }

    public void ClearAllErrors()
    {
      if (HasErrors)
      {
        errors.Clear();
        HasErrors = false;
        OnErrorsChanged(null);
      }
    }

    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

    private Dictionary<string, List<string>> errors =
        new Dictionary<string, List<string>>();
    public IEnumerable GetErrors(string propertyName)
    {
      List<string> errorsForProperty;
      bool err = errors.TryGetValue(propertyName, out errorsForProperty);
      if (!err) return null;
```

```
      return errorsForProperty;
    }

    private bool hasErrors = false;
    public bool HasErrors
    {
      get { return hasErrors; }
      protected set {
        if (SetProperty(ref hasErrors, value))
        {
          OnErrorsChanged(propertyName: null);
        }
      }
    }

    protected void OnErrorsChanged([CallerMemberName] string propertyName = null)
    {
      var errorsChanged = ErrorsChanged;
      if (errorsChanged != null)
      {
        errorsChanged(this, new DataErrorsChangedEventArgs(propertyName));
      }
    }
  }
}
```

The class `SomeDataWithNotifications` is the data object that is bound to the XAML code. This class derives from the base class `NotifyDataErrorInfoBase` to inherit the implementation of the interface `INotifyDataErrorInfo`. The property `Val1` is validated asynchronously. For the validation, the method `CheckVal1` is invoked after the property is set. This method makes an asynchronous call to the method `ValidationSimulator.Validate`. After invoking the method, the UI thread can return to handle other events; and as soon as the result is returned, the `SetError` method of the base class is invoked if an error was returned. You can easily change the async invocation to call a Web service or perform another async activity (code file `ValidationDemo/SomeDataWithNotifications.cs`):

```
using System.Runtime.CompilerServices;
using System.Threading.Tasks;

namespace ValidationDemo
{
  public class SomeDataWithNotifications : NotifyDataErrorInfoBase
  {
    private int val1;
    public int Val1
    {
      get { return val1; }
      set
      {
        SetProperty(ref val1, value);
        CheckVal1(val1, value);
      }
    }

    private async void CheckVal1(int oldValue, int newValue,
        [CallerMemberName] string propertyName = null)
    {
      ClearErrors(propertyName);

      string result = await ValidationSimulator.Validate(newValue, propertyName);
      if (result != null)
```

```
      {
        SetError(result, propertyName);
      }
    }
  }
```

The `Validate` method of the `ValidationSimilator` has a delay of three seconds before checking the value, and returns an error message if the value is larger than 50:

```
public static class ValidationSimulator
{
  public static Task<string> Validate(int val,
      [CallerMemberName] string propertyName = null)
  {
    return Task<string>.Run(async () =>
      {
        await Task.Delay(3000);
        if (val > 50) return "bad value";
        else return null;
      });
  }
}
```

With data binding, just the `ValidatesOnNotifyDataErrors` property must be set to `True` to make use of the async validation of the interface `INotifyDataErrorInfo` (XAML file `ValidationDemo/NotificationWindow.xaml`):

```
<TextBox Grid.Row="0" Grid.Column="1"
    Text="{Binding Val1, ValidatesOnNotifyDataErrors=True}" Margin="8" />
```

Running the application, you can see the text box surrounded by the default red rectangle three seconds after wrong input was entered. Showing error information in a different way can be done in the same way you've seen it before—with error templates and triggers accessing validation errors.

Custom Validation Rules

To get more control of the validation you can implement a custom validation rule. A class implementing a custom validation rule needs to derive from the base class `ValidationRule`. In the previous two examples, validation rules have been used as well. Two classes that derive from the abstract base class `ValidationRule` are `DataErrorValidationRule` and `ExceptionValidationRule`. `DataErrorValidationRule` is activated by setting the property `ValidatesOnDataErrors` and uses the interface `IDataErrorInfo`; `ExceptionValidationRule` deals with exceptions and is activated by setting the property `ValidatesOnException`.

In the following example, a validation rule is implemented to verify a regular expression. The class `RegularExpressionValidationRule` derives from the base class `ValidationRule` and overrides the abstract method `Validate` that is defined by the base class. With the implementation, the `RegEx` class from the namespace `System.Text.RegularExpressions` is used to validate the expression defined by the `Expression` property:

```
public class RegularExpressionValidationRule : ValidationRule
{
  public string Expression { get; set; }
  public string ErrorMessage { get; set; }

  public override ValidationResult Validate(object value,
      CultureInfo cultureInfo)
  {
    ValidationResult result = null;
    if (value != null)
    {
```

```
            var regEx = new Regex(Expression);
            bool isMatch = regEx.IsMatch(value.ToString());
            result = new ValidationResult(isMatch, isMatch ?
               null: ErrorMessage);
          }
          return result;
        }
    }
```

> **NOTE** *Regular expressions are explained in Chapter 9, "Strings and Regular Expressions."*

Instead of using the `Binding` markup extension, now the binding is done as a child of the `TextBox.Text` element. The bound object defines an `Email` property that is implemented with the simple property syntax. The `UpdateSourceTrigger` property defines when the source should be updated. Possible options for updating the source are as follows:

- When the property value changes, which is every character typed by the user
- When the focus is lost
- Explicitly

`ValidationRules` is a property of the `Binding` class that contains `ValidationRule` elements. Here, the validation rule used is the custom class `RegularExpressionValidationRule`, where the `Expression` property is set to a regular expression that verifies whether the input is a valid e-mail address; and the `ErrorMessage` property, which outputs the error message if the data entered in the `TextBox` is invalid:

```
<Label Margin="5" Grid.Row="2" Grid.Column="0">Email:</Label>
<TextBox Margin="5" Grid.Row="2" Grid.Column="1">
  <TextBox.Text>
    <Binding Path="Email" UpdateSourceTrigger="LostFocus">
      <Binding.ValidationRules>
        <src:RegularExpressionValidationRule
            Expression="^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.
                       [0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|
                       [0-9]{1,3})(\]?)$"
            ErrorMessage="Email is not valid" />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

TREEVIEW

The `TreeView` control is used to display hierarchical data. Binding to a `TreeView` is very similar to the binding you've seen with the `ListBox`. What's different is the hierarchical data display—a `HierarchicalDataTemplate` can be used.

The next example uses hierarchical displays and the `DataGrid` control. The `Formula1` sample database is accessed with the ADO.NET Entity Framework. The mapping used is shown in Figure 36-16. The `Race` class contains information about the date of the race and is associated with the `Circuit` class. The `Circuit` class has information about the `Country` and the name of the race circuit. `Race` also has an association with `RaceResult`. A `RaceResult` contains information about the `Racer` and the `Team`.

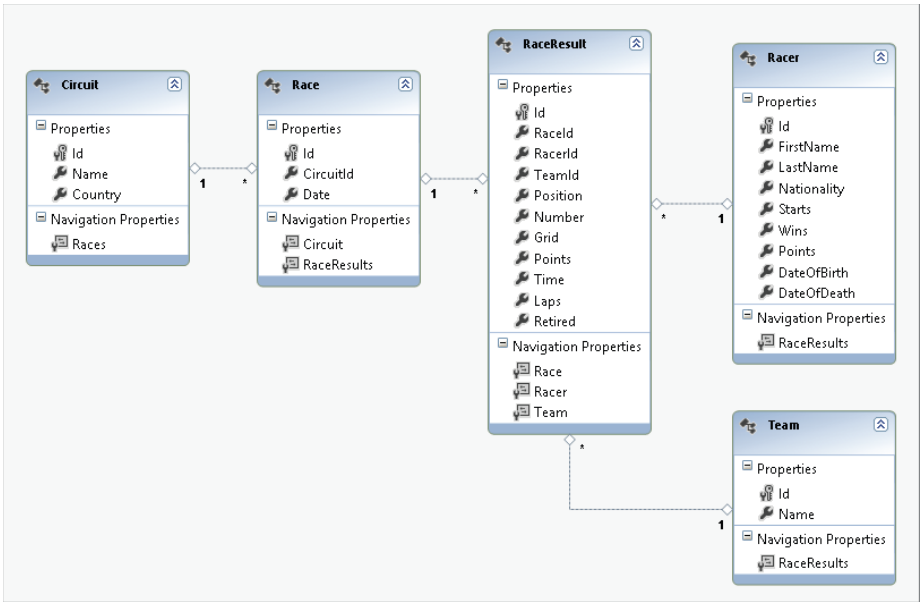


FIGURE 36-16

> **NOTE** *The ADO.NET Entity Framework is covered in Chapter 33, "ADO.NET Entity Framework."*

With the XAML code a `TreeView` is declared. `TreeView` derives from the base class `ItemsControl`, where binding to a list can be done with the `ItemsSource` property. `ItemsSource` is bound to the data context. The data context is assigned in the code-behind, as you will see next. Of course, this could also be done with an `ObjectDataProvider`. To define a custom display for the hierarchical data, `HierarchicalDataTemplate` elements are defined. The data templates here are defined for specific data types with the `DataType` property. The first `HierarchicalDataTemplate` is the template for the `Championship` class and binds the `Year` property of this class to the `Text` property of a `TextBlock`. The `ItemsSource` property defines the binding for the data template itself to specify the next level in the data hierarchy. If the `Races` property of the `Championship` class returns a collection, you bind the `ItemsSource` property directly to `Races`. However, because this property returns a `Lazy<T>` object, binding is done to `Races.Value`. The advantages of the `Lazy<T>` class are discussed later in this chapter.

The second `HierarchicalDataTemplate` element defines the template for the `F1Race` class and binds the `Country` and `Date` properties of this class. With the `Date` property a `StringFormat` is defined with the binding. The next level of the hierarchy is defined binding the `ItemsSource` to `Results.Value`.

The class `F1RaceResult` doesn't have a children collection, so the hierarchy stops here. For this data type, a normal `DataTemplate` is defined to bind the `Position`, `Racer`, and `Car` properties (XAML file `Formula1Demo/TreeUC.xaml`):

```
<UserControl x:Class="Formula1Demo.TreeUC"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
            xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
            xmlns:local="clr-namespace:Formula1Demo"
            mc:Ignorable="d"
            d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
    <TreeView ItemsSource="{Binding}" >
      <TreeView.Resources>
        <HierarchicalDataTemplate DataType="{x:Type local:Championship}"
                                  ItemsSource="{Binding Races.Value}">
          <TextBlock Text="{Binding Year}" />
        </HierarchicalDataTemplate>

        <HierarchicalDataTemplate DataType="{x:Type local:F1Race}"
                                  ItemsSource="{Binding Results.Value}">
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Country}" Margin="5,0,5,0" />
            <TextBlock Text="{Binding Date, StringFormat=d}" Margin="5,0,5,0" />
          </StackPanel>
        </HierarchicalDataTemplate>

        <DataTemplate DataType="{x:Type local:F1RaceResult}">
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Position}" Margin="5,0,5,0" />
            <TextBlock Text="{Binding Racer}" Margin="5,0,0,0" />
            <TextBlock Text=", " />
            <TextBlock Text="{Binding Car}" />
          </StackPanel>
        </DataTemplate>
      </TreeView.Resources>
    </TreeView>
  </Grid>
</UserControl>
```

Now for the code that fills the hierarchical control. In the code-behind file of the XAML code, `DataContext` is assigned to the `Years` property. The `Years` property uses a LINQ query, instead of the ADO.NET Entity Framework data context, to get all the years of the Formula-1 races in the database and to create a new `Championship` object for every year. With the instance of the `Championship` class, the `Year` property is set. This class also has a `Races` property to return the races of the year, but this information is not yet filled in (code file `Formula1Demo/TreeUC.xaml.cs`):

> **NOTE** *LINQ is discussed in Chapter 11, "Language Integrated Query," and Chapter 33.*

```
using System.Collections.Generic;
using System.Linq;
using System.Windows.Controls;

namespace Formula1Demo
{
  public partial class TreeUC : UserControl
  {
    private Formula1Entities data = new Formula1Entities();

    public TreeUC()
    {
      InitializeComponent();
```

```
        this.DataContext = Years;
      }

      public IEnumerable<Championship> Years
      {
        get
        {
          F1DataContext.Data = data;
          return data.Races.Select(r => new Championship
          {
            Year = r.Date.Year
          }).Distinct().OrderBy(c => c.Year);
        }
      }
    }
  }
```

The `Championship` class has a simple automatic property for the year. The `Races` property is of type `Lazy<IEnumerable<F1Race>>`. The `Lazy<T>` class was introduced with .NET 4 for lazy initialization. With a `TreeView` control, this class comes in very handy. If the data behind the tree is large and you do not want to load the full tree in advance, but only when a user makes a selection, lazy loading can be used. With the constructor of the `Lazy<T>` class, a delegate `Func<IEnumerable<F1Race>>` is used. With this delegate, `IEnumerable<F1Race>` needs to be returned. The implementation of the Lambda expression, assigned to the delegate, uses a LINQ query to create a list of `F1Race` objects that have the `Date` and `Country` property assigned (code file `Formula1Demo/Championship.cs`):

```
public class Championship
{
  public int Year { get; set; }
  public Lazy<IEnumerable<F1Race>> Races
  {
    get
    {
      return new Lazy<IEnumerable<F1Race>>(() =>
      {
        return from r in F1DataContext.Data.Races
               where r.Date.Year == Year
               orderby r.Date
               select new F1Race
               {
                 Date = r.Date,
                 Country = r.Circuit.Country
               };
      });
    }
  }
}
```

The `F1Race` class again defines the `Results` property that uses the `Lazy<T>` type to return a list of `F1RaceResult` objects (code file `Formula1Demo/F1Race.cs`):

```
public class F1Race
{
  public string Country { get; set; }
  public DateTime Date { get; set; }
  public Lazy<IEnumerable<F1RaceResult>> Results
  {
    get
    {
      return new Lazy<IEnumerable<F1RaceResult>>(() =>
      {
        return from rr in F1DataContext.Data.RaceResults
```

```
                where rr.Race.Date == this.Date
                select new F1RaceResult
                {
                  rr.Position,
                  Racer = rr.Racer.FirstName + " " + rr.Racer.LastName,
                  Car = rr.Team.Name
                };
        });
      }
    }
  }
```

The final class of the hierarchy is `F1RaceResult`, which is a simple data holder for `Position`, `Racer`, and `Car` (code file `Formula1Demo/Championship.cs`):

```
public class F1RaceResult
{
  public int Position { get; set; }
  public string Racer { get; set; }
  public string Car { get; set; }
}
```

When you run the application, you can see at first all the years of the championships in the tree view. Because of binding, the next level is already accessed—every `Championship` object already has the `F1Race` objects associated. The user doesn't need to wait for the first level after the year or an open year with the default appearance of a small triangle. As shown in Figure 36-17, the year 1984 is open. As soon as the user clicks a year to see the second-level binding, the third level is done and the race results are retrieved.

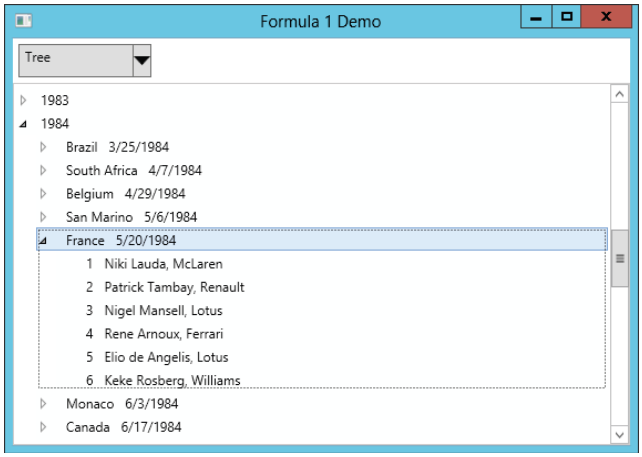


FIGURE 36-17

Of course, you can also customize the `TreeView` control and define different styles for the complete template or the items in the view.

DATAGRID

To display and edit data using rows and columns, the `DataGrid` control can be used. The `DataGrid` control is an `ItemsControl` and defines the `ItemsSource` property that is bound to a collection. The XAML code of this user interface also defines two `RepeatButton` controls that are used for paging functionality. Instead of loading all the race information at once, paging is used so users can step through pages. In a simple scenario,

only the `ItemsSource` property of the `DataGrid` needs to be assigned. By default, the `DataGrid` creates columns based on the properties of the bound data (XAML file `Formula1Demo/GridUC.xaml`):

```
<UserControl x:Class="Formula1Demo.GridUC"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
    <Grid.RowDefinitions>
      <RepeatButton Margin="5" Click="OnPrevious">Previous</RepeatButton>
      <RepeatButton Margin="5" Click="OnNext">Next</RepeatButton>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" Grid.Row="0">
      <Button Click="OnPrevious">Previous</Button>
      <Button Click="OnNext">Next</Button>
    </StackPanel>
    <DataGrid Grid.Row="1" ItemsSource="{Binding}" />
  </Grid>
</UserControl>
```

The code-behind uses the same `Formula1` database as the previous `TreeView` example. The `DataContext` of the `UserControl` is set to the `Races` property. This property returns `IEnumerable<object>`. Instead of assigning a strongly typed enumeration, an `object` is used to make it possible to create an anonymous class with the LINQ query. The LINQ query creates the anonymous class with `Year`, `Country`, `Position`, `Racer`, and `Car` properties and uses a compound to access `Races` and `RaceResults`. It also accesses other associations of `Races` to get country, racer, and team information. With the `Skip` and `Take` methods, paging functionality is implemented. The size of a page is fixed to 50 items, and the current page changes with the `OnNext` and `OnPrevious` handlers (code file `Formula1Demo/GridUC.xaml.cs`):

```
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Controls;

namespace Formula1Demo
{
  public partial class GridUC : UserControl
  {
    private int currentPage = 0;
    private int pageSize = 50;
    private Formula1Entities data = new Formula1Entities();
    public GridUC()
    {
      InitializeComponent();
      this.DataContext = Races;
    }

    public IEnumerable<object> Races
    {
      get
      {
        return (from r in data.Races
                from rr in r.RaceResults
                orderby r.Date ascending
                select new
                {
                  r.Date.Year,
                  r.Circuit.Country,
                  rr.Position,
```

```
                Racer = rr.Racer.FirstName + " " + rr.Racer.LastName,
                Car = rr.Team.Name
              }).Skip(currentPage * pageSize).Take(pageSize);
        }
      }

      private void OnPrevious(object sender, RoutedEventArgs e)
      {
        if (currentPage > 0)
        {
          currentPage--;
          this.DataContext = Races;
        }
      }

      private void OnNext(object sender, RoutedEventArgs e)
      {
        currentPage++;
        this.DataContext = Races;
      }
    }
  }
```

Figure 36-18 shows the running application with the default grid styles and headers.

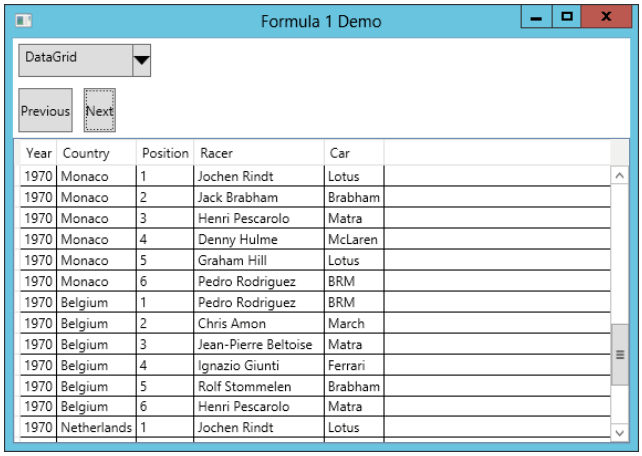


FIGURE 36-18

In the next `DataGrid` example, the grid is customized with custom columns and grouping.

Custom Columns

Setting the property `AutoGenerateColumns` of the `DataGrid` to `False` doesn't generate default columns. You can create custom columns with the `Columns` property. You can also specify elements that derive from `DataGridColumn`. You can use predefined classes, and `DataGridTextColumn` can be used to read and edit text. `DataGridHyperlinkColumn` is for displaying hyperlinks. `DataGridCheckBoxColumn` displays a check box for Boolean data. For a list of items in a column, you can use the `DataGridComboBoxColumn`. More `DataGridColumn` types will be available in the future, but if you need a different representation now, you can use the `DataGridTemplateColumn` to define and bind any elements you want.

The example code uses `DataGridTextColumn` elements that are bound to the `Position` and `Racer` properties. The `Header` property is set to a string for display. Of course, you can also use a template to define a complete custom header for the column (XAML file `Formula1Demo/GridUC.xaml.cs`):

```
<DataGrid ItemsSource="{Binding}" AutoGenerateColumns="False">
  <DataGrid.Columns>
    <DataGridTextColumn Binding="{Binding Position, Mode=OneWay}"
                        Header="Position" />
    <DataGridTextColumn Binding="{Binding Racer, Mode=OneWay}"
                        Header="Racer" />
  </DataGrid.Columns>
```

Row Details

When a row is selected, the `DataGrid` can display additional information for the row. This is done by specifying a `RowDetailsTemplate` with the `DataGrid`. A `DataTemplate` is assigned to the `RowDetailsTemplate`, which contains several `TextBlock` elements that display the car and points (XAML file `Formula1Demo/GridUC.xaml.cs`):

```
<DataGrid.RowDetailsTemplate>
  <DataTemplate>
    <StackPanel Orientation="Horizontal">
      <TextBlock Text="Car:" Margin="5,0,0,0" />
      <TextBlock Text="{Binding Car}" Margin="5,0,0,0" />
      <TextBlock Text="Points:" Margin="5,0,0,0" />
      <TextBlock Text="{Binding Points}" />
    </StackPanel>
  </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

Grouping with the DataGrid

The Formula-1 races have several rows that contain the same information, such as the year and the country. For such data, grouping can be helpful to organize the information for the user.

For grouping, the `CollectionViewSource` can be used in XAML code. It also supports sorting and filtering. With code-behind you can also use the `ListCollectionView` class, which is used only by the `CollectionViewSource`.

`CollectionViewSource` is defined within a `Resources` collection. The source of `CollectionViewSource` is the result from an `ObjectDataProvider`. The `ObjectDataProvider` invokes the `GetRaces` method of the `F1Races` type. This method has two `int` parameters that are assigned from the `MethodParameters` collection. The `CollectionViewSource` uses two descriptions for grouping—first by the `Year` property and then by the `Country` property (XAML file `Formula1Demo/GridGroupingUC.xaml`):

```
<Grid.Resources>
  <ObjectDataProvider x:Key="races" ObjectType="{x:Type local:F1Races}"
                      MethodName="GetRaces">
    <ObjectDataProvider.MethodParameters>
      <sys:Int32>0</sys:Int32>
      <sys:Int32>20</sys:Int32>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
  <CollectionViewSource x:Key="viewSource"
                        Source="{StaticResource races}">
    <CollectionViewSource.GroupDescriptions>
      <PropertyGroupDescription PropertyName="Year" />
      <PropertyGroupDescription PropertyName="Country" />
    </CollectionViewSource.GroupDescriptions>
  </CollectionViewSource>
</Grid.Resources>
```

How the group is displayed is defined with the DataGrid GroupStyle property. With the GroupStyle element you need to customize the ContainerStyle as well as the HeaderTemplate and the complete panel. To dynamically select the GroupStyle and HeaderStyle, you can also write a container style selector and a header template selector. It is very similar in functionality to the data template selector described earlier.

The GroupStyle in the example sets the ContainerStyle property of the GroupStyle. With this style, the GroupItem is customized with a template. The GroupItem appears as the root element of a group when grouping is used. Displayed within the group is the name, using the Name property, and the number of items, using the ItemCount property. The third column of the Grid contains all the normal items using the ItemsPresenter. If the rows are grouped by country, the labels of the Name property would all have a different width, which doesn't look good. Therefore, the SharedSizeGroup property is set with the second column of the grid to ensure all items are the same size. The shared size scope needs to be set for all elements that have the same size. This is done in the DataGrid setting Grid.IsSharedSizeScope="True":

```
<DataGrid.GroupStyle>
  <GroupStyle>
    <GroupStyle.ContainerStyle>
      <Style TargetType="{x:Type GroupItem}">
        <Setter Property="Template">
          <Setter.Value>
            <ControlTemplate >
              <StackPanel Orientation="Horizontal" >
                <Grid>
                  <Grid.ColumnDefinitions>
                    <ColumnDefinition SharedSizeGroup="LeftColumn" />
                    <ColumnDefinition />
                    <ColumnDefinition />
                  </Grid.ColumnDefinitions>
                  <Label Grid.Column="0" Background="Yellow"
                         Content="{Binding Name}" />
                  <Label Grid.Column="1" Content="{Binding ItemCount}" />
                  <Grid Grid.Column="2" HorizontalAlignment="Center"
                        VerticalAlignment="Center">
                    <ItemsPresenter/>
                  </Grid>
                </Grid>
              </StackPanel>
            </ControlTemplate>
          </Setter.Value>
        </Setter>
      </Style>
    </GroupStyle.ContainerStyle>
  </GroupStyle>
</DataGrid.GroupStyle>
```

The class F1Races that is used by the ObjectDataProvider uses LINQ to access the Formula1 database and returns a list of anonymous types with Year, Country, Position, Racer, Car, and Points properties. The Skip and Take methods are used to access part of the data (code file Formula1Demo/F1Races.cs):

```
using System.Collections.Generic;
using System.Linq;

namespace Formula1Demo
{
  public class F1Races
  {
    private int lastpageSearched = -1;
    private IEnumerable<object> cache = null;
    private Formula1Entities data = new Formula1Entities();

    public IEnumerable<object> GetRaces(int page, int pageSize)
```

```
    {
      if (lastpageSearched == page)
        return cache;
      lastpageSearched = page;

      var q = (from r in data.Races
               from rr in r.RaceResults
               orderby r.Date ascending
               select new
               {
                 Year = r.Date.Year,
                 Country = r.Circuit.Country,
                 Position = rr.Position,
                 Racer = rr.Racer.Firstname + " " + rr.Racer.Lastname,
                 Car = rr.Team.Name,
                 Points = rr.Points
               }).Skip(page * pageSize).Take(pageSize);
      cache = q;
      return cache;
    }
  }
}
```

Now all that's left is for the user to set the page number and change the parameter of the `ObjectDataProvider`. In the user interface, a `TextBox` and a `Button` are defined (XAML file `Formula1Demo/GridGroupingUC.xaml`):

```
<StackPanel Orientation="Horizontal" Grid.Row="0">
  <TextBlock Margin="5" Padding="4" VerticalAlignment="Center">
    Page:
  </TextBlock>
  <TextBox Margin="5" Padding="4" VerticalAlignment="Center"
     x:Name="textPageNumber" Text="0" />
  <Button Click="OnGetPage">Get Page</Button>
</StackPanel>
```

The `OnGetPage` handler of the button in the code-behind accesses the `ObjectDataProvider` and changes the first parameter of the method. It then invokes the `Refresh` method so the `ObjectDataProvider` requests the new page (code file `Formula1Demo/GridGroupingUC.xaml.cs`):

```
private void OnGetPage(object sender, RoutedEventArgs e)
{
  int page = int.Parse(textPageNumber.Text);
  var odp = (sender as FrameworkElement).FindResource("races")
            as ObjectDataProvider;
  odp.MethodParameters[0] = page;
  odp.Refresh();
}
```

Running the application, you can see grouping and row detail information, as shown in Figure 36-19.

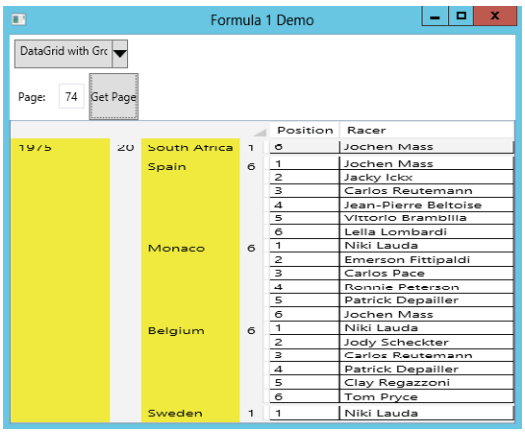


FIGURE 36-19

Live Shaping

A new feature with WPF 4.5 is *live shaping*. You've seen the collection view source with its support for sorting, filtering, and grouping. However, if the collection changes over time in that sorting, filtering, or grouping returns different results, the `CollectionViewSource` didn't help—until now. For live shaping, a new interface, `ICollectionViewLiveShaping`, is

used. This interface defines the properties `CanChangeLiveFiltering`, `CanChangeLiveGrouping`, and `CanChangeLiveSorting` to check the data source if these live shaping features are available. The properties `IsLiveFiltering`, `IsLiveGrouping`, and `IsLiveSorting` enable turning on the live shaping features—if available. With `LiveFilteringProperties`, `LiveGroupingProperties`, and `LiveSortingProperties`, you can define the properties of the source that should be used for live filtering, grouping, and sorting.

The sample application shows how the results of a Formula 1 race—this time the race from Barcelona in 2012—change lap by lap.

A racer is represented by the `Racer` class. This type has the simple properties `Name`, `Team`, and `Number`. These properties are implemented using auto properties, as the values of this type don't change when the application is run (code file `LiveShaping/Racer.cs`):

```
public class Racer
{
  public string Name { get; set; }
  public string Team { get; set; }
  public int Number { get; set; }

  public override string ToString()
  {
    return Name;
  }
}
```

The class `Formula1` returns a list of all racers who competed at the Barcelona race 2012 (code file `LiveShaping/Formula1.cs`):

```
public class Formula1
{
  private List<Racer> racers;
  public IEnumerable<Racer> Racers
  {
    get
    {
      return racers ?? (racers = GetRacers());
    }
  }

  private List<Racer> GetRacers()
  {
    return new List<Racer>()
    {
      new Racer { Name="Sebastian Vettel", Team="Red Bull Racing", Number=1 },
      new Racer { Name="Mark Webber", Team="Red Bull Racing", Number=2 },
      new Racer { Name="Jenson Button", Team="McLaren", Number=3 },
      new Racer { Name="Lewis Hamilton", Team="McLaren", Number=4 },
      new Racer { Name="Fernando Alonso", Team="Ferrari", Number=5 },
      new Racer { Name="Felipe Massa", Team="Ferrari", Number=6 },
      new Racer { Name="Michael Schumacher", Team="Mercedes", Number=7 },
      new Racer { Name="Nico Rosberg", Team="Mercedes", Number=8 },
      new Racer { Name="Kimi Raikkonen", Team="Lotus", Number=9 },
      new Racer { Name="Romain Grosjean", Team="Lotus", Number=10 },
      new Racer { Name="Paul di Resta", Team="Force India", Number=11 },
      new Racer { Name="Nico Hülkenberg", Team="Force India", Number=12 },
      new Racer { Name="Kamui Kobayashi", Team="Sauber", Number=14 },
      new Racer { Name="Sergio Perez", Team="Sauber", Number=15 },
      new Racer { Name="Daniel Riccardio", Team="Toro Rosso", Number=16 },
      new Racer { Name="Jean-Eric Vergne", Team="Toro Rosso", Number=17 },
```

```
            new Racer { Name="Pastor Maldonado", Team="Williams", Number=18 },

            //... more racers in the source code download
          };
        }
      }
```

Now it gets more interesting. The `LapRacerInfo` class is the type that is shown in the `DataGrid` control. The class derives from the base class `BindableObject` to get an implementation of `INotifyPropertyChanged` as you've seen earlier. The properties `Lap`, `Position`, and `PositionChange` change over time. `Lap` gives the current lap number, `Position` gives the position in the race in the specified lap, and `PositionChange` provides information about how the position changed from the previous lap. If the position did not change, the state is `None`; if the position is lower than in the previous lap, it is `Up`; if it is higher, then it is `Down`; and if the racer is out of the race, the `PositionChange` is `Out`. This information can be used within the UI for a different representation (code file `LiveShaping/LapRacerInfo.cs`):

```
public enum PositionChange
{
  None,
  Up,
  Down,
  Out
}

public class LapRacerInfo : BindableObject
{
  public Racer Racer { get; set; }
  private int lap;
  public int Lap
  {
    get { return lap; }
    set { SetProperty(ref lap, value); }
  }
  private int position;
  public int Position
  {
    get { return position; }
    set { SetProperty(ref position, value); }
  }
  private PositionChange positionChange;
  public PositionChange PositionChange
  {
    get { return positionChange; }
    set { SetProperty(ref positionChange, value); }
  }
}
```

The class `LapChart` contains all the information about all laps and racers. This class could be changed to access a live Web service to retrieve this information, and then the application could show the current live results from an active race.

The method `SetLapInfoForStart` creates the initial list of `LapRacerInfo` items and fills the position to the grid position. The grid position is the first number of the `List<int>` collection that is added to the `positions` dictionary. Then, with every invocation of the `NextLap` method, the items inside the `lapInfo` collection change to a new position and set the `PositionChange` state information (code file `LiveShaping/LapChart.cs`):

```
public class LapChart
{
  private Formula1 f1 = new Formula1();
  private List<LapRacerInfo> lapInfo;
  private int currentLap = 0;
```

```
private const int PostionOut = 999;
private int maxLaps;
public LapChart()
{
  FillPositions();
  SetLapInfoForStart();
}

private Dictionary<int, List<int>> positions =
    new Dictionary<int, List<int>>();
private void FillPositions()
{
  positions.Add(18, new List<int> { 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2,
      2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 3, 3, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1 });
  positions.Add(5, new List<int> { 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2,
      2, 2, 2, 2, 1, 1, 1, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
      2, 2, 2, 2, 2 });
  positions.Add(10, new List<int> { 3, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 9, 7, 6,
      6, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4,
      4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
      4, 4, 4, 4, 4 });
  // more position information with the code download

  maxLaps = positions.Select(p => p.Value.Count).Max() - 1;
}

private void SetLapInfoForStart()
{
  lapInfo = positions.Select(x => new LapRacerInfo
  {
    Racer = f1.Racers.Where(r => r.Number == x.Key).Single(),
    Lap = 0,
    Position = x.Value.First(),
    PositionChange = PositionChange.None
  }).ToList();
}

public IEnumerable<LapRacerInfo> GetLapInfo()
{
  return lapInfo;
}

public bool NextLap()
{
  currentLap++;
  if (currentLap > maxLaps) return false;

  foreach (var info in lapInfo)
  {
    int lastPosition = info.Position;
    var racerInfo = positions.Where(x => x.Key == info.Racer.Number).Single();

    if (racerInfo.Value.Count > currentLap)
    {
      info.Position = racerInfo.Value[currentLap];
    }
    else
    {
      info.Position = lastPosition;
```

```
            }
            info.PositionChange = GetPositionChange(lastPosition, info.Position);

            info.Lap = currentLap;
          }
          return true;
        }

        private PositionChange GetPositionChange(int oldPosition, int newPosition)
        {
          if (oldPosition == PositionOut ||| newPosition == PositionOut)
            return PositionChange.Out;
          else if (oldPosition == newPosition)
            return PositionChange.None;
          else if (oldPosition < newPosition)
            return PositionChange.Down;
          else
            return PositionChange.Up;
        }
      }
```

In the main window, the `DataGrid` is specified and contains some `DataGridTextColumn` elements that are bound to properties of the `LapRacerInfo` class that is returned from the collection shown previously. `DataTrigger` elements are used to define a different background color for the row depending on whether the racer has a better or worse position compared to the previous lap by using the enumeration value from the `PositionChange` property (XAML file `LiveShaping/MainWindow.xaml`):

```
<DataGrid IsReadOnly="True" ItemsSource="{Binding}"
    DataContext="{StaticResource cvs}" AutoGenerateColumns="False">
  <DataGrid.CellStyle>
    <Style TargetType="DataGridCell">
      <Style.Triggers>
        <Trigger Property="IsSelected" Value="True">
          <Setter Property="Background" Value="{x:Null}" />
          <Setter Property="BorderBrush" Value="{x:Null}" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </DataGrid.CellStyle>
  <DataGrid.RowStyle>
    <Style TargetType="DataGridRow">
      <Style.Triggers>
        <Trigger Property="IsSelected" Value="True">
          <Setter Property="Background" Value="{x:Null}" />
          <Setter Property="BorderBrush" Value="{x:Null}" />
        </Trigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="None">
          <Setter Property="Background" Value="LightGray" />
        </DataTrigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="Up">
          <Setter Property="Background" Value="LightGreen" />
        </DataTrigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="Down">
          <Setter Property="Background" Value="Yellow" />
        </DataTrigger>
        <DataTrigger Binding="{Binding PositionChange}" Value="Out">
          <Setter Property="Background" Value="Red" />
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </DataGrid.RowStyle>
  <DataGrid.Columns>
```

```
      <DataGridTextColumn Binding="{Binding Position}" />
      <DataGridTextColumn Binding="{Binding Racer.Number}" />
      <DataGridTextColumn Binding="{Binding Racer.Name}" />
      <DataGridTextColumn Binding="{Binding Racer.Team}" />
      <DataGridTextColumn Binding="{Binding Lap}" />
    </DataGrid.Columns>
  </DataGrid>
```

> **NOTE** *Data triggers are explained in Chapter 35, "Core WPF."*

The data context specified with the `DataGrid` control is found in the resources of the window with the `CollectionViewSource`. The collection view source is bound to the data context that you'll see soon is specified with the code-behind. The important property set here is `IsLiveSortingRequested`. The value is set to `true` to change the order of the elements in the user interface. The property used for sorting is `Position`. As the position changes, the items are reordered in real time:

```
<Window.Resources>
  <CollectionViewSource x:Key="cvs" Source="{Binding}"
      IsLiveSortingRequested="True">
    <CollectionViewSource.SortDescriptions>
      <scm:SortDescription PropertyName="Position" />
    </CollectionViewSource.SortDescriptions>
  </CollectionViewSource>
</Window.Resources>
```

Now, you just need to get to the code-behind source code where the data context is set and the live values are changed dynamically. In the constructor of the main window, the `DataContext` property is set to the initial collection of type `LapRacerInfo`. Next, a background task invokes the `NextLap` method every three seconds to change the values in the UI with the new positions. The background task makes use of an async Lambda expression. The implementation could be changed to get live data from a Web service (code file `LiveShaping/MainWindow.xaml.cs`).

```
public partial class MainWindow : Window
{
  private LapChart lapChart = new LapChart();
  public MainWindow()
  {
    InitializeComponent();
    this.DataContext = lapChart.GetLapInfo();

    Task.Run(async () =>
      {
        bool raceContinues = true;
        while (raceContinues)
        {
          await Task.Delay(3000);
          raceContinues = lapChart.NextLap();
        }
      });
  }
}
```

Figure 36-20 shows a run of the application while in lap 14, with a leading Fernando Alonso driving a Ferrari.

1	5	Fernando Alonso	Ferrari	14	None
2	18	Pastor Maldonado	Williams	14	None
3	9	Kimi Raikkonen	Lotus	14	None
4	4	Lewis Hamilton	McLaren	14	None
5	8	Nico Rosberg	Mercedes	14	None
6	10	Romain Grosjean	Lotus	14	None
7	1	Sebastian Vettel	Red Bull Racing	14	None
8	3	Jenson Button	McLaren	14	None
9	14	Kamui Kobayashi	Sauber	14	None
10	2	Mark Webber	Red Bull Racing	14	Up
11	11	Paul di Resta	Force India	14	Up
12	17	Jean-Eric Vergne	Toro Rosso	14	Up
13	6	Felipe Massa	Ferrari	14	Up
14	12	Nico Hülkenberg	Force India	14	Up
15	16	Daniel Riccardio	Toro Rosso	14	Up
16	15	Sergio Perez	Sauber	14	Up
17	20	Heikki Kovalainen	Caterham	14	Down
18	24	Timo Glock	Marussia	14	Down
19	25	Charles Pic	Marussia	14	None
20	21	Witali Petrow	Caterham	14	None
21	22	Pedro de la Rosa	HRT	14	None
22	23	Narain Karthikeyan	HRT	14	None
99	19	Bruno Senna	Williams	14	Out
99	7	Michael Schumacher	Mercedes	14	Out

FIGURE 36-20

SUMMARY

This chapter covered some features of WPF that are extremely important for business applications. For clear and easy interaction with data, WPF data binding provides a leap forward. You can bind any property of a .NET class to a property of a WPF element. The binding mode defines the direction of the binding. You can bind .NET objects and lists, and define a data template to create a default look for a .NET class.

Command binding makes it possible to map handler code to menus and toolbars. You've also seen how easy it is to copy and paste with WPF because a command handler for this technology is already included in the `TextBox` control. You've also seen many more WPF features, such as using a `DataGrid`, the `CollectionViewSource` for sorting and grouping, and all this with live shaping as well.

The next chapter goes into another facet of WPF: working with documents.

37

Creating Documents with WPF

WHAT'S IN THIS CHAPTER?

- Creating flow documents
- Creating fixed documents
- Creating XPS documents
- Printing documents

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `http://www.wrox.com/remtitle.cgi?isbn=1118314425` on the Download Code tab. The code for this chapter is divided into the following major examples:

- Show Fonts
- Text Effects
- Table
- Flow Documents
- Create XPS
- Printing

INTRODUCTION

Creating documents is a large part of WPF. The namespace `System.Windows.Documents` supports creating both flow documents and fixed documents. This namespace contains elements with which you can have a rich Word-like experience with flow documents, and create WYSIWYG fixed documents.

Flow documents are geared toward screen reading; the content of the document is arranged based on the size of the window and the flow of the document changes if the window is resized. *Fixed documents* are mainly used for printing and page-oriented content and the content is always arranged in the same way.

This chapter teaches you how to create and print flow documents and fixed documents, and covers the namespaces `System.Windows.Documents`, `System.Windows.Xps`, and `System.IO.Packaging`.

TEXT ELEMENTS

To build the content of documents, you need document elements. The base class of these elements is `TextElement`. This class defines common properties for font settings, foreground and background, and text effects. `TextElement` is the base class for the classes `Block` and `Inline`, whose functionality is explored in the following sections.

Fonts

An important aspect of text is how it looks, and thus the importance of the font. With the `TextElement`, the font can be specified with the properties `FontWeight`, `FontStyle`, `FontStretch`, `FontSize`, and `FontFamily`:

- `FontWeight` — Predefined values are specified by the `FontWeights` class, which offers values such as `UltraLight`, `Light`, `Medium`, `Normal`, `Bold`, `UltraBold`, and `Heavy`.
- `FontStyle` — Values are defined by the `FontStyles` class, which offers `Normal`, `Italic`, and `Oblique`.
- `FontStretch` — Enables you to specify the degrees to stretch the font compared to the normal aspect ratio. `FrontStretch` defines predefined stretches that range from 50% (`UltraCondensed`) to 200% (`UltraExpanded`). Predefined values in between the range are `ExtraCondensed` (62.5%), `Condensed` (75%), `SemiCondensed` (87.5%), `Normal` (100%), `SemiExpanded` (112.5%), `Expanded` (125%), and `ExtraExpanded` (150%).
- `FontSize` — This is of type `double` and enables you to specify the size of the font in device-independent units, inches, centimeters, and points.
- `FontFamily` — Use this to define the name of the preferred font-family, e.g., Arial or Times New Roman. With this property you can specify a list of font family names so if one font is not available, the next one in the list is used. (If neither the selected font nor the alternate font are available, a flow document falls back to the default `MessageFontFamily`.) You can also reference a font family from a resource or use a URI to reference a font from a server. With fixed documents there's no fallback on a font not available because the font is available with the document.

To give you a feel for the look of different fonts, the following sample WPF application includes a `ListBox`. The `ListBox` defines an `ItemTemplate` for every item in the list. This template uses four `TextBlock` elements whereby the `FontFamily` is bound to the `Source` property of a `FontFamily` object. With different `TextBlock` elements, `FontWeight` and `FontStyle` are set (XAML file `ShowFonts/ShowFontsWindow.xaml`):

```
<ListBox ItemsSource="{Binding}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal" >
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" Text="{Binding Path=Source}" />
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" FontStyle="Italic" Text="Italic" />
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" FontWeight="UltraBold" Text="UltraBold" />
        <TextBlock Margin="3, 0, 3, 0" FontFamily="{Binding Path=Source}"
          FontSize="18" FontWeight="UltraLight" Text="UltraLight" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

In the code-behind, the data context is set to the result of the `SystemFontFamilies` property of the `System.Windows.Media.Font` class. This returns all the available fonts (code file `ShowFonts/ShowFontsWindow.xaml.cs`):

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

    this.DataContext=Fonts.SystemFontFamilies;
  }
}
```

Running the application, you get a large list of system font families with italic, bold, ultrabold, and ultralight characteristics, as shown in Figure 37-1.

FIGURE 37-1

TextEffect

Now let's have a look into `TextEffect`, as it is also common to all document elements. `TextEffect` is defined in the namespace `System.Windows.Media` and derives from the base class `Animatable`, which enables the animation of text.

`TextEffect` enables you to animate a clipping region, the foreground brush, and a transformation. With the properties `PositionStart` and `PositionCount` you specify the position in the text to which the animation applies.

For applying the text effects, the `TextEffects` property of a `Run` element is set. The `TextEffect` element specified within the property defines a foreground and a transformation. For the foreground, a `SolidColorBrush` with the name `brush1` is used that is animated with a `ColorAnimation` element. The transformation makes use of a `ScaleTransformation` with the name `scale1`, which is animated from two `DoubleAnimation` elements (XAML file `TextEffectsDemo/MainWindow.xaml`):

```
<TextBlock>
  <TextBlock.Triggers>
    <EventTrigger RoutedEvent="TextBlock.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <ColorAnimation AutoReverse="True" RepeatBehavior="Forever"
              From="Blue" To="Red" Duration="0:0:16"
              Storyboard.TargetName="brush1"
              Storyboard.TargetProperty="Color" />
          <DoubleAnimation AutoReverse="True"
              RepeatBehavior="Forever"
              From="0.2" To="12" Duration="0:0:16"
              Storyboard.TargetName="scale1"
              Storyboard.TargetProperty="ScaleX" />
          <DoubleAnimation AutoReverse="True"
              RepeatBehavior="Forever"
              From="0.2" To="12" Duration="0:0:16"
              Storyboard.TargetName="scale1"
              Storyboard.TargetProperty="ScaleY" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </TextBlock.Triggers>
  <Run FontFamily="Segoe UI">
      cn|elements
```

```
        <Run.TextEffects>
          <TextEffect PositionStart="0" PositionCount="30">
            <TextEffect.Foreground>
              <SolidColorBrush x:Name="brush1" Color="Blue" />
            </TextEffect.Foreground>
            <TextEffect.Transform>
              <ScaleTransform x:Name="scale1" ScaleX="3" ScaleY="3" />
            </TextEffect.Transform>
          </TextEffect>
        </Run.TextEffects>
      </Run>
    </TextBlock>
```

Running the application, you can see the changes in size and color as shown in Figures 37-2 and 37-3.

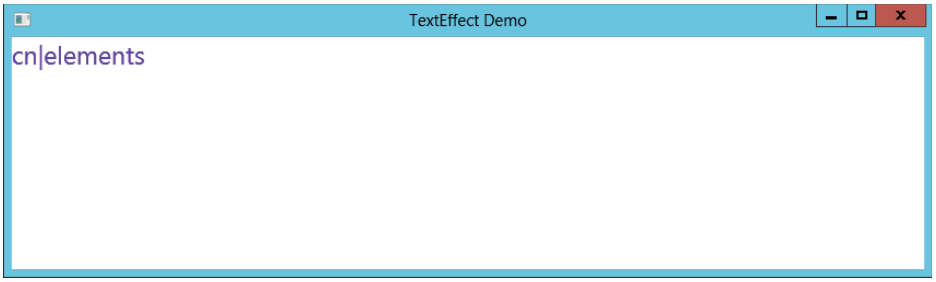


FIGURE 37-2

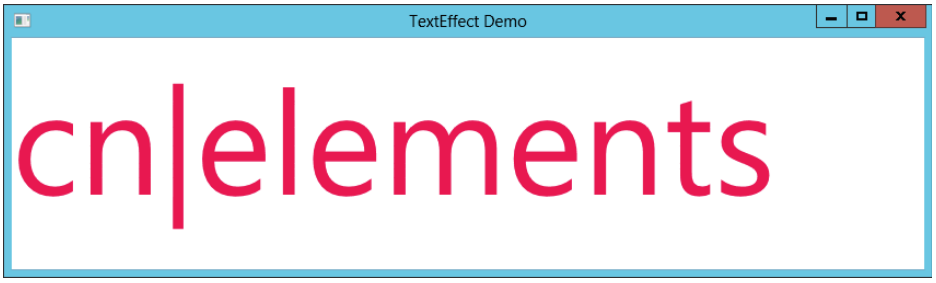


FIGURE 37-3

Inline

The base class for all inline flow content elements is `Inline`. You can use `Inline` elements within a paragraph of a flow document. Because within a paragraph one `Inline` element can follow another, the `Inline` class provides the `PreviousInline` and `NextInline` properties to navigate from one element to another. You can also get a collection of all peer inlines with `SiblingInlines`.

The `Run` element that was used earlier to write some text is an `Inline` element for formatted or unformatted text, but there are many more. A new line after a `Run` element can be done with the `LineBreak` element.

The `Span` element derives from the `Inline` class and enables the grouping of `Inline` elements. Only `Inline` elements are allowed within the content of `Span`. The self-explanatory `Bold`, `Hyperlink`, `Italic`, and `Underline` classes all derive from `Span` and thus have the same functionality to enable `Inline` elements as its content, but to act on these elements differently. The following XAML code demonstrates using

Bold, Italic, Underline, and LineBreak, as shown in Figure 37-4 (XAML file FlowDocumentsDemo/FlowDocument1.xaml):

```
<Paragraph FontWeight="Normal">
  <Span>
    <Span>Normal</Span>
    <Bold>Bold</Bold>
    <Italic>Italic</Italic>
    <LineBreak />
    <Underline>Underline</Underline>
  </Span>
</Paragraph>
```

AnchoredBlock is an abstract class that derives from Inline and is used to anchor Block elements to flow content. Figure and Floater are concrete classes that derive from AnchoredBlock. Because these two inline elements become interesting in relation to blocks, these elements are discussed later in this chapter.

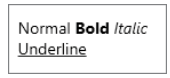


FIGURE 37-4

Another Inline element that maps UI elements that have been used in previous chapters is InlineUIContainer. InlineUIContainer enables adding all UIElement objects (for example, a Button) to the document. The following code segment adds an InlineUIContainer with ComboBox, RadioButton, and TextBox elements to the document (the result is shown in Figure 37-5) (XAML file FlowDocumentsDemo/FlowDocument2.xaml):

> **NOTE** *Of course, you can also style the UI elements as shown in Chapter 35, "Core WPF."*

```
<Paragraph TextAlignment="Center">
  <Span FontSize="36">
    <Italic>cn|elements</Italic>
  </Span>
  <LineBreak />
  <LineBreak />
  <InlineUIContainer>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <ComboBox Width="40"  Margin="3"  Grid.Row="0">
        <ComboBoxItem>Filet Mignon</ComboBoxItem>
        <ComboBoxItem>Rib Eye</ComboBoxItem>
        <ComboBoxItem>Sirloin</ComboBoxItem>
      </ComboBox>
      <StackPanel Grid.Row="0" Grid.RowSpan="2" Grid.Column="1">
        <RadioButton>Raw</RadioButton>
        <RadioButton>Medium</RadioButton>
        <RadioButton>Well done</RadioButton>
      </StackPanel>
      <TextBox Grid.Row="1" Grid.Column="0" Width="140"></TextBox>
    </Grid>
  </InlineUIContainer>
</Paragraph>
```

Block

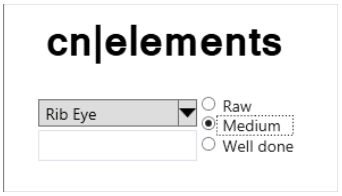


FIGURE 37-5

`Block` is an abstract base class for block-level elements. Blocks enable grouping elements contained to specific views. Common to all blocks are the properties `PreviousBlock`, `NextBlock`, and `SiblingBlocks` that enable you to navigate from block to block. Setting `BreakPageBefore` and `BreakColumnBefore` page and column breaks are done before the block starts. A `Block` also defines a border with the `BorderBrush` and `BorderThickness` properties.

Classes that derive from `Block` are `Paragraph`, `Section`, `List`, `Table`, and `BlockUIContainer`. `BlockUIContainer` is similar to `InlineUIContainer` in that you can add elements that derive from `UIElement`.

`Paragraph` and `Section` are simple blocks; `Paragraph` contains inline elements, and `Section` is used to group other `Block` elements. With the `Paragraph` block you can determine whether a page or column break is allowed within the paragraph or between paragraphs. `KeepTogether` can be used to disallow breaking within the paragraph; `KeepWithNext` tries to keep one paragraph and the next together. If a paragraph is broken by a page or column break, `MinWidowLines` defines the minimum number of lines that are placed after the break; `MinOrphanLines` defines the minimum number of lines before the break.

The `Paragraph` block also enables decorating the text within the paragraph with `TextDecoration` elements. Predefined text decorations are defined by `TextDecorations`: `Baseline`, `Overline`, `Strikethrough`, and `Underline`.

The following XAML code shows multiple `Paragraph` elements. One `Paragraph` element with a title follows another with the content belonging to this title. These two paragraphs are connected with the attribute `KeepWithNext`. It's also assured that the paragraph with the content is not broken by setting `KeepTogether` to `True` (XAML file `FlowDocumentsDemo/ParagraphDemo.xaml`):

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              ColumnWidth="300" FontSize="16" FontFamily="Georgia">
  <Paragraph FontSize="36">
    <Run>Lyrics</Run>
  </Paragraph>
  <Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
    <Bold>
      <Run>Mary had a little lamb</Run>
    </Bold>
  </Paragraph>
  <Paragraph KeepTogether="True">
    <Run>Mary had a little lamb,</Run>
    <LineBreak />
    <Run>little lamb, little lamb,</Run>
    <LineBreak />
    <Run>Mary had a little lamb,</Run>
    <LineBreak />
    <Run>whose fleece was white as snow.</Run>
    <LineBreak />
    <Run>And everywhere that Mary went,</Run>
    <LineBreak />
    <Run>Mary went, Mary went,</Run>
    <LineBreak />
    <Run>and everywhere that Mary went,</Run>
    <LineBreak />
    <Run>the lamb was sure to go.</Run>
  </Paragraph>
  <Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
    <Bold>
      <Run>Humpty Dumpty</Run>
```

```
      </Bold>
    </Paragraph>
    <Paragraph KeepTogether="True">
      <Run>Humpty dumpty sat on a wall</Run>
      <LineBreak />
      <Run>Humpty dumpty had a great fall</Run>
      <LineBreak />
      <Run>All the King's horses</Run>
      <LineBreak />
      <Run>And all the King's men</Run>
      <LineBreak />
      <Run>Couldn't put Humpty together again</Run>
    </Paragraph>
  </FlowDocument>
```

The result is shown in Figure 37-6.

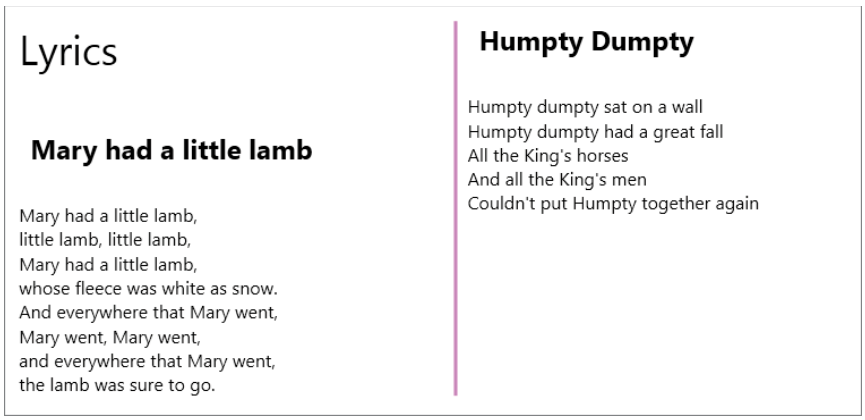


FIGURE 37-6

Lists

The `List` class is used to create textual unordered or ordered lists. `List` defines the bullet style of its items by setting the `MarkerStyle` property. `MarkerStyle` is of type `TextMarkerStyle` and can be a number (`Decimal`), a letter (`LowerLatin` and `UpperLatin`), a roman numeral (`LowerRoman` and `UpperRoman`), or a graphic (`Disc`, `Circle`, `Square`, `Box`). `List` can only contain `ListItem` elements, which in turn can only contain `Block` elements.

Defining the following list with XAML results in the output shown in Figure 37-7 (XAML file `FlowDocumentsDemo/ListDemo.xaml`):

```
<List MarkerStyle="Square">
  <ListItem>
    <Paragraph>Monday</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Tuesday</Paragraph>
  </ListItem>
  <ListItem>
    <Paragraph>Wednesday</Paragraph>
  </ListItem>
</List>
```

Tables

FIGURE 37-7

The `Table` class is very similar to the `Grid` class presented in Chapter 35 to define rows and columns. The following example demonstrates creating a `FlowDocument` with a `Table`. To create tables you can add `TableColumn` objects to the `Columns` property. With `TableColumn` you can specify the `width and background`.

The `Table` also contains `TableRowGroup` objects. The `TableRowGroup` has a `Rows` property whereby `TableRow` objects can be added. The `TableRow` class defines a `Cells` property that enables adding `TableCell` objects. `TableCell` objects can contain any `Block` element. Here, a `Paragraph` is used that contains the `Inline` element `Run` (code file `TableDemo/MainWindow.xaml.cs`):

```
var doc = new FlowDocument();
var t1 = new Table();
t1.Columns.Add(new TableColumn
{ Width = new GridLength(50, GridUnitType.Pixel) });
t1.Columns.Add(new TableColumn
{ Width = new GridLength(1, GridUnitType.Auto) });
t1.Columns.Add(new TableColumn
{ Width = new GridLength(1, GridUnitType.Auto) });

var titleRow = new TableRow { Background = Brushes.LightBlue };
var titleCell = new TableCell
{ ColumnSpan = 3, TextAlignment = TextAlignment.Center };
titleCell.Blocks.Add(
    new Paragraph(new Run("Formula 1 Championship 2011")
    { FontSize=24, FontWeight = FontWeights.Bold }));
titleRow.Cells.Add(titleCell);

var headerRow = new TableRow
{ Background = Brushes.LightGoldenrodYellow };
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Pos"))
{ FontSize = 14, FontWeight=FontWeights.Bold}));
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Name"))
{ FontSize = 14, FontWeight = FontWeights.Bold }));
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Points"))
{ FontSize = 14, FontWeight = FontWeights.Bold }));

var rowGroup = new TableRowGroup();
rowGroup.Rows.Add(titleRow);
rowGroup.Rows.Add(headerRow);

string[][] results = new string[][]
{
  new string[] { "1.", "Sebastian Vettel", "392" },
  new string[] { "2.", "Jenson Button", "270" },
  new string[] { "3.", "Mark Webber", "258" },
  new string[] { "4.", "Fernando Alonso", "257" },
  new string[] { "5.", "Lewis Hamilton", "227"}
};

List<TableRow> rows = results.Select(row =>
  {
    var tr = new TableRow();
    foreach (var cell in row)
    {
      tr.Cells.Add(new TableCell(new Paragraph(new Run(cell))));
    }
    return tr;
  }).ToList();
```

```
rows.ForEach(r => rowGroup.Rows.Add(r));

t1.RowGroups.Add(rowGroup);
doc.Blocks.Add(t1);

reader.Document = doc;
```

Running the application, you can see the nicely formatted table as shown in Figure 37-8.

Formula 1 Championship 2011

Pos	Name	Points
1.	Sebastian Vettel	392
2.	Jenson Button	270
3.	Mark Webber	258
4.	Fernando Alonso	257
5.	Lewis Hamilton	227

FIGURE 37-8

Anchor to Blocks

Now that you've learned about the `Inline` and `Block` elements, you can combine the two by using the `Inline` elements of type `AnchoredBlock`. `AnchoredBlock` is an abstract base class with two concrete implementations, `Figure` and `Floater`.

The `Floater` displays its content parallel to the main content with the properties `HorizontalAlignment` and `Width`.

Starting with the earlier example, a new paragraph is added that contains a `Floater`. This `Floater` is aligned to the left and has a width of 120. As shown in Figure 37-9, the next paragraph flows around it (XAML file `FlowDocumentsDemo/ParagraphKeepTogether.xaml`):

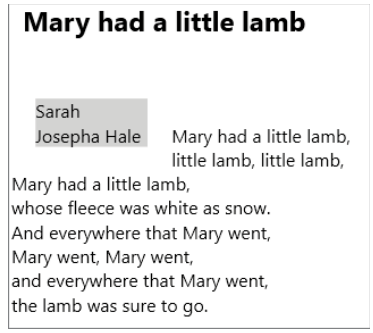


FIGURE 37-9

```
<Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
  <Bold>
    <Run>Mary had a little lamb</Run>
  </Bold>
</Paragraph>
<Paragraph>
  <Floater HorizontalAlignment="Left" Width="120">
    <Paragraph Background="LightGray">
      <Run>Sarah Josepha Hale</Run>
    </Paragraph>
  </Floater>
</Paragraph>
<Paragraph KeepTogether="True">
  <Run>Mary had a little lamb</Run>
  <LineBreak />
  <!-- ... -->
</Paragraph>
```

A `Figure` aligns horizontally and vertically and can be anchored to the page, content, a column, or a paragraph. The `Figure` in the following code is anchored to the page center but with a horizontal and vertical offset. The `WrapDirection` is set so that both left and right columns wrap around the figure. Figure 37-10 shows the result of the wrap (XAML file `FlowDocumentsDemo/FigureAlignment.xaml`):

```
<Paragraph>
  <Figure HorizontalAnchor="PageCenter" HorizontalOffset="20"
      VerticalAnchor="PageCenter" VerticalOffset="20" WrapDirection="Both" >
    <Paragraph Background="LightGray" FontSize="24">
      <Run>Lyrics Samples</Run>
    </Paragraph>
  </Figure>
</Paragraph>
```

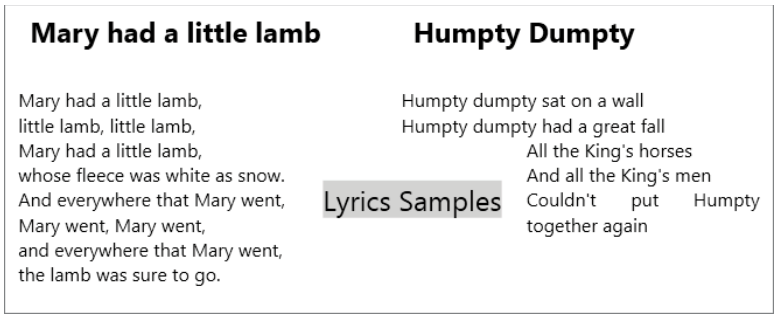


FIGURE 37-10

`Floater` and `Figure` are both used to add content that is not in the main flow. Although these two features seem similar, the characteristics of these elements are quite different. The following table explains the differences between `Floater` and `Figure`:

CHARACTERISTIC	FLOATER	FIGURE
Position	A floater cannot be positioned. It is rendered where space is available.	A figure can be positioned with horizontal and vertical anchors. It can be docked relative to the page, content, column, or paragraph.
Width	A floater can be placed only within one column. If the width is set larger than the column's size, it is ignored.	A figure can be sized across multiple columns. The width of a figure can be set to 0.5 pages or two columns.
Pagination	If a floater is larger than a column's height, the floater breaks and paginates to the next column or page.	If a figure is larger than a column's height, only the part of the figure that fits in the column is rendered; the other content is lost.

FLOW DOCUMENTS

With all the `Inline` and `Block` elements, now you know what should be put into a flow document. The class `FlowDocument` can contain `Block` elements, and the `Block` elements can contain `Block` or `Inline` elements, depending on the type of the `Block`.

A major functionality of the `FlowDocument` class is that it is used to break up the flow into multiple pages. This is done via the `IDocumentPaginatorSource` interface, which is implemented by `FlowDocument`.

Other options with a `FlowDocument` are to set up the default font and foreground and background brushes, and to configure the page and column sizes.

The following XAML code for the `FlowDocument` defines a default font and font size, a column width, and a ruler between columns:

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    ColumnWidth="300" FontSize="16" FontFamily="Georgia"
    ColumnRuleWidth="3" ColumnRuleBrush="Violet">
```

Now you just need a way to view the documents. The following list describes several viewers:

- `RichTextBox` — A simple viewer that also allows editing (as long as the `IsReadOnly` property is not set to `true`). The `RichTextBox` doesn't display the document with multiple columns but instead in scroll mode. This is similar to the Web layout in Microsoft Word. The scrollbar can be enabled by setting the `HorizontalScrollbarVisibility` to `ScrollbarVisibility.Auto`.
- `FlowDocumentScrollViewer` — A reader that is meant only to read but not edit documents. This reader enables zooming into the document. There's also a toolbar with a slider for zooming that can be enabled with the property `IsToolbarEnabled`. Settings such as `CanIncreaseZoom`, `CanDecreaseZoom`, `MinZoom`, and `MaxZoom` enable setting the zoom features.
- `FlowDocumentPageViewer` — A viewer that paginates the document. With this viewer you not only have a toolbar to zoom into the document, you can also switch from page to page.
- `FlowDocumentReader` — A viewer that combines the functionality of `FlowDocumentScrollViewer` and `FlowDocumentPageViewer`. This viewer supports different viewing modes that can be set from the toolbar or with the property `ViewingMode` that is of type `FlowDocumentReaderViewingMode`. This enumeration has the possible values `Page`, `TwoPage`, and `Scroll`. The viewing modes can also be disabled according to your needs.

The sample application to demonstrate flow documents defines several readers such that one reader can be chosen dynamically. Within the `Grid` element you can find the `FlowDocumentReader`, `RichTextBox`, `FlowDocumentScrollViewer`, and `FlowDocumentPageViewer`. With all the readers the `Visibility` property is set to `Collapsed`, so on startup none of the readers appear. The `ComboBox` that is the first child element within the grid enables the user to select the active reader. The `ItemsSource` property of the `ComboBox` is bound to the `Readers` property to display the list of readers. On selection of a reader, the method `OnReaderSelectionChanged` is invoked (XAML file `FlowDocumentsDemo/MainWindow.xaml`):

```
<Grid x:Name="grid1">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>
  <ComboBox ItemsSource="{Binding Readers}" Grid.Row="0" Grid.Column="0"
      Margin="4" SelectionChanged="OnReaderSelectionChanged"
     SelectedIndex="0">
    <ComboBox.ItemTemplate>
      <DataTemplate>
        <StackPanel>
          <TextBlock Text="{Binding Name}" />
        </StackPanel>
      </DataTemplate>
    </ComboBox.ItemTemplate>
  </ComboBox>
  <Button Grid.Column="1" Margin="4" Padding="3" Click="OnOpenDocument">
      Open Document</Button>
  <FlowDocumentReader ViewingMode="TwoPage" Grid.Row="1"
      Visibility="Collapsed" Grid.ColumnSpan="2" />
  <RichTextBox IsDocumentEnabled="True" HorizontalScrollBarVisibility="Auto"
      VerticalScrollBarVisibility="Auto" Visibility="Collapsed"
      Grid.Row="1" Grid.ColumnSpan="2" />
  <FlowDocumentScrollViewer Visibility="Collapsed" Grid.Row="1"
      Grid.ColumnSpan="2" />
  <FlowDocumentPageViewer Visibility="Collapsed" Grid.Row="1"
      Grid.ColumnSpan="2" />
</Grid>
```

The `Readers` property of the `MainWindow` class invokes the `GetReaders` method to return to return the readers to the `ComboBox` data binding. The `GetReaders` method returns the list assigned to the variable `documentReaders`. In case `documentReaders` was not yet assigned, the `LogicalTreeHelper` class is used

to get all the flow document readers within the grid `grid1`. As there is not a base class for a flow document reader nor an interface implemented by all readers, the `LogialTreeHelper` looks for all elements of type `FrameworkElement` that have a property `Document`. The `Document` property is common to all flow document readers. With every reader a new anonymous object is created with the properties `Name` and `Instance`. The `Name` property is used to appear in the `ComboBox` to enable the user to select the active reader, and the `Instance` property holds a reference to the reader to show the reader if it should be active (code file `FlowDocumentsDemo/MainWindow.xaml.cs`):

```
public IEnumerable<object> Readers
{
  get
  {
    return GetReaders();
  }
}

private List<object> documentReaders = null;
private IEnumerable<object> GetReaders()
{
  return documentReaders ?? (documentReaders =
    LogicalTreeHelper.GetChildren(grid1).OfType<FrameworkElement>()
      .Where(el => el.GetType().GetProperties()
          .Where(pi => pi.Name == "Document").Count() > 0)
      .Select(el => new
      {
        Name = el.GetType().Name,
        Instance = el
      }).Cast<object>().ToList());
}
```

When the user selects a flow document reader, the method `OnReaderSelectionChanged` is invoked. The XAML code that references this method was shown earlier. Within this method the previously selected flow document reader is made invisible by setting it to collapsed, and the variable `activeDocumentReader` is set to the selected reader:

```
private void OnReaderSelectionChanged(object sender,
                                      SelectionChangedEventArgs e)
{
  dynamic item = (sender as ComboBox).SelectedItem;

  if (activedocumentReader != null)
  {
    activedocumentReader.Visibility = Visibility.Collapsed;
  }
  activedocumentReader = item.Instance;
}

private dynamic activedocumentReader = null;
```

> **NOTE** *The sample code makes use of the* `dynamic` *keyword—the variable* `activeDocumentReader` *is declared as* `dynamic` *type. The* `dynamic` *keyword is used because the* `SelectedItem` *from the* `ComboBox` *either returns a* `FlowDocumentReader`*,* *a* `FlowDocumentScrollViewer`*,* *a* `FlowDocumentPageViewer`*, or a* `RichTextBox`*. All these types are flow document readers that offer a* `Document` *property of type* `FlowDocument`*. However, there's no common base class or interface defining this property. The* `dynamic` *keyword allows accessing these different types from the same variable and using the* `Document` *property. The* `dynamic` *keyword is explained in detail in Chapter 12, "Dynamic Language Extensions."*

When the user clicks the button to open a document, the method `OnOpenDocument` is invoked. With this method the `XamlReader` class is used to load the selected XAML file. If the reader returns a `FlowDocument` (which is the case when the root element of the XAML is the `FlowDocument` element), the `Document` property of the `activeDocumentReader` is assigned, and the `Visibility` is set to visible:

```
private void OnOpenDocument(object sender, RoutedEventArgs e)
{
  try
  {
    var dlg = new OpenFileDialog();
    dlg.DefaultExt = "*.xaml";
    dlg.InitialDirectory = Environment.CurrentDirectory;
    if (dlg.ShowDialog() == true)
    {
      using (FileStream xamlFile = File.OpenRead(dlg.FileName))
      {
        var doc = XamlReader.Load(xamlFile) as FlowDocument;
        if (doc != null)
        {
          activedocumentReader.Document = doc;
          activedocumentReader.Visibility = Visibility.Visible;
        }
      }
    }
  }
  catch (XamlParseException ex)
  {
    MessageBox.Show(string.Format("Check content for a Flow document, {0}",
        ex.Message));
  }
}
```

The running application is shown in Figure 37-11. This figure shows a flow document with the `FlowDocumentReader` in `TwoPage` mode.

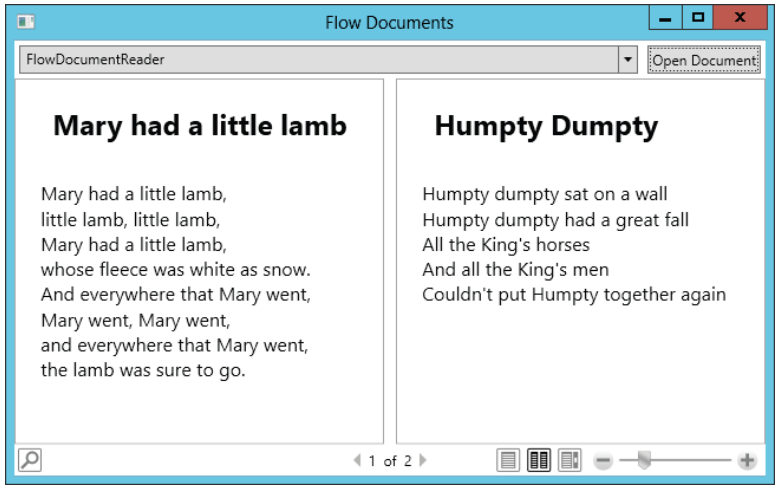


FIGURE 37-11

FIXED DOCUMENTS

Fixed documents always define the same look, the same pagination, and use the same fonts—no matter where the document is copied or used. WPF defines the class `FixedDocument` to create fixed documents, and the class `DocumentViewer` to view fixed documents.

This section uses a sample application to create a fixed document programmatically by requesting user input for a menu plan. The data for the menu plan is the content of the fixed document. Figure 37-12 shows the main user interface of this application, where the user can select a day with the `DatePicker` class, enter menus for a week in a `DataGrid`, and click the Create Doc button to create a new `FixedDocument`. This application uses `Page` objects that are navigated within a `NavigationWindow`. Clicking the Create Doc button navigates to a new page that contains the fixed document.

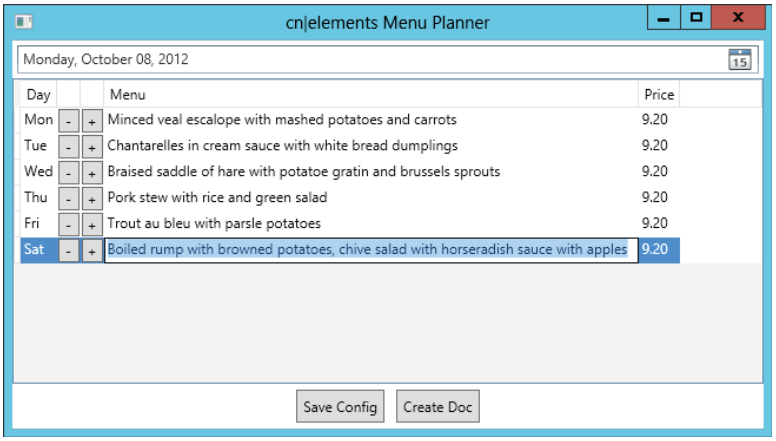


FIGURE 37-12

The event handler for the Create Doc button, `OnCreateDoc`, navigates to a new page. To do this, the handler instantiates the new page, `DocumentPage`. This page includes a handler, `NavigationService_LoadCompleted`, that is assigned to the `LoadCompleted` event of the `NavigationService`. Within this handler the new page can access the content that is passed to the page. Then the navigation is done by invoking the `Navigate` method to `page2`. The new page receives the object `menus` that contains all the menu information needed to build the fixed page. `menus` is a readonly variable of type `ObservableCollection<MenuEntry>` (code file `CreateXps/MenuPlannerPage.xaml.cs`):

```
private void OnCreateDoc(object sender, RoutedEventArgs e)
{
  if (menus.Count == 0)
  {
    MessageBox.Show("Select a date first", "Menu Planner",
        MessageBoxButton.OK);
    return;
  }
  var page2 = new DocumentPage();
  NavigationService.LoadCompleted +=
      page2.NavigationService_LoadCompleted;
  NavigationService.Navigate(page2, menus);
}
```

Within the DocumentPage, a DocumentViewer is used to provide read access to the fixed document. The fixed document is created in the method NavigationService_LoadCompleted. With the event handler, the data that is passed from the first page is received with the ExtraData property of NavigationEventArgs.

The received ObservableCollection<MenuEntry> is assigned to a menus variable that is used to build the fixed page (code file CreateXps/DocumentPage.xaml.cs):

```
internal void NavigationService_LoadCompleted(object sender,
    NavigationEventArgs e)
{
  menus = e.ExtraData as ObservableCollection<MenuEntry>;
  fixedDocument = new FixedDocument();
  var pageContent1 = new PageContent();
  fixedDocument.Pages.Add(pageContent1);
  var page1 = new FixedPage();
  pageContent1.Child = page1;
  page1.Children.Add(GetHeaderContent());
  page1.Children.Add(GetLogoContent());
  page1.Children.Add(GetDateContent());
  page1.Children.Add(GetMenuContent());
  viewer.Document = fixedDocument;
  NavigationService.LoadCompleted -= NavigationService_LoadCompleted;
}
```

Fixed documents are created with the FixedDocument class. The FixedDocument element only contains PageContent elements that are accessible via the Pages property. The PageContent elements must be added to the document in the order in which they should appear on the page. PageContent defines the content of a single page.

PageContent has a Child property such that a FixedPage can be associated with it. To the FixedPage you can add elements of type UIElement to the Children collection. This is where you can add all the elements you've learned about in the last two chapters, including a TextBlock element that itself can contain Inline and Block elements.

In the sample code, the children to the FixedPage are created with helper methods GetHeaderContent, GetLogoContent, GetDateContent, and GetMenuContent.

The method GetHeaderContent creates a TextBlock that is returned. The TextBlock has the Inline element Bold added, which in turn has the Run element added. The Run element then contains the header text for the document. With FixedPage.SetLeft and FixedPage.SetTop the position of the TextBox within the fixed page is defined:

```
private static UIElement GetHeaderContent()
{
  var text1 = new TextBlock
  {
    FontFamily = new FontFamily("Segoe UI"),
    FontSize = 34,
    HorizontalAlignment = HorizontalAlignment.Center
  };
  text1.Inlines.Add(new Bold(new Run("cn|elements")));
  FixedPage.SetLeft(text1, 170);
  FixedPage.SetTop(text1, 40);
  return text1;
}
```

The method `GetLogoContent` adds a logo in the form of an `Ellipse` with a `RadialGradientBrush` to the fixed document:

```
private static UIElement GetLogoContent()
{
  var ellipse = new Ellipse
  {
    Width = 90,
    Height = 40,
    Fill = new RadialGradientBrush(Colors.Yellow, Colors.DarkRed)
  };
  FixedPage.SetLeft(ellipse, 500);
  FixedPage.SetTop(ellipse, 50);
  return ellipse;
}
```

The method `GetDateContent` accesses the `menus` collection to add a date range to the document:

```
private UIElement GetDateContent()
{
  Contract.Requires(menus != null);
  Contract.Requires(menus.Count > 0);

  string dateString = String.Format("{0:d} to {1:d}",
      menus[0].Day, menus[menus.Count - 1].Day);
  var text1 = new TextBlock
  {
    FontSize = 24,
    HorizontalAlignment = HorizontalAlignment.Center
  };
  text1.Inlines.Add(new Bold(new Run(dateString)));
  FixedPage.SetLeft(text1, 130);
  FixedPage.SetTop(text1, 90);
  return text1;
}
```

Finally, the method `GetMenuContent` creates and returns a `Grid` control. This grid contains columns and rows that contain the date, menu, and price information:

```
private UIElement GetMenuContent()
{
  var grid1 = new Grid { ShowGridLines = true };

  grid1.ColumnDefinitions.Add(new ColumnDefinition
  { Width= new GridLength(50)});
  grid1.ColumnDefinitions.Add(new ColumnDefinition
  { Width = new GridLength(300)});
  grid1.ColumnDefinitions.Add(new ColumnDefinition
  { Width = new GridLength(70) });
  for (int i = 0; i < menus.Count; i++)
  {
    grid1.RowDefinitions.Add(new RowDefinition
    { Height = new GridLength(40) });
    var t1 = new TextBlock(new Run(String.Format(
                          "{0:ddd}", menus[i].Day)));
    var t2 = new TextBlock(new Run(menus[i].Menu));
    var t3 = new TextBlock(new Run(menus[i].Price.ToString()));
    var textBlocks = new TextBlock[] { t1, t2, t3 };

    for (int column = 0; column < textBlocks.Length; column++)
```

```
            {
              textBlocks[column].VerticalAlignment = VerticalAlignment.Center;
              textBlocks[column].Margin = new Thickness(5, 2, 5, 2);
              Grid.SetColumn(textBlocks[column], column);
              Grid.SetRow(textBlocks[column], i);
              grid1.Children.Add(textBlocks[column]);
            }
          }
          FixedPage.SetLeft(grid1, 100);
          FixedPage.SetTop(grid1, 140);
          return grid1;
        }
```

Run the application to see the created fixed document shown in Figure 37-13.

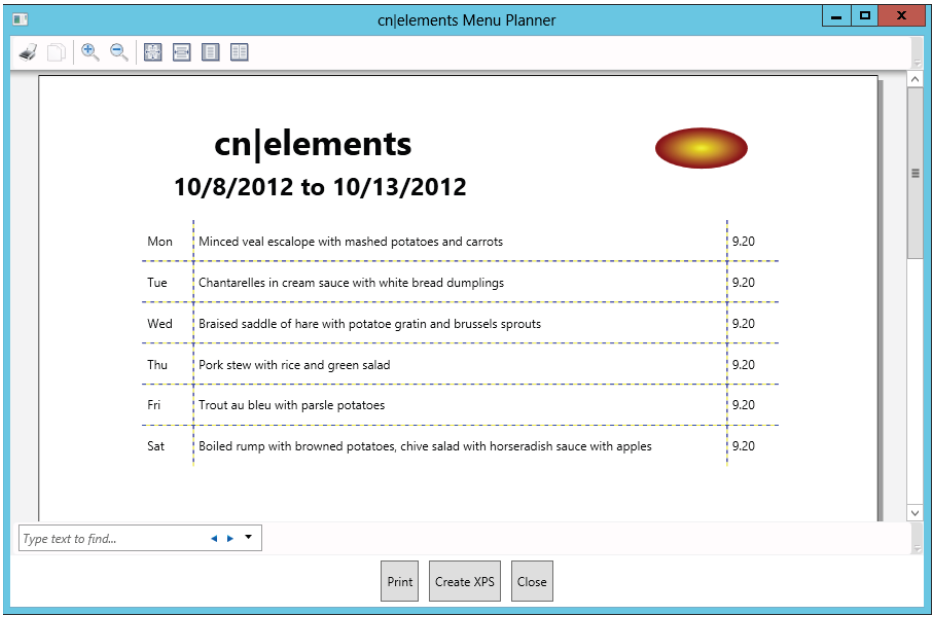


FIGURE 37-13

XPS DOCUMENTS

With Microsoft Word you can save a document as a PDF or a XPS file. XPS is the *XML Paper Specification*, a subset of WPF. Windows includes an XPS reader.

.NET includes classes and interfaces to read and write XPS documents with the namespaces `System.Windows.Xps`, `System.Windows.Xps.Packaging`, and `System.IO.Packaging`.

XPS is packaged in the zip file format, so you can easily analyze an XPS document by renaming a file with an `.xps` extension to `.zip` and opening the archive.

An XPS file requires a specific structure in the zipped document that is defined by the XML Paper Specifications (which you can download from `http://www.microsoft.com/whdc/xps/xpsspec.mspx`). The structure is based on the Open Packaging Convention (OPC) that Word documents (OOXML or Office Open XML) are based on as well. Within such a file you can find different folders for metadata, resources (such as fonts and pictures), and the document itself. Within the document folder of an XPS document is the XAML code representing the XPS subset of XAML.

To create an XPS document, you use the `XpsDocument` class from the namespace `System.Windows.Xps.Packaging`. To use this class, you need to reference the assembly `ReachFramework` as well. With this class you can add a thumbnail (`AddThumbnail`) and fixed document sequences (`AddFixedDocumentSequence`) to the document, as well as digitally sign the document. A fixed document sequence is written by using the interface `IXpsFixedDocumentSequenceWriter`, which in turn uses an `IXpsFixedDocumentWriter` to write the document within the sequence.

If a `FixedDocument` already exists, there's an easier way to write the XPS document. Instead of adding every resource and every document page, you can use the class `XpsDocumentWriter` from the namespace `System.Windows.Xps`. For this class the assembly `System.Printing` must be referenced.

With the following code snippet you can see the handler to create the XPS document. First, a filename for the menu plan is created that uses a week number in addition to the name `menuplan`. The week number is calculated with the help of the `GregorianCalendar` class. Then the `SaveFileDialog` is opened to enable the user overwrite the created filename and select the directory where the file should be stored. The `SaveFileDialog` class is defined in the namespace `Microsoft.Win32` and wraps the native file dialog. Then a new `XpsDocument` is created whose filename is passed to the constructor. Recall that the XPS file uses a `.zip` format to compress the content. With the `CompressionOption` you can specify whether the compression should be optimized for time or space.

Next, an `XpsDocumentWriter` is created with the help of the static method `XpsDocument.CreateXpsDocumentWriter`. The `Write` method of the `XpsDocumentWriter` is overloaded to accept different content or content parts to write the document. Examples of acceptable options with the `Write` method are `FixedDocumentSequence`, `FixedDocument`, `FixedPage`, `string`, and a `DocumentPaginator`. In the sample code, only the `fixedDocument` that was created earlier is passed:

```
private void OnCreateXPS(object sender, RoutedEventArgs e)
{
  var c = new GregorianCalendar();
  int weekNumber = c.GetWeekOfYear(menus[0].Day,
      CalendarWeekRule.FirstFourDayWeek, DayOfWeek.Monday);
  string fileName = String.Format("menuplan{0}", weekNumber);
  var dlg = new SaveFileDialog
  {
    FileName = fileName,
    DefaultExt = "xps",
    Filter = "XPS Documents|*.xps|All Files|*.*",
    AddExtension = true
  };
  if (dlg.ShowDialog() == true)
  {
    var doc = new XpsDocument(dlg.FileName, FileAccess.Write,
                              CompressionOption.Fast);
    XpsDocumentWriter writer = XpsDocument.CreateXpsDocumentWriter(doc);
    writer.Write(fixedDocument);
    doc.Close();
  }
}
```

By running the application to store the XPS document, you can view the document with an XPS viewer, as shown in Figure 37-14.

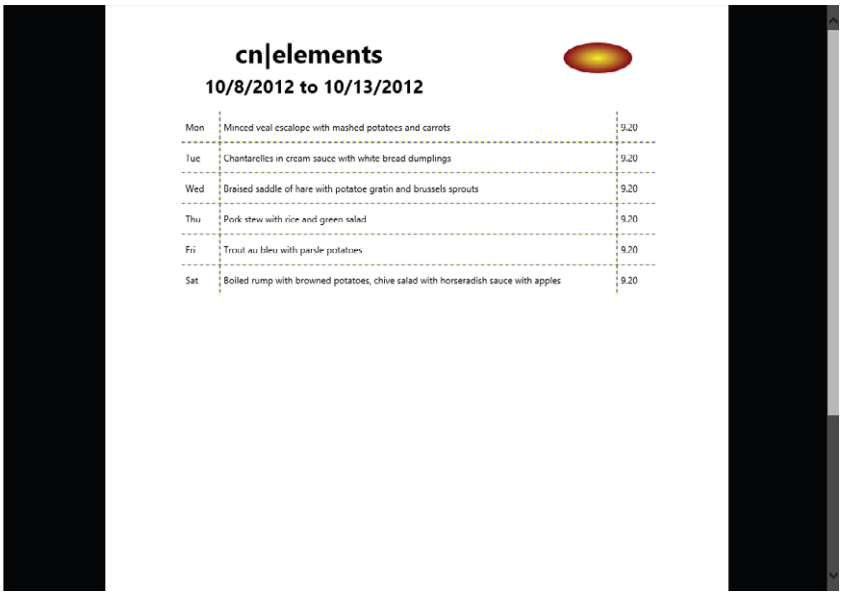


FIGURE 37-14

To one overload of the `Write` method of the `XpsDocumentWriter` you can also pass a `Visual`, which is the base class of `UIElement`, and thus you can pass any `UIElement` to the writer to create an XPS document easily. This functionality is used in the following printing example.

PRINTING

The simplest way to print a `FixedDocument` that is shown onscreen with the `DocumentViewer` is to invoke the `Print` method of the `DocumentViewer` with which the document is associated. This is all that needs to be done with the menu planner application in an `OnPrint` handler. The `Print` method of the `DocumentViewer` opens the `PrintDialog` and sends the associated `FixedDocument` to the selected printer (code file `CreateXPS/DocumentPage.xaml.cs`):

```
private void OnPrint(object sender, RoutedEventArgs e)
{
  viewer.Print();
}
```

Printing with the PrintDialog

If you want more control over the printing process, the `PrintDialog` can be instantiated, and the document printed with the `PrintDocument` method. The `PrintDocument` method requires a `DocumentPaginator` with the first argument. The `FixedDocument` returns a `DocumentPaginator` object with the `DocumentPaginator` property. The second argument defines the string that appears with the current printer and in the printer dialogs for the print job:

```
var dlg = new PrintDialog();
if (dlg.ShowDialog() == true)
{
  dlg.PrintDocument(fixedDocument.DocumentPaginator, "Menu Plan");
}
```

Printing Visuals

It's also simple to create `UIElement` objects. The following XAML code defines an `Ellipse`, a `Rectangle`, and a `Button` that is visually represented with two `Ellipse` elements. With the `Button`, there's a `Click` handler `OnPrint` that starts the print job of the visual elements (XAML file `PrintingDemo/MainWindow.xaml`):

```
<Canvas x:Name="canvas1">
  <Ellipse Canvas.Left="10" Canvas.Top="20" Width="180" Height="60"
      Stroke="Red" StrokeThickness="3" >
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Offset="0" Color="LightBlue" />
        <GradientStop Offset="1" Color="DarkBlue" />
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Rectangle Width="180" Height="90" Canvas.Left="50" Canvas.Top="50">
    <Rectangle.LayoutTransform>
      <RotateTransform Angle="30" />
    </Rectangle.LayoutTransform>
    <Rectangle.Fill>
      <LinearGradientBrush>
        <GradientStop Offset="0" Color="Aquamarine" />
        <GradientStop Offset="1" Color="ForestGreen" />
      </LinearGradientBrush>
    </Rectangle.Fill>
    <Rectangle.Stroke>
      <LinearGradientBrush>
        <GradientStop Offset="0" Color="LawnGreen" />
        <GradientStop Offset="1" Color="SeaGreen" />
      </LinearGradientBrush>
    </Rectangle.Stroke>
  </Rectangle>
  <Button Canvas.Left="90" Canvas.Top="190" Content="Print" Click="OnPrint">
    <Button.Template>
      <ControlTemplate TargetType="Button">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
          </Grid.RowDefinitions>
          <Ellipse Grid.Row="0" Grid.RowSpan="2" Width="60"
              Height="40" Fill="Yellow" />
          <Ellipse Grid.Row="0" Width="52" Height="20"
              HorizontalAlignment="Center">
            <Ellipse.Fill>
              <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                <GradientStop Color="White" Offset="0" />
                <GradientStop Color="Transparent" Offset="0.9" />
              </LinearGradientBrush>
            </Ellipse.Fill>
          </Ellipse>
          <ContentPresenter Grid.Row="0" Grid.RowSpan="2"
              HorizontalAlignment="Center"
              VerticalAlignment="Center" />
        </Grid>
      </ControlTemplate>
    </Button.Template>
  </Button>
</Canvas>
```

In the `OnPrint` handler, the print job can be started by invoking the `PrintVisual` method of the `PrintDialog`. `PrintVisual` accepts any object that derives from the base class `Visual` (code file `PrintingDemo/MainWindow.xaml.cs`):

```
private void OnPrint(object sender, RoutedEventArgs e)
{
    var dlg = new PrintDialog();
    if (dlg.ShowDialog() == true)
    {
        dlg.PrintVisual(canvas1, "Print Demo");

    }
}
```

To programmatically print without user intervention, the `PrintDialog` classes from the namespace `System.Printing` can be used to create a print job and adjust print settings. The class `LocalPrintServer` provides information about print queues and returns the default `PrintQueue` with the `DefaultPrintQueue` property. You can configure the print job with a `PrintTicket`. `PrintQueue.DefaultPrintTicket` returns a default `PrintTicket` that is associated with the queue. The `PrintQueue` method `GetPrintCapabilities` returns the capabilities of a printer, and depending on those you can configure the `PrintTicket` as shown in the following code segment. After configuration of the print ticket is complete, the static method `PrintQueue.CreateXpsDocumentWriter` returns an `XpsDocumentWriter` object. The `XpsDocumentWriter` class was used previously to create an XPS document. You can also use it to start a print job. The `Write` method of the `XpsDocumentWriter` accepts not only a `Visual` or `FixedDocument` as the first argument but also a `PrintTicket` as the second argument. If a `PrintTicket` is passed with the second argument, the target of the writer is the printer associated with the ticket and thus the writer sends the print job to the printer:

```
var printServer = new LocalPrintServer();
PrintQueue queue = printServer.DefaultPrintQueue;
PrintTicket ticket = queue.DefaultPrintTicket;
PrintCapabilities capabilities =
    queue.GetPrintCapabilities(ticket);
if (capabilities.DuplexingCapability.Contains(
    Duplexing.TwoSidedLongEdge))
  ticket.Duplexing = Duplexing.TwoSidedLongEdge;
if (capabilities.InputBinCapability.Contains(InputBin.AutoSelect))
  ticket.InputBin = InputBin.AutoSelect;
if (capabilities.MaxCopyCount > 3)
  ticket.CopyCount = 3;
if (capabilities.PageOrientationCapability.Contains(
    PageOrientation.Landscape))
  ticket.PageOrientation = PageOrientation.Landscape;
if (capabilities.PagesPerSheetCapability.Contains(2))
  ticket.PagesPerSheet = 2;
if (capabilities.StaplingCapability.Contains(Stapling.StapleBottomLeft))
  ticket.Stapling = Stapling.StapleBottomLeft;
XpsDocumentWriter writer = PrintQueue.CreateXpsDocumentWriter(queue);
writer.Write(canvas1, ticket);
```

SUMMARY

In this chapter you learned how WPF capabilities can be used with documents, how to create flow documents that adjust automatically depending on the screen sizes, and fixed documents that always look the same. You've also seen how to print documents and how to send visual elements to the printer.

The next chapter continues with XAML, showing how it can be used with Windows 8 applications.

Professional WPF with C# and .NET 4.5

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

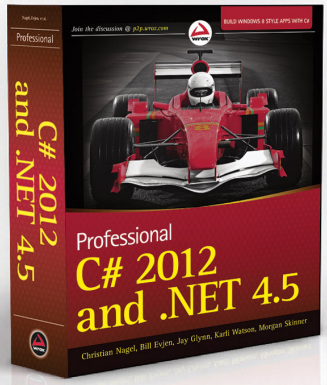
Published simultaneously in Canada

ISBN: 978-1-118-64438-6

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Go further with Wrox and Visual Studio 2012

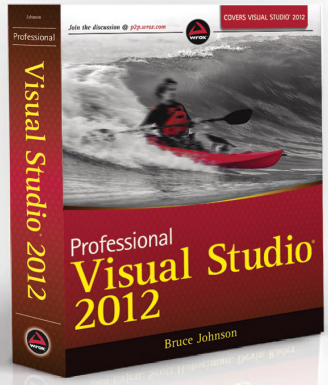


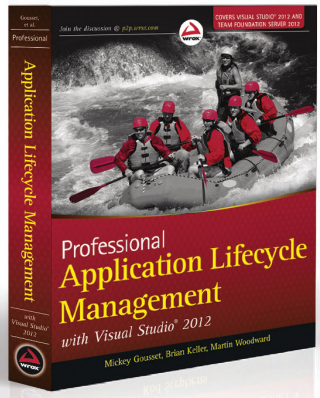
Professional C# 2012 and .NET 4.5
ISBN: 978-1-118-31442-5
By Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner

After a quick refresher on C# basics, the team of expert authors dives in to deliver unparalleled coverage on recent language and framework additions, as well as new test driven development and concurrent programming features. Ultimately, this book provides you with everything you need to know about C# 2012 and .NET 4.5 so that you can maximize the potential of these dynamic technologies.

Professional Visual Studio 2012
ISBN: 978-1-118-33770-7
By Bruce Johnson

This book is what you need to get up and running quickly on Visual Studio 2012. Written by a Microsoft Visual C# MVP, it guides you through the integrated development environment (IDE), showing you how to maximize each of the tools and features.





Professional Application Lifecycle Management with Visual Studio 2012
ISBN: 978-1-118-31408-1
By Mickey Gousset, Brian Keller, and Martin Woodward

Focused on the latest release of Visual Studio, this edition shows you how to use the Application Lifecycle Management (ALM) capabilities of Visual Studio 2012 to streamline software design, development, and testing. Divided into six main parts, this timely and authoritative title covers Team Foundation Server, stakeholder engagement, project management, architecture, software development, and testing. Whether serving as a stepby-step guide or a reference for designing software solutions, this book offers a nuts-and-bolts approach tousing Microsoft's flagship development tools to solve real-world challenges throughout the application lifecycle.