WPF Succinctly

by Buddy James

Syncfusion | Technology Resource Portal

WPF Succinctly

By

Buddy James

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Edited by
This publication was edited by Praveen Ramesh, director of development, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Buddy James a Microsoft Certified Solutions Developer from the Nashville, Tennessee area. He is a software engineer, an author, a blogger (www.refactorthis.net), a mentor, a thought leader, a technologist, a data scientist, and a husband. He enjoys working with design patterns, data mining, C#, WPF, Silverlight, WinRT, XAML, ASP.NET, Python, CouchDB, RavenDB, Hadoop, Android (MonoDroid), iOS (MonoTouch), and machine learning. He loves technology and loves to develop software, collect data, analyze the data, and learn from the data. When he is not coding, he's determined to make a difference in the world by using data and machine learning techniques. You can follow him on Twitter at @budbjames.

Introduction

Target audience

This book is intended for software developers with an understanding of the .NET Framework who want to get up and running quickly using Windows Presentation Framework or WPF. A basic understanding of the .NET Framework class library (FCL) and Windows Forms development is required. It is assumed that you know your way around the MSDN developer documentation at http://msdn.microsoft.com/en-us/library/ms123401.aspx.

Conventions used throughout this book

This book is meant to serve as a reference to allow seasoned developers to quickly get up to speed on developing in Microsoft's WPF. There are specific formats that you will see throughout this book to illustrate tips and tricks or other important concepts.

Note: This will identify things to note throughout the book.

Tip: This will identify tips and tricks throughout the book.

Please understand that this book is not meant to be a beginner's guide. The purpose of this book is to provide code examples to allow experienced .NET developers to get up and running with WPF as quickly as possible. I will provide basic descriptions of the important aspects of WPF as well as some code examples. With this information, you will be ready to experiment and learn this wonderful new technology.

Without further interruption, it's time to learn WPF!

Software requirements

To get the most out of this book and the included examples, you will need to have a version of the Visual Studio IDE installed on your computer. At the time of this writing, the most current available stable edition of Visual Studio Express is Visual Studio 2012. You can download Visual Studio Express 2012 for free here: http://www.microsoft.com/visualstudio/11/en-us/products/express.

Example code

This e-book will cover a lot of complex concepts in a small amount of time. As such, I will provide as much code as possible to assist in absorbing the ideas that are covered. You will find included in this text many examples of source code that illustrate the concepts covered in each chapter.

The code samples can be downloaded at https://github.com/SyncfusionSuccinctlyE-Books/WPF-Succinctly.

You can use the code to solidify your understanding of the subject matter. I've made it a point to keep the samples small and self-contained to conserve space. The code samples are cohesive units of code that clearly illustrate the concepts discussed. All examples are formatted with the default Visual Studio source code colors.

All examples in this book are written in C#.

Example code snippet

```
using System;


namespace WPFinctly.Examples
{
    public class Foo
    {
        public void Bar()
        {
            Console.WriteLine("Welcome to WPF !");
        }
    }
}
```

Additional resources

These additional resources will help you in your quest to get up to speed on XAML and WPF:

WPF resources (MSDN): http://msdn.microsoft.com/en-us/library/ms754130.aspx.

Introduction to WPF (MSDN): http://msdn.microsoft.com/en-us/library/aa970268.aspx.

Getting started WPF (MSDN): http://msdn.microsoft.com/en-us/library/ms742119.aspx.

refactorthis.net (my blog): http://www.refactorthis.net.

Chapter 1 WPF Origins

A brief history before WPF

The only way to fully understand where you are is to understand where you've been. WPF has completely revolutionized the way that we develop desktop applications. I'm not simply talking about writing applications, but core changes in the way that the operating system renders the GUI of your applications.

User32 and GDI/GDI+

GDI/GDI+ has provided drawing support to the Windows OS for things such as images, shapes, and text. These technologies are known to be relatively complicated to use and they provide poor performance. The User32 subsystem has provided the common look and feel for Windows elements such as buttons, text boxes, windows, etc.

Any framework that came along before WPF simply provided optimized wrappers to GDI/GDI+ and User32. These wrappers provided improved APIs, but the performance was still poor as the underlying technologies were the same.

And then there was DirectX

Microsoft created DirectX in an effort to provide a toolkit to allow developers to create video games that could bypass the limitations of the GDI/User32 subsystems. The DirectX API was extremely complex and therefore it was not ideal for creating line-of-business application user interfaces.

WPF to the rescue

WPF is the most comprehensive graphical technology change to the Windows operating system since Windows 95. WPF uses DirectX behind the scenes to render graphical content, allowing WPF to take advantage of hardware acceleration. This means that your applications will use your GPU (your graphics card's processor) as much as possible when rendering your WPF applications.

User32 in a limited capacity

WPF uses User32 in a limited capacity to handle the routing of your input controls (your mouse and keyboard, for instance). However, all drawing functions have been passed on through DirectX to provide monumental improvements in performance.

WPF and Windows 7/Windows Vista

WPF will perform best under Windows 7 or Windows Vista. This is because these operating systems allow the technology to take advantage of the Windows Display Driver Model (WDDM). WDDM allows scheduling of multiple GPU operations at the same time. It also provides a mechanism to page video card memory with normal system memory when the video card memory threshold is exceeded.

One of the first jobs of the WPF infrastructure is to evaluate your video card and provide a score or rating called a tier value. WPF recognizes three distinct tier values. The tier value descriptions follow, as provided by the Microsoft Developer Network documentation on WPF, available at https://msdn.microsoft.com/en-us/library/ms742196(v=vs.100).aspx:

Rendering Tier 0: No graphics hardware acceleration. All graphics features use software acceleration. The DirectX version level is lower than version 9.0.

Rendering Tier 1: Some graphics features use graphics hardware acceleration. The DirectX version level is higher than or equal to version 9.0.

Rendering Tier 2: Most graphics features use graphics hardware acceleration. The DirectX version level is higher than or equal to version 9.0.

The System.Windows.Media.RenderCapability.Tier property

To programmatically retrieve the rendering tier, you will need to use the static **System.Windows.Media.RenderCapability.Tier** property. You have to shift the property by 16 bits to access the tier value. These extra bits allow for extensibility, such as the possibility of storing information regarding support for other features or tiers in the future. To illustrate, we will create a WPF application. I will illustrate how to create a WPF project step by step, as this will be the starting point for each example throughout the entire book. Please note: I'm using Visual Studio 2010 Professional; however, the steps should be the same regardless of the version.

Open Visual Studio and select **New Project.**

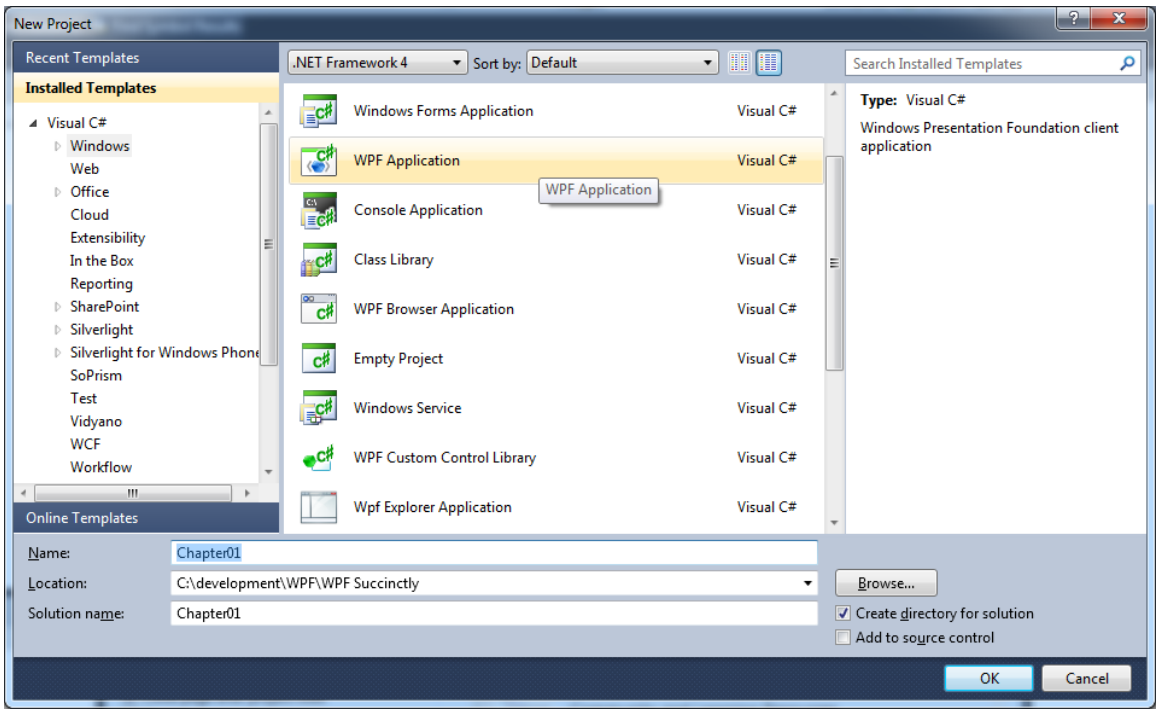

Figure 1: New Project Window

A WPF application contains a window as well as a code behind. The **MainWindow.xaml** file contains the XAML markup. The **MainWindow.xaml.cs** file contains the code-behind logic. The idea of a code behind should be familiar to you if you have any experience creating Windows Forms applications. The XAML markup may seem confusing at first, but don't worry, we will cover the markup in the chapters to come.

Here is a screenshot of the **MainWindow.xaml** file in design mode:

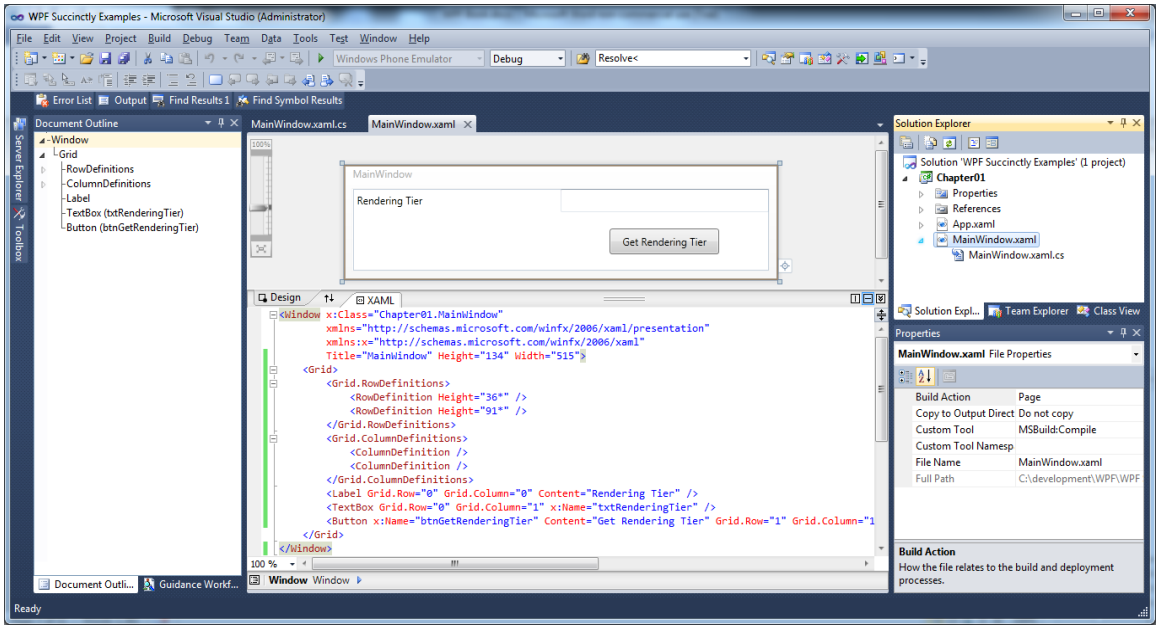

Figure 2: MainWindow.xaml File in Design Mode

Tip: In Design mode, you can split the designer to see the rendered output at the top and the XAML markup at the bottom.

To the right in the **Solution Explorer** you will find the **MainWindow.xaml** and **MainWindow.xaml.cs** code-behind files. First, we'll start with the XAML.

```
<Window x:Class="Chapter01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="134" Width="515">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="36*" />
            <RowDefinition Height="91*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Label Grid.Row="0" Grid.Column="0" Content="Rendering Tier" />
        <TextBox Grid.Row="0" Grid.Column="1" x:Name="txtRenderingTier" />
        <Button x:Name="btnGetRenderingTier" Content="Get Rendering Tier"
Grid.Row="1" Grid.Column="1" Width="130" Height="30"
Click="btnGetRenderingTier_Click" />
    </Grid>
</Window>
```

Note: The WPF Window can be compared to the WinForms Form object.

This is the XAML (Extensible Application Markup Language) that defines the user interface. XAML is an XML-based markup language, and is covered in detail in Chapter 2. Basically we have a label control, a textbox control, and a button. The grid control is used to control the layout of the other controls. The main unit of display that we are using in this example is a WPF Window.

Here are the contents of the MainWindow.xaml.cs code behind:

```
using System.Windows;
using System.Windows.Media;

namespace Chapter01
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml.
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void btnGetRenderingTier_Click(object sender, RoutedEventArgs e)
        {
            //Shift the value 16 bits to retrieve the rendering tier.
            int currentRenderingTier = (RenderCapability.Tier >> 16);

            switch (currentRenderingTier)
            {
                //DirectX version level less than 7.0.
                case 0:
                    txtRenderingTier.Text =
                                    string.Format("{0} No hardware acceleration.",
currentRenderingTier.ToString());
                    break;
                //DirectX version level greater than 7.0 but less than 9.0.
                case 1:
                    txtRenderingTier.Text =
                                    string.Format("{0} Partial hardware
acceleration.",
currentRenderingTier.ToString());
```

```
                break;
            //DirectX version level greater than or equal to 9.0.
            case 2:
                txtRenderingTier.Text =
                                string.Format("{0} Total hardware acceleration.",
                                                currentRenderingTier.ToString());
                break;
        }
    }
}
```

Tip: You can use the Rendering Tier to adjust the user interface. This way you can limit animations and other GPU-intensive operations to the graphics cards that support them.

Note: The code behind looks similar to the code behind of a WinForms form.

Chapter 2 Inside WPF

What is XAML?

At the heart of WPF is a new XML-style declarative markup language called Extensible Application Markup Language, or XAML (pronounced “Zammel”). The nature of XAML allows the user interface to be completely independent from the logic in the code behind. As such, it’s incredibly easy to allow a graphics designer to work in a design tool such as Expression Blend while a developer works to complete the business logic in Visual Studio. XAML is a vast and powerful language. This separation of design and development is one of the driving reasons that Microsoft created XAML and WPF.

XAML allows the instantiation of .NET objects via declarative markup. You can import namespaces to dictate the elements available to your markup. For instance, you can create a class in C# that models a customer and declare the customer class as an application-level resource. Then you can access this resource via XAML from anywhere in your application.

In a classic WinForms application, you would design your user interface in Visual Studio and the IDE would emit C# or VB.NET in a partial class that represented the layout of your code. Each control, be it a button, text box, or otherwise, would have a corresponding line of code.

In WPF and XAML, you use the IDE of your choice to create XAML markup. This markup isn’t translated into code. It’s serialized into a set of tags that are loaded on the fly to generate the UI of your application.

Elements represent objects, and attributes represent the objects’ property values.

Here’s an example:

```
1 <Window x:Class="Chapter01.ControlsExample"
2         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4         Title="ControlsExample" Height="300" Width="300">
5    <Grid>
6
7    </Grid>
8 </Window>
```

Elements as objects, attributes as properties

Line one is the definition of the Window object. Just like any other XML-based language, elements can be nested within other elements. The **x:Class** attribute specifies the name of the Window’s class. This is the class in which we will access the window programmatically from the code behind. Line two specifies the **xmlns** attribute, which represents a URI namespace. The URI looks like a web resource, though it’s not.

The http://schemas.microsoft.com/winfx/2006/xaml/presentation namespace URI defines all of the standard WPF controls. The **xmlns** is equivalent to the **using** statement in C#. Microsoft chose this naming scheme because by using a URI with the Microsoft domain, it's unlikely that any other developer will use the same URIs when defining namespaces. There are roughly twelve different standard WPF assembly namespaces and all of them start with **System.Windows**. By specifying the URI, the code will search each of the namespaces and automatically resolve that the Window element maps to the **System.Windows.Window** class. It will find the Grid resides in the **System.Windows.Controls** assembly namespace.

Since line 2 imports the namespace in which all of the WPF elements reside, we do not add a suffix. Line 3 represents another default WPF namespace that contains many utility classes that dictate how your XAML document is interpreted. We apply the **x** suffix to this namespace. Then to access elements of this namespace, we will prefix the element with **x:**. For example, **<x:MyElement>**.

The following explanation of the **xmlns** attribute is reproduced from the XAML Overview (WPF) page provided on the Microsoft Developer Network documentation on WPF at https://msdn.microsoft.com/en-us/library/ms752059(v=vs.110).aspx.

> The **xmlns** attribute specifically indicates the default XAML namespace. Within the default XAML namespace, object elements in the markup can be specified without a prefix. For most WPF application scenarios, and for almost all of the examples given in the WPF sections of the SDK, the default XAML namespace is mapped to the WPF namespace http://schemas.microsoft.com/winfx/2006/xaml/presentation. The **xmlns:x** attribute indicates an additional XAML namespace, which maps the XAML language namespace http://schemas.microsoft.com/winfx/2006/xaml.
>
> This usage of **xmlns** to define a scope for usage and mapping of a namespace is consistent with the XML 1.0 specification. XAML namespace is different from XML namespace only in that a XAML namespace also implies something about how the namespace's elements are backed by types when it comes to type resolution and parsing the XAML.
>
> Note that the **xmlns** attributes are only strictly necessary on the root element of each XAML file. The **xmlns** definitions will apply to all descendant elements of the root element (this behavior is again consistent with the XML 1.0 specification for **xmlns**). The **xmlns** attributes are also permitted on other elements underneath the root, and would apply to any descendant elements of the defining element. However, frequent definition or redefinition of XAML namespaces can result in a XAML markup style that is difficult to read.
>
> The WPF implementation of its XAML processor includes an infrastructure that has awareness of the WPF core assemblies. The WPF core assemblies are known to contain the types that support the WPF mappings to the default XAML namespace. This is enabled through configuration that is part of your project build file and the WPF build and project systems. Therefore, declaring the default XAML namespace as the default **xmlns** is all that is necessary to reference XAML elements that come from WPF assemblies.

XAML elements

In XAML, the objects that make up your UI are represented as elements. In the previous example, we had a root Window element that contained a grid control. The grid control is an object and is represented as an element. Please note that the grid is also a child element since it's contained within the opening and closing **Window** element tags. Also note that in some cases this parent/child relationship can be used to express object properties and values. Perhaps an example will help with this concept.

```
1 <Window x:Class="Chapter01.ControlsExample"
2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4       Title="ControlsExample" Height="300" Width="300">
5    <Grid>
6       <Button Width="100" Height="50" Name="btnTestButton">
7           Button Text
8       </Button>
9   </Grid>
10 </Window>
```

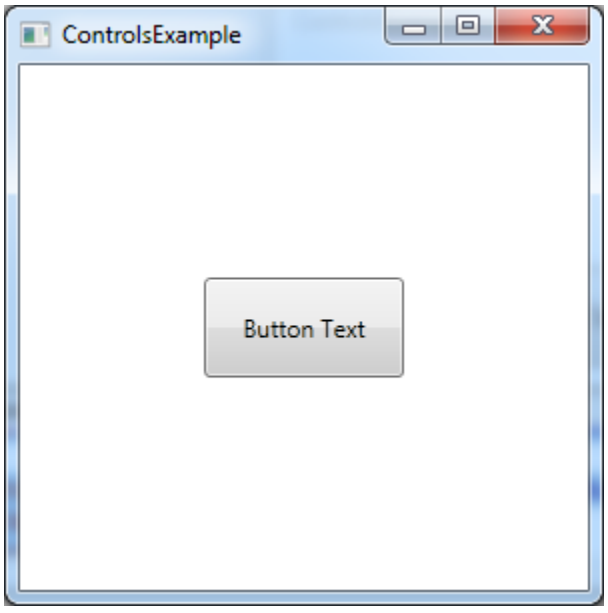

Figure 3: Rendered Window, Grid, and Button Elements

Output

As we look at the elements, you'll notice that we have a window, a grid, and a button. The grid is a child element of the window and the button is a child of the grid. The button has a text value in between the opening and closing tags. This automatically sets the **Content** property of the **Button** object to **Button Text**. This is a prime example of how an element's child can represent an object or a property value much like an element attribute.

XAML attributes

As I've stated in the previous section, attributes represent object properties. Perhaps one of the most important element attributes is the **name** attribute. The **name** attribute provides access to the object from the code in the code behind. If your element doesn't have a **name** attribute set, you won't be able to manipulate the object using C# or VB.NET code in the window code behind.

Due to the nature of XAML markup, all attribute values are specified as strings. There are many properties that must convert the string representation of an attribute value to a more complex data type. A perfect example of this is the **Background** property. The **Background** property will take a string name of a predefined set of colors and convert it to the proper **SolidBrush** object of the color upon compilation. This is achieved by a class called **System.ComponentModel.TypeConverter**.

Chapter 3 WPF Controls Overview

A solid understanding of the default controls available in .NET and WPF can save a lot of time in development. In fact, having a strong understanding of the basics is virtually required. Most WPF controls were designed to be extended. This is the beauty of WPF. You have the ability to work with controls at their lowest levels to tweak the elements that make up their appearance. Remember that controls are the basis for virtually any 2-D WPF application. They range from the basic label and button to complex tree views and grids. WPF allows nearly endless customization and nesting of controls. You will quickly notice that certain properties such as background, font size, etc., are shared between most WPF controls.

WPF controls

The following two lists are reproduced from the article WPF Controls - A Visual Quick Start by Josh Fischer, published on the Code Project website on November 21, 2008. The original article is available at codeproject.com/Articles/31137/WPF-Controls-A-Visual-Quick-Start.

The default WPF control types are:

- Button
- Text
- Shapes
- Containers
- Media
- Toolbars
- Scrolls
- Panels and lists
- Miscellaneous

Buttons

- **Button**: Indicates an area that can be clicked.
- **ToggleButton**: A button that remains selected after it is clicked.
- **CheckBox**: Indicates a positive, negative, or indeterminate state represented by a check box.
- **RadioButton**: Allows exclusive selection between a range of options.

Layout controls

The layout of controls is critical to an application's usability. Arranging controls based on fixed pixel coordinates may work in some instances. However, it can introduce problems, particularly when you need to support different screen resolutions or with different font sizes and resizing windows. WPF provides a rich set of built-in layout panels to help you avoid layout problems.

Grid

Some of the content in this section is reproduced from Christian Mosers' Grid Panel tutorial at wpftutorial.net/GridLayout.html.

The **Grid** is a layout panel that arranges its child controls in a tabular structure of rows and columns. Typically, its functionality is similar to the HTML table. A cell can contain multiple controls that can span over multiple cells and even overlap themselves. The alignment of the controls is defined by the **HorizontalAlignment** and **VerticalAlignment** properties of the **Grid Control**.

A grid has one row and column by default. To create any additional rows and columns, you have to add **RowDefinition** items to the **RowDefinitions** collection and **ColumnDefinition** items to the **ColumnDefinitions** collection.

The following figure shows a grid with four rows and two columns.

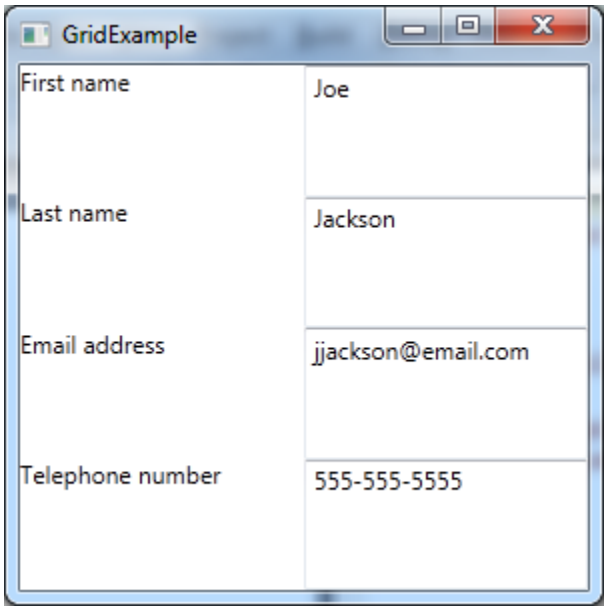

Figure 4: Grid Layout Panel

And here is the XAML markup:

```
<Window x:Class="WPF.GridExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="GridExample" Height="300" Width="300">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
```

```
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition  />
            <RowDefinition  />
            <RowDefinition />
            <RowDefinition  />
        </Grid.RowDefinitions>

        <TextBlock Grid.Column="0" Grid.Row="0" Text="First name" />
        <TextBox Grid.Column="1" Grid.Row="0" Text="Joe" />

        <TextBlock Grid.Column="0" Grid.Row="1" Text="Last name" />
        <TextBox Grid.Column="1" Grid.Row="1" Text="Jackson" />

        <TextBlock Grid.Column="0" Grid.Row="2" Text="Email address" />
        <TextBox Grid.Column="1" Grid.Row="2" Text="jjackson@email.com" />

        <TextBlock Grid.Column="0" Grid.Row="3" Text="Telephone number" />
        <TextBox Grid.Column="1" Grid.Row="3" Text="555-555-5555" />
    </Grid>
</Window>
```

The following table shows the methods of sizing grid columns and rows:

Table 1: Methods for Sizing Grid Columns and Rows.

Fixed	Fixed size of logical units (1/96 inch).
Auto	Takes as much space as needed by the contained control.
Star (*)	Takes as much space as available (after filling all auto and fixed sized columns), proportionally divided over all star-sized columns. So 3*/5* means the same as 30*/50*. Remember that star sizing does not work if the grid size is calculated based on its content.

StackPanel

Some of the content in this section is reproduced from Christian Mosers' WPF StackPanel tutorial at wpftutorial.net/StackPanel.html.

In WPF, the **StackPanel** is a simple and useful layout panel. It stacks the child elements below or beside each other depending on the value of the **Orientation** property. The **Orientation** property's possible values are **Vertical** and **Horizontal**. The **StackPanel** is especially useful when creating a list of controls. All WPF **ItemsControls** like **ComboBox**, **ListBox**, or **Menu** use a **StackPanel** as their internal layout panel.

For example:

```
<Window x:Class="WPFExamples.StackPanelExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="StackPanel"
Height="300" Width="300">
    <StackPanel>
        <TextBlock Margin="10" FontSize="20">How do you like your coffee?</TextBlock>
        <Button Margin="10">Black</Button>
        <Button Margin="10">With milk</Button>
        <Button Margin="10">Latte macchiato</Button>
        <Button Margin="10">Cappuccino</Button>
    </StackPanel>
</Window>
```

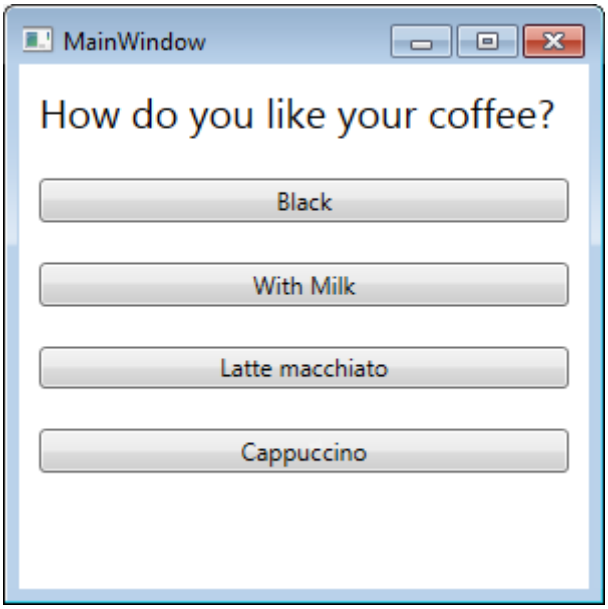

Figure 5: StackPanel Layout Panel

As you can see in the previous figure, the text and buttons are stacked on top of each other when using the **StackPanel** as the container. The default value of the **Orientation** property is **Vertical**.

I find that the **StackPanel** can be very effective for lining up confirmation buttons at the bottom of a dialog window.

For instance, take a look at the following example:

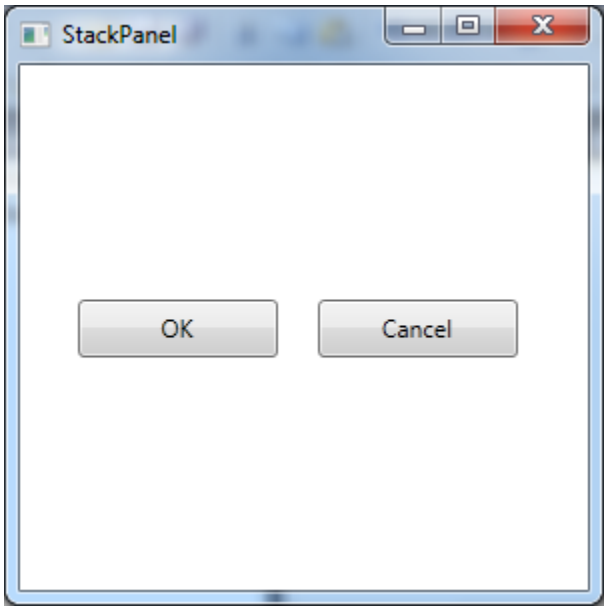

Figure 6: Horizontally Oriented StackPanel

```
<Window x:Class="WPFExamples.StackPanelExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StackPanel" Height="300" Width="300">
    <StackPanel Orientation="Horizontal" Height="49" Width="247">
        <Button Width="100" Margin="10">OK</Button>
        <Button Width="100" Margin="10">Cancel</Button>
    </StackPanel>
</Window>
```

DockPanel

Some of the content in this section is reproduced from Christian Mosers' WPF Dock Panel tutorial at wpftutorial.net/DockPanel.html.

The **DockPanel** defines an area where you can arrange your child elements either horizontally or vertically relative to each other. The dock panel is a layout panel that provides easy docking of the elements to the left, right, top, bottom, or even center of the panel. The dock-side of an element is defined by the attached property **DockPanel.Dock**. To dock an element to the center of the panel, the element must be the last child of the panel and the **LastChildFill** property must be set to **true**.

For example:

```
<DockPanel LastChildFill="True">
    <Button Content="Dock=Top" DockPanel.Dock="Top"/>
    <Button Content="Dock=Bottom" DockPanel.Dock="Bottom"/>
    <Button Content="Dock=Left"/>
    <Button Content="Dock=Right" DockPanel.Dock="Right"/>
    <Button Content="LastChildFill=True"/>
</DockPanel>
```

WrapPanel

Some of the content in this section is reproduced from Christian Mosers' WPF Wrap Panel tutorial at wpftutorial.net/WrapPanel.html.

The **WrapPanel** is similar to the **DockPanel**; however, it does not stack all child elements to one row. It wraps them to new lines if no space is left in the width of the container. The orientation can be set to **Horizontal** or **Vertical**. The **WrapPanel** can also be used to arrange tabs of a tab control, menu items in a toolbar, or items in a Windows Explorer-like list.

```
<WrapPanel Orientation="Horizontal">
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
    <Button Content="Button" />
</WrapPanel>
```

Canvas

Some of the content in this section and the one that follows is reproduced from Christian Mosers' WPF Canvas Panel tutorial at wpftutorial.net/Canvas.html.

The **Canvas** is the most basic layout panel in WPF; it allows you to position its child elements using exact coordinates. The **Canvas** container can be used as a surface for drawing shapes and animations.

Typically, the **Canvas** is used to group 2-D graphic elements together. The **Canvas** is not a great container for other user interface elements because it does not perform any resizing optimizations when its parent container is resized. The position-based layout is problematic in resizing scenarios. Observe the following example in which I've placed a **Rectangle**, an **Ellipse**, and a **Path** element.

```
<Canvas>
    <Rectangle Canvas.Left="40" Canvas.Top="31" Width="63" Height="41" Fill="Blue"  />
    <Ellipse Canvas.Left="130" Canvas.Top="79" Width="58" Height="58" Fill="Blue"  />
    <Path Canvas.Left="61" Canvas.Top="28" Width="133" Height="98" Fill="Blue"
          Stretch="Fill" Data="M61,125 L193,28"/>
</Canvas>
```

Example to override the z-order of elements

Normally the z-order of elements inside a **Canvas** element is specified by the order in XAML, but you can override the natural z-order by explicitly defining the **Canvas.ZIndex** property on the element. See the example that follows.

```
<Canvas>
    <Ellipse Fill="Green" Width="60" Height="60" Canvas.Left="30" Canvas.Top="20"
             Canvas.ZIndex="1"/>
    <Ellipse Fill="Blue"  Width="60" Height="60" Canvas.Left="60" Canvas.Top="40"/>
</Canvas>
```

Layout best practices

Some of the content in this list is reproduced from Christian Mosers' Introduction to WPF Layout tutorial at wpftutorial.net/LayoutProperties.html.

- Avoid fixed positions; instead, use the **Alignment** properties in combination with **Margins** to position elements in a panel.
- Avoid fixed sizes. Set the **Width** and **Height** of elements to **Auto** whenever possible.
- Don't use the Canvas panel to specify the layout of common controls; instead use it to arrange vector graphics.

- Use a StackPanel to lay out dialog confirmation buttons on a dialog.
- Use a Grid to lay out complex user interfaces and data entry forms.

TextBlock

The **TextBlock** control represents a block of text that cannot be edited. You can use the **TextBlock** much like a label control. The **TextBlock** is also used in control templates to display text.

TextBox

The TextBox control has been around as long as the Windows operating system. It is the main control used to capture text entry from a user. WPF takes data entry a step further with a concept known as data binding. The TextBox supports events and properties related to the entry of data from the keyboard.

Animation

Animated transitional view example

WPF provides a very rich API related to animations. Animation can be a complex topic and though it's great for enriching your user interface, it does not change the way in which line-of-business applications are created. Thus, instead of covering every aspect of animation in WPF, I plan to provide an example application that will pique your interest so that you can study the topic on your own. Here is an article to get you started: MSDN Animation Overview, http://msdn.microsoft.com/en-us/library/ms752312.aspx.

In this section, I will show you how to build a "transitional view" switching mechanism that uses WPF animation to add some flair to your applications. The effect is accomplished by implementing your user interface views as WPF user controls that inherit from a class that I call **TransitionControl**. The **TransitionControl** will provide the methods that will display an animated effect as you switch between **UserControls**. Here is the code:

TransitionControl.xaml

```
<UserControl x:Class="ScreenSlideTransitionExample.TransitionControl"
             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"
             >

</UserControl>
```

TransitionControl.xaml.cs

```
using System;
using System.Windows.Controls;

namespace ScreenSlideTransitionExample
{
    /// <summary>
    /// Interaction logic for ScreenOne.xaml
    /// </summary>
    public partial class TransitionControl : UserControl
    {
        public MainWindow ParentWindow { get; set; }
        public TransitionControl CurrentScreen { get; set; }

        public TransitionControl(MainWindow parent)
        {
            this.ParentWindow = parent;
        }

        public void ChangeScreen(TransitionControl screen)
        {
            if (screen == null)
                throw new ArgumentNullException("Unable to navigate to next screen. A null
reference section occurred");

            this.CurrentScreen = screen;

            this.ParentWindow.ChangeContent(screen);
        }
    }
}
```

MainWindow.xaml

```
<Window x:Class="ScreenSlideTransitionExample.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Canvas x:Name="TransitionContainer" />
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Animation;
```

```
namespace ScreenSlideTransitionExample
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        readonly Duration _animationDuration = new
Duration(TimeSpan.FromSeconds(1.0));


        public MainWindow()
        {
            InitializeComponent();
            ChangeContent( new ScreenOne( new TransitionControl( this ) ) );
        }



        DoubleAnimation CreateDoubleAnimation(double from, double to, EventHandler
completedEventHandler)
        {
            DoubleAnimation doubleAnimation = new DoubleAnimation(from, to,
_animationDuration);


            if (completedEventHandler != null)
            {
                doubleAnimation.Completed += completedEventHandler;
            }


            return doubleAnimation;
        }


        void SlideAnimation(UIElement newContent, UIElement oldContent, EventHandler
completedEventHandler)
        {


            double leftStart = Canvas.GetLeft(oldContent);
            Canvas.SetLeft(newContent, leftStart - Width);
```

```
            TransitionContainer.Children.Add(newContent);


            if (double.IsNaN(leftStart))
            {
                leftStart = 0;
            }


            DoubleAnimation outAnimation = CreateDoubleAnimation(leftStart, leftStart
+ Width, null);
            DoubleAnimation inAnimation = CreateDoubleAnimation(leftStart - Width,
leftStart, completedEventHandler);


            oldContent.BeginAnimation(Canvas.LeftProperty, outAnimation);
            newContent.BeginAnimation(Canvas.LeftProperty, inAnimation);
        }


        public void ChangeContent(UIElement newContent)
        {
            if (TransitionContainer.Children.Count == 0)
            {
                TransitionContainer.Children.Add(newContent);
                return;
            }


            if (TransitionContainer.Children.Count == 1)
            {
                TransitionContainer.IsHitTestVisible = false;
                UIElement oldContent = TransitionContainer.Children[0];


                EventHandler onAnimationCompletedHandler = delegate(object sender,
EventArgs e)
                {
                    TransitionContainer.IsHitTestVisible = true;
                    TransitionContainer.Children.Remove(oldContent);
                    if (oldContent is IDisposable)
                    {
```

```
                    (oldContent as IDisposable).Dispose();
                }
                oldContent = null;
            };

            SlideAnimation(newContent, oldContent, onAnimationCompletedHandler);
        }
    }
}
```

ScreenOne.xaml

```
<UserControl x:Class="ScreenSlideTransitionExample.ScreenOne"
             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" Height="350" Width="525"
             Background="BlanchedAlmond">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBlock Text="Screen one" FontFamily="Verdana"  FontSize="30" Grid.Row="0"
/>
        <Button Name="btnChangeContent" Grid.Row="2" Content="Change content"
Click="btnChangeContent_Click" Margin="267,59,140,12" />
    </Grid>
</UserControl>
```

ScreenOne.xaml.cs

```
using System.Windows;
using System.Windows.Controls;

namespace ScreenSlideTransitionExample
{
    /// <summary>
    /// Interaction logic for ScreenOne.xaml
    /// </summary>
    public partial class ScreenOne : UserControl
    {
        private TransitionControl _transitionControl;
```

```
        public ScreenOne(TransitionControl transitionControl)
        {
            InitializeComponent();

            _transitionControl = transitionControl;
        }

        private void btnChangeContent_Click(object sender, RoutedEventArgs e)
        {
            _transitionControl.ParentWindow.ChangeContent(
                new ScreenTwo( new TransitionControl( _transitionControl.ParentWindow )
) );
        }
    }
}
```

ScreenTwo.xaml

```
<UserControl x:Class="ScreenSlideTransitionExample.ScreenTwo"
             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" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBlock Text="Screen two" FontFamily="Verdana"  FontSize="30" Grid.Row="0"
/>
        <Button Name="btnChangeContent" Content="Change content"
Margin="241,68,159,72" Click="btnChangeContent_Click" />
    </Grid>
</UserControl>
```

ScreenTwo.xaml.cs

```
using System.Windows;
using System.Windows.Controls;

namespace ScreenSlideTransitionExample
```

```
{
    /// <summary>
    /// Interaction logic for ScreenTwo.xaml
    /// </summary>
    public partial class ScreenTwo : UserControl
    {
        private TransitionControl _transCtrl;

        public ScreenTwo(TransitionControl transitionControl)
        {
            InitializeComponent();
            _transCtrl = transitionControl;
        }

        private void btnChangeContent_Click(object sender, RoutedEventArgs e)
        {
            var transCtrl = new TransitionControl(_transCtrl.ParentWindow);
            var screenOne = new ScreenOne(transCtrl);

            _transCtrl.ParentWindow.ChangeContent(screenOne);
        }
    }
}
```

Type Converters

The **System.ComponentModel.TypeConverter** class provides a unified way of converting XAML string attribute values to corresponding object value types. The MSDN documentation for the **TypeConverter** class can be found at http://msdn.microsoft.com/en-us/library/system.componentmodel.typeconverter.aspx.

Here is an example:

```
1 <Window x:Class="Chapter._02.MainWindow"
2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4       Title="MainWindow" Height="350" Width="525">
5   <Grid>
6       <Grid.RowDefinitions>
7           <RowDefinition />
8           <RowDefinition />
9       </Grid.RowDefinitions>
10      <Grid.ColumnDefinitions>
```

```
11          <ColumnDefinition />
12          <ColumnDefinition />
13      </Grid.ColumnDefinitions>
14      <TextBlock x:Name="txtDisplay" Text="This is a textblock control. Can't be
15       edited." Grid.Column="0" Grid.Row="0" />
16      <Button x:Name="btnDisplayButton" Grid.Column="1" Grid.Row="0"
Background="Blue"
17       Content="Button" Width="100" Height="40" />
18      <TextBlock x:Name="txtHexRGBColor" Text="Background as a three digit hex value
19       RGB." Grid.Column="0" Grid.Row="1" />
20      <Button x:Name="btnHexRGBColor" Grid.Column="1" Grid.Row="1" Background="#299"
21       Content="Button"  Width="100" Height="40" />
22  </Grid>
23 </Window>
```

Note: Observe the grid attributes on the buttons and text blocks. This is how we position controls in our user interface.

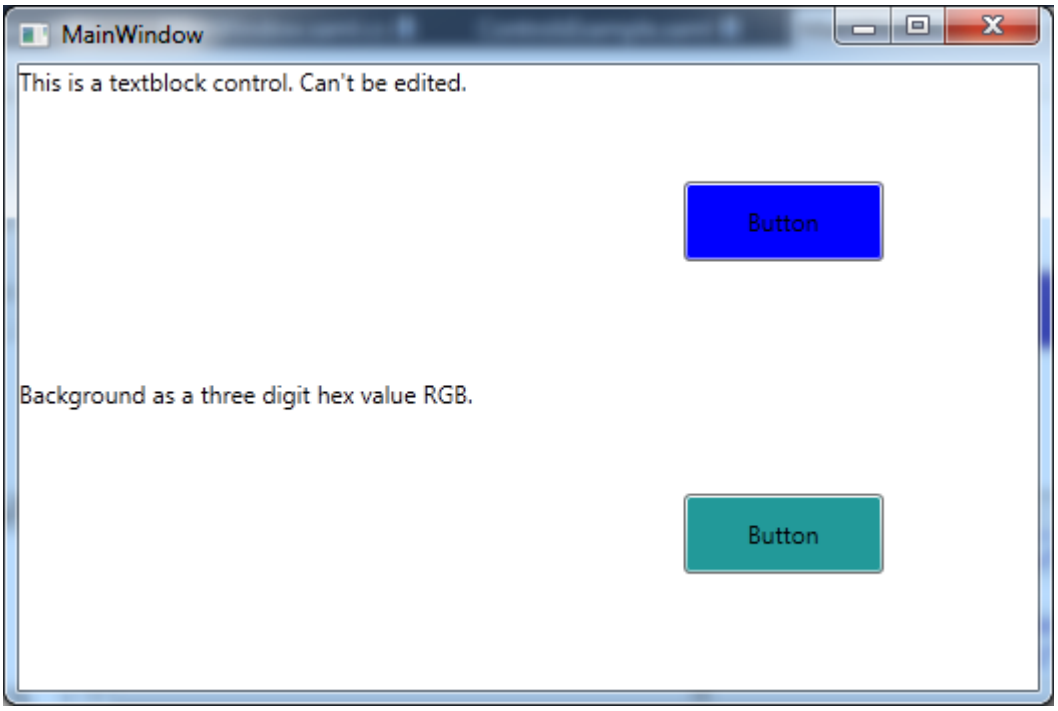

Figure 7: Using TypeConverters

As you can see, we have a window that contains two text blocks and two button objects inside of a grid. The grid has two columns and two rows defined. We've specified the **Background** attribute on each of the button controls. In the first button, we simply specified the string name of the color we wanted to use as the background. For the second button we specified the RGB values as a three digit hex value. The **System.Drawing.ColorConverter** class is responsible for providing this functionality.

Tip: You can create your own TypeConverter class by implementing the IValueConverter and inheriting from the TypeConverter base class.

Implementing a TypeConverter

The **TypeConverter** class existed before XAML was added to the .NET Framework. The purpose of the **TypeConverter** class, pre-XAML, was to provide string conversions for property dialogs in WinForms applications. As you become more proficient with WPF and XAML, you may find the need to implement your own **TypeConverter** class. To do this, you create a class that derives from the **TypeConverter** base class.

For the purposes of XAML value conversion, there are four relevant members that must be overridden.

Table 2: TypeConverter Methods

Method	Method Description
CanConvertTo()	A support method that returns a Boolean indicating whether the value can be converted to the specified type.
CanConvertFrom()	A support method that returns a Boolean indicating whether the value can be converted from a specified type.
ConvertTo()	Converts the given value object to the specified type.
ConvertFrom()	Converts the given value to the type of this converter.

You must apply the **TypeConverterAttribute** to the class that implements **TypeConverter**. Here is an example of the use of the attribute [**TypeConverter(typeof(MyCustomConverter))**].

Person.cs

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

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using TypeConverterExample.TypeConverters;

namespace TypeConverterExample
{
    /// <summary>
    /// Represents a person's information.
    /// </summary>
    [TypeConverter(typeof(PersonInfoTypeConverter))]
    public class Person
    {
        /// <summary>
        /// Represents the person's name.
        /// </summary>
        public string Name { get; set; }
        /// <summary>
        /// Represents the person's age.
        /// </summary>
        public int Age { get; set; }
        /// <summary>
        /// Represents the person's birth date.
        /// </summary>
        public DateTime BirthDate { get; set; }

        /// <summary>
        /// The default constructor.
        /// </summary>
        public Person(){}

        /// <summary>
        /// An overloaded constructor to allow for easy creation of a person and their
properties.
        /// </summary>
        /// <param name="name">The person's name.</param>
```

```
        /// <param name="age">The person's age.</param>
        /// <param name="birthDate">The person's birth date.</param>
        public Person(string name, int age, DateTime birthDate)
        {
            if (string.IsNullOrEmpty(name))
                throw new ArgumentNullException("name");

            if (age <= 0)
                throw new ArgumentException("age must be greater than 0");

            this.Name = name;
            this.Age = age;
            this.BirthDate = birthDate;
        }

        /// <summary>
        /// This method will take the details.
        /// </summary>
        /// <param name="xamlPropertyValue">Represents the string value of the XAML
property.</param>
        /// <returns>A person object to the caller.</returns>
        public static Person Parse(string xamlPropertyValue)
        {
            if (string.IsNullOrEmpty(xamlPropertyValue))
                return new Person();

            string[] propertyValues = xamlPropertyValue.Split(',');

            if (propertyValues.Length != 3)
                throw new FormatException("Please specify the Name, Age, and Birthdate
of the person.");

            Person returnValue = new Person();
            returnValue.Name = propertyValues[0];

            try
            {
```

```
                int personAge = int.Parse(propertyValues[1]);
                returnValue.Age = personAge;
            }
            catch (Exception ex)
            {
                throw new Exception(string.Format("Unable to parse the person's age.
The value = {0}. The error message = {1}",
                    propertyValues[1], ex.ToString()));
            }

            try
            {
                DateTime personBirthDate = DateTime.Parse(propertyValues[2]);
                returnValue.BirthDate = personBirthDate;
            }
            catch (Exception ex)
            {
                throw new Exception(string.Format("Unable to parse the person's birth
date. The value = {0}. The error message = {1}",
                    propertyValues[2], ex.ToString()));
            }

            return returnValue;
        }

        public override string ToString()
        {
            return string.Format("{0}, {1}, {2}",
                this.Name, this.Age.ToString(), this.BirthDate.ToString());
        }
    }
}
```

PersonInfoTypeConverter.cs

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

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel;
namespace TypeConverterExample.TypeConverters
{
    /// <summary>
    /// Represents an ITypeConverter implementation that will take the PersonInfoControl's
XAML string values.
    /// and convert them into a PersonInfo object.
    /// </summary>
    public class PersonInfoTypeConverter : TypeConverter
    {
        /// <summary>
        /// Determines if the source type can be converted.
        /// </summary>
        /// <param name="context">The type description context.</param>
        /// <param name="sourceType">The type to convert to Person.</param>
        /// <returns>True if the sourceType can be converted.</returns>
        public override bool CanConvertFrom(ITypeDescriptorContext context, Type
sourceType)
        {
            //Since we are dealing with a XAML property, the value must be of type string.
            if (sourceType == typeof (string))
                return true;

            //Otherwise, try the base implementation.
            return base.CanConvertFrom(context, sourceType);
        }

        /// <summary>
        /// Determines whether the destination type can be converted.
        /// </summary>
        public override bool CanConvertTo(ITypeDescriptorContext context, Type
destinationType)
        {
            return base.CanConvertTo(context, destinationType);
```

```
        }

        /// <summary>
        /// Converts from the Person value to the XAML string representation.
        /// </summary>
        /// <param name="context">Type description context.</param>
        /// <param name="culture">The string culture for localization purposes.</param>
        /// <param name="value">The value to convert.</param>
        /// <returns>A person object with the properties specified in the value
string.</returns>
        public override object ConvertFrom(ITypeDescriptorContext context,
            System.Globalization.CultureInfo culture, object value)
        {

            if (value == null)
                throw new ArgumentNullException("value");

            try
            {
                Person convertedPerson = Person.Parse((string)value);
                return convertedPerson;
            }
            catch (Exception ex)
            {
                //Log the exception and try the base implementation.
                return base.ConvertTo(context, culture, value, destinationType);
            }
        }

        /// <summary>
        /// Converts a person object to the XAML representation.
        /// </summary>
        /// <param name="context">The type description context.</param>
        /// <param name="culture">The string culture for localization.</param>
        /// <param name="value">The XAML string representing the person.</param>
        /// <param name="destinationType">The type to convert to (Person).</param>
        /// <returns></returns>
```

```
        public override object ConvertTo(ITypeDescriptorContext context,
System.Globalization.CultureInfo culture, object value, Type destinationType)
        {
            if (value == null)
                throw new ArgumentNullException("value");


            if (value is Person)
            {
                var personObject = (Person) value;
                var xamlAttributeValue = personObject.ToString();


                return xamlAttributeValue;
            }
            return base.ConvertFrom(context, culture, value);


        }
    }
}
```

The core WPF class hierarchy

Object: The base class for all .NET classes.

DispatcherObject: This is a base class for any object that can only be accessed on the thread for which the object was created. Most WPF classes derive from DisaptcherObject, so naturally the ones that do are not thread-safe.

DependencyObject: This is the base class for any object that has the ability to support DependencyProperties.

DependencyProperties: The base class provides the GetValue and SetValue methods, which DependencyProperties must use in order to work.

Freezable: The base class for all objects that can be “frozen” into a read-only state. Once frozen, they cannot be unfrozen.

Visual: The base class for all objects that have a visual representation.

UIElement: The base class for all visual objects with support for routed events, command binding, layout, and focus support.

ContentElement: A base class similar to UIElement but for pieces of content that don't have rendering behavior on their own. Instead, ContentElements are hosted in a Visual-derived class to be rendered on the screen.

FrameworkElement: The base class that adds support for styles, data binding, resources, and a few common mechanisms for Windows-based controls such as tooltips and context menus.

FrameworkContentElement: The analog to FrameworkElement for content.

Control: The base class for familiar controls such as Button, ListBox, and StatusBar. Control adds many properties to its FrameworkElement base class, such as Foreground, Background, and FontSize. Controls also support templates that enable you to completely replace their visual tree.

Resource dictionaries

Whether you are designing a custom type converter class or you are writing a **ViewModel** for data-binding to implement the MVVM pattern (we will discuss this pattern in detail later in this book), you have to tell XAML of the existence of your class. You can do so by creating a resource dictionary. You can create a resource dictionary at the Application, Window, and UserControl levels. A XAML dictionary contains elements known as resources. This essentially allows you to import your own C# objects to use in XAML data binding. With this approach, it's possible to write an application without writing a single line of code in your code behind.

Chapter 4 WPF Applications

Navigation-based Windows WPF applications

WPF applications can be designed to use a more navigation-based user interface much like a website. To accomplish this design, you will generally use pages instead of window objects. Pages can be hosted in two built-in navigation containers: **Frames** and **NavigationWindows**. They provide the ability to browse from page to page as well as provide a journal mechanism to keep track of the navigation history.

Much like its name implies, a **Frame** container works a lot like the IFRAME element of a website. A **NavigationWindow** is a top-level element where a frame can take up a rectangular region within another element. The **NavigationWindow** has a bar across the top with back and forward buttons by default, much like a web browser. This bar can be shown or hidden by changing the **ShowsNavigationUI** property of the parent page markup.

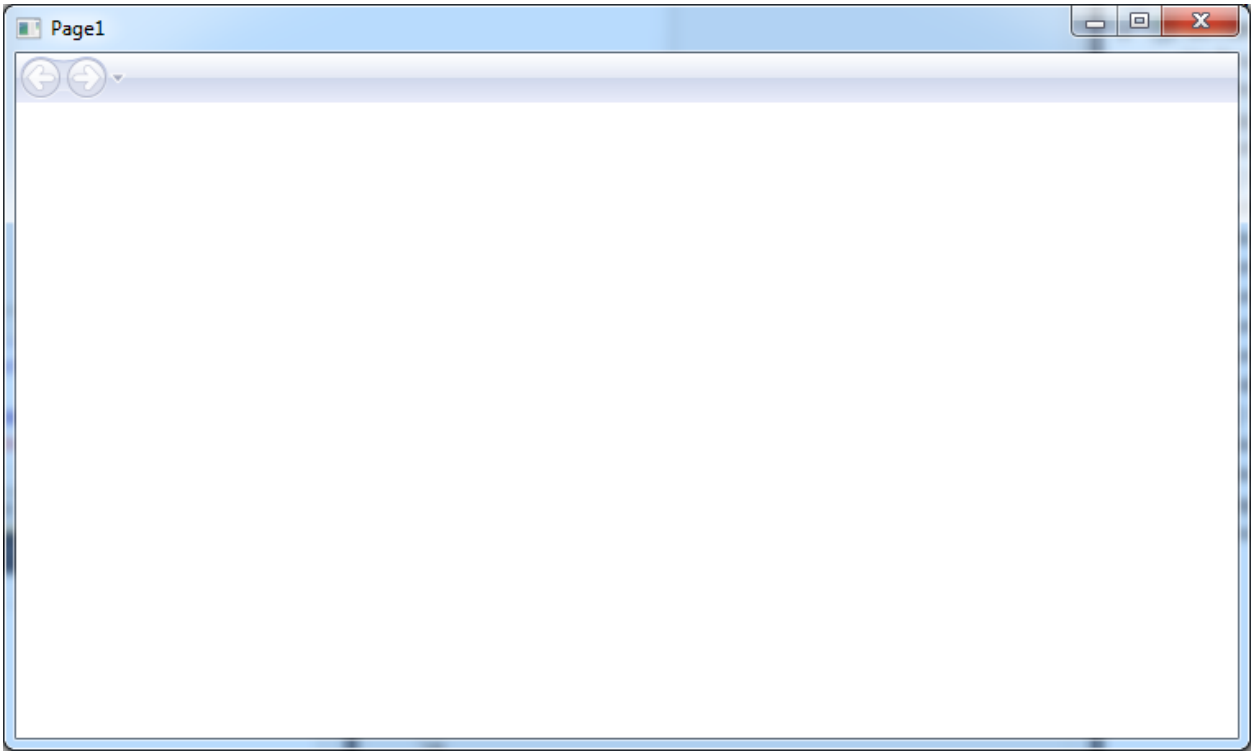

Figure 8: NavigationWindow with Navigation Bar

Data binding

Some of the content in this section is reproduced from the Data Binding and LINQ to DataSet documentation at docs.microsoft.com/en-us/dotnet/framework/data/adonet/data-binding-and-linq-to-dataset, and the Data Binding Overview documentation at docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview.

Data binding is a process that establishes a connection between the application UI and the business logic. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data reflect changes automatically. Data binding also means that if an outer representation of the data in an element changes, then the underlying data can automatically be updated to reflect the change. One example is if a user edits the value in a **TextBox** element, then the underlying data value is automatically updated to reflect that change.

For WPF, this concept is expanded to include the binding of a broad range of properties to a variety of data sources. Also, WPF dependency properties of elements can be bound to CLR objects, ADO.NET objects, other elements, or XML data.

Basic data binding concepts

Much of the content in this section and the next is reproduced from the Data Binding Overview documentation at docs.microsoft.com/en-us/dotnet/framework/wpf/data/data-binding-overview.

Regardless of what element you are binding and the nature of your data source, each binding always follows the model illustrated by the following figure:

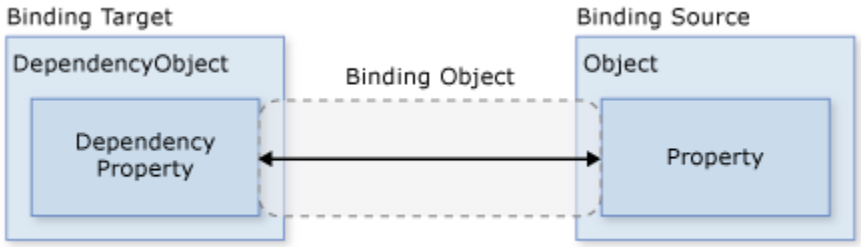

Figure 9: Data Binding Diagram

As illustrated by the previous figure, data binding is essentially the bridge between your binding target and the binding source. The figure demonstrates the following fundamental WPF data binding concepts:

- Typically, each binding has four components: a binding target object, a target property, a binding source, and a path to the value in the binding source to use. For example, if you want to bind the content of a **TextBox** to the **Name** property of an **Employee** object, your target object is the **TextBox**, the target property is the **Text** property, the value to use is **Name**, and the source object is the **Employee** object.

- The target property must be a dependency property. In **UIElement** properties, most are dependency properties, and most dependency properties, except read-only ones, support data binding by default. Only the **DependencyObject** types can define dependency properties, and all **UIElements** derive from **DependencyObject**.

- Although not specified in the figure, it should be noted that the binding source object is not restricted to being a custom CLR object. To provide some examples, your binding source may be a **UIElement**, any list object, a CLR object that is associated with ADO.NET data or web services, or an **XmlNode** that contains your XML data.

Data flow direction

As mentioned previously and indicated by the arrow in the previous figure, the data flow of a binding can go from the binding target to the binding source, (e.g., the source value changing when a user edits the value of a **TextBox**), from the binding source to the binding target (e.g., a **TextBox**'s content updating when changes in the binding source occur if the binding source provides the proper notifications), or both. If you want your application to enable users to change the data and propagate it back to the source object, you can control this behavior by setting the **Mode** property of your **Binding** object. The following figure illustrates the different types of data flow:

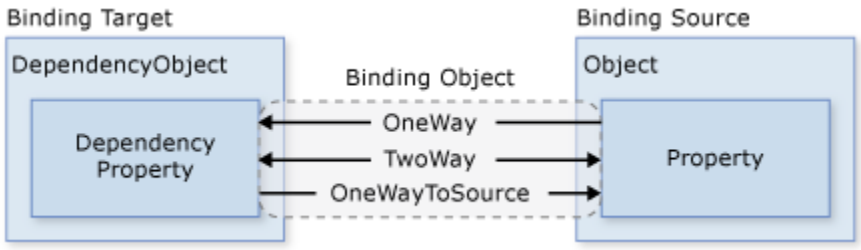

Figure 10: Data Flow in Data Binding

- The **OneWay** binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property. This type of binding is appropriate if the control being bound is implicitly read-only. For instance, you may bind to a source such as a stock ticker, or perhaps your target property has no control interface provided for making changes, such as a data-bound **TextBlock** control. If there is no need to monitor the changes of the target property, the **OneWay** binding mode avoids the overhead of the **TwoWay** binding mode.

- The **TwoWay** binding causes changes to either the source property or the target property to automatically update the other. This type of binding is appropriate for editable forms or other fully interactive user interface scenarios. Most properties default to **OneWay** binding, but some dependency properties (typically properties of user-editable controls such as the **Text** property of **TextBox** and the **IsChecked** property of **CheckBox**) default to **TwoWay** binding. A programmatic way to determine whether a dependency property binds one-way or two-way by default is to get the property metadata using **GetMetadata** and then check the Boolean value of the **BindsTwoWayByDefault** property.

- **OneWayToSource** is the reverse of **OneWay** binding; it updates the source property when the target property changes. One example scenario is if you only need to re-evaluate the source value from the UI.

- **OneTime** binding is not illustrated in Figure 10. It causes the source property to initialize the target property, but subsequent changes do not propagate. This means that if the data context undergoes a change or the object in the data context changes, then the change is not reflected in the target property. This type of binding is appropriate if you are using data where either a snapshot of the current state is appropriate to use or the data is truly static. This type of binding is also useful if you want to initialize your target property with some value from a source property and the data context is not known in advance. This is essentially a simpler form of **OneWay** binding that provides better performance in cases where the source value does not change.

DataContext

Some of the content in this section is reproduced from the "DataContext in WPF" article by Kishore Gaddam for CodeProject, published at codeproject.com/Articles/321899/DataContext-in-WPF.

The **DataContext** property is a vital part of one of the most important concepts in WPF: data binding. When implementing data binding in WPF, the C# representation involves creating a **Binding** object. The **Binding** object needs a data source, and there are a few ways to specify the data source. One way to define the source of the data-binding operation is to set the **Binding.Source** property to the object in which you wish to retrieve as well as store the bound control's data. There is also an attached property called the **DataContext**. This **DataContext** can be set at the Window level and propagate down to each data-bound control by way of property value inheritance. You can also set the **ElementName** and **RelativeSource** properties in the **Binding** object. The **DataContext** property is the most common way to define the source of a data-binding expression.

User interface elements in WPF have a **DataContext** attached property. That property has the standard dependency property value inheritance behavior by default. This means if you set the **DataContext** on an element to a **Student** object, the **DataContext** property on all of the parent's logical descendant elements will inherit the **Student** object reference. This means that all data bindings contained within the root element's tree will automatically bind against the **Student** object, unless explicitly told to bind against something else.

INotifyPropertyChanged

The **INotifyPropertyChanged** interface is another key element involved in the data-binding functionality of any WPF application. The **INotifyPropertyChanged** interface provides an event handler that will raise an event to indicate that a data-bound value has changed, thus communicating this change to the XAML user interface.

In the following example, we will set the **DataContext** value of the **MainWindow** in the code behind instead of the XAML markup. I will show you how this can be done using XAML only in the MVVM chapter. In this example, we have a window, a grid, two text boxes, and a button. The **TextBox** controls will represent the **FirstName** property and **LastName** property of a **Person** object. We will set the **DataContext** of the window equal to an instance of a **Person** object in which we wish to allow users to modify the **Person**'s names.

Person.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleData-binding
{
    public class Person : INotifyPropertyChanged
    {

        private string _FirstName;
        private string _LastName;
        private string _FullName;

        #region "Public Properties"

        public string FirstName
        {
            get
            {
                return _FirstName;
            }
            set
            {
                if (_FirstName != value)
                {
                    _FirstName = value;
```

```
                this.FullName = string.Format("{0} {1}", _FirstName, _LastName);
                OnPropertyChanged("FirstName");
            }
        }
    }

    public string LastName
    {
        get
        {
            return _LastName;
        }
        set
        {
            if (_LastName != value)
            {
                _LastName = value;
                this.FullName = string.Format("{0} {1}", _FirstName, _LastName);
                OnPropertyChanged("LastName");

            }
        }
    }

    public string FullName
    {
        get
        {
            return _FullName;
        }
        set
        {
            _FullName = value;
            OnPropertyChanged("FullName");
        }
```

```
        }

        #endregion


        #region "INotifyPropertyChanged members"

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;

            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }

        #endregion
    }
}
```

MainWindow.xaml

```
<Window x:Class="SimpleData-binding.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="218.905" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="87*" />
            <RowDefinition Height="47*" />
            <RowDefinition Height="54*" />
            <RowDefinition Height="54*" />
            <RowDefinition Height="54*" />
```

```
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" FontSize="40"
                   Text="Enter Person's Name" />

        <TextBlock Grid.Column="0" Grid.Row="1" FontSize="20"
                   Text="First Name" />

        <TextBox Name="txtFirstName" Grid.Column="1" Grid.Row="1" FontSize="20"
                 Text="{Binding Path=FirstName}" />

        <TextBlock Grid.Column="0" Grid.Row="2" FontSize="20"
                   Text="Last Name" />

        <TextBox Name="txtLastName" Grid.Column="1" Grid.Row="2" FontSize="20"
                 Text="{Binding Path=LastName}" />

        <TextBlock Grid.Column="0" Grid.Row="3" FontSize="20"
                   Text="Full Name" />

        <TextBlock Name="txtFullName" Grid.Column="1" Grid.Row="3" FontSize="20"
                 Text="{Binding Path=FullName}" />

        <Button Name="btnClose" Grid.Row="4" Grid.Column="1"
                Width="140" Height="30" Content="Close" Click="btnClose_Click" />

    </Grid>
</Window>
```

MainWindow.xaml.cs

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

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace SimpleData-binding
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml.
    /// </summary>
    public partial class MainWindow : Window
    {
        private Person _person;

        public MainWindow()
        {
            InitializeComponent();

            //Create an instance of the Person object.
            _person = new Person {FirstName = "<Enter first name>",
                LastName = "<Enter last name>" };

            //Set the DataContext of the Window.
            this.DataContext = _person;
        }

        private void btnClose_Click(object sender, RoutedEventArgs e)
```

```
        {
            this.Close();
        }
    }
}
```

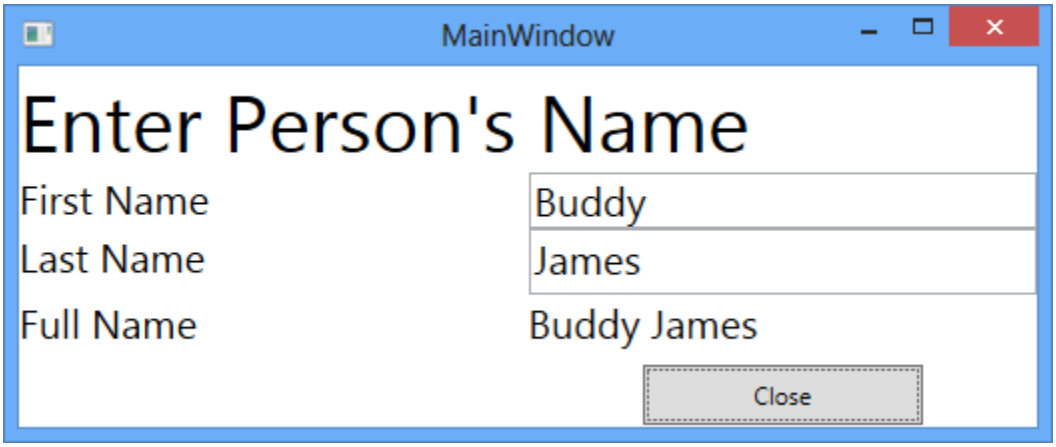

Figure 11: Rendered Data-Bound Window

MultiBinding

As you can see, data binding is a very important and powerful concept in WPF. In the previous example, we had to implement a rather ugly implementation regarding the person's full name display. This is only one example of using more than one element's values to produce the value of a single data-bound element.

As it turns out, WPF offers a much better way to implement this scenario. I'm referring to MultiBindings. A MultiBinding involves creating an **IMultiValueConverter** and binding the converter to the **txtFullName** element. The **IMultiValueConverter** interface works just like the **IValueConverter** interface except that it accepts an array of values to convert. The array of values comes from the element properties that we define in the **MultiBinding xaml** definition. A brief example will make things clearer.

As you can see, the **Convert** method takes an array of values. It iterates through the array and concatenates the strings into a single value that it returns. We haven't implemented the **ConvertBack** method because it's not used in this example.

FullnameConverter.cs

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

```
using System.Text;
using System.Threading.Tasks;
using System.Windows.Data;

namespace MultiBinding
{
    /// <summary>
    /// This class is a MultiValueConverter. It works much like a ValueConverter,
except that it takes an array of values in the convert
    /// method and it returns an array of values from the ConvertBack routine.
    /// </summary>
    public class FullnameConverter : IMultiValueConverter
    {
        public object Convert(object[] values, Type targetType, object parameter,
            System.Globalization.CultureInfo culture)
        {
            StringBuilder fullNameBuilder = new StringBuilder();
            foreach (object name in values)
            {
                fullNameBuilder.AppendFormat("{0} ", name.ToString());
            }
                return fullNameBuilder.ToString();
        }

        public object[] ConvertBack(object value, Type[] targetTypes, object
parameter,
            System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }

}
```

The **Person** class no longer contains a full name property for data binding.

Person.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MultiBinding
{
    public class Person : INotifyPropertyChanged
    {
        #region "private members"

        private string _FirstName;
        private string _LastName;

        #endregion

        #region "Public Properties"

        public string FirstName
        {
            get
            {
                return _FirstName;
            }
            set
            {
                if (_FirstName != value)
                {
                    _FirstName = value;
                    OnPropertyChanged("FirstName");
                }
            }
```

```
        }

        public string LastName
        {
            get
            {
                return _LastName;
            }
            set
            {
                if (_LastName != value)
                {
                    _LastName = value;
                    OnPropertyChanged("LastName");

                }
            }
        }

        #endregion

        #region "INotifyPropertyChanged members"

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;

            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
```

```
        #endregion
    }
}
```

We create a resource dictionary in the window definition and create a resource to our custom **MultiValueConverter**. Next, we change the **txtFullName** element to use a **MultiBinding** definition. The definition specifies the converter to use, as well as the elements and their properties that we will use as the values array in the **Convert** method of the converter.

MainWindow.xaml

```
<Window x:Class="MultiBinding.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="218.905" Width="525"
        xmlns:tc="clr-namespace:MultiBinding">
    <Window.Resources>
        <tc:FullnameConverter x:Key="fullNameConverter" />
    </Window.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="87*" />
            <RowDefinition Height="47*" />
            <RowDefinition Height="54*" />
            <RowDefinition Height="54*" />
            <RowDefinition Height="54*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" FontSize="40"
                   Text="Enter Person's Name" />

        <TextBlock Grid.Column="0" Grid.Row="1" FontSize="20"
                   Text="First Name" />

        <TextBox Name="txtFirstName" Grid.Column="1" Grid.Row="1" FontSize="20"
```

```
                Text="{Binding Path=FirstName}" />

        <TextBlock Grid.Column="0" Grid.Row="2" FontSize="20"
                   Text="Last Name" />

        <TextBox Name="txtLastName" Grid.Column="1" Grid.Row="2" FontSize="20"
                 Text="{Binding Path=LastName}" />

        <TextBlock Grid.Column="0" Grid.Row="3" FontSize="20"
                   Text="Full Name" />

        <TextBox Name="txtFullName" Grid.Column="1" Grid.Row="3" FontSize="20">
            <TextBox.Text>
            <MultiBinding Converter="{StaticResource fullNameConverter}">
                <Binding ElementName="txtFirstName" Path="Text" />
                <Binding ElementName="txtLastName" Path="Text" />
            </MultiBinding>
            </TextBox.Text>
        </TextBox>

        <Button Name="btnClose" Grid.Row="4" Grid.Column="1"
                Width="140" Height="30" Content="Close" Click="btnClose_Click" />

    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
```

```
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace MultiBinding
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml.
    /// </summary>
    public partial class MainWindow : Window
    {
        private Person _person;

        public MainWindow()
        {
            InitializeComponent();//Create an instance of the Person object.

            _person =
                new Person { FirstName = "<Enter first name>", LastName = "<Enter last
name>" };

            //Set the DataContext of the Window.
            this.DataContext = _person;
        }

        private void btnClose_Click(object sender, RoutedEventArgs e)
        {
            this.Close();
        }
    }
}
```

As you can see, this implementation is flexible compared to our previous implementation. This technique is useful in situations where you need to data-bind to an aggregate computation such as the average of text box values, or any situation where you need one element to bind to the values of multiple element values.

Chapter 5 WPF and MVVM

Model-View-ViewModel (MVVM)

When you begin to learn about WPF and XAML, you will hear of MVVM or the Model-View-ViewModel design pattern. MVVM is a user interface design pattern invented by John Gossman and based on the Presenter Model pattern by Martin Fowler.

The Model-View-ViewModel pattern allows developers to completely separate the user interface from the logic required to facilitate the user interface and the business logic and data. Separation of UI, business logic, and data allows for a nice, clean, testable piece of software. WPF data binding is extremely powerful because it allows you to separate your user interface from your business logic. The MVVM pattern provides excellent examples of how to use WPF data binding in new ways to achieve a completely decoupled application design.

We will start with the **Model**. The Model represents the data that you wish to interact with. This could be information from a database, properties of your business objects, or even an XML file. Any software package will deal with data, and it's best if your user interface is decoupled from your specific data model because your data is likely to change. Generally in WPF, your Model will implement the **INotifyPropertyChanged** interface. This interface provides members that will facilitate change notification. This means that your user interface will need to know when your Model changes without being tightly coupled to your Model. This goes both ways. Your Model will need to know when the user has changed data in your Model via user interface controls. We will cover the **INotifyPropertyChanged** interface in more detail with examples, but know that its members provide events that facilitate change notification.

Next we have the **ViewModel**. The ViewModel is a class that sits between your XAML View and your data model. This layer of abstraction is key because if your ViewModel is designed in a loosely coupled nature, your model and views are open to change. The ViewModel is typically implemented as a class with properties that point to your model's properties. You will generally use WPF data binding to bind to ViewModel. The ViewModel doesn't care about the specifics of your view or your model. This makes unit testing extremely easy.

At the top of the pattern we have the **Views**, which are WPF Windows or UserControls. The View makes up the visual part of your application. When you begin studying WPF, you will come across some MVVM purists who believe that there should be absolutely no code in your View's code behind. Although this does offer the most flexible design, it's also one of the more difficult designs to accomplish. It will require that you eliminate certain common design choices such as control event handlers and the like.

I will show an example of a no code-behind approach. However, this is not required to implement the MVVM pattern. Understand that the point behind MVVM is that you want your model and your view to be decoupled and to communicate only via the **ViewModel**. The purpose of this decoupled architecture is to allow you to easily change the view or model without breaking existing code. You also want to be able to fully unit test all of your code without worrying about using difficult tools because you've created a tightly coupled XAML view to your business logic.

MVVM example

Here is a short MVVM example to illustrate how this can work together. Imagine that you have a contact manager application. The purpose of this application is to keep up with your many contacts. Naturally you will need to save a contact's name, telephone number, email address, and home address. This will represent your Model. Examine the following **Contact** class diagram.

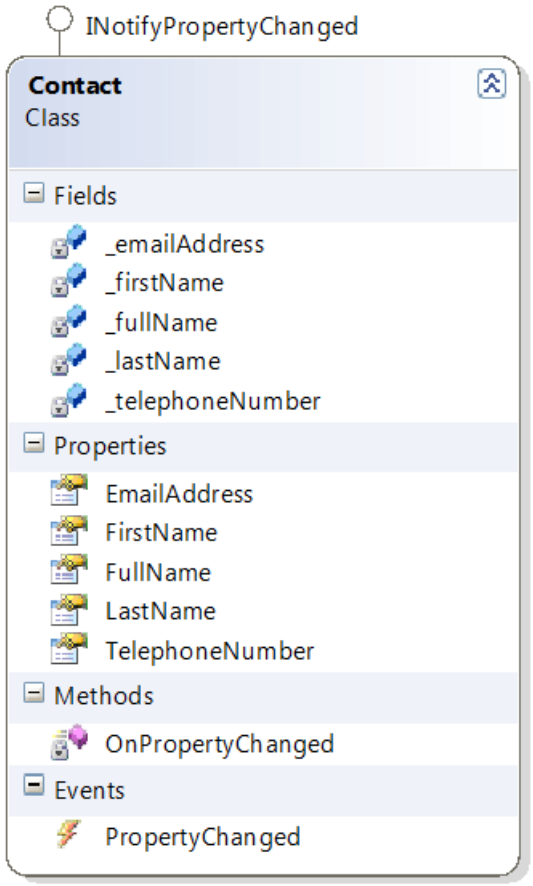

Figure 12: Contact Class Diagram

We will start by implementing a class that we'd like to use for data binding. Naturally you will probably implement a contact class. This class would have a property for the contact's first name, last name, phone number, and email address.

Contact.cs

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

```
using System.ComponentModel;

namespace ContactMvvm
{
    public class Contact : INotifyPropertyChanged
    {
        #region "private fields"

        private string _firstName;
        private string _fullName;
        private string _lastName;
        private string _telephoneNumber;
        private string _emailAddress;
        #endregion

        #region "Public Properties"
        public string FirstName
        {
            get { return _firstName; }
            set
            {
                _firstName = value;
                OnPropertyChanged("FirstName");
            }
        }

        public string EmailAddress
        {
            get { return _emailAddress; }
            set
            {
                _emailAddress = value;
                OnPropertyChanged("EmailAddress");
            }
        }
```

```
        public string LastName
        {
            get { return _lastName; }
            set
            {
                _lastName = value;
                OnPropertyChanged("LastName");
            }
        }

        public string TelephoneNumber
        {
            get { return _telephoneNumber; }
            set
            {
                _telephoneNumber = value;
                OnPropertyChanged("TelephoneNumber");
            }
        }
        #endregion

        #region "INotifyPropertyChanged members"

        public event PropertyChangedEventHandler PropertyChanged;

        //This routine is called each time a property value has been set. This will
        //cause an event to notify WPF via data-binding that a change has occurred.
        private void OnPropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;

            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
```

```
            }
        }
        #endregion
    }
}
```

Don't be overwhelmed by the **INotifyPropertyChanged** implementation. This will be covered in another chapter.

Please note that this is a plain CLR object, complete with a namespace, class name, and properties. This is the Model in MVVM; it represents our data.

Next we will create a **ViewModel** class that will take the data from our Model and present the data to the View through WPF data binding. The **ViewModel** class inherits from a class called **ObservableCollection<>**. **ObservableCollection<>** is a generic class that accepts our model as the generic type. This class basically turns our **ViewModel** into a collection of **Contact** objects with built-in change notifications to facilitate the WPF data binding mechanisms.

When we use **ObservableCollection**, WPF data binding knows when items are added or removed. I've created one private routine **PrepareContactCollection**, which creates three instances of our **Contact** class and adds them to the **ViewModel'**s internal collection. We will bind a **ListView**'s **ItemsSource** property to the **ViewModel** as well as bind the main grid's **DataContext** property to the **ViewModel**. By setting the **DataContext** of the top-level grid, we will cause the child controls to inherit the **DataContext** for data binding. The **ItemsSource** property of the **ListView** will populate the **ListView** with each item that exists in our **ObservableCollection** of contacts. Here is a class diagram and the code:

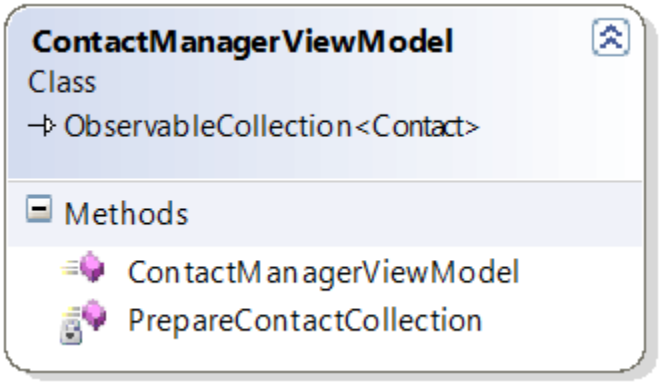

Figure 13: Class Diagram of an ObservableCollection

ContactManagerViewModel.cs

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

```
using System.Linq;
using System.Text;
using System.Collections.ObjectModel;

namespace ContactMvvm
{
    public class ContactManagerViewModel : ObservableCollection<Contact>
    {
        #region "constructor"
        public ContactManagerViewModel()
        {
            PrepareContactCollection();
        }

        #endregion

        #region "private routines"
        private void PrepareContactCollection()
        {
            //Create new contacts and add them to the ViewModel's
            //ObservableCollection.
            var ContactOne = new Contact
            {
                FirstName = "John",
                LastName = "Doe",
                EmailAddress = "jdoe@email.com",
                TelephoneNumber = "555-555-5555"

            };
            Add(ContactOne);

            var ContactTwo = new Contact
            {
                FirstName = "Bob",
                LastName = "Watson",
```

```
                EmailAddress = "bwatson@email.com",
                TelephoneNumber = "555-555-5555"

            };
            Add(ContactTwo);

            var ContactThree = new Contact
            {
                FirstName = "Joe",
                LastName = "Johnson",
                EmailAddress = "jjohnson@email.com",
                TelephoneNumber = "555-555-5555"

            };

            Add(ContactThree);
        }
        #endregion
    }
}
```

Tip: Notice that by inheriting from ObservableCollection<Contact> we gain an Add() method. This allows us to populate the ViewModel with instances of our Contact Model.

Now we want to write a XAML user interface and bind our **ViewModel** to the controls. The first thing that we will do is import the object's namespace so we can access the object in XAML. We will do this by specifying **xmlns:viewModel** in the **Window** element. **viewModel** is an arbitrary value and can be anything that you wish to associate with your newly imported namespace.

Next we will create a resource dictionary and add a static resource that represents an instance of this **ViewModel**. This allows us to bind to our **ViewModel** without writing any C# code! Here is the XAML markup.

ContactManager.xaml

```
<Window x:Class="ExceptionValidation.ContactManager"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ContactManager" Height="300" Width="300"
```

```
        xmlns:viewModel="clr-namespace:ContactMvvm">
    <Window.Resources>
        <viewModel:ContactManagerViewModel x:Key="contactViewModel" />
    </Window.Resources>
    <Grid DataContext="{StaticResource ResourceKey=contactViewModel}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="93*" />
            <RowDefinition Height="29*" />
            <RowDefinition Height="27*" />
            <RowDefinition Height="30*" />
            <RowDefinition Height="81*" />
        </Grid.RowDefinitions>
        <ListView Grid.ColumnSpan="2" BorderThickness="2" x:Name="lstContacts"
IsSynchronizedWithCurrentItem="True" ItemsSource="{StaticResource
ResourceKey=contactViewModel}" DisplayMemberPath="FirstName" Margin="0,0,0,28"
Grid.RowSpan="2" />


        <TextBlock Grid.Column="0" Grid.Row="1" Text="First name" />
        <TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=FirstName}" />

        <TextBlock Grid.Column="0" Grid.Row="2" Text="Last name" />
        <TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Path=LastName}" />

        <TextBlock Grid.Column="0" Grid.Row="3" Text="Email address" />
        <TextBox Grid.Column="1" Grid.Row="3" Text="{Binding Path=EmailAddress}" />

        <TextBlock Grid.Column="0" Grid.Row="4" Text="Telephone number" />
        <TextBox Grid.Column="1" Grid.Row="4" Text="{Binding Path=TelephoneNumber}" />
    </Grid>
</Window>
```

Notice that we set the grid's **DataContext** property binding to the **ViewModel** static resource that we created in the dictionary. The **DataContext** provides a binding context for your WPF controls. When we set the **DataContext** on the grid, this value will propagate down to the child elements via property inheritance—this is why all of the **TextBox Binding** values can be set to properties of the Model. This is because the **DataContext** is set to our **ViewModel**, which is an **ObservableCollection<Contact>**. So when we set our binding paths, WPF knows we are dealing with a **Contact** object and the values are mapped to the correct properties. This will also work if we had properties inside of our **ViewModel** class. We could bind to these values as well.

The following figure shows the program's output:

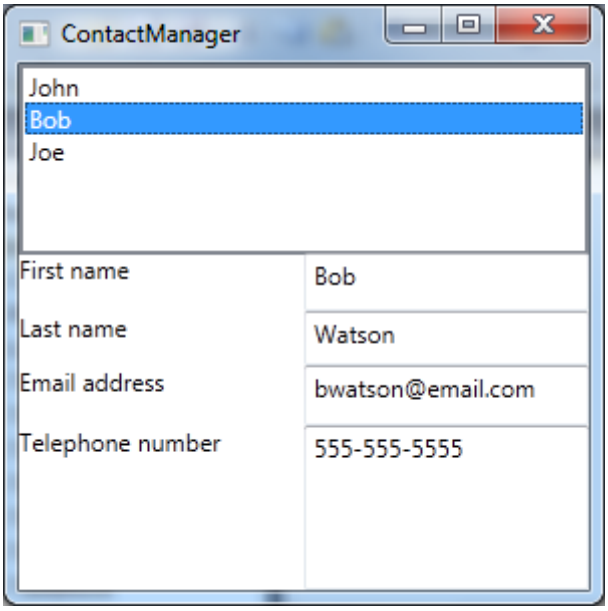

Figure 14: Completed Contact Manager

As you can see, we've created a data-bound WPF window with data from our custom **Contact** object. You can update the values in the text box and the values will change in the **ListView**. You can click in the **ListView** and the values will update based on your selection. The user interface is 100% data-bound and we didn't write any event handlers in the code behind! We were able to do it in 100% XAML. Also note that resource dictionaries are not limited to CLR objects. You can store XAML brushes, third-party controls, styles, type converters, or any other type of objects for use in your XAML markup.

Chapter 6 WPF Commands

The ICommand interface: an alternative to event handlers

In the early days of rapid application development, most Windows applications were built on an event-based model. You would create your controls and wire up event handlers in the code-behind to handle the events of your controls. This is still a popular way of handling the logic of Windows user interface controls. However, with the advent of WPF and XAML, the data binding mechanisms have provided alternatives to the old event-driven system. This is especially true when it comes to button clicks. What is this great programming magic that I speak of? It can be summed up by defining the command-driven approach to user interface processing.

Commands and XAML

The button control, as well as other WPF controls, have a property called **Command**. This property allows a developer to specify an instance of an implementation of the **ICommand** interface to accomplish a specified task without adding any code to the Window's code behind. The **ICommand** interface can be found in the **System.Windows.Input** namespace and it provides a generic implementation of a loosely coupled command. Consider the following code, which demonstrates the typical event-driven implementation of a button click handler:

```
<Window x:Class="WPFExample.ButtonClick"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="ButtonClick" Height="300" Width="300">
    <Grid>
        <Button x:Name="btnFireEvent" Content="Fire event"
               Width="100" Height="100" Click="btnFireEvent_Click" />
    </Grid>
</Window>
```

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

```
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace WPFExample
{
    /// <summary>
    /// Interaction logic for ButtonClick.xaml.
    /// </summary>
    public partial class ButtonClick : Window
    {
        public ButtonClick()
        {
            InitializeComponent();
        }

        private void btnFireEvent_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Hello Event Handler!");
        }
    }
}
```

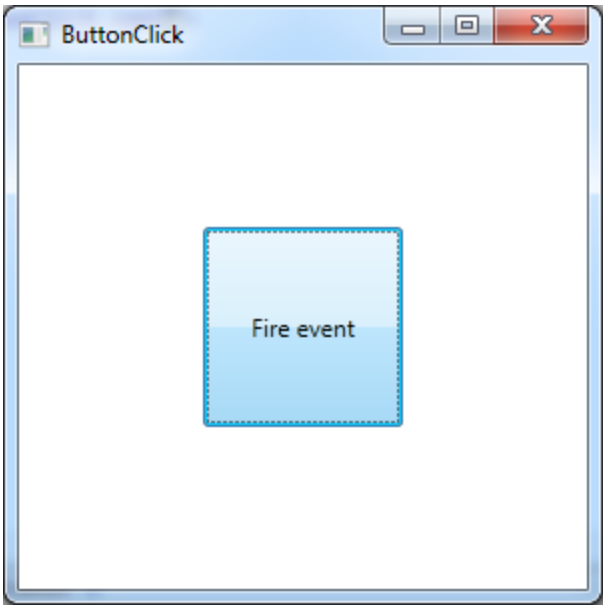

Figure 15: Event-Driven Button

There should be nothing new in this example. We basically have a XAML window with a button defined. The button's XAML contains a **Click** attribute that points to an event handler method that can be found in the C# code behind. When the user clicks the button, the method will be called and the message box will appear. The fundamental problem with this implementation is that the XAML is tightly coupled to the code behind due to the event handler declaration. This coupling makes the scenario very tough to unit test. As you can see, this would break many of the benefits that we gain from implementing the MVVM design pattern.

The **ICommand** interface can help by allowing us to implement a class to act as a generic command. As you've seen in the previous examples, we can easily use WPF data binding to reference the **ICommand** implementation which would negate the need for the tightly coupled code-behind event handler. The **ICommand** interface has the following members:

Table 3: ICommand Members

Member	Method Description
CanExecute()	A support method that returns a Boolean indicating whether the command is in an executable state. The WPF data binding mechanism will check the return value of this method and enable or disable the associated control based on the value.
Execute()	A method that contains the code that should be executed to accomplish the task associated with the command.

Member	Method Description
CanExecuteChanged()	An event that is triggered when changes occur that affect whether the command should execute.

Here is an example. Please note that the example is implemented with the MVVM pattern in mind.

We will start with the same XAML as before with a few changes. First we create a resource that references an instance of our **ViewModel**. Our **ViewModel** has one property of **ICommand** called **ButtonClickCommand** and a method called **ShowMessageBox(string message)**. We then set the main data context to our **ViewModel**. This allows us to bind to the **ButtonClickCommand** property to the **Command** attribute of the **Button** element. We also set the **CommandParameter** attribute to **{Binding}**, which equals our **ViewModel**.

You'll notice that the **Execute** method of the **Command** class takes a parameter of type object. With the data binding that we've set up below, we've bound the **Command** to an **ICommand** property on the **ViewModel** and we will pass the **ViewModel** as an object to the **Execute** method on the **ICommand** implementation. The following code will illustrate the idea better:

```
<Window x:Class="WPFExample.ButtonClick"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ButtonClick" Height="300" Width="300"
        xmlns:viewModel="clr-namespace:WPFCommand">


    <Window.Resources>
        <viewModel:CommandViewModel x:Key="commandViewModel" />
    </Window.Resources>
    <Grid DataContext="{StaticResource ResourceKey=commandViewModel}">
        <Button x:Name="btnFireEvent" Content="Fire event"
                Width="100" Height="100" Command="{Binding Path=ButtonClickCommand}"
CommandParameter="{Binding}" />
    </Grid>
</Window>
```

CommandViewModel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Input;
using System.Windows;

namespace WPFCommand
{
    public class CommandViewModel
    {
        public ICommand ButtonClickCommand
        {
            get
            {
                return new ButtonClickCommand();
            }
        }

        public void ShowMessagebox(string message)
        {
            MessageBox.Show(message);
        }
    }
}
```

ButtonClickCommand.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Input;

namespace WPFCommand
```

```
{
    public class ButtonClickCommand : ICommand
    {
        public bool CanExecute(object parameter)
        {
            return true;
        }

        public event EventHandler CanExecuteChanged;

        public void Execute(object parameter)
        {
            var viewModel = (CommandViewModel)parameter;

            viewModel.ShowMessagebox("Hello decoupled command!");
        }
    }
}
```

As you can see, by using the command binding behavior in XAML, we are able to execute a method on our **ViewModel** by way of a generic command class and all without a single line of code behind. The result is a fully testable solution that plays great with MVVM!

Existing commands

Some of the content in this section is reproduced from the Commands in WPF page on ASP Free, available at www.aspfree.com/c/a/braindump/commands-in-wpf/.

There are five different classes that provide static properties that represent WPF's built-in commands.

The **ApplicationCommands** class contains properties for the following commands right out of the box: Close, Copy, Cut, Delete, Find, Help, New, Open, Paste, Print, PrintPreview, Properties, Redo, Replace, Save, SaveAs, SelectAll, Stop, Undo, and more.

The **ComponentCommands** class contains properties for MoveDown, MoveLeft, MoveRight, MoveUp, ScrollByLine, ScrollPageDown, ScrollPageLeft, ScrollPageRight, ScrollPageUp, SelectToEnd, SelectToHome, SelectToPageDown, SelectToPageUp, and more.

The **MediaCommands** class contains properties for the following command properties: ChannelDown, ChannelUp, DecreaseVolume, FastForward, IncreaseVolume, MuteVolume, NextTrack, Pause, Play, PreviousTrack, Record, Rewind, Select, Stop, and more.

The **NavigationCommands** class contains the following command properties: BrowseBack, BrowseForward, BrowseHome, BrowseStop, Favorites, FirstPage, GoToPage, LastPage, NextPage, PreviousPage, Refresh, Search, Zoom, and more.

The **EditingCommands** class contains the following command properties: AlignCenter, AlignJustify, AlignLeft, AlignRight, CorrectSpellingError, DecreaseFontSize, DecreaseIndentation, EnterLineBreak, EnterParagraphBreak, IgnoreSpellingError, IncreaseFontSize, IncreaseIndentation, MoveDownByLine, MoveDownByPage, MoveDownByParagraph, MoveLeftByCharacter, MoveLeftByWord, MoveRightByCharacter, MoveRightByWord and more.

MVVM base class implementations

I find that it's helpful to create some base classes and interfaces to make working with the MVVM design pattern easier while improving your overall application's design. I will provide a sample base class that I use for my **ViewModel**s and **Command**s in the following section.

Base ViewModel class example

It's a very common practice for your **ViewModel** to implement the **INotifyPropertyChanged** interface. This will facilitate the change notification associated with WPF data binding.

The typical implementation involves calling a method from each property setter that passes the property name by string to another method that raises the **PropertyChanged** event. This can be messy because it's never a good idea to rely on magic strings. To solve this problem, I've implemented a routine that takes a lambda expression instead of the property name string value. This allows strongly typed property names to be passed instead of magic strings.

Finally, there is the **IDataErrorInfo** interface, which will validate your properties that are marked with validation attributes. If the property value does not match the validation attribute, your view will be notified via change notification so the view can be updated to show the error.

ViewModelBase.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Linq.Expressions;
using System.ComponentModel;
```

```
namespace Mvvm.Infrastructure
{
    public abstract class ViewModelBase : INotifyPropertyChanged, IDataErrorInfo
    {
        #region Fields

        /// <summary>
        /// A dictionary of property names, property values. The property name is the
        /// key to find the property value.
        /// </summary>
        private readonly Dictionary<string, object> _values = new Dictionary<string,
object>();

        #endregion

        #region Protected

        /// <summary>
        /// Sets the value of a property.
        /// </summary>
        /// <typeparam name="T">The type of the property value.</typeparam>
        /// <param name="propertySelector">Expression tree contains the property
definition.</param>
        /// <param name="value">The property value.</param>
        protected void SetValue<T>(Expression<Func<T>> propertySelector, T value)
        {
            string propertyName = GetPropertyName(propertySelector);

            SetValue<T>(propertyName, value);
        }

        /// <summary>
        /// Sets the value of a property.
        /// </summary>
        /// <typeparam name="T">The type of the property value.</typeparam>
        /// <param name="propertyName">The name of the property.</param>
```

```
        /// <param name="value">The property value.</param>
        protected void SetValue<T>(string propertyName, T value)
        {
            if (string.IsNullOrEmpty(propertyName))
            {
                throw new ArgumentException("Invalid property name", propertyName);
            }

            _values[propertyName] = value;
            NotifyPropertyChanged(propertyName);
        }


        /// <summary>
        /// Gets the value of a property.
        /// </summary>
        /// <typeparam name="T">The type of the property value.</typeparam>
        /// <param name="propertySelector">Expression tree contains the property
        ///definition.</param>
        /// <returns>The value of the property or default value if one doesn't
        ///exist.</returns>
        protected T GetValue<T>(Expression<Func<T>> propertySelector)
        {
            string propertyName = GetPropertyName(propertySelector);

            return GetValue<T>(propertyName);
        }


        /// <summary>
        /// Gets the value of a property.
        /// </summary>
        /// <typeparam name="T">The type of the property value.</typeparam>
        /// <param name="propertyName">The name of the property.</param>
        /// <returns>The value of the property or default value.</returns>
        protected T GetValue<T>(string propertyName)
        {
            if (string.IsNullOrEmpty(propertyName))
```

```
            {
                throw new ArgumentException("Invalid property name", propertyName);
            }

            object value;
            if (!_values.TryGetValue(propertyName, out value))
            {
                value = default(T);
                _values.Add(propertyName, value);
            }

            return (T)value;
        }

        /// <summary>
        /// Validates current instance properties using data annotations.
        /// </summary>
        /// <param name="propertyName">This instance property to validate.</param>
        /// <returns>Relevant error string on validation failure or <see
cref="System.String.Empty"/> on validation success.</returns>
        protected virtual string OnValidate(string propertyName)
        {
            if (string.IsNullOrEmpty(propertyName))
            {
                throw new ArgumentException("Invalid property name", propertyName);
            }

            string error = string.Empty;
            var value = GetValue(propertyName);
            var results = new List<ValidationResult>(1);
            var result = Validator.TryValidateProperty(
                value,
                new ValidationContext(this, null, null)
                {
                    MemberName = propertyName
                },
```

```
            results);

        if (!result)
        {
            var validationResult = results.First();
            error = validationResult.ErrorMessage;
        }

        return error;
    }

    #endregion

    #region Change Notification

    /// <summary>
    /// Raised when a property on this object has a new value.
    /// </summary>
    public event PropertyChangedEventHandler PropertyChanged;

    /// <summary>
    /// Raises this object's PropertyChanged event.
    /// </summary>
    /// <param name="propertyName">The property that has a new value.</param>
    protected void NotifyPropertyChanged(string propertyName)
    {
        this.VerifyPropertyName(propertyName);

        PropertyChangedEventHandler handler = this.PropertyChanged;
        if (handler != null)
        {
            var e = new PropertyChangedEventArgs(propertyName);
            handler(this, e);
        }
    }
```

```
        /// <summary>
        /// Raises the object's PropertyChanged event.
        /// </summary>
        /// <typeparam name="T">The type of property that has changed.</typeparam>
        /// <param name="propertySelector">Expression tree contains the property
        ///definition.</param>
        protected void NotifyPropertyChanged<T>(Expression<Func<T>> propertySelector)
        {
            var propertyChanged = PropertyChanged;
            if (propertyChanged != null)
            {
                string propertyName = GetPropertyName(propertySelector);
                propertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }
        #endregion // INotifyPropertyChanged Members.

        #region Data Validation
        string IDataErrorInfo.Error
        {
            get
            {
                throw new NotSupportedException("IDataErrorInfo.Error is not supported,
use IDataErrorInfo.this[propertyName] instead.");
            }
        }
        string IDataErrorInfo.this[string propertyName]
        {
            get
            {
                return OnValidate(propertyName);
            }
        }
        #endregion
```

```
        #region "Private members"
        private string GetPropertyName(LambdaExpression expression)
        {
            var memberExpression = expression.Body as MemberExpression;
            if (memberExpression == null)
            {
                throw new InvalidOperationException();
            }
            return memberExpression.Member.Name;
        }

        private object GetValue(string propertyName)
        {
            object value;
            if (!_values.TryGetValue(propertyName, out value))
            {
                var propertyDescriptor =
TypeDescriptor.GetProperties(GetType()).Find(propertyName, false);
                if (propertyDescriptor == null)
                {
                    throw new ArgumentException("Invalid property name", propertyName);
                }
                value = propertyDescriptor.GetValue(this);
                _values.Add(propertyName, value);
            }
            return value;
        }
        #endregion
    }
}
```

The following is an example of a **ViewModel** that implements this base class.

PersonViewModel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.DataAnnotations;

namespace Mvvm.Infrastructure
{
    /// <summary>
    /// Represents person data.
    /// </summary>
    public class PersonViewModel : ViewModelBase
    {
        /// <summary>
        /// Gets or sets the person's first name.
        /// </summary>
        /// <remarks>
        /// Empty string and null are not allowed.
        /// Allow minimum of 2 and up to 40 uppercase and lowercase.
        /// </remarks>
        [Required]
        [RegularExpression(@"^[a-zA-Z''-'\s]{2,40}$")]
        public string FirstName
        {
            get { return GetValue(() => FirstName); }
            set { SetValue(() => FirstName, value); }
        }

        /// <summary>
        /// Gets or sets the person's last name.
        /// </summary>
        /// <remarks>
        /// Empty string and null are not allowed.
        /// </remarks>
```

```
        [Required]
        public string LastName
        {
            get { return GetValue(() => LastName); }
            set { SetValue(() => LastName, value); }
        }

        /// <summary>
        /// Gets or sets the person's age.
        /// </summary>
        /// <remarks>
        /// Only values between 1 and 120 are allowed.
        /// </remarks>
        [Range(1, 120)]
        public int Age
        {
            get { return GetValue(() => Age); }
            set { SetValue(() => Age, value); }
        }
    }
}
```

Please note that you will need to add a reference to the **System.ComponentModel.DataAnnotation** assembly in order to support the validation attributes.

Chapter 7 Advanced WPF Concepts

Property value inheritance

WPF dependency properties support a concept called property value inheritance. Property value inheritance is a mechanism by which a property of a parent element will propagate down to the same property on child elements. For instance, you can set the **FontSize** of a window object and all text controls will share the same font size unless the size is explicitly set on a child control.

Routed events

Routed events are much like your typical Windows Forms events except for some key differences. Event routing provides a mechanism for events to originate in an element and propagate the event up the tree to other elements that are listed for the event. This is much like the property value inheritance in dependency properties.

WPF documents

Many times you will find yourself with a lot of content to display in a window. WPF provides classes for formatting your content on the screen. This is accomplished with WPF documents. Documents allow you to display large amounts of content without worrying about the size of the window in which the content is contained. The WPF document types are separated into fixed documents and flow documents.

Fixed documents

Fixed documents are generally documents meant for print. Microsoft has included one type of fixed document to be used in WPF. These fixed documents are called XPS documents (XML Paper Specification). The XPS file structure is actually a compressed file that contains files for each element of the document (images, fonts, and text content). The WPF programming model provides a lot of interesting options for dealing with fixed documents. You can load, print, write, and even annotate the documents using WPF programming techniques.

Tip: You can rename an XPS document by replacing its extension with .zip and then view the contents of the compressed file in your preferred zip compression tool.

To display a fixed XPS document in a WPF application, use the **DocumentViewer** object. This object will create user interface elements for searching and zooming in the document. You must add a reference to the **ReachFramework** assembly in order to access the **System.Windows.Xps.Packaging.XpsDocument** object. The following example shows how to load an XPS file for viewing in a WPF application.

MainWindow.xaml

```
<Window x:Class="XPSDocumentViewer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <DocumentViewer x:Name="xpsDocViewer">

        </DocumentViewer>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Xps.Packaging;
using System.IO;

namespace XPSDocumentViewer
{
```

```
    /// <summary>
    /// Interaction logic for MainWindow.xaml.
    /// </summary>
    public partial class MainWindow : Window
    {
        XpsDocument mydoc = new XpsDocument("lettertest.xps", FileAccess.Read);

        public MainWindow()
        {
            InitializeComponent();
            xpsDocViewer.Document = mydoc.GetFixedDocumentSequence();
            mydoc.Close();
        }
    }
}
```

The following figure is a screenshot of the application:

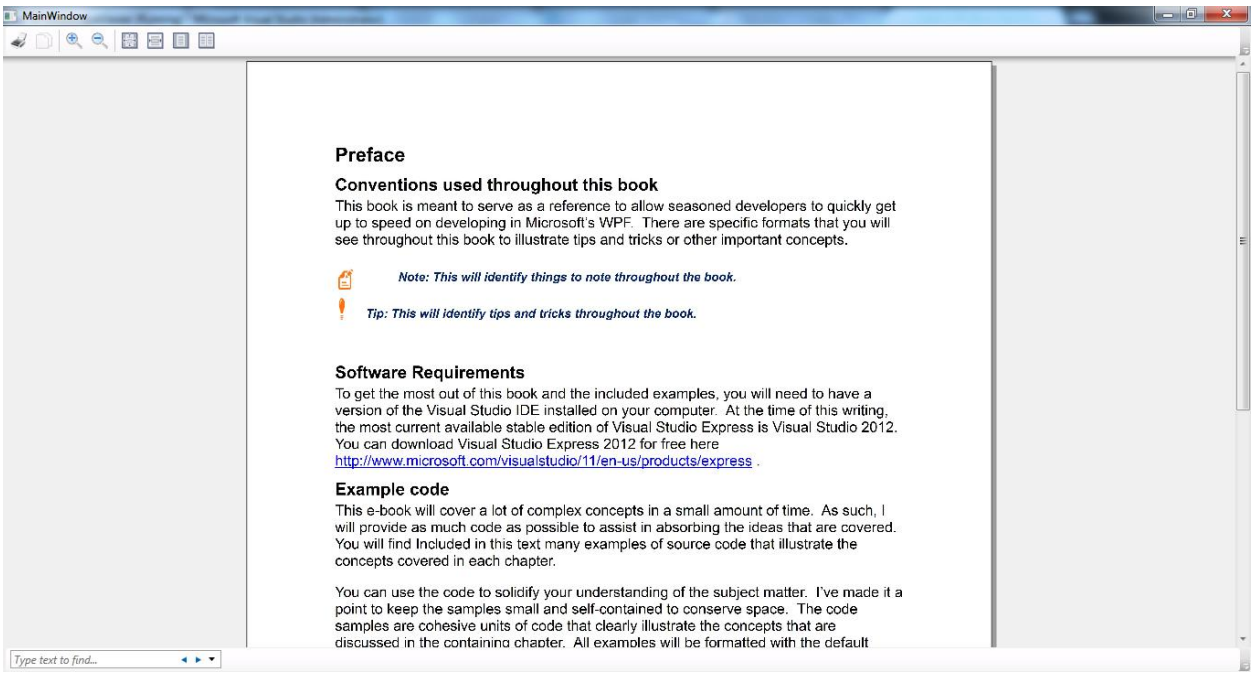

Figure 16: Rendered Fixed Document

Flow documents

The other type of WPF document is called a flow document. A WPF flow document is created by combining different flow elements within a container. **FlowElements** are unique in that they don't inherit from the **UIElement** and **FrameworkElement** classes. Instead, they represent a completely separate branch of classes that derive from **ContentElement** and **FrameworkContentElement**.

Content element classes are scaled down in comparison to other **UIElements**. They have events for handling many of the same basic features like mouse and keyboard events and drag-and-drop operations. The main difference between content and non-content elements is that content elements do not handle their own rendering. They rely on the container in which they are placed to render them. This allows the container to defer rendering to perform some last-minute optimizations, such as paragraph layout and hyphenating words.

Flow documents can be resized and their container will handle the rendering to make sure that the **FlowElements** are rendered as you expect in reference to their container's size and position.

The following is a list of the flow content elements used to lay out your flow document:

- Block
- Paragraph
- Table
- ListItem
- TableRow
- Inline
- Span
- InlineUIContainer
- Figure
- List
- Section
- TableCell
- BlockUIContainer
- TableRowGroup
- Run
- LineBreak
- AnchoredBlock
- Floater

One of the flow document containers is the **FlowDocumentScrollViewer**. As the name suggests, this container allows your content to be scrolled. A flow document can be laid out much like an HTML document. There are even table-related classes for showing tabular data. Here is an example of a table-based flow document.

MainWindow.xaml

```
<Window x:Class="FlowDocument.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <FlowDocumentScrollViewer>
        <FlowDocument>
            <Paragraph FontSize="20pt">Here is an example of a flow document
```

```
paragraph.</Paragraph>
            <Table>
                <TableRowGroup>
                    <TableRow>
                        <TableCell>
                            <Paragraph>
                                Cell one
                            </Paragraph>
                        </TableCell>
                        <TableCell>
                            <Paragraph>
                                Cell Two
                            </Paragraph>
                        </TableCell>
                        <TableCell>
                            <Paragraph>
                                Cell Three
                            </Paragraph>
                        </TableCell>
                        <TableCell>
                            <Paragraph>
                                Cell Four
                            </Paragraph>
                        </TableCell>
                    </TableRow>
                    <TableRow>
                        <TableCell>
                            <Paragraph>
                                Data one
                            </Paragraph>
                        </TableCell>
                        <TableCell>
                            <Paragraph>
                                Data Two
                            </Paragraph>
```

```
                    </TableCell>
                    <TableCell>
                        <Paragraph>
                            Data Three
                        </Paragraph>
                    </TableCell>
                    <TableCell>
                        <Paragraph>
                            Data Four
                        </Paragraph>
                    </TableCell>
                </TableRow>
            </TableRowGroup>
        </Table>
    </FlowDocument>
</FlowDocumentScrollViewer>
</Window>
```

The following screenshots show the application's output; they represent the same output with the window size changed.

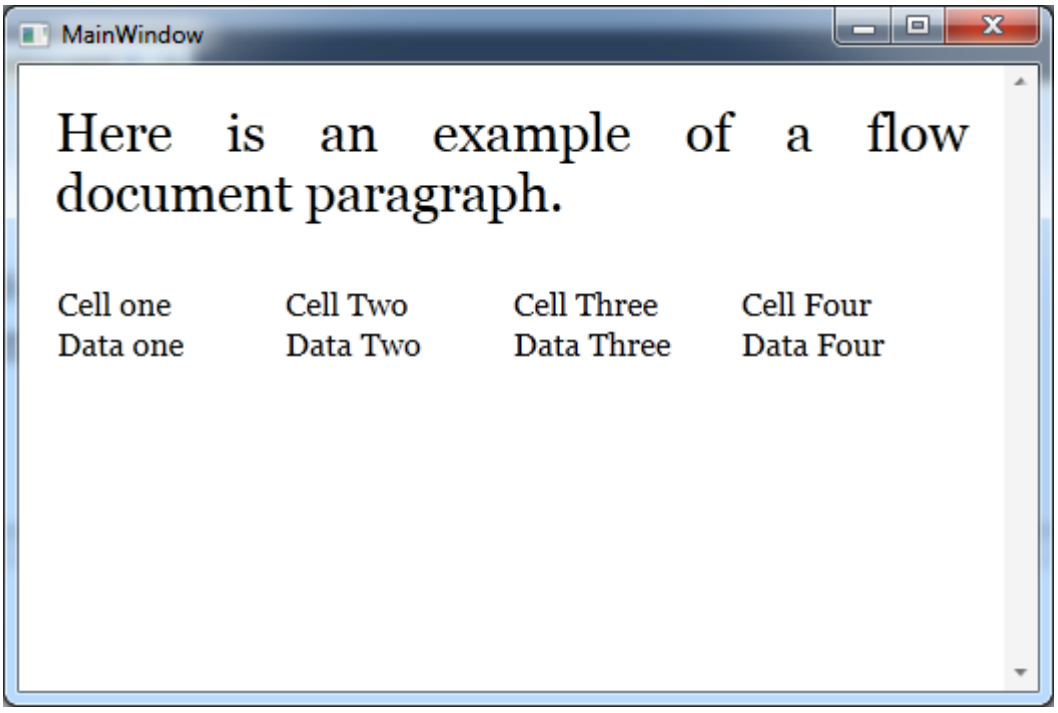

Figure 17: Rendered Flow Document

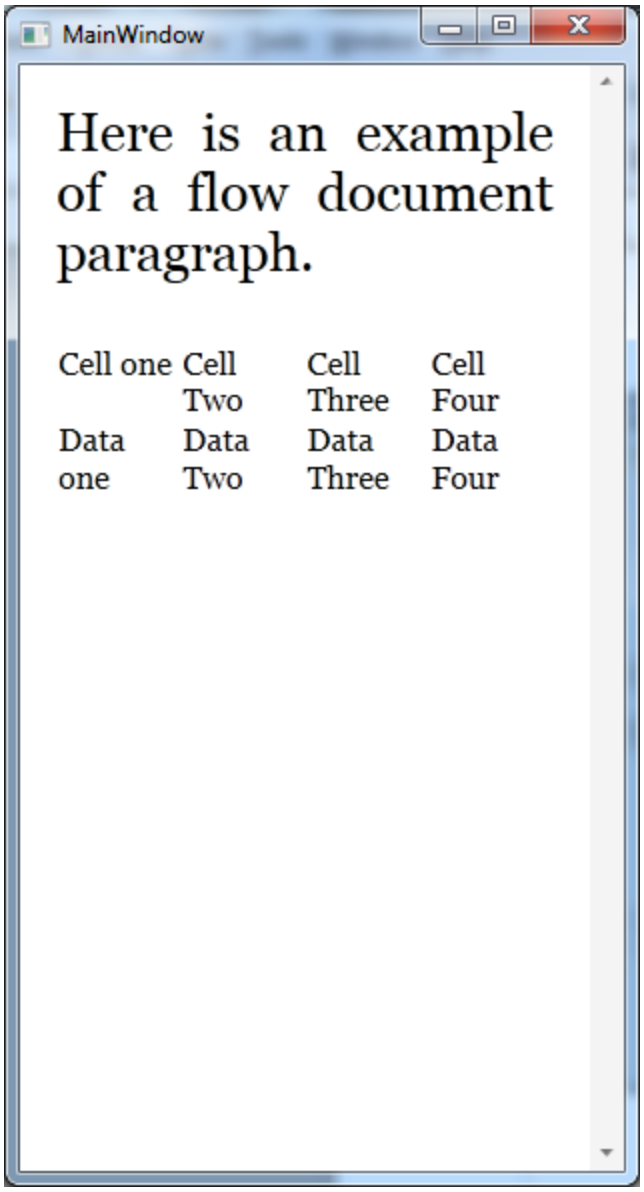

Figure 18: Rendered Flow Document Resized

Chapter 8 WPF Control Styles and Templates

The frameless window effect

A nice trick for developing unique-looking WPF windows is to set your **WindowStyle** property to **none** and then add a bit of opacity to the window. You can then choose a shape and color to create a window that is anything but traditional. Here is just one example of the many different possibilities for customization of your WPF windows:

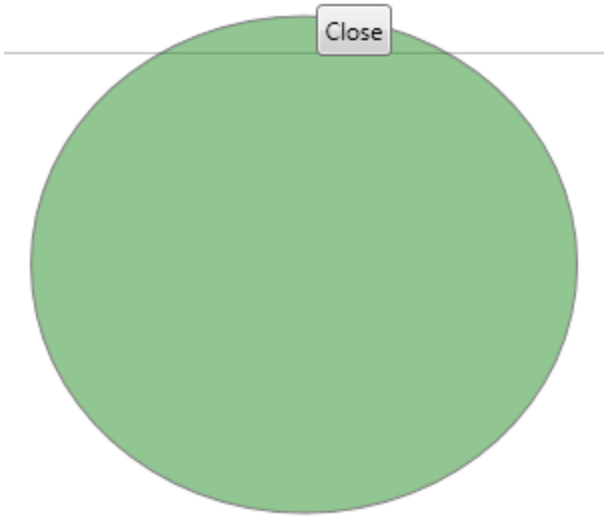

Figure 19: Frameless Window

UniqueWindow.xaml

```
<Window x:Class="ExceptionValidation.UniqueWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="UniqueWindow" Height="300" Width="300"
        WindowStyle="None" AllowsTransparency="True" Background="Transparent"
        MouseLeftButtonDown="Window_MouseLeftButtonDown">
    <Grid>
        <Ellipse Fill="ForestGreen" Opacity="0.5"  Height="249"  Name="ellipse1"
Stroke="Black" Width="274">
```

```
        </Ellipse>
        <Button Content="Close" Click="button1_Click" Height="26"
HorizontalAlignment="Left" Margin="156,20,0,0" Name="button1" VerticalAlignment="Top"
Width="38" />
    </Grid>
</Window>
```

UniqueWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace ExceptionValidation
{
    /// <summary>
    /// Interaction logic for UniqueWindow.xaml.
    /// </summary>
    public partial class UniqueWindow : Window
    {
        public UniqueWindow()
        {
            InitializeComponent();
        }

        private void Window_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
        {
            DragMove();
```

```
        }

        private void button1_Click(object sender, RoutedEventArgs e)
        {
            this.Close();
        }
    }
}
```

Logical tree vs. visual tree

In order to completely redesign the look of a control, you will need to understand and modify all of the elements that make up the control. When you design a window, you create a hierarchy of parent-child element relationships to define your user interface. This is called the logical tree.

WPF controls such as **Button** or **StackPanel** are made up of a collection of elements, much like your Window is constructed of elements to define its look. The difference is that you as a developer are unable to see the control hierarchy that makes up the WPF controls. This is problematic because later, when you decide to redesign a control, often you will need to understand how the control was originally constructed. The hierarchy of elements that make up a control is known as the visual tree.

With a little recursion and some C# code, we can create an application that will display the visual tree of an entire window.

VisualTreeBuilder.xaml

```
<Window x:Class="VisualTreeExplorer.VisualTreeBuilder"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="VisualTreeBuilder" Height="300" Width="300">
    <Grid>
        <TreeView  HorizontalAlignment="Left" Name="treeElements"
VerticalAlignment="Top"  />
    </Grid>
</Window>
```

VisualTreeBuilder.xaml.cs

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

```
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace VisualTreeExplorer
{
    /// <summary>
    /// Interaction logic for VisualTreeBuilder.xaml.
    /// </summary>
    public partial class VisualTreeBuilder : Window
    {
        public VisualTreeBuilder()
        {
            InitializeComponent();
        }

        public void BuildVisualTree(DependencyObject element)
        {
            treeElements.Items.Clear();

            TraverseElement(element, null);
        }

        private void TraverseElement(DependencyObject currentElement, TreeViewItem
parentItem)
        {
            var item = new TreeViewItem();
            item.Header = currentElement.GetType().Name;
            item.IsExpanded = true;
```

```
            if (parentItem == null)
            {
                treeElements.Items.Add(item);
            }
            else
            {
                parentItem.Items.Add(item);
            }

            for (int i = 0; i < VisualTreeHelper.GetChildrenCount(currentElement); i++)
            {
                // Process each contained element recursively.
                TraverseElement(VisualTreeHelper.GetChild(currentElement, i), item);
            }
        }
    }
}
```

MainWindow.xaml

```
<Window x:Class="VisualTreeExplorer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525" Loaded="Window_Loaded">
    <Grid>
        <StackPanel Height="271" HorizontalAlignment="Left" Margin="124,10,0,0"
Name="stackPanel1" VerticalAlignment="Top" Width="247" >
            <TextBlock Text="Test Text Block" />
            <TextBox Text="Test Text Box" />
            <Button Content="Button one" />
            <Button Content="Button two" />
            <Button Content="Button three" />

        </StackPanel>
    </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace VisualTreeExplorer
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml.
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            var visualTreeBuilder = new VisualTreeBuilder();
            visualTreeBuilder.BuildVisualTree(this);
            visualTreeBuilder.Show();
        }
    }
}
```

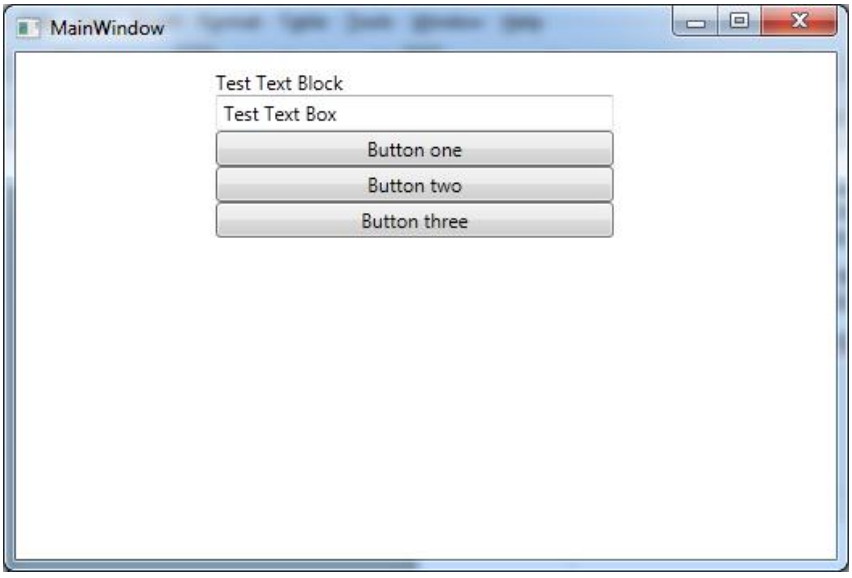

Figure 20: Original Window

As you can see, our **MainWindow** is made up of various elements for display purposes only. Next we create a window that uses the **VisualTreeHelper** class to build a view that represents the visual tree of the window's element hierarchy. This code can be very useful for restyling your WPF applications.

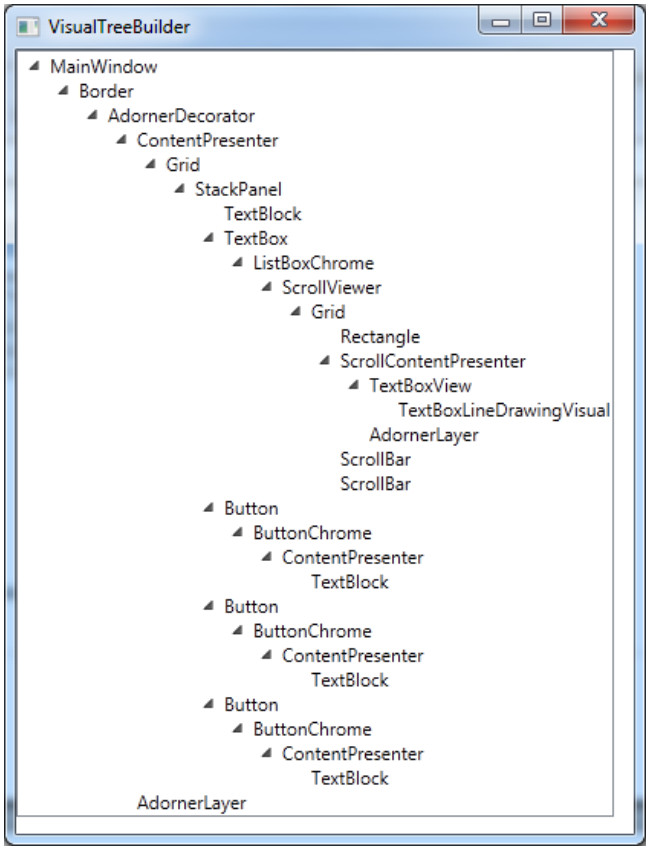

Figure 21: Window Displaying Tree of Visual Elements

Templates

Each WPF control has a “lookless” design. This means that the look of the control can be completely changed from its default appearance. The behavior of the control is baked into the classes that represent the control, and the appearance is defined by what is known as a control template.

A control template defines the visual aspects of a control. For instance, a Button control is actually the combination of many smaller XAML elements that together define the look of the button.

Templates vs. styles

Templates are different from styles in many ways. With styles you can set the properties of a control to alter the appearance. Templates allow you to change the appearance as well as the behavior of your controls. They allow you to completely restructure the make-up of a particular control.

In order to create a custom control template for the **Button** control, we will start by defining our **ControlTemplate** in the **Window** resources collection. You will usually want to create the template in an application-level resource so you can reuse the template throughout your application. You set the **TargetType** property of the **ControlTemplate** to the **Button** type definition. We will draw a border and background and place the button's content inside. We will use a **Border** control for the border. All content controls require a **ContentPresenter** as a container for the content of the control.

Triggers

Triggers are used in styles and templates to change a control's property when another property value is changed. To add some visual effects to your button, we will want to respond to the **IsMouseOver** and **IsPressed** triggers. We will show and hide a rectangle on the **Button.IsKeyboardFocused** property trigger to show when the button has focus.

MainWindow.xaml

```
<Window x:Class="ButtonControlTemplate.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <ControlTemplate x:Key="CustomButtonTemplate" TargetType="{x:Type Button}">
            <Border Name="Border" BorderBrush="Blue" BorderThickness="3"
CornerRadius="2"
```

```
                Background="BlueViolet" TextBlock.Foreground="White">
            <Grid>
                <Rectangle Name="FocusCue" Visibility="Hidden" Stroke="Black"
                           StrokeThickness="1" StrokeDashArray="1 2"
                           SnapsToDevicePixels="True" ></Rectangle>
                <ContentPresenter RecognizesAccessKey="True"
Margin="{TemplateBinding Padding}"></ContentPresenter>
            </Grid>
        </Border>
        <ControlTemplate.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
                <Setter TargetName="Border" Property="Background" Value="DarkBlue"
/>
            </Trigger>
            <Trigger Property="IsPressed" Value="True">
                <Setter TargetName="Border" Property="Background" Value="Purple" />
                <Setter TargetName="Border" Property="BorderBrush"
Value="DarkKhaki" />
            </Trigger>
            <Trigger Property="IsKeyboardFocused" Value="True">
                <Setter TargetName="FocusCue" Property="Visibility" Value="Visible"
/>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
</Window.Resources>
<Grid>
    <Button Width="100" Height="30" Margin="10" Template="{StaticResource
CustomButtonTemplate}">Button Template in action</Button>
</Grid>
</Window>
```

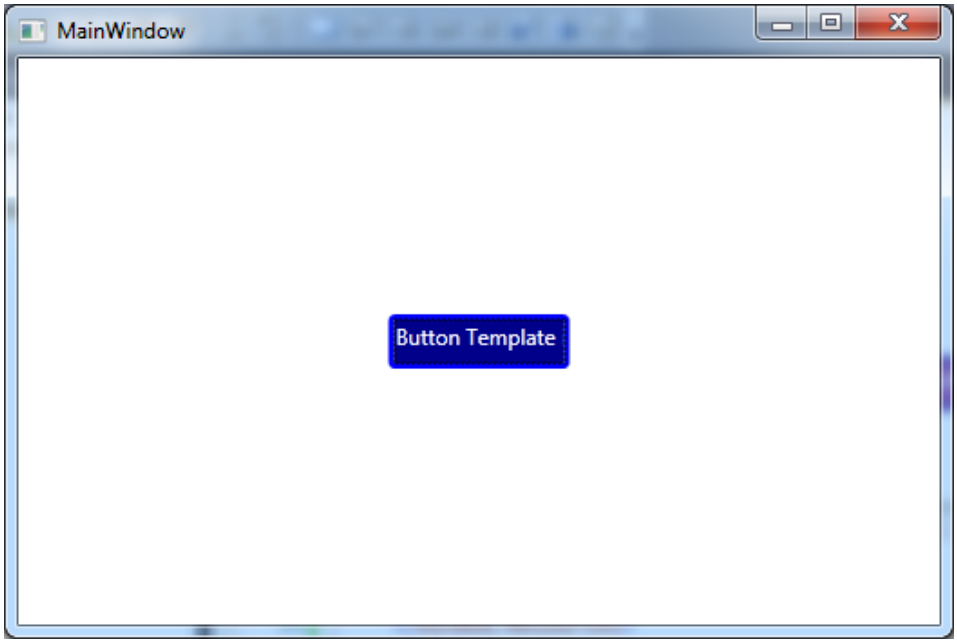

Figure 22: Button with Associated Trigger

Data templates

When data binding to a list control, you are forced to specify a single property, **DisplayMemberPath**. Wouldn't it be nice if you could define several controls, each with its own binding display so you end up with a flexible new list item for each item in the bound collection? You can accomplish this with data templates. A data template can be applied to two different types of controls: content controls and list controls.

The content controls use data templates through the **ContentTemplate** property. The **ItemsControl** list controls support data templates through the **ItemTemplate** property.

MainWindow.xaml

```
<Window x:Class="BindObservableCollectionToListbox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <ListBox Name="lstProducts" Width="300" Height="200">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Border BorderThickness="3" CornerRadius="6"
BorderBrush="AliceBlue" Background="DarkBlue">
```

```
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition />
                            <ColumnDefinition />
                            <ColumnDefinition />
                        </Grid.ColumnDefinitions>
                        <Grid.RowDefinitions>
                            <RowDefinition />
                        </Grid.RowDefinitions>
                        <Ellipse Grid.Column="0" Grid.Row="0" Width="20" Height="20"
Fill="Azure" />
                        <TextBlock Grid.Column="1" Grid.Row="0"
x:Name="txtProductName" Text="{Binding ProductName}" Foreground="White" />
                        <TextBlock Grid.Column="2" Grid.Row="0"
x:Name="txtProductPrice" Text="{Binding ProductPrice}" Foreground="White" />
                    </Grid>
                </Border>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
  </Grid>
</Window>
```

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
```

```
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace BindObservableCollectionToListbox
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml.
    /// </summary>
    public partial class MainWindow : Window
    {
        public ProductViewModel ViewModel { get; set; }

        public MainWindow()
        {
            InitializeComponent();

            ViewModel = new ProductViewModel();

            lstProducts.DataContext = ViewModel;
            lstProducts.ItemsSource = ViewModel;

        }
    }
}
```

Product.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BindObservableCollectionToListbox
{
    public class Product : INotifyPropertyChanged
```

```
    {
        private string _productName;
        private string _productPrice;

        public string ProductName
        {
            get
            {
                return _productName;
            }
            set
            {
                _productName = value;
                OnPropertyChanged("ProductName");
            }
        }

        public string ProductPrice
        {
            get
            {
                return _productPrice;
            }
            set
            {
                _productPrice = value;
                OnPropertyChanged("ProductPrice");
            }
        }

        public override string ToString()
        {
            return ProductName;
        }

        public event PropertyChangedEventHandler PropertyChanged;
```

```
        private void OnPropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;

            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

ProductViewModel.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BindObservableCollectionToListbox
{
    public class ProductViewModel : ObservableCollection<Product>
    {
        public ProductViewModel()
        {
            this.Add(new Product { ProductName = "Toy Truck", ProductPrice = "$20.99"
});
            this.Add(new Product { ProductName = "Baseball Glove", ProductPrice =
"$4.99" });
            this.Add(new Product { ProductName = "Baseball Bat", ProductPrice = "$7.99"
});
        }
    }
}
```

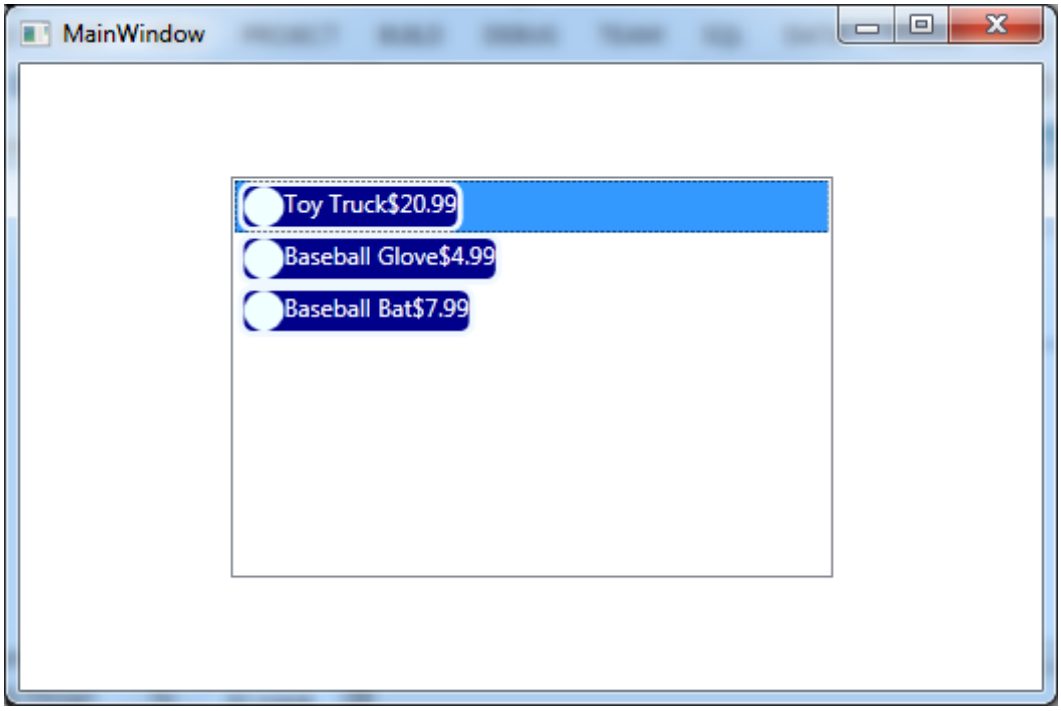

Figure 23: Content Controls with Data Templates

Chapter 9 WPF Tools and Frameworks

Expression Blend

As you learn more about WPF and XAML, your user interface designs will become more complex. Microsoft has created a tool that is geared toward the WPF user interface design specifically. The tool is called Expression Blend and it allows a designer to create extremely complex and stunning visual effects oriented to WPF XAML user interfaces. The application works side by side with Visual Studio, so a designer can work on designing and a developer can work on coding. Although we will not cover Expression Blend in detail, you can download a trial version at http://www.microsoft.com/en-us/download/details.aspx?id=10156.

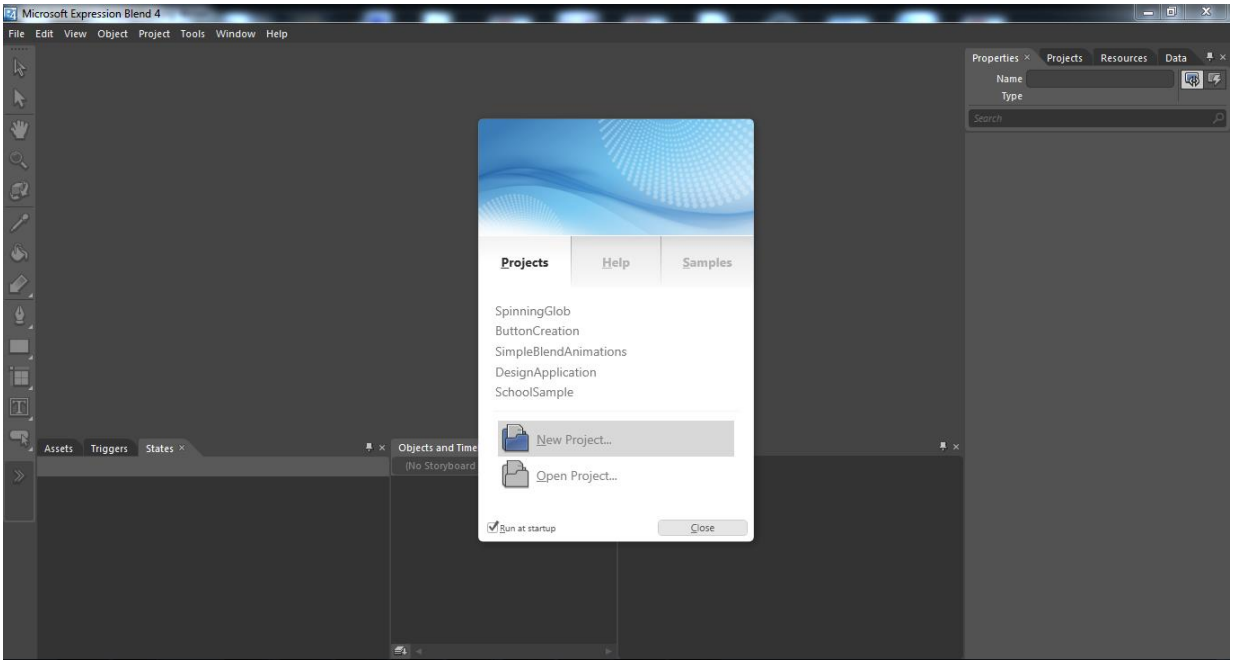

Figure 24: Microsoft Expression Blend

Examining complex user interfaces using Snoop

Snoop is an application that allows you as a developer to attach to any WPF application to view and modify the visual tree's elements and their properties. As you change the properties of the Snooped application's user interface, you can actually see the changes take effect in the application. You can view and change anything from visual properties to data binding properties and bound values. This is a must-have tool for any WPF developer.

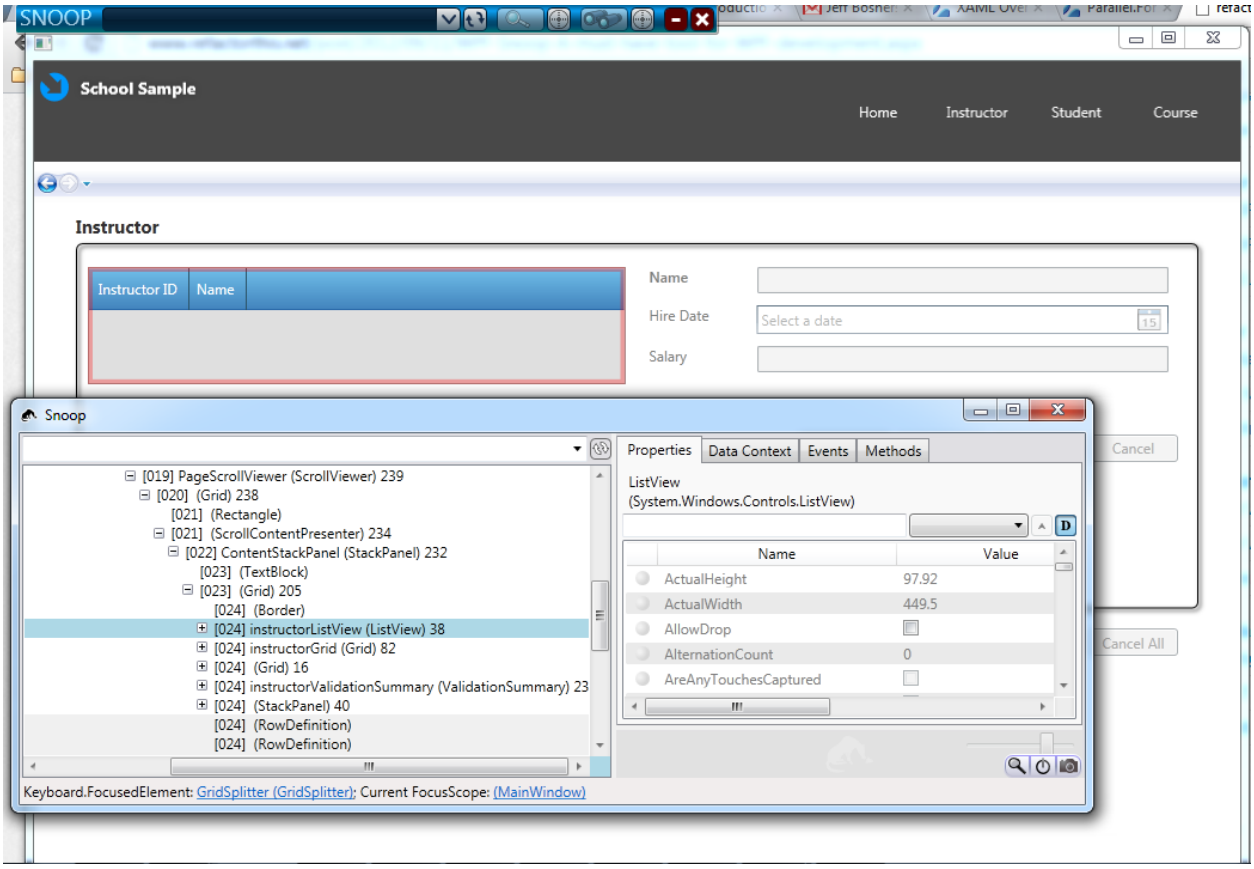

Figure 25: Snoop

Building modular WPF composite applications using Prism

As your experience with WPF grows, you will begin to look around for MVVM frameworks to facilitate the application of the pattern in the most effective way possible. One of the better-known frameworks used in WPF applications is the Prism framework. The Prism framework is a part of Microsoft's patterns and practices suite of software. It provides classes that allow you to create loosely coupled modules that are dynamically loaded and displayed inside of a main shell. The shell is divided into regions, which you dynamically fill with user control views that are found in your modules. You can read more about Prism here: http://www.microsoft.com/en-us/download/details.aspx?id=28950.

WPF XAML vs. WinRT XAML

With the release of Windows 8 and WinRT, developers now have another technology that uses XAML as the UI markup language. There are key differences between WPF XAML and WinRT XAML. This is because WPF is based on the .NET CLR, and WinRT supports a subset of the .NET Framework and is written in native code. Nevertheless, your WPF skills will come in handy if you decide to develop Windows 8 or WinRT applications. Here is a link to an MSDN article titled “Porting Silverlight or WPF XAML/code to a Windows Store app (Windows)”: http://msdn.microsoft.com/en-us/library/windows/apps/xaml/br229571.aspx.

Conclusion

This concludes the *WPF Succinctly* e-book. WPF is an amazing technology that allows you to develop stunning user interfaces for your Windows applications. The technology is limited only by your imagination. You are given full control to completely reimagine the existing user interface controls or to create new controls of your own. I hope you have enjoyed reading and working through the examples.