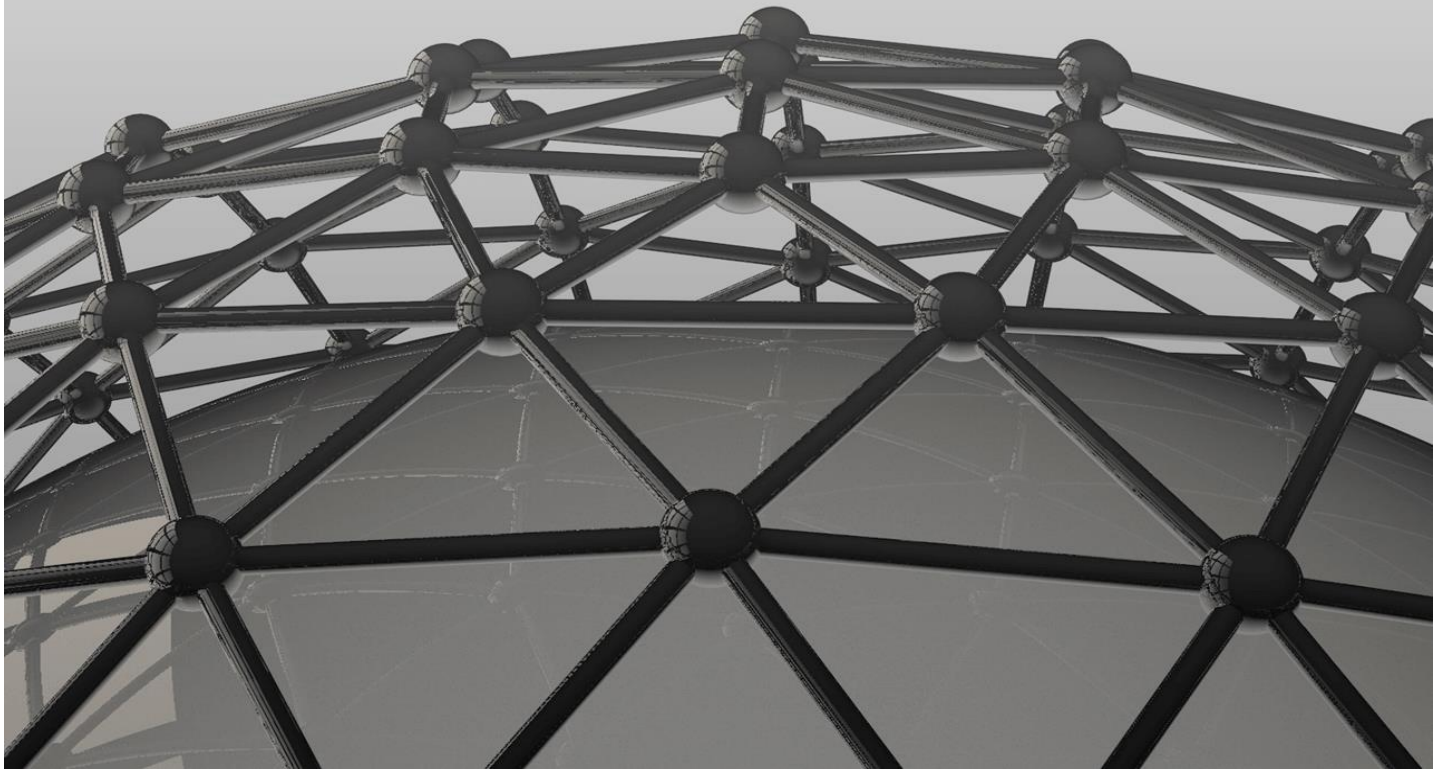
PROGRAMMING <XAML\>

Mahesh Chand



Programming XAML Beginners Guide

This free book is provided by courtesy of C# Corner and Mindcracker Network and its authors. Feel free to share this book with your friends and co-workers. Please do not reproduce, republish, edit or copy this book.

Mahesh Chand
Foundar C# Corner

Sam Hobbs
Editor, C# Corner

ABOUT THE AUTHOR

Mahesh Chand founded C# Corner in 1999 as a hobby code sharing website. Today, C# Corner reaches over 3 Million users each month and has become one of the most popular online communities for developers. Mahesh is a Software Solutions Architect and 9-time Microsoft MVP. Holding a Bachelor's degree in Mathematics and Physics and a Master's degree in Computer Science, Mahesh has written half a dozen books with publishers including Addison-Wesley and APress.

In his day job, Mahesh is a technical architect, startup advisor, and mentor. Some of the companies he has worked with includes Microsoft, J&J, Unisys, Adidas, Juniper, McGraw-Hill, and Exelon.

A Message from the Author

"C# Corner is a community with the main goal – learn, share and educate. You could help grow this community by telling your co-workers and share on your social media Twitter and Facebook accounts "

- Mahesh Chand

Table of Contents

XAML Language

This book discusses the Extensible Application Markup Language (XAML) programming language used to create user interfaces for Windows. XAML was first time introduced as a part of Microsoft .NET Framework 3.5. Officially speaking, XAML is a new descriptive programming language developed by Microsoft to write user interfaces for next-generation managed applications.

XAML is used to build user interfaces for Windows and Mobile applications that use Windows Presentation Foundation (WPF) and Windows Runtime.

This book is an introduction to the XAML language. In this book, we will learn how to define XAML elements, create controls and build screens using XAML Controls. By the end of this chapter, you will be able create user interfaces using XAML.

Purpose of XAML

The purpose of XAML is simple, to create user interfaces using a markup language that looks like XML. Most of the time, you will be using a designer to create your XAML but you're free to directly manipulate XAML by hand.

A typical user interface developed for Windows has a window or page as a parent container with one or more child containers and other user interface controls. Figure 1 shows a window with file child controls including two Radio Button controls, a Button control, a TextBox control and one TextBlock control.

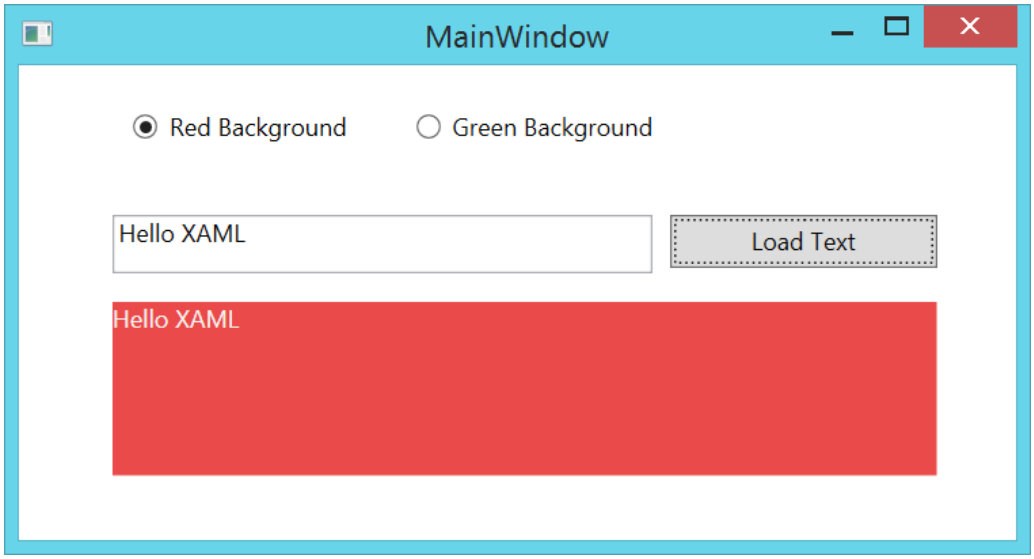


Figure 1

XAML allows us to represent the parent containers and child controls using markup script elements. Each parent, child containers and child control on the user interface is represented by a XAML element. XAML gives is the flexibility to build user interfaces at both design-time as well as run-time. Design-time usually means using a designer to drag-and-drop, resize and move screen layouts and control positons. Run-time usually means the containers and controls layout positioning is done by the code.

Hello XAML

XAML uses the XML format for elements and attributes. Each element in XAML represents an object which is an instance of a type. The scope of a type (class, enumeration etc.) is defined a namespace that physically resides in an assembly (DLL) of the .NET Framework library.

Similar to XML, a XAML element syntax always starts with an open angle bracket (<) and ends with a close angle bracket (>). Each element tag also has a start tag and an end tag. For example, a Button object is represented by the <Button> object element. The code snippet in Listing 1 represents a Button object element.

```
<Button></Button>
```

Listing 1

Alternatively, you can use a self-closing format to close the bracket. The code snippet in Listing 2 is equivalent to Listing 1.

```
<Button />
```

Listing 2

An object element in XAML represents a type. A type can be a control, a class or other objects defined in the framework library.

Hello XAML Sample

Let's create a WPF application using Visual Studio 2014.

Open Visual Studio and select a New Project > Windows Desktop > WPF Application. See Figure 2.

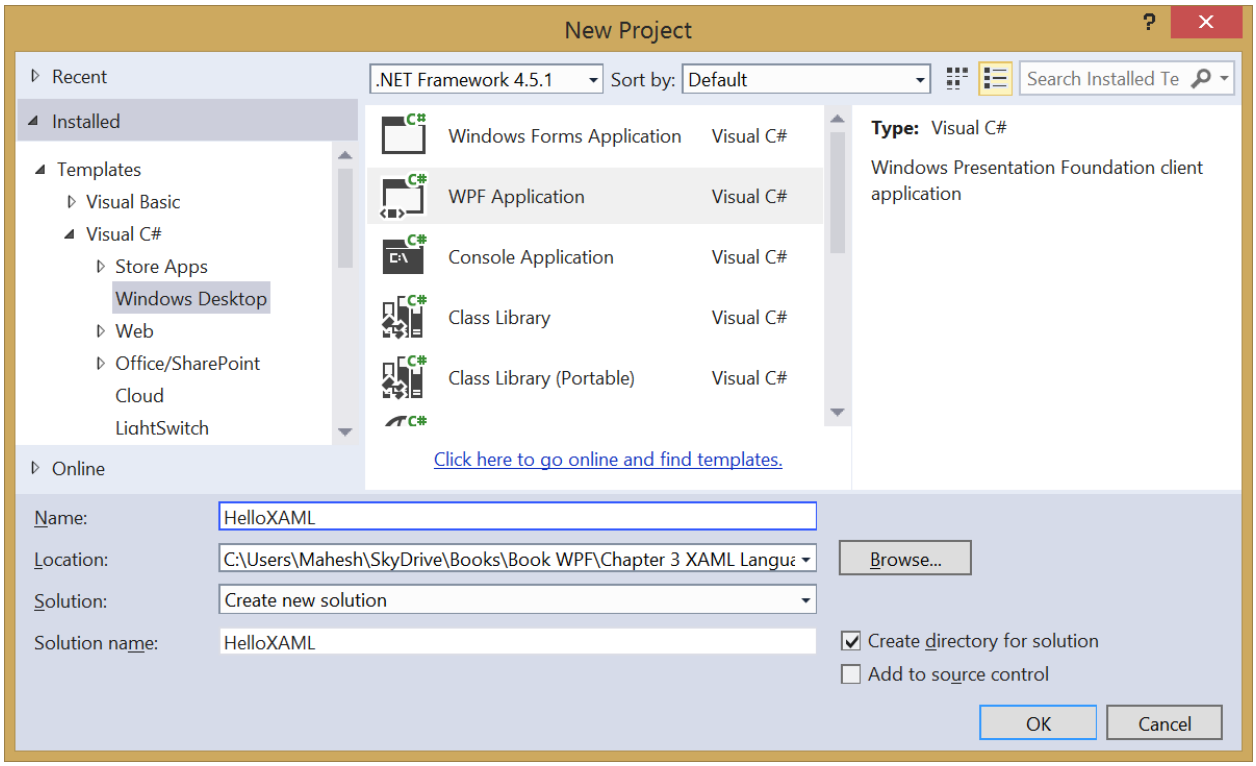


Figure 2.

Select OK and your WPF project is created.

You will land in Visual Studio designer with the MainWindow.xaml file opened in design mode.

See Figure 3. As you can see from Figure 3, Visual Studio presents two views, the design view and the XAML code view.

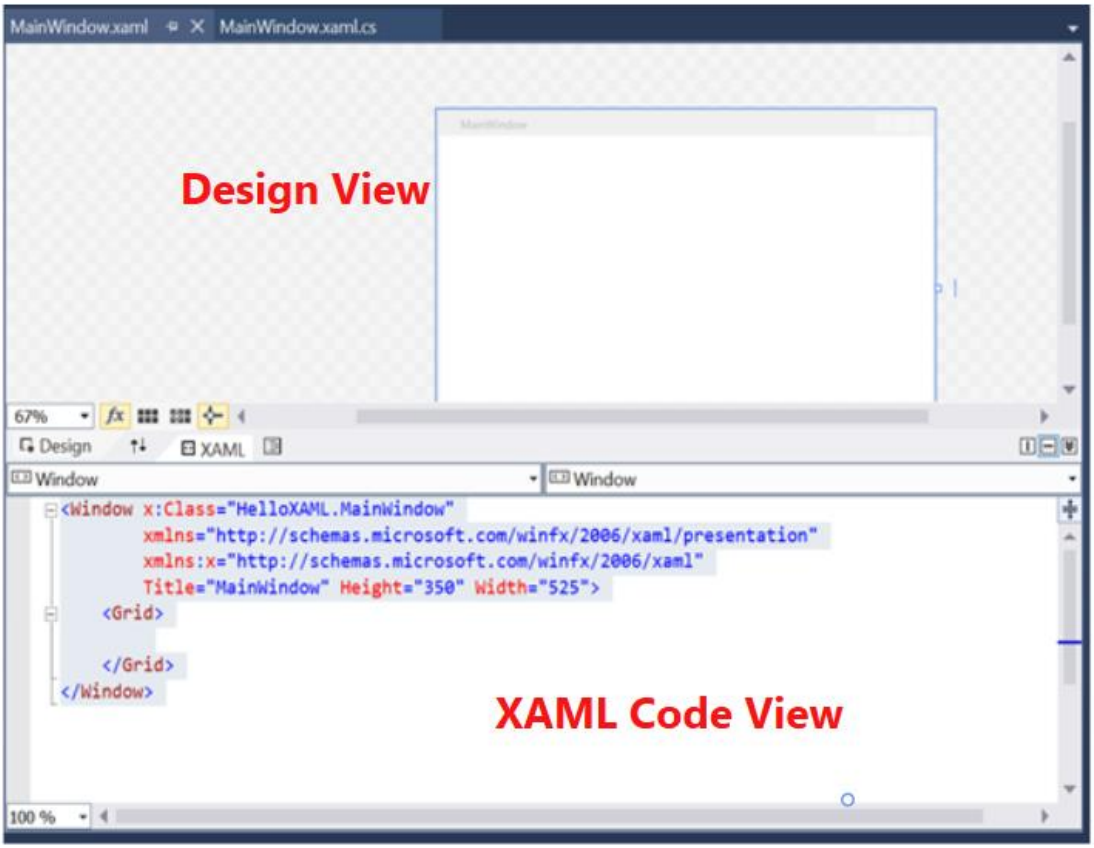


Figure 3.

On the design view, we can use the designer to create user interfaces by dragging and dropping controls from the Toolbox to the window, move them around, resize and reposition them, and set their properties. For all the action taken at the design-time, the designer generates the XAML code for you.

In the XAML code view, we can do the same thing by typing in the XAML code by hand. As soon as the correct XAML code is entered, the designer view is refreshed immediately to see the output of the code.

For learning purposes, for now, we will be entering most of the XAML code by hand.

Listing 3 lists the default XAML of MainWindow.xaml that has a root Window element and a child Grid element.

```
<Window x:Class="HelloXAML.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>
    </Grid>
</Window>
```

Listing 3

We will now add text to the window that will show “Hello XAML” on the screen.

The TextBlock control of XAML is used to display text. Let’s type in the code listed in Listing 4. The code snippet in Listing 4 creates a TextBlock control with its content “Hello XAML”.

```
<TextBlock>Hello XAML</TextBlock>
```

Listing 4

Alternatively, you can replace the code in Listing 4 with code in Listing 5.

```
<TextBlock Text="Hello XAML" />
```

Listing 5

The output of Listing 4 generates Figure 4.

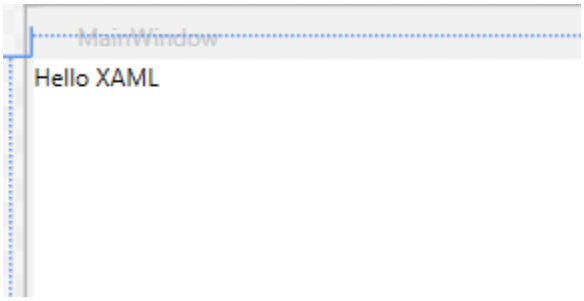


Figure 4

The attributes of XAML object elements represent properties of the types (containers and control). The code snippet in Listing 6 sets width, height, foreground and some font related properties of a TextBlock.

```
<TextBlock Width="300" Height="50"
        Foreground="Orange" FontFamily="Verdana"
       FontSize="15" FontWeight="Bold">
   Hello XAML
</TextBlock>
```

Listing 6

The output looks as in Figure 5.

Hello XAML

Figure 5

The Root Elements

Each XAML document must have a root element. The root element usually works as a container and defines the namespaces and basic properties of the element. Three most common root elements are <Windows />, <Page />, and <UserControl >. The <ResourceDirectory /> and <Application /> are other two root elements that can be used in a XAML file.

The Window element represents a Window container. The code snippet in Listing 7 shows a Window element with its Height, Width, Title and x:Name attributes. The x:Name attribute of an

element represents the ID of an element used to access the element in the code-behind. The code snippet also sets xmlns and xmlns:x attributes that represent the namespaces used in the code. The x:Class attribute represents the code-behind class name.

```
<Window x:Class="HelloXAML.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>
```

Listing 7

The Page element represents a page container. The code snippet in Listing 8 creates a page container. The code also sets the FlowDirection attribute that represents the flow direct of the contents of the page.

```
<Page
        xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
        xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
        x:Class="WPFApp.Page1"
        x:Name="Page"
        WindowTitle="Page"
        FlowDirection="LeftToRight"
        Width="640" Height="480"
        WindowWidth="640" WindowHeight="480">
</Page>
```

Listing 8

The UserControl element represents a user control container. The code snippet in Listing 9 represents a user control container.

```
<UserControl x:Class="HelloXAML.Page"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Width="400" Height="300">
</UserControl>
```

Listing 9

Namespaces

The part of the root element of each XAML are two or more attributes pre-fixed with xmlns and xmlns:x. See Listing 10.

```
<Window
        xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
        xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
        …… >
</Window>
```

Listing 10

The xmlns attribute indicates the default XAML namespace so the object elements in used in XAML can be specified without a prefix. The xmlns:x attribute indicates an additional XAML namespace, which maps the XAML language namespace http://schemas.microsoft.com/winfx/2006/xaml.

Additionally, the x: prefix is used with more than just the namespace. Here are some common x:prefix syntaxes that are used in XAML.

- x:Key: Sets a unique key for each resource in a ResourceDictionary.
- x:Class: Class name provides code-behind for a XAML page.
- x:Name: Unique run-time object name for the instance that exists in run-time code after an object element is processed.
- x:Static: Enables a reference that returns a static value that is not otherwise a XAML-compatible property.
- x:Type: Constructs a Type reference based on a type name.

We will see these directives in action in the later parts of this chapter.

XAML Markup and Code-Behind

Since XAML is a markup language it may not be an ideal choice for writing lengthy code. XAML provides an option to associate a XAML file with a code file such as C# or VB.NET. The code file (.cs or .vb) can host all the backend code while the XAML file (.xaml) presents the user interface (UI) screen. The code file associated with a XAML file is also known as code-behind.

In our HelloXAML application, if you open the Solution Explorer, there is a MainWindow.xaml file and if you expand it, there is a MainWindow.xaml.cs file. See Figure 6.

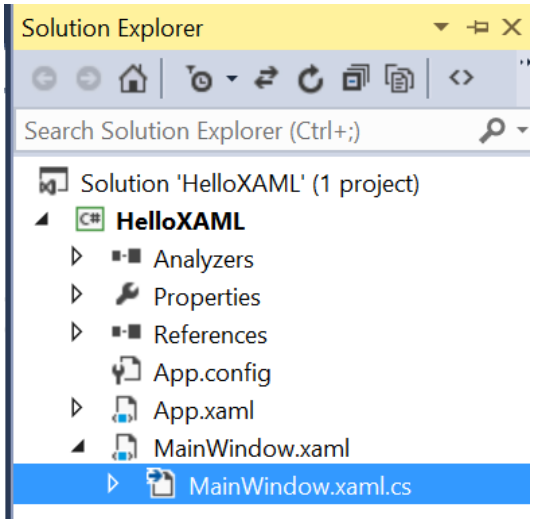


Figure 6

Double-clicking on MainWindow.xaml will open the designer view with the XAML at the bottom of the designer (or top depending on your settings). The XAML listed in Listing 11 contains all the XAML code for the Window and its child controls.

```
<Window x:Class="HelloXAML.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>
   <TextBlock Width="300" Height="50" Foreground="Orange" FontFamily="Verdana"
   FontSize="15" FontWeight="Bold" x:Name="HelloTextBlock"> Hello XAML
   </TextBlock>
</Grid>
</Window>
```

Listing 11

The x:Class directive is used to associate a code file with the XAML file. If you look at the code listed in Listing 12, the first line of the code contains a x:Class directive pointing to the HelloXAML.MainWindow.

```
<Window x:Class="HelloXAML.MainWindow"
.....>
```

Listing 12

In this example, the HelloXAML is the namespace and MainWindow is the class that contains all the code associated with the XAML controls. The physical location of the code is hosted in the MainWindow.cs file. Double-click on the MainWindow.xaml.cs file and the code is listed in Figure 7 that contains a partial class MainWindow inherited from the Window type.

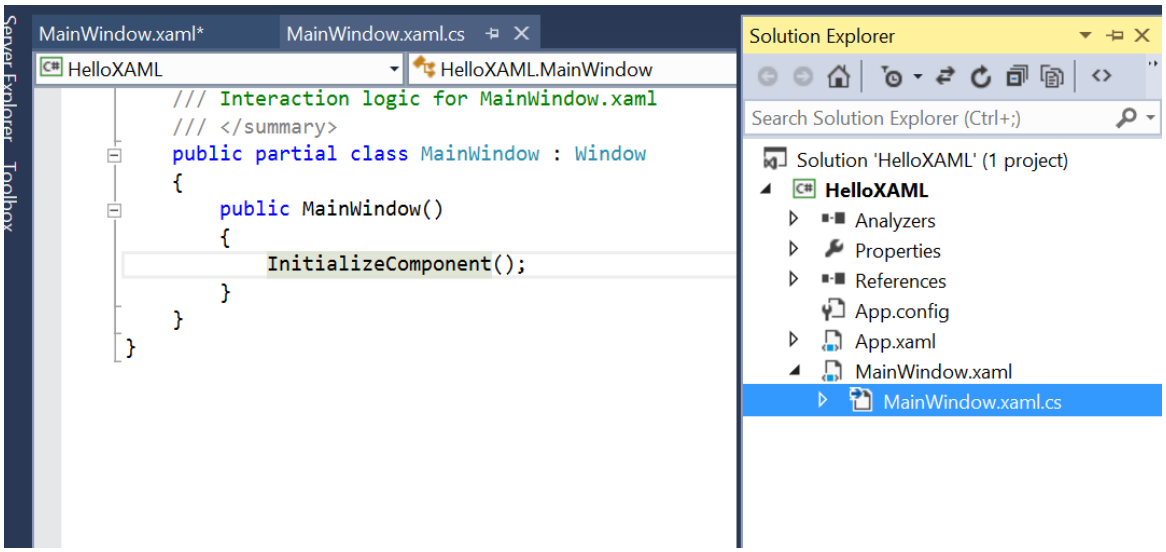


Figure 7

This is the code-behind where we will be writing our C# or VB.NET code.

x:Name

All element objects defined in the XAML file can be represented with a unique name, x:Name that can be used to access the objects in the code-behind file. Let's update our XAML code for the TextBlock element by adding a new x:Name value as listed in Listing 13.

```
<Grid>
   <TextBlock Width="300" Height="50" Foreground="Orange" FontFamily="Verdana"
   FontSize="15" FontWeight="Bold" x:Name="HelloTextBlock"> Hello XAML
   </TextBlock>
</Grid>
```

Listing 13

Now, let's return to our code-behind MainWindow.xaml.cs file and add one line after the InitializeComponent(). See Listing 14.

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
      // Update TextBlock property at run-time
      HelloTextBlock.Text = "Hello XAML Updated";
   }
}
```

Listing 14

In Listing 14, you can see we can access the TextBlock object using the x:Name value HelloTextBlock and set the HelloTextBlock.Text property to "Hello XAML Updated". Now, if you build and run the code, the new outlook looks like Figure 8.

Figure 8

Design-time versus Run-time

Accessing and setting an element object attributes and properties in XAML is called design-time and accessing and setting an element object properties and attributes in code-behind are called at run-time or dynamically. In Listing 14, we set the TextBlock's Text property at run-time. Run-time code is always executed after the design-time.

Inline Code

In the preceding examples, we used a code-behind file to write C# code. But you can also write your code in the XAML file itself. Using C# code in XAML itself is called inline coding. The x:Code directive is used to write inline code in XAML. The code that is defined inline can interact with the XAML on the same page.

The code listed in Listing 15 demonstrates the use of inline coding using x:Code. The code must be surrounded by <CDATA[...]]> to escape the contents for XML, so that a XAML processor (interpreting either the XAML schema or the WPF schema) will not try to interpret the contents literally as XML.

```
<Grid>
  <TextBlock Width="300" Height="50" Foreground="Orange" FontFamily="Verdana"
  FontSize="15" FontWeight="Bold" x:Name="HelloTextBlock"> Hello XAML
  </TextBlock>
    <x:Code>
      <![CDATA[
      // Put code here
    ]]>
    </x:Code>
  </Grid>
```

Listing 15

Note: I highly recommend not using inline coding. It defeats the purpose of code maintainability.

Elements and Attributes

A type in WPF or Windows RT is represented by an XAML element. The <Page> and <Button> elements represent a page and a button control respectively. The XAML Button element listed in

Listing 16 represents a button control.

```
<Button />
```

Listing 16

Each of the elements such as <Page> or <Button> have attributes that can be set within the element itself. An attribute of an element represents a property of the type. For example, a Button has Height, Width, Background and Foreground properties that represent the height, width, foreground color and background color of the button respectively. The Content property of the Button represents the text of a button control. The x:Name property represents the unique ID of a control that may be used to access a control in the code behind.

The code snippet in Listing 17 sets the ID, height, width, background color, foreground color, font name and size and content of a button control.

```
<Button Content="Click Me" Width="200" Height="50"
      Background="Orange" Foreground="Blue"
      FontSize="20" FontFamily="Georgia" FontWeight="Bold"
      x:Name="ClickButton">
</Button>
```

Listing 17

Figure 9 is the result of Listing 17. As you can see from Figure 9, the button has a White foreground and Red background, with the size specified in the code.

Figure 9

Content Property

Each XAML object element is capable of displaying different content types. XAML provides a special property called Content that works to display the content of the element depending on the element capabilities. For example, a Content property of a Button can be a set to a string, an

object, a UIElement, or even and container. However, the Content property of a ListBox is set using the Items property.

Note: Some XAML object elements may not have the Content property available directly. It must be set through a property.

The code snippet in Listing 18 creates a Button control and sets its Content property to a string "Hello XAML". The output looks as in Figure 10.

```
<Button Height="50" Margin="10,10,350,310" Content="Hello XAML" />
```

Listing 18

Listing 19 is an alternative way to set the Content property of a Button.

```
<Button Height="50" Margin="10,10,350,310">Hello XAML</Button>
```

Listing 19

The output of Listing 19 looks the same as in Figure 10.

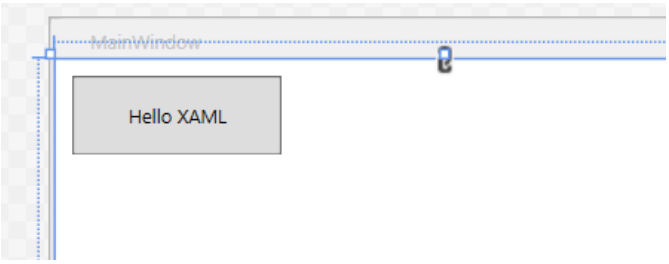


Figure 10

A Button element can display other child elements as its content. The code listed in Listing 20 sets a Rectangle element as the content of the Button.

```
<Button Height="80" Margin="10,80,300,170">
<Rectangle Height="60" Width="120" Fill="Green"/>
</Button>
```

Listing 20

The output of Listing 20 looks the same as in Figure 11.

Figure 11

Content property can also be a container or a parent element hosting child elements. The code listed in Listing 21 sets a StackPanel container with 5 child elements as the content of the Button.

```
<Button Margin="10,201,100,40">
      <StackPanel Orientation="Horizontal">
        <Ellipse Height="60" Width="60" Fill="Red"/>
        <TextBlock TextAlignment="Center"><Run Text=" Red Circle"/></TextBlock>
        <TextBlock TextAlignment="Center"><Run Text="     "/></TextBlock>
        <Rectangle Height="60" Width="120" Fill="Green"></Rectangle>
        <TextBlock TextAlignment="Center"><Run Text=" Green Rectangle"/></TextBlock
      </StackPanel>
</Button>
```

Listing 21

The output of Listing 21 looks the same as in Figure 12.

Figure 11

The final XAML code is listed in Listing 22.

```
<Window x:Class="ContentPropertySample.MainWindow"
      xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
      xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
      Title="MainWindow" Height="400" Width="500"
      <Grid x:Name="ParentGrid">
           <Button Height="50" Margin="10,10,350,310" Content="Hello XAML" />
           <Button Height="80" Margin="10,80,300,170">
                <Rectangle Height="60" Width="120" Fill="Green"></Rectangle>
           </Button>
           <Button Margin="10,201,100,40">
               <StackPanel Orientation="Horizontal">
                    <Ellipse Height="60" Width="60" Fill="Red"/>
                    <TextBlock TextAlignment="Center"><Run Text=" Red Circle"/></TextBlock>
                    <TextBlock TextAlignment="Center"><Run Text="     "/></TextBlock>
                    <Rectangle Height="60" Width="120" Fill="Green"></Rectangle>
                    <TextBlock TextAlignment="Center"><Run Text=" Green
                                        Rectangle"/></TextBlock>
               </StackPanel>
           </Button>
       </Grid>
</Window>
```

Listing 22

The output of Listing 22 looks the same as in Figure 13.

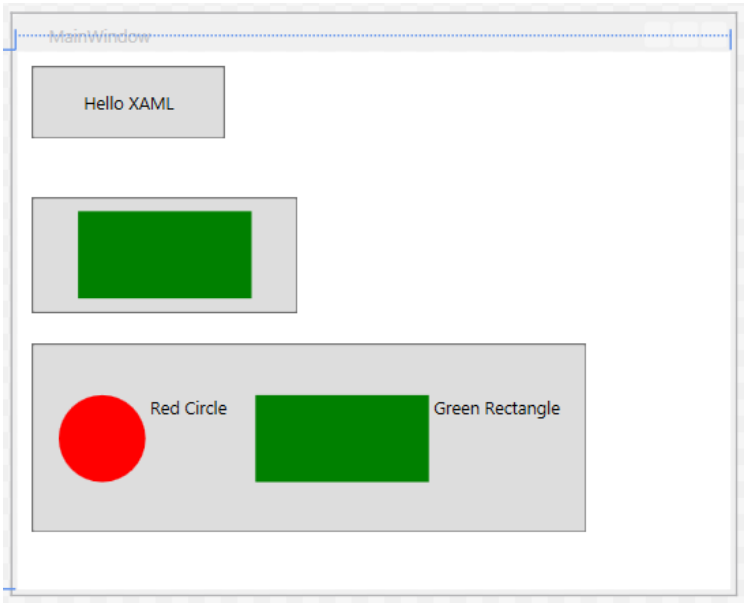


Figure 13

As you can imagine from the preceding examples, you can pretty much host any user interfaces as the content of a XAML element.

The code listed in Listing 23 creates the preceding Button controls dynamically in the code and sets their Content properties to a string, a Rectangle and a StackPanel respectively.

```
// Button with string content
Button helloButton = new Button();
helloButton.Margin = new Thickness(10,10,350,310);
helloButton.Content = "Hello XAML";
// Button with a UIElement
Button buttonWithRectangle = new Button();
buttonWithRectangle.Height = 80;
buttonWithRectangle.Margin = new Thickness(10, 80, 300, 170);
// Create a Rectangle
Rectangle greenRectangle = new Rectangle();
greenRectangle.Height = 60;
greenRectangle.Width = 120;
greenRectangle.Fill = Brushes.Green;
// Set Rectangle as Button.Content
```

```
buttonWithRectangle.Content = greenRectangle;
// Button with a Container, StackPanel
Button buttonWithStackPanel = new Button();
buttonWithStackPanel.Margin = new Thickness(10, 10, 350, 310);
// Create a StackPanel and set its orinetation to horizontal
StackPanel stackPanel = new StackPanel();
stackPanel.Orientation = Orientation.Horizontal;
// Create an Ellipse
Ellipse redEllipse = new Ellipse();
redEllipse.Width = 60;
redEllipse.Height = 60;
redEllipse.Fill = Brushes.Red;
// Add to StackPanel
stackPanel.Children.Add(redEllipse);
// Create a TextBlock
TextBlock textBlock1 = new TextBlock();
textBlock1.TextAlignment = TextAlignment.Left;
textBlock1.Text = "Red Circle";
// Add to StackPanel
stackPanel.Children.Add(textBlock1);
// Create a TextBlock
TextBlock space = new TextBlock();
space.TextAlignment = TextAlignment.Center;
space.Text = "      ";
// Add to StackPanel
stackPanel.Children.Add(space);
// Create a Rectangle
Rectangle greenRectangle2 = new Rectangle();
greenRectangle2.Height = 60;
greenRectangle2.Width = 120;
greenRectangle2.Fill = Brushes.Green;
// Add to StackPanel
stackPanel.Children.Add(greenRectangle2);
// Create a TextBlock
TextBlock textBlock2 = new TextBlock();
textBlock2.TextAlignment = TextAlignment.Left;
```

```
textBlock2.Text = "Green Rectangle";
// Add to StackPanel
stackPanel.Children.Add(textBlock2);
// Set StackPaenl as Button.Content
buttonWithStackPanel.Content = stackPanel;
// Add dynamic button controls to the Window
ParentGrid.Children.Add(helloButton);
ParentGrid.Children.Add(buttonWithRectangle);
ParentGrid.Children.Add(buttonWithStackPanel);
```

Listing 23

In this article, we saw the meaning of the Content property available to XAML elements and how to use it in our application.

Events

Windows controls have most of the common events such as Click, GotFocus, LostFocus, KeyUp, KeyDown, MouseEnter, MouseLeave, MouseLeftButtonDown, MouseRightButtonDown and MouseMove. An event in XAML has an event handler that is defined in the code-behind and the code is executed when the event is raised.

Let's see this by an example.

Create a new WPF Application and add a Button and a TextBlock control to the Window. Position and format the control the way you like. My final code is listed in Listing 18.

```
<Grid >
    <Button x:Name="HelloButton" Content="Click Me"
        Width="150" Height="40" Margin="11,10,357.667,280.667"
        FontSize="16" />
    <TextBlock x:Name="HelloTextBlock" Width="400" Height="100"
         Margin="10,57,208.667,163.667" Background="LightGray"
         FontSize="30" Foreground="Orange"/>
</Grid>
```

Listing 18

We will now add the Button click event handler and write the code that will add some text to the TextBlock on the button click. As shown in Listing 19, add the click event hander called HelloButton_Click.

```
<Button x:Name="HelloButton" Content="Click Me"
        Width="150" Height="40" Margin="11,10,357.667,280.667"
        FontSize="16" Click="HelloButton_Click" />
```

Listing 19

Now go to the code-behind and add the following code listed in Listing 20. The code in Listing 20 is the HelloButton's click event handler where we update the Text of the TextBlock.

```
void HelloButton_Click(object sender, RoutedEventArgs e)
{
   HelloTextBlock.Text = "HelloButton is clicked.";
}
```

Listing 20

Build and run the application and click on the button, the output will change to Figure 10.

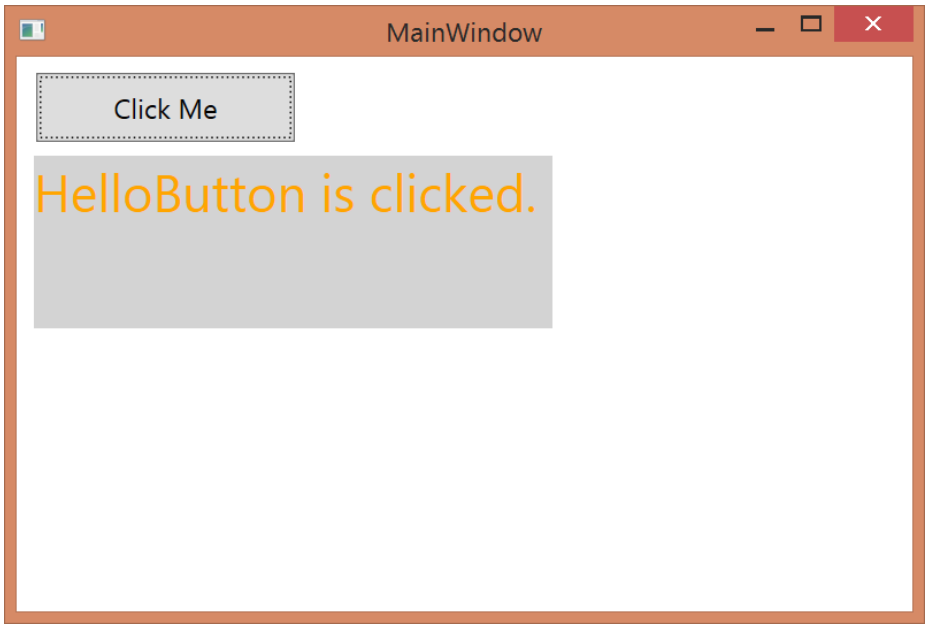


Figure 10

Creating Controls at Run-time

In the previous sections, we created controls at design-time by defining controls in the XAML code. Each XAML element has a corresponding type in .NET Framework that may be used to create and work with a control. For example, the Button type may be used to create and work with a button control.

The code listed in Listing 21 creates a button dynamically and sets its properties.

```
private void CreateControlsDynamically()
{
   Button clickButton = new Button();
   clickButton.Width = 200;
   clickButton.Height = 50;
   clickButton.Background = new SolidColorBrush(Colors.Orange);
   clickButton.Foreground = new SolidColorBrush(Colors.Black);
   clickButton.FontSize = 20;
   clickButton.FontWeight = FontWeights.Bold;
   clickButton.Name = "HelloButton";
   clickButton.Content = "Click Me";
   LayoutRoot.Children.Add(clickButton);
}
```

Listing 21

Container, Parent and Child Controls

XAML uses the tree presentation to represents its parent and child controls. Figure 11 is a typical screen with a parent container, a child container and a few child controls.

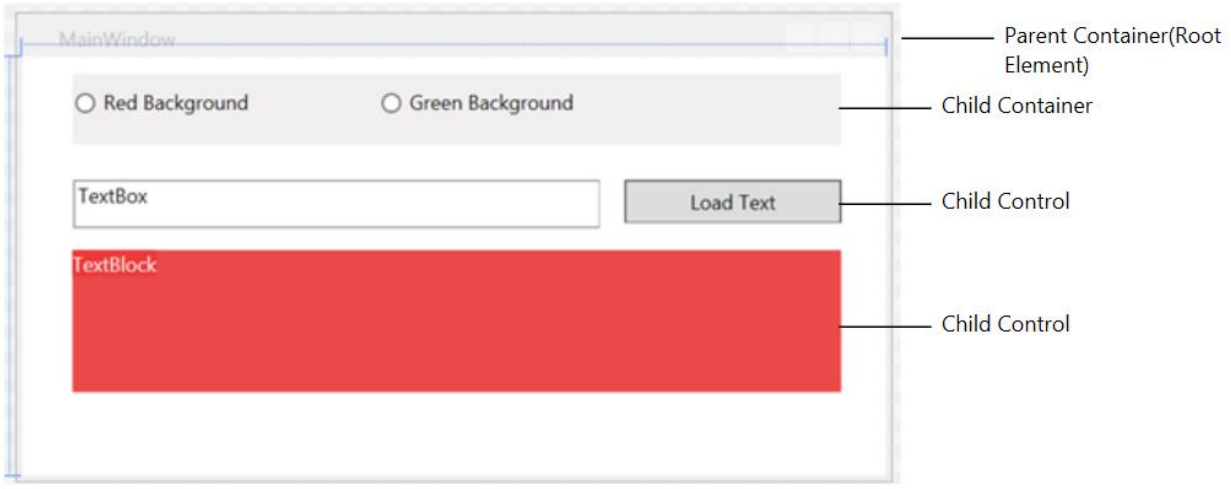


Figure 11

The tree representation of Figure 10 is displayed in Figure 12.

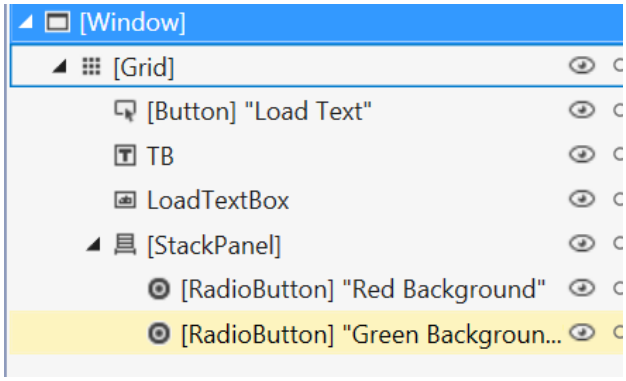


Figure 12

As you can see from Figure 12, the Window root element has a child container, Grid. The Grid container hosts a Button, TextBlock, TextBox and StackPanel. The StackPanel is also being used as a container for two Radio Button controls.

The XAML presentation of Figure 12 is listed in Listing 22.

```
<Window x:Class="HelloXAMLSample.MainWindow"
      xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
      xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
      Title="MainWindow" Height="280.666" Width="525">
   <Grid>
      <Button Content="Load Text" HorizontalAlignment="Left" Margin="361,73,0,0"
            VerticalAlignment="Top" Width="131" Height="26"
            RenderTransformOrigin="0.377,2.526" Click="Button_Click"/>
      <TextBlock x:Name="TB" HorizontalAlignment="Left" Margin="30,116,0,0"
               TextWrapping="Wrap" VerticalAlignment="Top" Height="85"
               Width="462" Background="#FFEB4A4A" Foreground="#FFF5F3F3">
         <Run Background="#FFEE3A3A" Text="TextBlock"/></TextBlock>
      <TextBox HorizontalAlignment="Left" Height="29" Margin="30,73,0,0"
            TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top"
            Width="317" x:Name="LoadTextBox"/>
      <StackPanel HorizontalAlignment="Left" Height="43" Margin="30,10,0,0"
VerticalAlignment="Top" Width="462" Background="#FFF1EFEF" Orientation="Horizontal">
         <RadioButton Content="Red Background" Margin="0,10,0,0.333" Width="184"
Checked="RadioButton_Checked_1"/>
         <RadioButton Content="Green Background" Margin="0,10,0,0.333" Width="207"/>
      </StackPanel>
   </Grid>
</Window>
```

Listing 22

Shapes

In XAML, each graphics shape, such as a line or a rectangle, is represented by an element object. For example, the <Line /> element represents a line and the <Rectangle /> element represents a rectangle shape. These elements have attributes that represent object properties. In the .NET Framework, each XAML element object is associated with a type. For example, the Line and Rectangle classes represent a line and a rectangle shape respectively.

The code snippet in Listing 23 creates a line and sets its stroke and stroke thickness properties. X1, Y1 is the starting point and X2, Y2 is the end point of the line.

```
<Line Stroke="#000fff" StrokeThickness="2" X1="100"  Y1="100" X2="300" Y2="100"/>
```

Listing 23

The code snippet in Listing 24 creates a line, a rectangle, an ellipse, a polygon and a polyline shape using XAML elements.

```
<Canvas x:Name="LayoutRoot" Background="White">
  <!-- Create a line in XAML -->
  <Line Canvas.Left="10" Canvas.Top="20"
    X1="0" Y1="0"
    X2="250" Y2="0"
    Stroke="Red"
    StrokeThickness="2" />
  <!-- Create a Rectangle in XAML -->
  <Rectangle Canvas.Left="10" Canvas.Top="40"
    Width="200"
    Height="100"
    Fill="Blue"
    Stroke="Black"
    StrokeThickness="2" />
  <!-- Create an Ellipse in XAML -->
  <Ellipse Canvas.Left="10" Canvas.Top="150"
    Width="200"
    Height="100"
    Fill="Yellow"
    Stroke="Black"
    StrokeThickness="2" />
  <!-- Create a polygon-->
  <Polygon Canvas.Left="200" Canvas.Top="0"
    Points="50, 100 200, 100 200, 200 300, 30"
    Stroke="Black" StrokeThickness="4"
    Fill="LightGreen"  />
  <!-- Create a polyline in XAML-->
  <Polyline Canvas.Left="250" Canvas.Top="100"
    Points="10,100 100,200 200,30 250,200 200,150"
```

```
    Stroke="Black"
    StrokeThickness="4"
    />
</Canvas>
```

Listing 24

The output generated by Listing 24 looks as in Figure 13 with a line, a rectangle, an ellipse, a polygon and a polyline.

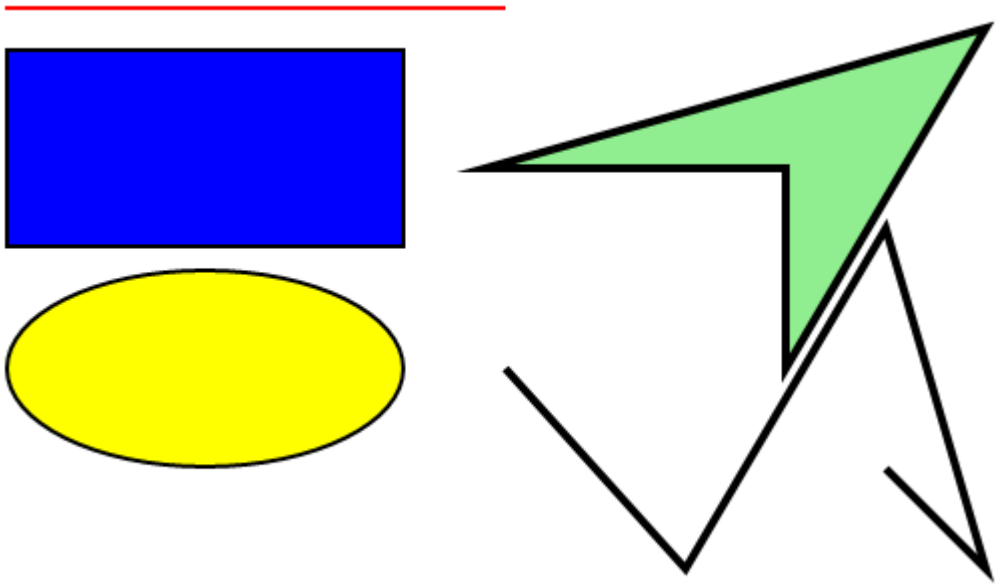


Figure 13

Brushes

Brushes are responsible for drawing and painting anything on a surface. In XAML, each brush is represented by an element object. For example, the SolidColorBrush element represents a solid brush. There are five brushes available in XAML: Solid Brush, Linear Gradient Brush, Radial Gradient Brush, Visual Brush and Image Brush.

The code snippet in Listing 25 draws various rectangles using these brushes.

```
<Canvas x:Name="LayoutRoot" Background="White">
  <!-- SolidColorBrush-->
```

```
<Rectangle Canvas.Left="10" Canvas.Top="10"
   Width="200"
   Height="100"
   Stroke="Black"
   StrokeThickness="4">
         <Rectangle.Fill>
            <SolidColorBrush Color="Blue" />
         </Rectangle.Fill>
</Rectangle>
<!-- LinearGradientBrush-->
<Rectangle Canvas.Left="10" Canvas.Top="120"
        Width="200" Height="100">
   <Rectangle.Fill>
      <LinearGradientBrush StartPoint="0,0" EndPoint="1,1" >
         <GradientStop Color="Blue" Offset="0" />
         <GradientStop Color="Red" Offset="1.0" />
      </LinearGradientBrush>
   </Rectangle.Fill>
</Rectangle>
<!-- RadialGradientBrush -->
<Rectangle Canvas.Left="10" Canvas.Top="230"
   Width="200" Height="100" Stroke="Black" >
   <Rectangle.Fill>
      <RadialGradientBrush
   GradientOrigin="0.5,0.5"
   Center="0.5,0.5" >
         <RadialGradientBrush.GradientStops>
            <GradientStop Color="Blue" Offset="0" />
            <GradientStop Color="Red" Offset="1.0" />
         </RadialGradientBrush.GradientStops>
      </RadialGradientBrush>
   </Rectangle.Fill>
</Rectangle>
<!-- ImageBrush-->
<Rectangle Canvas.Left="230" Canvas.Top="10"
   Width="200"
```

```
        Height="100"
        Stroke="Black"
        StrokeThickness="4">
            <Rectangle.Fill>
              <ImageBrush ImageSource="dock.jpg" />
            </Rectangle.Fill>
    </Rectangle>
</Canvas>
```

Listing 25

The output generated by Listing 25 looks as in Figure 14 that has four rectangles drawn using various brushes.

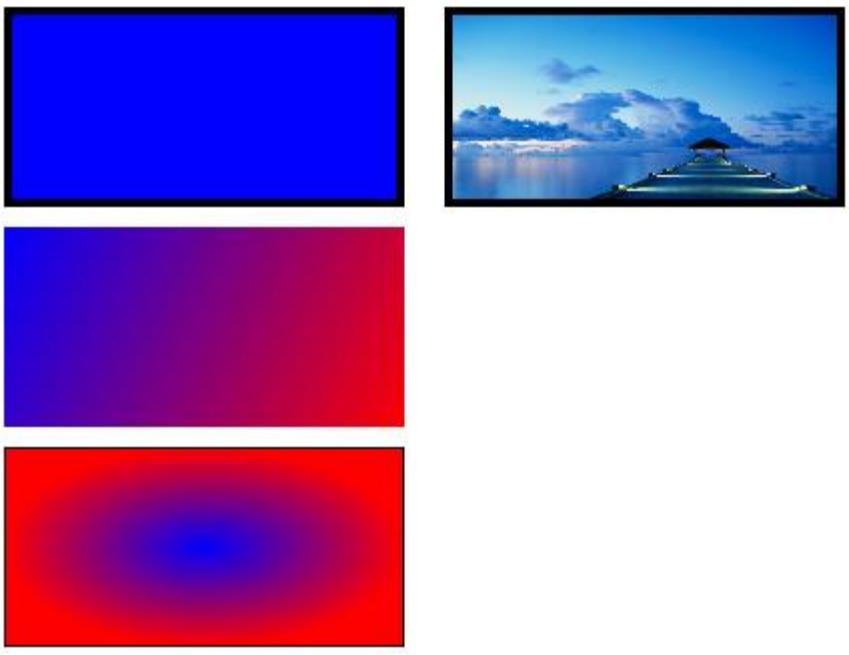


Figure 14

Special Characters in XAML

XAML uses the Unicode UTF-8 file format for special characters. However, there is a set of commonly-used special characters that are handled differently. These special characters follow

the World Wide Web Consortium (W3C) XML standard for encoding. Here is a list of these special characters.

< = <

> = >

& = &

“ = &qout;

The code snippet in Listing 26 shows how to use these characters.

```
<TextBlock>
   &lt;   <!-- Less than symbol -->
   &gt;   <!-- Greater than symbol -->
   &amp;  <!-- Ampersand symbol -->
   &quot; <!-- Double quote symbol -->
</TextBlock>
```

Listing 26

Read and Write XAML In Code

XAML is mostly used at design-time but there may be a time when you may want to create XAML dynamically and/or load XAML in your code. The XamlWriter and the XamlReader classes are used to create and read XAML in code. The XamlWriter and the XamlReader classes are defined in the System.Windows.Markup namespace and must be imported to use the classes.

```
using System.Windows.Markup;
```

The XamlWriter.Save method takes an object as an input and creates a string containing the valid XAML. The code snippet in Listing 27 creates a Button control using code and saves it in a string using the XamlWriter.Save method.

```
// Create a Dynamic Button.
Button helloButton = new Button();
helloButton.Height = 50;
helloButton.Width = 100;
```

```
helloButton.Background = Brushes.AliceBlue;
helloButton.Content = "Click Me";
// Save the Button to a string.
string dynamicXAML = XamlWriter.Save(helloButton);
```

Listing 27

The XamlReader.Load method reads the XAML input in the specified Stream and returns an object that is the root of the corresponding object tree. The code snippet in Listing 28 creates a XmlReader from a XAML and then creates a Button control using the XamlReader.Load method.

```
// Load the button
XmlReader xmlReader = XmlReader.Create(new StringReader(dynamicXAML));
Button readerLoadButton = (Button)XamlReader.Load(xmlReader);
```

Listing 28

Styling Controls in XAML

XAML is the universal language for Windows Presentation Foundation (WPF), Silverlight, and Windows Store app user interfaces such as Windows, Pages and controls. In this article, we will learn how to create and use styles on UI elements using XAML. Once you know how this is done in XAML, you can use the same approach in your WPF, Silverlight, and Windows Store apps.

This sample is created using a WPF application using Visual Studio 2012.

Styling is a way to group similar properties in a single Style element and apply on multiple XAML elements.

Let's have a look at the XAML code in Listing 1 that generates Figure 1. This code creates a Window with three Button controls, a TextBlock and a TextBox.

```
<Window x:Class="StylesSample.Window1"

  xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
  xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
  Title="Window1" Height="291" Width="455">
  <Grid Height="236" Width="405">
    <TextBlock Margin="12,52,26,83" Name="textBlock1"
```

```
            Background="Gray" Foreground="Orange"
            FontFamily="Georgia" FontSize="12"
            Width="370" Height="100" />
    <TextBox Height="30" Margin="11,16,155,0" Name="textBox1" VerticalAlignment="Top"
/>
    <Button HorizontalAlignment="Right" Margin="0,14,26,0"
        Name="button1" VerticalAlignment="Top"
        Height="30" Width="120"
        FontFamily="Verdana" FontSize="14" FontWeight="Normal"
        Foreground="White" Background="DarkGreen"
        BorderBrush="Black" >
      Browse
    </Button>
    <Button HorizontalAlignment="Right" Margin="0,0,30,39" Name="button2"
        VerticalAlignment="Bottom"
        Height="30" Width="120"
        FontFamily="Verdana" FontSize="14" FontWeight="Normal"

        Foreground="White" Background="DarkGreen"
        BorderBrush="Black" >
      Spell Check
    </Button>
    <Button Margin="129,0,156,39" Name="button3" VerticalAlignment="Bottom"
        Height="30" FontFamily="Verdana" FontSize="14" FontWeight="Normal"
        Foreground="White" Background="DarkGreen"
        BorderBrush="Black" >
      Save File
    </Button>
  </Grid>
</Window>
```

Listing 29

The output of Listing 1 XAML creates a window that looks as in Figure 1. As you can see from Figure 1, all three buttons have the same width, height, background, foreground, and fonts.

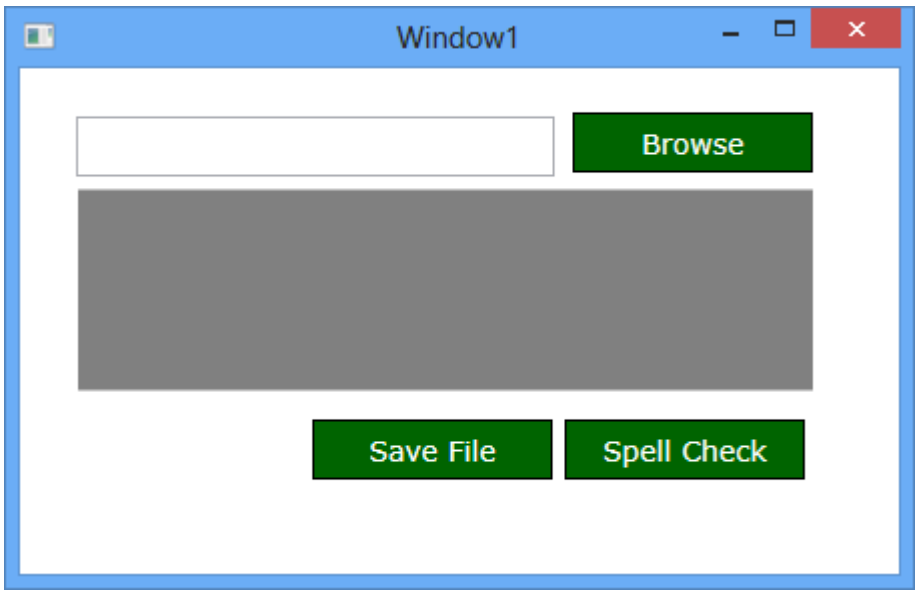


Figure 1

Here is the Button element code that sets Height, Width, Foreground, Background and Font properties.

```
<Button HorizontalAlignment="Right" Margin="0,14,26,0"
        Name="button1" VerticalAlignment="Top"
        Height="30" Width="120"
        FontFamily="Verdana" FontSize="14" FontWeight="Normal"
        Foreground="White" Background="DarkGreen"
        BorderBrush="Black" >
    Browse
</Button>
```

Listing 30

All of the three buttons have the same values.

Now, imagine you have a large application with many windows and pages and they have many buttons with the same size and look. That means you will have to repeat the same XAML in every window that you need a Button.

Let's say, your application is finished and now your client wants to change a Green background color to a Red background color. You will have to go to each page and find and change the background color from Green to Red.

But there is a better way to do that. You can create a resource style and use that for every Button control. The next time that you need to change a property, all you need to do is change the value in the resource.

The Style element in XAML represents a style. A Style element is usually added to the resources of a FrameworkElement. The x:Key is the unique key identifier of the style. The TargetType is the element type such as a Button.

The code snippet in Listing 2 adds a Style to the Window Resources and within the Style we use a Setter to set the property type and their values. The code snippet sets Width, Height, FontFamily, FontSize, FontWeight, Foreground, Background and BorderBrush properties.

```
<Window.Resources>
  <!-- Green Button Style -->
  <Style x:Key="GreenButtonStyle" TargetType="Button" >
    <Setter Property="Width" Value="120"/>
    <Setter Property="Height" Value="30"/>
    <Setter Property="FontFamily" Value="Verdana"/>
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="FontWeight" Value="Normal"/>
    <Setter Property="Foreground" Value="White"/>
    <Setter Property="Background" Value="DarkGreen"/>
    <Setter Property="BorderBrush" Value="Black"/>
  </Style>
</Window.Resources>
```

Listing 31

Note: If you do not specify the TargetType, you can explicitly specify it in the Setter as in the following:

```
<Setter Property="Button.Width" Value="120"/>
```

Listing 32

Once a Style is added to the resource dictionary, you can use it by using the Style property of a FrameworkElement. The code snippet in Listing 3 sets the Style of a Button using the StaticResource Markup Extension.

```
<Button HorizontalAlignment="Right" Margin="0,14,26,0"
      Name="button1" VerticalAlignment="Top"
      Style="{StaticResource GreenButtonStyle}"  >
   Browse
</Button>
```

Listing 33

Now we can replace Listing 1 with the much cleaner and manageable code listed in Listing 4. If we need to change the background color of Buttons from Green to Red, all we have to do is, change the Background property in the resources.

```
<Window x:Class="StylesSample.Window1"
   xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
   xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
   Title="Window1" Height="291" Width="455">
   <Window.Resources>
    <Style x:Key="GreenButtonStyle" TargetType="Button" >
         <Setter Property="Width" Value="120"/>
         <Setter Property="Height" Value="30"/>
         <Setter Property="FontFamily" Value="Verdana"/>
         <Setter Property="FontSize" Value="14"/>
         <Setter Property="FontWeight" Value="Normal"/>
         <Setter Property="Foreground" Value="White"/>
         <Setter Property="Background" Value="DarkGreen"/>
         <Setter Property="BorderBrush" Value="Black"/>
      </Style>
   </Window.Resources>
   <Grid Height="236" Width="405">
      <TextBlock Margin="12,52,26,83" Name="textBlock1"
            Background="Gray" Foreground="Orange"
```

```
            FontFamily="Georgia" FontSize="12"
            Width="370" Height="100" />
    <TextBox Height="30" Margin="11,16,155,0" Name="textBox1" VerticalAlignment="Top"
/>
    <Button HorizontalAlignment="Right" Margin="0,14,26,0"
          Name="button1" VerticalAlignment="Top"
          Style="{StaticResource GreenButtonStyle}"  >
        Browse
    </Button>
    <Button HorizontalAlignment="Right" Margin="0,0,30,39" Name="button2"
          VerticalAlignment="Bottom"
          Style="{StaticResource GreenButtonStyle}" >
        Spell Check
    </Button>
    <Button Margin="129,0,156,39" Name="button3" VerticalAlignment="Bottom"

          Style="{StaticResource GreenButtonStyle}" >

        Save File

    </Button>

  </Grid>

</Window>
```

Listing 34

Style Element

In the previous example, we saw how a Style element can be used within the resources to group multiple properties of elements and set them using the Style property of elements. However, Style functionality does not end here. Style can be used to group and share not only properties but also resources, and event handlers on any FrameworkElement or FrameworkContentElement.

Styles are resources and are used as any other resource and applies to the current element, parent element, root element and even at the application level. The scope of styles are similar to any other resource. The resource lookup process first looks up for local styles and if not found, it traverses to the parent element in the logical tree and so on. In the end, the resource lookup process looks for styles in the application and themes.

A Style element in XAML represents a style. The typical definition of a Style element looks as in the following:

```
<Style>
  Setters
</Style>
```

As you can see from the definition of Style, a Style has one more Setter element. Each Setter consists of a property and a value. The property is the name of the property and value is the actual value of that property of the element to that the style will be applied to.

Setters Property

The Setters property of Type represents a collection of Setter and EventSetter objects. Listing 4 uses the Setters property and adds a Setter and EventSetter objects.

The code snippet in Listing 4 sets the Setters property of a Style by adding a few Setter elements and one EventSetter element using XAML at design-time.

```
<Grid>
  <Grid.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Width" Value="200"/>
      <Setter Property="Height" Value="30"/>
      <Setter Property="Foreground" Value="White"/>
      <Setter Property="Background" Value="DarkGreen"/>
      <Setter Property="BorderBrush" Value="Black"/>
      <EventSetter Event="Click" Handler="Button1_Click"/>
    </Style>
  </Grid.Resources>
```

```
  <Button>Click me</Button>
</Grid>
```

Listing 35

BasedOn Property

Styles support inheritance. That means we can create styles based on existing styles. When you inherit a style from an existing style, settings from the parent style are available in the inherited style. To inherit a style from another style, we set the BasedOn property to StaticResource Markup Extension as the style it is being inherited from.

The code snippet in Listing 5 creates a Style BackForeColorStyle that sets the Background and Foreground properties of the control. Then we create a FontStyle style that sets font properties but is inherited from the BackForeColorStyle. The last style ButtonAllStyle is inherited from FontStyle. In the end, we set the Style of the button.

```
<Grid Name="RootLayout">
  <Grid.Resources>
    <Style x:Key="BackForeColorStyle">
      <Setter Property="Control.Background" Value="Green"/>

      <Setter Property="Control.Foreground" Value="White"/>
    </Style>
    <Style x:Key="FontStyle" BasedOn="{StaticResource BackForeColorStyle}">
      <Setter Property="Control.FontFamily" Value="Verdana"/>
      <Setter Property="Control.FontSize" Value="14"/>
      <Setter Property="Control.FontWeight" Value="Normal"/>
    </Style>
    <Style x:Key="ButtonAllStyle" BasedOn="{StaticResource FontStyle}">
      <Setter Property="Button.Width" Value="120"/>
      <Setter Property="Button.Height" Value="30"/>
    </Style>
  </Grid.Resources>
    <Button Name="Button1" Style="{StaticResource ButtonAllStyle}" >
     Click me
```

```
  </Button>
</Grid>
```

Listing 36

TargetType Property

The TargetType property can be used to get and set the type for which a style is intended. If the TargetType property of a Style is set and you assign a style to an element that is not the type set in TargetType, you will get an error.

If the TargetType property is not set, you must set the x:Key property of a Style.

Let's take a quick look at the code in Listing 5. This code creates a Style named GreenButtonStyle and sets very many Button properties.

```
<Grid>
  <Grid.Resources>
    <!-- Green Button Style -->
    <Style x:Key="GreenButtonStyle" >
      <Setter Property="Button.Width" Value="120"/>
      <Setter Property="Button.Height" Value="30"/>
      <Setter Property="Button.FontFamily" Value="Verdana"/>
      <Setter Property="Button.FontSize" Value="14"/>
      <Setter Property="Button.FontWeight" Value="Normal"/>
      <Setter Property="Button.Foreground" Value="White"/>
      <Setter Property="Button.Background" Value="DarkGreen"/>
      <Setter Property="Button.BorderBrush" Value="Black"/>
    </Style>
  </Grid.Resources>
  <Button HorizontalAlignment="Right" Margin="0,14,26,0"
      Name="button1" VerticalAlignment="Top"
      Style="{StaticResource GreenButtonStyle}"  >
    Browse
  </Button>
</Grid>
```

Listing 37

Now can simply replace the Style code in Listing 5 with that in Listing 6 where you may notice that we have set TargetType = "Button" but have removed Button in front of the properties. Setting TargetType fixes that this style can be applied to a Button element only.

```
<!-- Green Button Style -->
        <Style x:Key="GreenButtonStyle" TargetType="Button" >
           <Setter Property="Width" Value="120"/>
           <Setter Property="Height" Value="30"/>
           <Setter Property="FontFamily" Value="Verdana"/>
           <Setter Property="FontSize" Value="14"/>
           <Setter Property="FontWeight" Value="Normal"/>
           <Setter Property="Foreground" Value="White"/>
           <Setter Property="Background" Value="DarkGreen"/>
           <Setter Property="BorderBrush" Value="Black"/>
        </Style>
```

Listing 38

Triggers Property

Styles can use triggers within them. The Triggers property of Style represents the triggers applicable on a Style. For example, the following code snippet adds a Trigger for a button when the Button is in a pressed state; it will change the Foreground color of the button to Orange.

```
<Style.Triggers>
    <Trigger Property="IsPressed" Value="true">
        <Setter Property = "Foreground" Value="Orange"/>
     </Trigger>
</Style.Triggers>
```

Listing 6 shows the complete code of implementing triggers within a style.

```
<Grid>
  <Grid.Resources>
     <!-- Green Button Style -->
```

```
    <Style x:Key="GreenButtonStyle" TargetType="Button" >
      <Setter Property="Width" Value="120"/>
      <Setter Property="Height" Value="30"/>
      <Setter Property="FontFamily" Value="Verdana"/>
      <Setter Property="FontSize" Value="14"/>
      <Setter Property="FontWeight" Value="Normal"/>
      <Setter Property="Foreground" Value="White"/>
      <Setter Property="Background" Value="DarkGreen"/>
      <Setter Property="BorderBrush" Value="Black"/>
      <Style.Triggers>
        <Trigger Property="IsPressed" Value="true">
          <Setter Property = "Foreground" Value="Orange"/>
        </Trigger>
      </Style.Triggers>
    </Style>
  </Grid.Resources>
  <Button HorizontalAlignment="Right" Margin="0,14,26,0"
      Name="button1" VerticalAlignment="Top"
      Style="{StaticResource GreenButtonStyle}"  >
    Browse
  </Button>
</Grid>
```

Listing 39

XAML has become quite popular in recent days after the popularity of WPF, Silverlight, Windows Phone and Windows Store apps. In this article, we saw how to use XAML to apply style on XAML elements.

Collection Elements

XAML provides UI element objects that can host child collection items. XAML also provides support to work with .NET collection types as data sources.

XAML Collections

A collection element usually is a parent control with child collection elements. The child collection elements are ItemCollection that implements IList<object>. A ListBox element is a collection of ListBoxItem elements.

The code in Listing 1 creates a ListBox with a few ListBoxItems.

```
<ListBox Margin="10,10,0,13" Name="listBox1" HorizontalAlignment="Left"
      VerticalAlignment="Top" Width="194" Height="200">
  <ListBoxItem Content="Coffie"></ListBoxItem>
  <ListBoxItem Content="Tea"></ListBoxItem>
  <ListBoxItem Content="Orange Juice"></ListBoxItem>
  <ListBoxItem Content="Milk"></ListBoxItem>
  <ListBoxItem Content="Iced Tea"></ListBoxItem>
  <ListBoxItem Content="Mango Shake"></ListBoxItem>
</ListBox>
```

Listing 40

The preceding code in Listing 1 generates Figure 1.

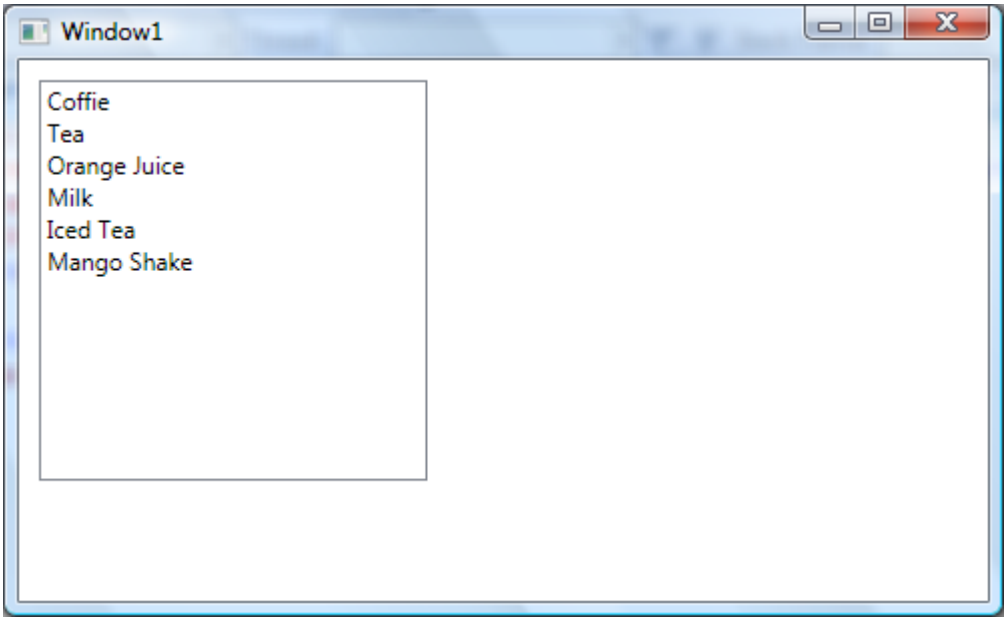


Figure 1

Add Collection Items

In the previous section, we saw how to add items to a ListBox at design-time from XAML. We can add items to a ListBox from the code.

Let's change our UI and add a TextBox and a button control to the page. The XAML code for the TextBox and Button controls looks as in the following:

```
<TextBox Height="23" HorizontalAlignment="Left" Margin="8,14,0,0"
         Name="textBox1" VerticalAlignment="Top" Width="127" />

<Button Height="23" Margin="140,14,0,0" Name="button1" VerticalAlignment="Top"
        HorizontalAlignment="Left" Width="76" Click="button1_Click">
    Add Item
</Button>
```

Listing 41

The final UI looks as in Figure 2.

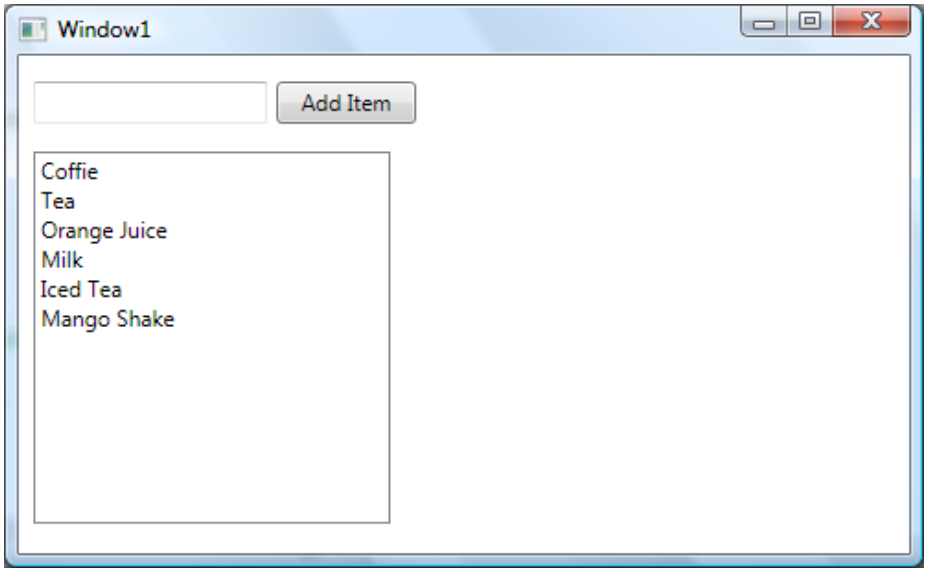


Figure 2.

We can access collection items using the Items property. In a button click event handler, we add the contents of a TextBox to the ListBox by calling the ListBox.Items.Add method. The following code adds TextBox contents to the ListBox items.

```
private void button1_Click(object sender, RoutedEventArgs e)
{
   listBox1.Items.Add(textBox1.Text);
}
```

Listing 42

In a button click event handler, we add the contents of a TextBox to the ListBox by calling the ListBox.Items.Add method.

Now if you enter text into the TextBox and click the Add Item button, it will add the contents of the TextBox to the ListBox.

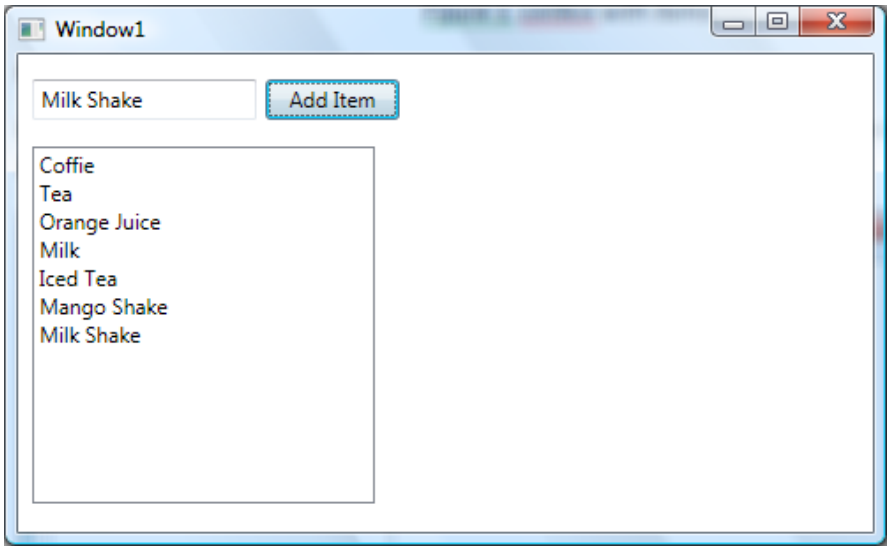


Figure 3. Adding ListBox items dynamically

Delete Collection Items

We can use either the ListBox.Items.Remove or ListBox.Items.RemoveAt method to delete an item from a collection of items in a ListBox. The RemoveAt method takes the index of the item in the collection.

Now, we will modify our application and add a new button called Delete Item. The XAML code for this button looks as below.

```
<Button Height="23" Margin="226,14,124,0" Name="DeleteButton"
      VerticalAlignment="Top" Click="DeleteButton_Click">
   Delete Item</Button>
```

Listing 42

The button click event handler looks as in the following. On this button click, we find the index of the selected item and call the ListBox.Items.RemoveAt method as in the following.

```
private void DeleteButton_Click(object sender, RoutedEventArgs e)
{
  listBox1.Items.RemoveAt
     (listBox1.Items.IndexOf(listBox1.SelectedItem));
}
```

Listing 43

Collection Types

XAML also allows developers to access .NET class library collection types form the scripting language. The code snippet in Listing 44 creates an array of String types, a collection of strings. To use the Array and String types, we must import the System namespace.

The code in Listing 2 creates an Array of String objects in XAML. As you may have noticed in Listing , you must import the System namespace in XAML using xmlns.

```
<Window x:Class="XamlCollectionsSample.MainWindow"
      xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
      xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
      xmlns:sys="clr-namespace:System;assembly=mscorlib"
      Title="MainWindow" Height="402.759" Width="633.345">
   <Window.Resources>
      <x:Array x:Key="AuthorList" Type="{x:Type sys:String}">
         <sys:String>Mahesh Chand</sys:String>
         <sys:String>Praveen Kumar</sys:String>
         <sys:String>Raj Beniwal</sys:String>
```

```
        <sys:String>Neel Beniwal</sys:String>
        <sys:String>Sam Hobbs</sys:String>
      </x:Array>
    </Window.Resources>
</Window>
```

Listing 44

The ItemsSource property if ListBox in XAML is used to bind the ArrayList. See Listing 3.

```
<ListBox Name="lst" Margin="5" ItemsSource="{StaticResource AuthorList}" />
```

Listing 45

XAML also allows developers to access .NET class library collection types form the scripting language.

Data Binding

Before I discuss data binding in general, I must confess, Microsoft experts have made a big mess related to data-binding in .NET 3.0 and 3.5. Instead of making things simpler, they have made them complicated. Perhaps they have some bigger plans for the future but so far I have seen binding using dependency objects and properties, LINQ and DLINQ and WCF and ASP.NET Web Services and it is all a big mess. It's not even close to the ADO.NET model we had in .NET 1.0 and 2.0. I hope they clean up this mess in near future.

When it comes to data binding, we need to first understand the data. Here is a list of ways a data can be consumed from:

- objects
- a relational database such as SQL Server
- a XML file
- other controls

Data Binding with Objects

The ItemsSource property of ListBox is used to bind a collection of IEnuemerable such as an ArrayList to the ListBox control.

```
// Bind ArrayList with the ListBox
LeftListBox.ItemsSource = LoadListBoxData();
private ArrayList LoadListBoxData()
{
   ArrayList itemsList = new ArrayList();
   itemsList.Add("Coffie");
   itemsList.Add("Tea");
   itemsList.Add("Orange Juice");
   itemsList.Add("Milk");
   itemsList.Add("Mango Shake");
   itemsList.Add("Iced Tea");
   itemsList.Add("Soda");
   itemsList.Add("Water");
   return itemsList;
}
```

Listing 46

Sample: Transferring data from one ListBox to Another

We've seen many requirements where a page has two ListBox controls and the left ListBox displays a list of items and using a button we can add items from the left ListBox and add them to the right side ListBox and using the remove button we can remove items from the right side ListBox and add them back to the left side ListBox.

This sample shows how we can move items from one ListBox to another. The final page looks as in Figure 7. The Add button adds the selected item to the right side ListBox and removes it from the left side ListBox. The Remove button removes the selected item from the right side ListBox and adds it back to the left side ListBox.

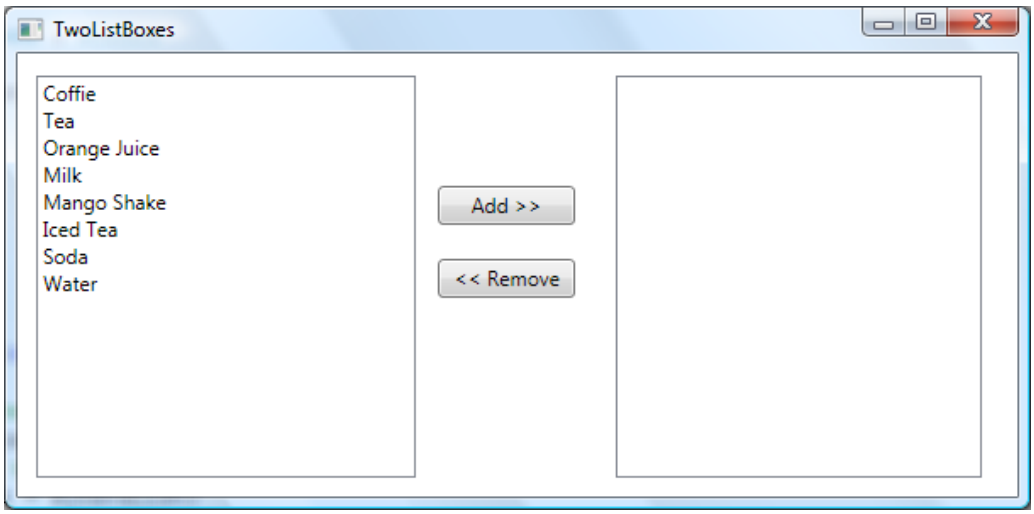


Figure 7

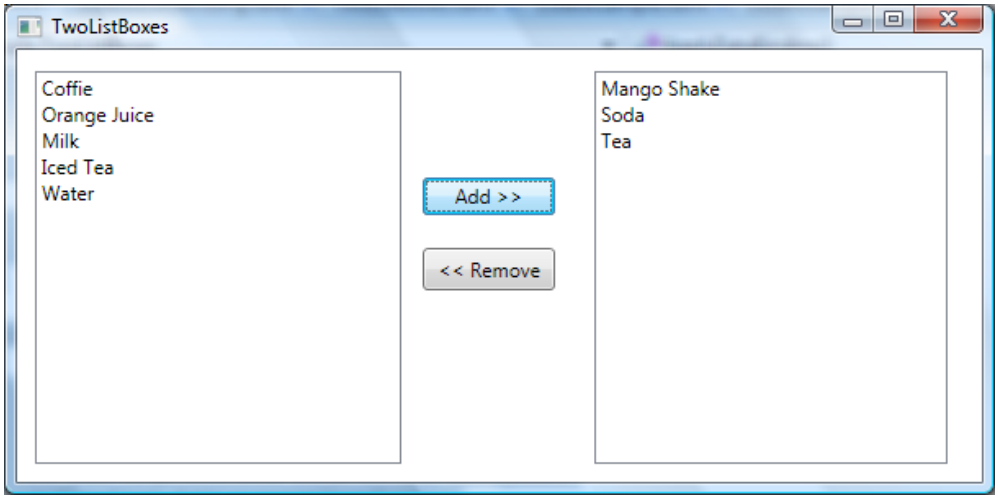


Figure 8

The following XAML code generates two ListBox controls and two Button controls.

```
<ListBox Margin="11,13,355,11" Name="LeftListBox" />
<ListBox Margin="0,13,21,11" Name="RightListBox" HorizontalAlignment="Right" Width="216" />
<Button Name="AddButton" Height="23" Margin="248,78,261,0" VerticalAlignment="Top"
      Click="AddButton_Click">Add &gt;&gt;</Button>
<Button Name="RemoveButton" Margin="248,121,261,117"
      Click="RemoveButton_Click">&lt;&lt; Remove</Button>
```

Listing 47

On the Window loaded event, we create and load data items to the ListBox by setting the ItemsSource property to an ArrayList.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
   // Get data from somewhere and fill in my local ArrayList
   myDataList = LoadListBoxData();
   // Bind ArrayList with the ListBox
   LeftListBox.ItemsSource = myDataList;
}
/// <summary>
/// Generate data. This method can bring data from a database or XML file
/// or from a Web service or generate data dynamically
/// </summary>
/// <returns></returns>
private ArrayList LoadListBoxData()
{
   ArrayList itemsList = new ArrayList();
   itemsList.Add("Coffie");
   itemsList.Add("Tea");
   itemsList.Add("Orange Juice");
   itemsList.Add("Milk");
   itemsList.Add("Mango Shake");
   itemsList.Add("Iced Tea");
   itemsList.Add("Soda");
   itemsList.Add("Water");
   return itemsList;
}
```

Listing 48

On an Add button click event handler, we get the value and index of the selected item in the left side ListBox and add that to the right side ListBox and remove that item from the ArrayList, which is our data source.

The ApplyBinding method simply removes the current binding of the ListBox and rebinds with the updated ArrayList.

```
private void AddButton_Click(object sender, RoutedEventArgs e)
{
   // Find the right item and it's value and index
   currentItemText = LeftListBox.SelectedValue.ToString();
   currentItemIndex = LeftListBox.SelectedIndex;
   RightListBox.Items.Add(currentItemText);
   if (myDataList != null)
   {
      myDataList.RemoveAt(currentItemIndex);
   }
   // Refresh data binding
   ApplyDataBinding();
}
/// <summary>
/// Refreshes data binding
/// </summary>
private void ApplyDataBinding()
{
   LeftListBox.ItemsSource = null;
   // Bind ArrayList with the ListBox
   LeftListBox.ItemsSource = myDataList;
}
```

Listing 49

Similarly, on the Remove button click event handler, we get the selected item text and index from the right side ListBox and add that to the ArrayList and remove it from the right side ListBox.

```
private void RemoveButton_Click(object sender, RoutedEventArgs e)
{
   // Find the right item and it's value and index
   currentItemText = RightListBox.SelectedValue.ToString();
   currentItemIndex = RightListBox.SelectedIndex;
   // Add RightListBox item to the ArrayList
   myDataList.Add(currentItemText);
```

```
 RightListBox.Items.RemoveAt(RightListBox.Items.IndexOf(RightListBox.SelectedItem));
   // Refresh data binding
   ApplyDataBinding();
}
```

Listing 50

Data Binding with a Database

We use the Northwind.mdf database that comes with SQL Server. In our application, we will read data from the Customers table. The Customers table columns looks as in Figure 9.

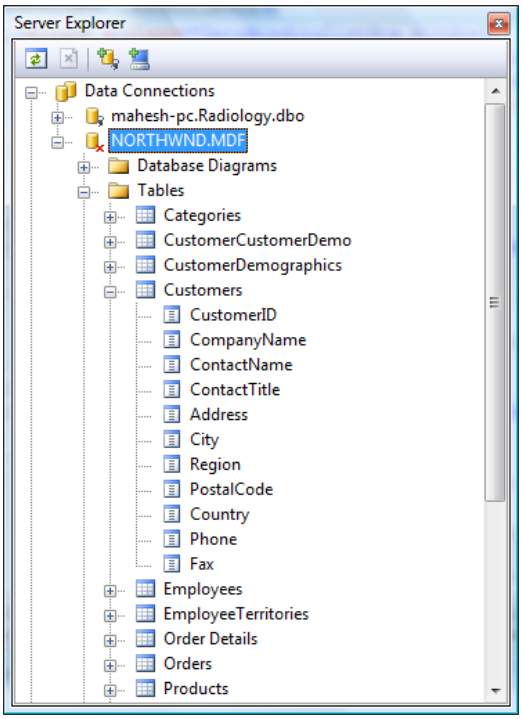


Figure 9

We will read ContactName, Address, City, and Country columns in a WPF ListBox control. The final ListBox looks as in Figure 10.

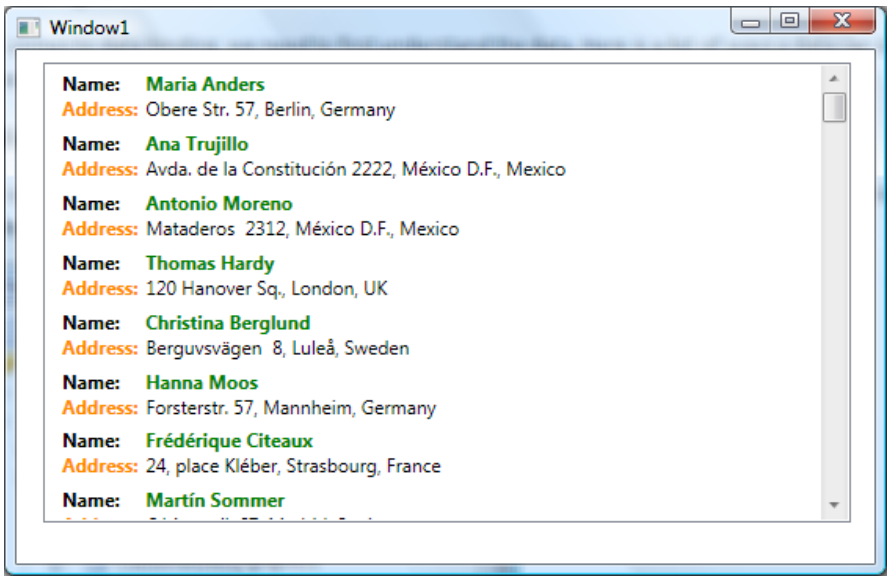


Figure 10

Now let's look at our XAML file. We create a resource DataTemplate type called listBoxTemplate. A data template is used to represent data in a formatted way. The data template has two dock panels where the first panel shows the name and the second panel shows address, city and country columns using TextBlock controls.

```
<Window.Resources>
  <DataTemplate x:Key="listBoxTemplate">
    <StackPanel Margin="3">
      <DockPanel >
        <TextBlock FontWeight="Bold" Text="Name:"
         DockPanel.Dock="Left"
         Margin="5,0,10,0"/>
        <TextBlock Text="  " />
        <TextBlock Text="{Binding ContactName}" Foreground="Green" FontWeight="Bold" />
      </DockPanel>
      <DockPanel >
        <TextBlock FontWeight="Bold" Text="Address:" Foreground ="DarkOrange"
         DockPanel.Dock="Left"
         Margin="5,0,5,0"/>
        <TextBlock Text="{Binding Address}" />
         <TextBlock Text=", " />
```

```
            <TextBlock Text="{Binding City}" />
             <TextBlock Text=", " />
            <TextBlock Text="{Binding Country}" />
          </DockPanel>
        </StackPanel>
    </DataTemplate>
</Window.Resources>
```

Listing 51

Now we add a ListBox control and set its ItemsSource property as the first DataTable of the DataSet and set ItemTemplate to the resource defined above.

```
<ListBox Margin="17,8,15,26" Name="listBox1"  ItemsSource="{Binding Tables[0]}"
ItemTemplate="{StaticResource listBoxTemplate}" />
```

Listing 52

Now in our code behind, we define the following variables.

```
public SqlConnection connection; public SqlCommand command;
string sql = "SELECT ContactName, Address, City, Country FROM Customers";
string connectionString = @"Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;Integrated
Security=True;Connect Timeout=30;User Instance=True";
```

Listing 53

Now on the Windows_Loaded method, we call the BindData method and in the BindData method, we create a connection, data adapter and fill in the DataSet using the SqlDataAdapter.Fill() method.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
   BindData();
}
private void BindData()
{
   DataSet dtSet = new DataSet();
```

```
    using (connection = new SqlConnection(connectionString))
    {
        command = new SqlCommand(sql, connection);
        SqlDataAdapter adapter = new SqlDataAdapter();
        connection.Open();
        adapter.SelectCommand = command;
        adapter.Fill(dtSet, "Customers");
        listBox1.DataContext = dtSet;
    }
}
```

Listing 54

Data Binding with XML

Now let's look at how we can bind XML data to a ListBox control. XmlDataProvider binds XML data in WPF.

Here is an XmlDataProvider defined in XAML that contains books data. The XML data is defined within the x:Data tag.

```
<XmlDataProvider x:Key="BooksData" XPath="Inventory/Books">
    <x:XData>
        <Inventory xmlns="">
            <Books>
                <Book Category="Programming" >
                    <Title>A Programmer's Guide to ADO.NET</Title>
                    <Summary>Learn how to write database applications using ADO.NET and C#.
                    </Summary>
                    <Author>Mahesh Chand</Author>
                    <Publisher>APress</Publisher>
                </Book>
                <Book Category="Programming" >
                    <Title>Graphics Programming with GDI+</Title>
                    <Summary>Learn how to write graphics applications using GDI+ and C#.
                    </Summary>
                    <Author>Mahesh Chand</Author>
```

```
            <Publisher>Addison Wesley</Publisher>
          </Book>
          <Book Category="Programming" >
            <Title>Visual C#</Title>
            <Summary>Learn how to write C# applications.
            </Summary>
            <Author>Mike Gold</Author>
            <Publisher>APress</Publisher>
          </Book>
          <Book Category="Programming" >
            <Title>Introducing Microsoft .NET</Title>
            <Summary>Programming .NET
            </Summary>
            <Author>Mathew Cochran</Author>
            <Publisher>APress</Publisher>
          </Book>
          <Book Category="Database" >
            <Title>DBA Express</Title>
            <Summary>DBA's Handbook
            </Summary>
            <Author>Mahesh Chand</Author>
            <Publisher>Microsoft</Publisher>
          </Book>
        </Books>
      </Inventory>
    </x:XData>
</XmlDataProvider>
```

Listing 55

To bind an XmlDataProvider, we set the Source property inside the ItemsSource of a ListBox to the x:Key of XmlDataProvider and XPath is used to filter the data. In the ListBox.ItemTempate, we use the Binding property.

```
<ListBox Width="400" Height="300" Background="LightGray">
    <ListBox.ItemsSource>
      <Binding Source="{StaticResource BooksData}"
```

```
    XPath="*[@Category='Programming'] "/>
  </ListBox.ItemsSource>
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="Title: " FontWeight="Bold"/>
        <TextBlock Foreground="Green"  >
          <TextBlock.Text>
            <Binding XPath="Title"/>
          </TextBlock.Text>
        </TextBlock>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Listing 56

The output of the preceding code looks as in Figure 11.

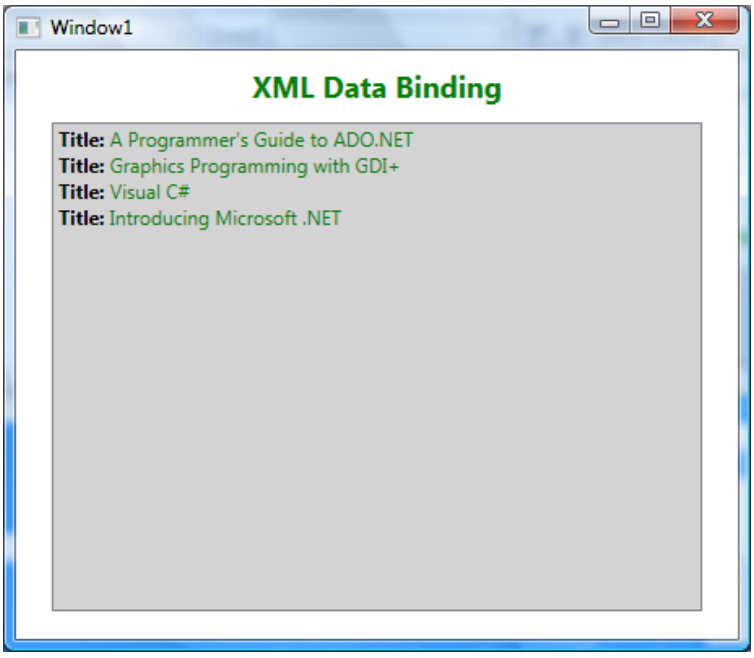


Figure 11

Data Binding with Controls

The last data binding type we will see is how to provide data exchange between a ListBox and other controls using data binding in WPF.

We will create an application that looks like Figure 12. In Figure 12, I have a ListBox with a list of colors, a TextBox, and a Canvas. When we pick a color from the ListBox, the text of the TextBox and the color of the Canvas changes dynamically to the color selected in the ListBox and it is possible to do all that in XAML without writing a single line of code in the code behind file.

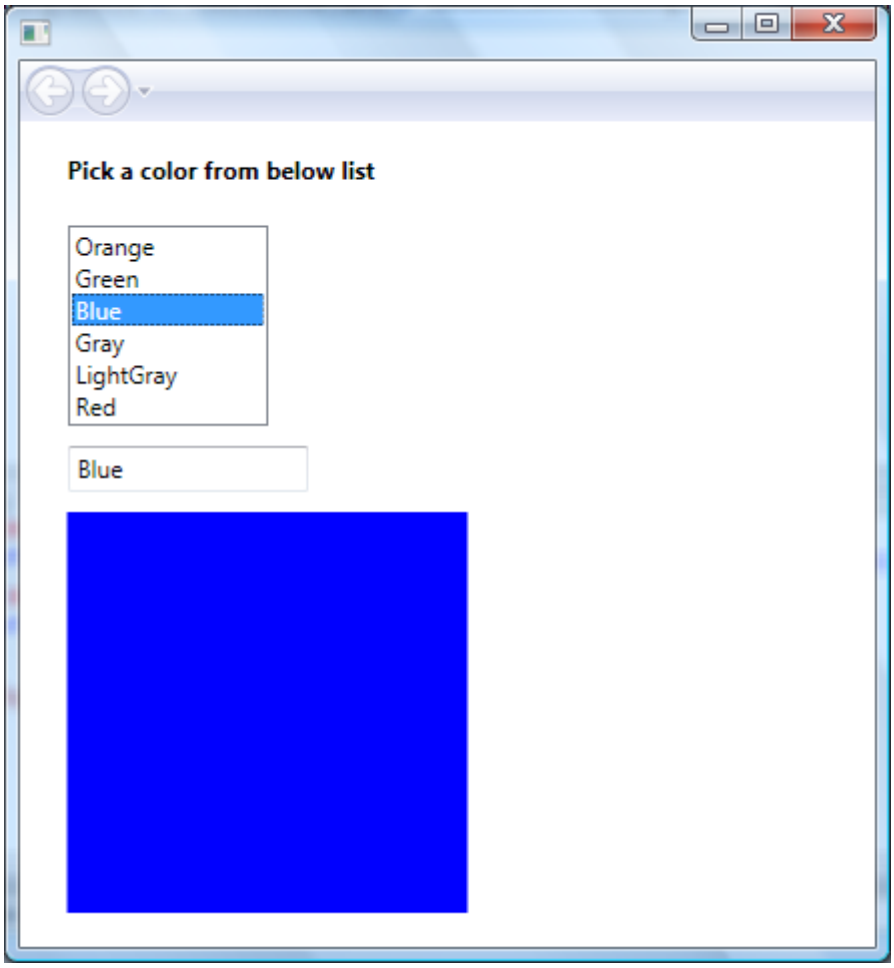


Figure 12.

The XAML code of the page looks as in the following.

```
<StackPanel Orientation="Vertical">
   <TextBlock Margin="10,10,10,10" FontWeight="Bold">
      Pick a color from below list
   </TextBlock>
   <ListBox Name="mcListBox" Height="100" Width="100"
         Margin="10,10,0,0" HorizontalAlignment="Left" >
      <ListBoxItem>Orange</ListBoxItem>
      <ListBoxItem>Green</ListBoxItem>
      <ListBoxItem>Blue</ListBoxItem>
      <ListBoxItem>Gray</ListBoxItem>
      <ListBoxItem>LightGray</ListBoxItem>
      <ListBoxItem>Red</ListBoxItem>
   </ListBox>
  <TextBox Height="23" Name="textBox1" Width="120" Margin="10,10,0,0"
HorizontalAlignment="Left"  >
      <TextBox.Text>
         <Binding ElementName="mcListBox" Path="SelectedItem.Content"/>
      </TextBox.Text>
   </TextBox>
   <Canvas Margin="10,10,0,0" Height="200" Width="200" HorizontalAlignment="Left" >
      <Canvas.Background>
         <Binding ElementName="mcListBox" Path="SelectedItem.Content"/>
      </Canvas.Background>
   </Canvas>
</StackPanel>
```

Listing 57

If you look at the TextBox XAML code, you will see the Binding within the TextBox.Text property, which sets the binding from TextBox to another control and another control ID is ElementName and another control's property is Path. So in the following code, we are setting the SelectedItem.Content property of ListBox to the TextBox.Text property.

```
      <TextBox.Text>
         <Binding ElementName="mcListBox" Path="SelectedItem.Content"/>
      </TextBox.Text>
```

Listing 58

Now the same applies to the Canvas.Background property, where we set it to the SelectedItem.Content of the ListBox. Now, every time you select an item in the ListBox, the TextBox.Text and Canvas.Background properties are set to that selected item in the ListBox.

```
<Canvas.Background>
   <Binding ElementName="mcListBox" Path="SelectedItem.Content"/>
</Canvas.Background>
```

Listing 59

MediaElement

The MediaElement tag in XAML allows you to play audios and videos in XAML. The Source attribute of the tag takes the full path of the audio or video file. The following code snippet uses the MediaElement to display a video.

```
<MediaElementName="VideoControl"Width="200"Height ="400"
            Source="C:\Windows\System32\oobe\images\intro.wmv" >
</MediaElement>
```

Listing 60

The MediaElement has Play, Pause, and Stop properties that are used to play, pause and stop an audio or video.

Summary

This book is an introduction to the XAML language. We started this book by discussing various tools available to write XAML code. After that we discussed basic controls, elements, attributes, user interfaces, and control events. After that, we discussed communication between controls followed by shapes and brushes. We still need to discuss resources, styles, templates, skins, themes, audio, video and much more. I will discuss these XAML syntaxes in the related chapters later in this book.