developmentor

Using XAML Markup

Estimated time for completion: 30 minutes

Goals:

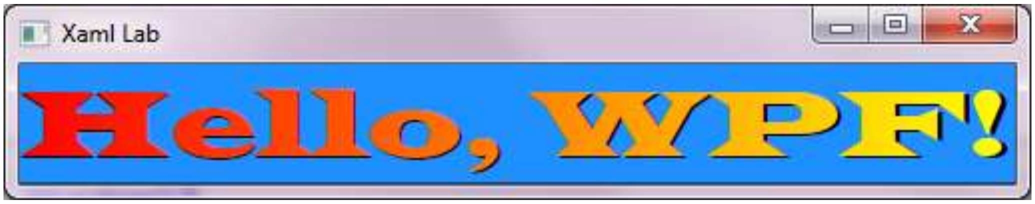
- Introduce the XAML syntax.
- Creating applications with XAML.
- Using XAML with custom types.

Overview:

This lab will introduce the student to building WPF applications with XAML.

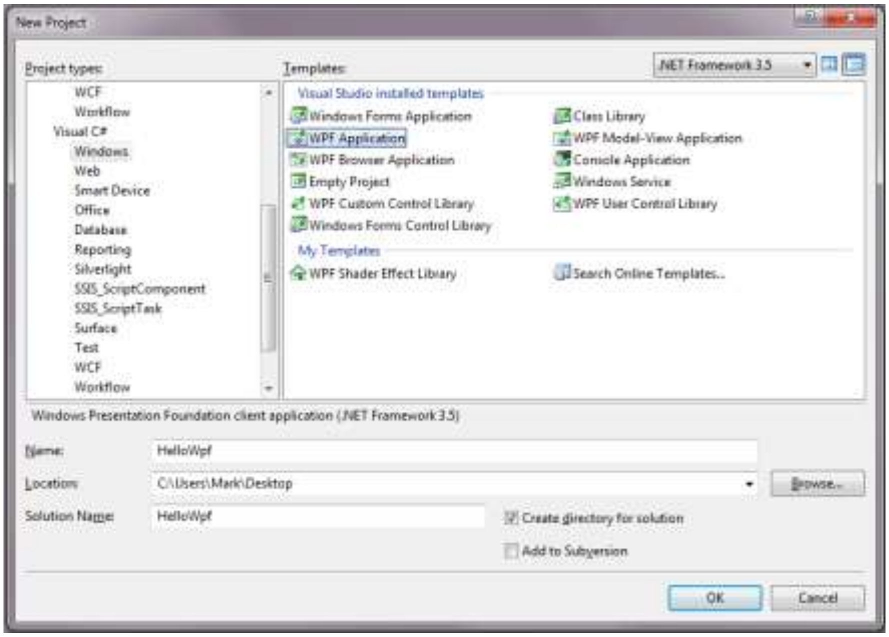
Part 1 – Writing a simple XAML application

In this part you will write a simple XAML application using Visual Studio. When you are finished, the page should look like:



Steps:

1. Create a new WPF Project using Visual Studio.
   a. Open Visual Studio 2008 or 2010.
   b. Select File | New Project
   c. Pick the WPF template from the list and give the project a name.



Windows Presentation Foundation client application (.NET Framework 3.5)

Name: HelloWpf

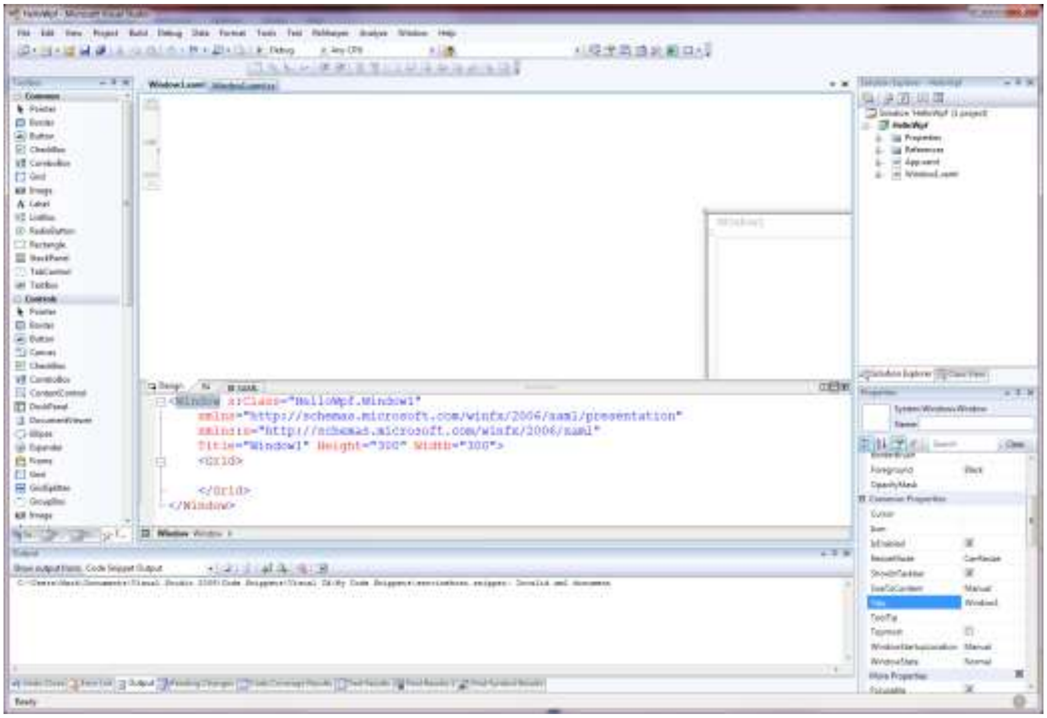
Location: C:\Users\Mark\Desktop

Solution Name: HelloWpf

☑ Create directory for solution

□ Add to Subversion

2. Visual Studio should open the **Window1.xaml** (or **MainWindow.xaml** if you are using Visual Studio 2010) file in the designer view initially. If it does not (this can be configured), double-click on the **Window1.xaml** file.



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

    </Grid>
</Window>
```

3. Set the window title to be “Xaml Lab”

4. Set the `Window.SizeToContent` property to be “WidthAndHeight”

5. Set the `Window.Background` property to be “DodgerBlue”

6. The root layout tag should be a `Grid` element – this will be the immediate child of the `Window` element at the root of the XAML document.

7. In the `Grid` element, add a `TextBlock` element and set the text property to "Hello WPF".

   a. Set the `FontSize` property of the `TextBlock` to "36pt"

   b. Set the `Foreground` property to a `LinearGradientBrush` going diagonally from red to yellow using the property element syntax (`TextBlock.Foreground`) shown in the slides.

   c. The following is the definition to use for the brush:

```
<LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
   <GradientStop Offset="0" Color="Red" />
   <GradientStop Offset="1" Color="Yellow" />
</LinearGradientBrush>
```

   d. Set the `FontFamily` to "Wide Latin" to make it a bit wider.

8. Now add a *second* `TextBlock` directly above the first. The `FontSize` should be slightly larger (try 36.25) and the Foreground should be "Black". This will be our shadow – since the object is defined above our original, it will be placed *below* the first one.

9. When you are finished, your code should look something like:

```
<Window x:Class="HelloWpf.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Xaml Lab" SizeToContent="WidthAndHeight" Background="DodgerBlue">
   <Grid>
        <TextBlock FontSize="36.25pt" Text="Hello, WPF!"
             Foreground="Black" FontFamily="Wide Latin" />

        <TextBlock FontSize="36pt" Text="Hello, WPF!"
                   FontFamily="Wide Latin">
            <TextBlock.Foreground>
                <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                    <GradientStop Offset="0" Color="Red" />
                    <GradientStop Offset="1" Color="Yellow" />
                </LinearGradientBrush>
            </TextBlock.Foreground>
        </TextBlock>

    </Grid>
</Window>
```

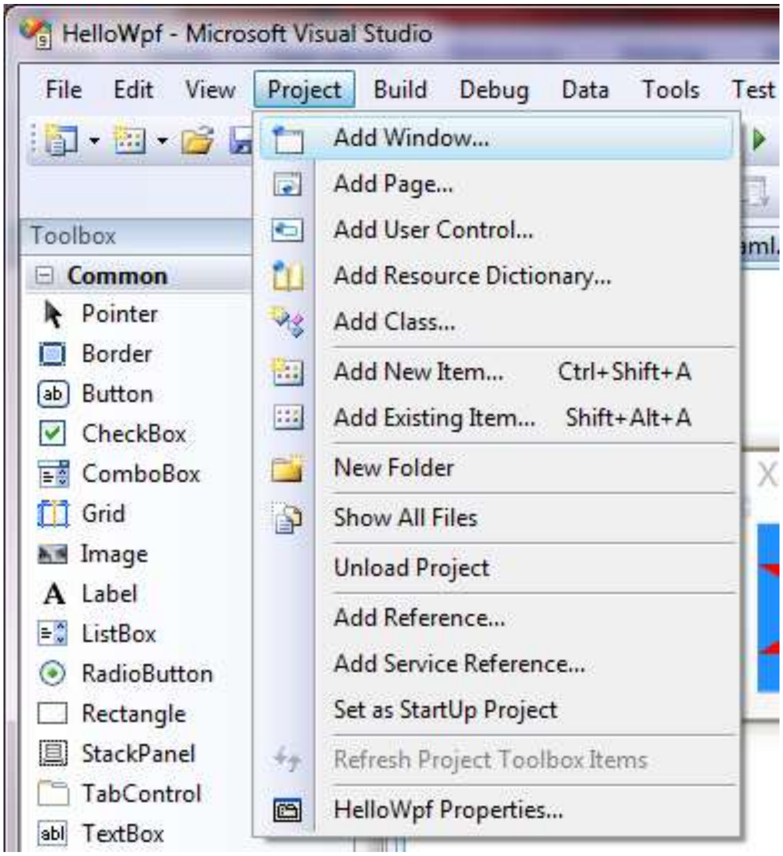
Run the program to see your results!

Part 2 – Using graphics inside XAML

In this part of the lab you will utilize the shapes that are part of the WPF framework.

Steps:

1. Add a new Window to your Visual Studio project.
   a. Right click on the project in the solution explorer and select "Add | Window…"
   b. Or click on the Project menu at the top and select Add Window…



2. You can name the window anything you like – the lab sample will keep the default name of Window2.xaml.
3. The designer view should open for Window2 – if not, double click on it in the solution explorer.
4. Like before, the default root layout element is a `Grid`, we want to use something else this time so change the `Grid` tag to a `Canvas` tag. Make sure you change both the start and closing tags!
5. In the `Canvas` element, add a `Rectangle` element. For the rectangle to be visible, you'll need to set its `Width`, `Height` and `Fill` properties.
   a. Set the `Width` to "150".
   b. Set the `Height` to "150".
   c. Set the `Fill` property to "Orange".

   d. Set the `RadiusX` and `RadiusY` properties to “10”
   e. Set the `Cursor` property to “Cross”.
   f. Set the `Stroke` property to “DarkOrange”
   g. Set the `StrokeThickness` property to “5”.

6. Your code should look something like:

```
<Canvas>
   <Rectangle Width="150" Height="150" Fill="Orange"
              RadiusX="10" RadiusY="10" Cursor="Cross"
              Stroke="DarkOrange" StrokeThickness="5" />
</Canvas>
```

7. Save the file.

8. Open the `App.xaml` file – this is the application object that starts up the WPF program initially. You should see a StartupUri tag applied in this XAML file – this identifies the initial window we want to display. Go ahead and change it from its current value (Window1.xaml or MainWindow.xaml) to our new Window – Window2.xaml. When you are done it should look like:

```
<Application x:Class="HelloWpf.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window2.xaml">
    <Application.Resources>

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

9. Run the program - the rectangle displayed by the page is a "live" object. To verify this, move the mouse over the rectangle and note that it will change to a cross.

Part 3 – Apply complex and attached properties

In this part, you'll set complex and attached properties using XAML.

Steps:

1. Having a solid color background is pretty boring. Your job is to change the background brush to a gradient brush.

2. In the previous example, the color “Orange” was automatically transformed into a brush. However, if the background is a gradient, it is not possible to express its value through a simple string.

3. Replace the **Fill** property value with a `LinearGradientBrush` object. Remember just as you did earlier, that in XAML a property may be set to a complex object using an

inner XML element where the name is specified in the form `Type.Property`. In this case, the element name should be `<Rectangle.Fill>`.

4. Set the `Fill` property to a gradient from Orange to Blue.

```
<Canvas>
   <Rectangle Width="150" Height="150" ~~Fill="Orange"~~
              RadiusX="10" RadiusY="10" Cursor="Cross"
              Stroke="DarkOrange" StrokeThickness="5">
      <Rectangle.Fill>
         <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
               <GradientStop Offset="0" Color="Orange" />
               <GradientStop Offset="1" Color="Blue" />
            </LinearGradientBrush.GradientStops>
         </LinearGradientBrush>
      </Rectangle.Fill>
   </Rectangle>
</Canvas>
```

5. Save the file and run the program.
   a. The background should now be a gradient going from Orange to Blue.
6. In the current implementation, the rectangle is on the top left corner of the canvas. Your job is now to set the Top and Left properties of the rectangle. Top and Left and example of attached properties. They are defined by the `Canvas` type but applied to its children.
   a. Set the `Left` of the rectangle to "100" by setting the `Canvas.Left` attribute.
   b. Set the `Top` of the rectangle to "80" using a full XAML property element syntax where it is placed in a child element.

```
<Window x:Class="HelloWpf.Window2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window2" Height="300" Width="300">
    <Canvas>
        <Rectangle Width="150" Height="150" Canvas.Left="100"
                   RadiusX="10" RadiusY="10" Cursor="Cross"
                   Stroke="DarkOrange" StrokeThickness="5">
            <Canvas.Top>80</Canvas.Top>
            <Rectangle.Fill>
                <LinearGradientBrush>
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0" Color="Orange" />
                        <GradientStop Offset="1" Color="Blue" />
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </Rectangle.Fill>
        </Rectangle>
    </Canvas>
</Window>
```

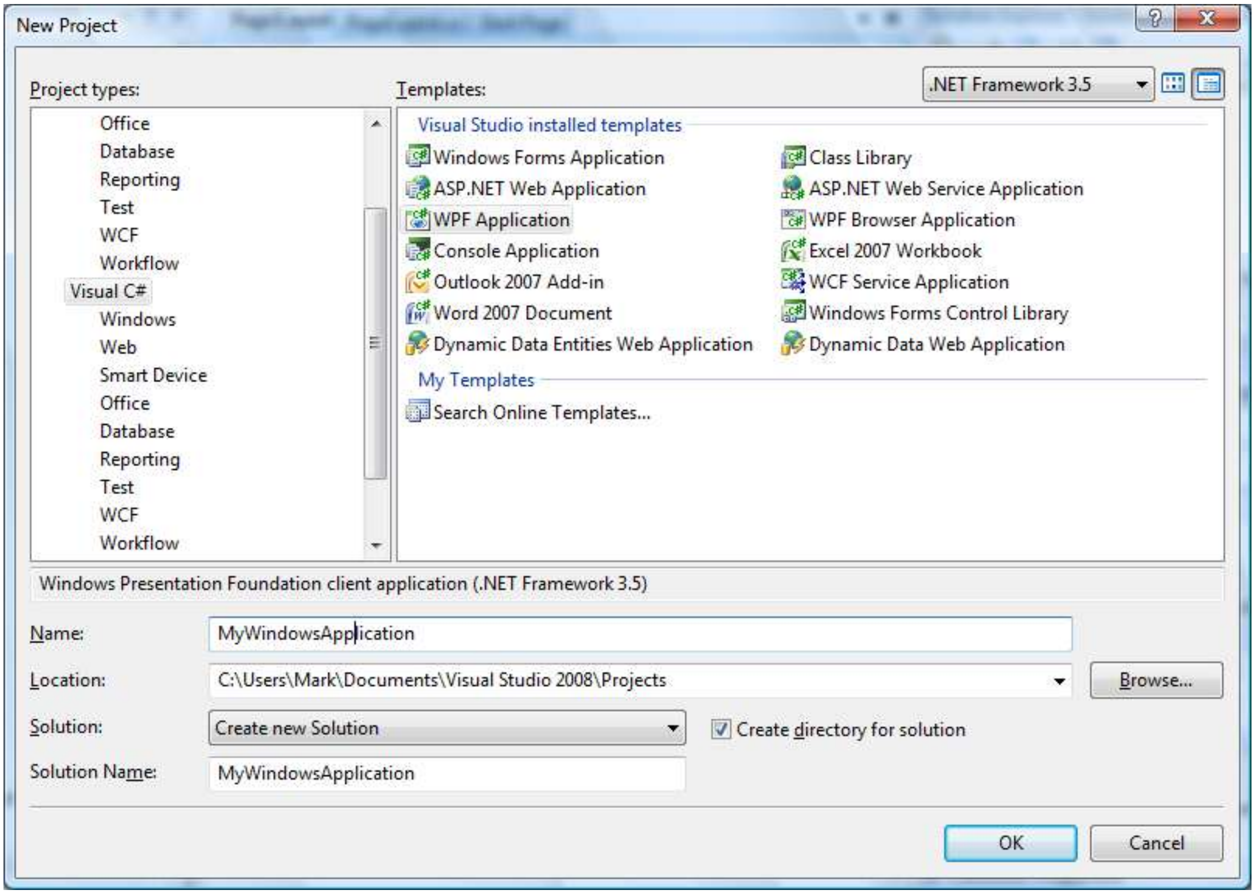
7. Save the file and run the program. As expected, the rectangle has moved away from the corner of the window.

Part 4 – Add C# event handlers

In the previous parts, the application did not contain any code. In this part, you'll create a Windows application that calls C# code in response to UI events.

Steps:

1. Create a new WPF Windows application project and name it **MyWindowsApplication**.



.NET Framework 3.5

Windows Presentation Foundation client application (.NET Framework 3.5)

Name: MyWindowsApplication

Location: C:\Users\Mark\Documents\Visual Studio 2008\Projects

Solution: Create new Solution

☑ Create directory for solution

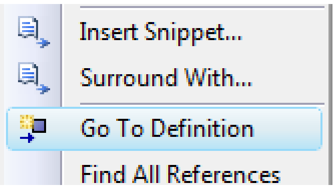
Solution Name: MyWindowsApplication

2. Examine the created project and notice that that it contains two XAML files: app.xaml and window1.xaml (this will be `MainWindow.xaml` if you are using Visual Studio 2010 – just substitute this file instead of `window1.xaml` below).

   a. `app.xaml` contains the markup for the application.

   b. `window1.xaml` contains the code for the main window.

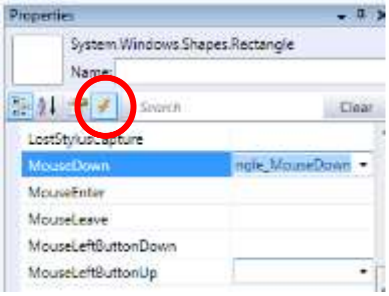
3. Open `app.xaml` and notice that the `StartupUri` attribute is set to `window1.xaml` - the main window.
4. Examine the content of `window1.xaml`. Notice it contains a `Window` element with a `Grid` inside. Notice it also specifies the corresponding class `x:Class="MyWindowsApplication.Window1`.
5. Verify that there is a file below `window1.xaml` called `window1.xaml.cs`. This is where you'll put your code.
6. Compile the project. In `window1.xaml.cs`, notice that the constructor calls a method named `InitializeComponent`. Navigate to the definition of this method by right-clicking on it and selecting "Go To Definition".



7. This should open an auto-generated intermediate file with the same name as the XAML and code behind files. If it does not (certain add-ins can affect this behavior), click the "Show All Files" button at the top of the solution explorer and navigate into the "obj/debug" directory. You should find the file in that location if the project was compiled properly – it will be named `window1.xaml.g.cs` or `window1.xaml.i.g.cs`.
8. As the "**g**" suggests, this method has been automatically generated during compilation. This is what the XAML compiler produced from the .XAML file. Notice that it is a partial class which adds to your code behind file. This is where:
    a. The .BAML is loaded and parsed, creating the actual objects based on the markup.
    b. Any named elements are backed with private fields accessible in your code behind.
    c. Event handlers are wired up to events.
9. Run the project. You should see the window with no content.
10. Go back to the XAML file for the main window. Inside the `Grid` element, add an orange rectangle like you did in the previous part. Compile and run. You should get a window with an orange rectangle.
    a. You can set the `Stroke` property to some color to see the edges of the rectangle if you want.
11. So far, the application does not do more than a markup application. Your job is now to change the fill of the rectangle to a random color each time you click on it.
12. Locate the `Rectangle` element and add a new `MouseDown` event handler.

a. If you type a name and hit enter, Visual Studio will give it some other name (Rectangle_MouseDown in this case) and add the method into code behind. This is likely a bug which will be fixed in a future release of the IDE. For now, you can accept the generated name, or type in the name and right-click on it and select "Go To Definition" – this will cause the IDE to generate your stubbed out method.

b. Alternatively, if you have Visual Studio 2008 SP1 or are using Visual Studio 2010, you can use the Events section of the properties window – it is the lightning bolt button on the toolbar in the properties window. Here you can type in a name for the event handler directly and it will be generated as desired.



13. Open the window1.xaml.cs file and locate the OnRectangleMouseDown handler. It should have the following signature:

```
void OnRectangleMouseDown(object sender, MouseButtonEventArgs args)
```

14. In order to change its fill color, you need a reference to the rectangle in OnRectangleMouseDown. There are several ways to accomplish this. The easiest way is to set the Name attribute in XAML to the desired C# field name.

   a. Set the Name attribute to "myRectangle".

   b. Compile. Notice that the code Window1.g.cs now contains a definition for a field named "myRectangle".

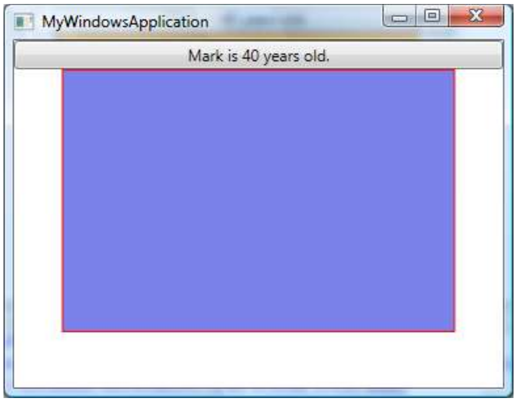
   c. Implement OnRectangleMouseDown to set the fill of the rectangle to a random color.

```
Random r = new Random();
 void OnRectangleMouseDown(object sender, MouseButtonEventArgs args)
 {
    Color c = new Color();
    c.A = (byte)r.Next(256);
    c.R = (byte)r.Next(256);
    c.G = (byte)r.Next(256);
    c.B = (byte)r.Next(256);
    myRectangle.Fill = new SolidColorBrush(c);
 }
```

15. Compile and test. Verify that the color changes each time you click on the rectangle. If you have time, add other controls such as buttons and list boxes.

Part 5 – Use XAML for custom types

In most cases, XAML will be used for defining UI elements. However, XAML can be used to create any valid .NET type that has a public, default constructor. In this part, you will define your type and use it directly from XAML. When you are finished your application should look something like:



Steps:

1. Create a new class named `Person` in the same namespace - `MyWindowsApplication`.
2. Add a few interesting properties such as `Age` and `Name`.
3. Override `ToString` to return the name and age.

```
public class Person
{
   public string Name { get; set; }
   public int Age { get; set; }

   public override string ToString()
   {
     return name + "is " + age + " years old.";
   }
}
```

4. In order to use this type from XAML, you need to map an XML namespace to the CLR namespace.
5. In the `window1.xaml` file, locate the existing XML namespace definitions and add a new XML namespace definition next to them.

```
xmlns:my="clr-namespace:MyWindowsApplication"
```

6. Locate the root Grid element in window1.xaml – change it to a StackPanel. You will need to change the closing tag as well.
7. Add a new Button as the first child of the StackPanel and set its content to a new Person.
8. In order to set the button content, add a child element using the “**my**” XML prefix you just defined.
9. Set the Age and Name properties to meaningful values.

```
<StackPanel>
   <Button>
      <my:Person Name=”Mark” Age=”40” />
   </Button>
    ...
</StackPanel>
```

10. Compile and run the application. Notice that the person is now inside the button and displaying the output from your ToString implementation.

Solution

Two solutions are provided in the **after** folder as part of the lab. The first, **HelloWpf.sln**, contains the first two parts of the lab. The second **MyWindowsApplication.sln** solution contains the final two parts. Both VS2008 SP1 and VS2010 versions are provided for your convenience.