Introduction to XAML – WPF 2010

An overview of XAML by JeremyBytes.com

Overview

Understanding XAML (eXtensible Application Markup Language) is a key to creating the latest .NET user experiences in WPF and Silverlight. We will introduce the basic concepts around XAML and take a look at various features such as namespaces, elements, properties, events, attached properties and some basic layout. We'll create a simple WPF application that covers these fundamentals. Although you will probably end up doing most of your UI design with a drag-and-drop tool such as Expression Blend, knowing the internals gives you a leg up in making the final tweaks to ensure an excellent user experience.

Visual Studio IntelliSense works in XAML files very well. This means that as we type, we get tag completion, attribute completion, and even value completion (and this has gotten even better with Visual Studio 2010). Depending on your preferences, you may find yourself doing a majority of your XAML in the Visual Studio editor and saving the visual design tools for complex cases.

WPF vs. Silverlight

The differences between WPF 4 and Silverlight 4 have grown narrower (compared to previous versions). For the XAML we'll be looking at here, the differences are almost non-existent. I'll point out these differences as we come across them. If you are interested, there is also a Silverlight 4 version of this walkthrough available at the download location below.

The Set Up

Our completed application will be a simple stop watch. We'll start with a skeleton of a WPF application that contains a class with the timer functionality (we'll fill in the UI ourselves). You can download the source code for the application here: http://www.jeremybytes.com/Downloads.aspx. The sample code we'll be looking at here is built for Visual Studio 2010, but there is also a version for Visual Studio 2008 available for download from the same location. The download includes the starter application and the completed code. This includes the following projects:

- WPFStopWatch – This is the application we will be working with. This is simply a new WPF project with a single class added: Ticker.cs.
- WPFStyledStopWatch – This is a completed application with a few more extras (additional styles, brushes, and custom form-handling). The additional features are outside of the scope of this introduction; you can peruse them at your leisure.

An Initial Window

When you start a new WPF application, you get the following XAML as a starter (MainWindow.xaml):

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

Let's take a look at a few of these elements.

XAML is XML

The first thing to note about XAML is that it is XML. If you need an overview of XML, you can go here: http://www.w3schools.com/xml/default.asp. Since the XAML is an XML document, it can only have a single root element. In this case, the root element is "`<Window>`" (but "`<Page>`" and "`<UserControl>`" are also common). Silverlight uses the "`<UserControl>`" as the default root element.

XAML Elements are .NET Classes

Each element in a XAML document refers to a .NET class. This means that both "`<Window>`" and "`<Grid>`" are .NET Classes.

XAML Namespaces

In order to reference a .NET class, we also need to reference the namespace. You can think of this as the "using" statements in a .cs file. Namespaces are added to XAML by using the `xmlns` attribute. You can see that by default, there are 2 namespaces included:

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

The first refers to the "presentation" namespace, the standard WPF namespace. This is also the default namespace, so any elements that do not have a prefix are assumed to come from this location. You'll notice that the namespace is written as a URI. This does not refer to a physical location, but rather the unique identifier of the namespace.

The second refers to the XAML namespace. This is identified by `xmlns:x`, with the "`x`" being an alias you can use as a prefix for elements from that namespace. We'll look at this a little more closely in just a bit.

Obviously, you can also add your own namespaces. We'll do that later on.

XAML Code Behind

The `x:Class` attribute references the code-behind for this XAML window. You'll notice the "`x:`" which means that the Class attribute comes from the XAML namespace noted above. The value of the

attribute references the "MainWindow" class in the "WPFStopWatch" namespace. If you go to the MainWindow.xaml.cs file, you'll see the partial class that is defined here.

The code behind is where we can put C# (or VB) code for things such as implementation of event handlers and other application logic. As a note, it is technically possible to create all of the XAML elements in code (WinForms apps create all of the UI in code), but that bypasses the advantages of having the XAML.

Other Starting Attributes

The other attributes of the Window element (`Title="MainWindow" Height="350" Width="525"`) are simply properties of the Window class (more below).

The Grid Element

The final element is the `<Grid>` element. We'll be filling this in soon. For now, note that the Window element only allows for a single child. This means that if we want to include more than one Control in our Window, we need to wrap them in some sort of layout control that allows multiple children. The Grid is just that sort of control.

Properties as Attributes

Properties of elements can be expressed in a couple of different ways. The first is by using an XML attribute. The `Title`, `Height`, and `Width` properties of the Window are examples of this. We'll go ahead and adjust a few of the values now. In addition, we'll add the "`Topmost`" property and set it to "`True`". This will keep the application on top of other active windows. This makes it more useful as a timer for other processes. Here's our Window markup:

```
<Window x:Class="WPFStopWatch.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF Stop Watch" Height="150" Width="250" Topmost="True">
```

Properties as Nested Elements

Properties can also be expressed as nested elements. This is often required when properties are of a complex type. Let's define some rows for our Grid. Here's what the markup looks like:

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

    </Grid>
```

You can see here that we have set the "`Grid.RowDefinitions`" property by creating a nested element. This particular element accepts a list of child objects ("`RowDefinition`"). We won't go into all of the options for the "`Height`" property of the rows; that's best left to a discussion on layout

controls. For now, just know that "Auto" means that the row will only take up as much space as its contained elements; "*" means to take up the remaining space. We won't define any columns for this application, but they are defined much the same way.

Attached Properties

An attached property is a property that doesn't belong to the element that specifies it. Let's take a look at an example. We'll add a TextBlock to the Grid. This will be the output for the time of our Stop Watch:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <!--Output Display-->
    <TextBlock Grid.Row="0"
               FontSize="48"
               HorizontalAlignment="Center" VerticalAlignment="Center"
               Text="00:00" />
</Grid>
```

First note that our `TextBlock` is nested inside our Grid markup. Most of the attributes are simply properties of the `TextBlock` class (`FontSize`, `HorizontalAlignment`, `VerticalAlignment`, `Text`).

But `Grid.Row` is not. Grid.Row is a property of the Grid class. But since it is an attached property, we can use this in any element contained in the Grid to let the Grid know how to handle the layout. In this case, we are indicating that the TextBlock should be placed in the first row (row 0) of the grid.

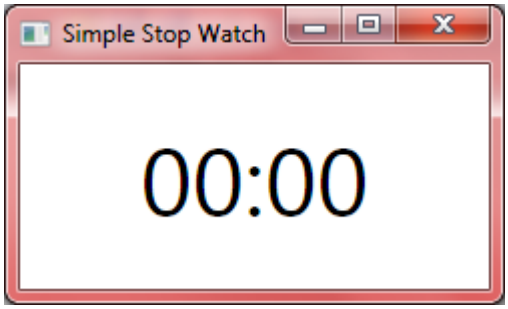
Naming is Optional

An interesting difference between XAML and WinForms UIs: you don't need to name all of your UI elements. Note that our TextBlock does not have a "Name" property specified. This is perfectly fine since we do not need to refer to this control anywhere else in our code or XAML. If we do need to refer to the control, then we simply include a "Name" attribute.

Comments

Comments in XAML are marked like XML & HTML comments, with the "`<!-- -->`" tag. You'll note that I use comments to help split up my markup into easily maintainable sections. This is especially important when you have more complex layouts with dozens of elements.

Here's the output of our application so far:



Just a quick note on the layout: since there are no elements in the second row of the grid (set to `Auto` height), it takes up no space at all.

Adding a Button

Let's add a button to the second row in the grid. We're going to add a total of three buttons, so we'll start with adding a StackPanel (another container) to hold the buttons. A StackPanel simply stacks elements either vertically (default) or horizontally. Here's our StackPanel markup:

```
<!--Button Panel-->
<StackPanel Grid.Row="1" Orientation="Horizontal"
            HorizontalAlignment="Center" Margin="5">
</StackPanel>
```

By using an attached property, we denote that this element should be in the second row of our grid. But interestingly enough, we can place the StackPanel markup before the TextBlock markup (if we want) – as long as both elements are enclosed inside the Grid tags. The physical position of the elements in the XAML does not matter in this case; the attached property does. For maintainability, however, you probably want to keep your XAML elements in a logical order.

Our first button will be a "Start" button. Here's our markup:

```
<!--Button Panel-->
<StackPanel Grid.Row="1" Orientation="Horizontal"
            HorizontalAlignment="Center" Margin="5">
    <!--Button with explicit Content tag-->
    <Button x:Name="startButton"
            FontSize="16"
            Margin="3"
            Padding="7,3,7,3">
        <Button.Content>Start</Button.Content>
    </Button>
</StackPanel>
```

The Name Property

For this `Button`, we have filled in the `Name` property; this is because we will be hooking up an event handler in just a bit. Notice that we are using `x:Name`, which means the Name property from the XAML namespace (with the "`x:`" referring to that namespace). You can also specify simply `Name`, which is the

property from the default ("presentation") namespace. These end up referring to the same underlying property. Most XAML elements define a `Name` property; if not, then you can use `x:Name` which is always available.

TypeConverter-Enabled Properties

The `Margin` and `Padding` properties are examples of TypeConverter-enabled properties. The Margin refers to the amount of space outside the control (between the edge of the control and its container); Padding refers to the amount of space inside the control (between the edge of the control and the contents). Both of these properties are of type "Thickness".

A Thickness is a complex type that has top, bottom, left, and right values. Fortunately, the .NET framework provides us with built-in type converters that allow us to use a shorthand in our attributes. In our example above, `Margin="3"` means that we have a Thickness of 3 units on each side. For the `Padding="7,3,7,3"` attribute, the Left and Right values are "7" and the Top and Bottom values are "3".

Button Content

The final thing to note is the `Content` property of the button. If you look, you will see that the Button does not have a "Text" or "Caption" property (as we might expect in a WinForms app); instead, it has "Content". This means that we can place whatever other visual elements we want inside a button. We could add a Grid or Stackpanel and put in additional text, an icon, or even moving video. For someone who has tried to do a button in WinForms that contains both text and an icon, this is hugely appreciated.

Since we are only showing text on our button, we simply have the word "Start" between the Content tags.

Adding a Style

We want all of our buttons to share certain properties, namely `FontSize`, `Margin`, and `Padding`. Rather than repeating these properties for each of our buttons, we can create a Style that puts these in a single location. We will do this by adding a Static Resource to our Window:

```
<Window x:Class="WPFStopWatch.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF Stop Watch" Height="150" Width="250" Topmost="True">
    <Window.Resources>
        <Style TargetType="Button" x:Key="ButtonStyle">
            <Setter Property="FontSize" Value="16"/>
            <Setter Property="Margin" Value="3"/>
            <Setter Property="Padding" Value="7,3,7,3"/>
        </Style>
    </Window.Resources>
```

You can see that we added a "`<Window.Resources>`" element. We won't go into Resources in detail here, but Resources are very valuable, and you will want to research this topic further.

The `Style` element has a few interesting properties. The first is the `TargetType`. This specifies that we will be applying this style to Buttons (or Button descendents). We will not be able to apply this style to any other type of control. Next is `x:Key`. As noted by the prefix, this comes from the XAML namespace. The Key is a unique value that we can use to reference this Style in our Window. Note that it is "Key" and not "Name"; "Key" is required for Resources.

The Style contains `Setter` elements which simply specify a Property name and a Value. We have three Setters; one for each of the properties we want to set in common.

New in Visual Studio 2010: The IntelliSense can now help you fill in the Property and Value fields. Since we have set the TargetType to Button, Visual Studio will give us a list of valid properties (such as FontSize and Margin) as code-completion. This is addition is greatly appreciated if you type in Styles by hand.

Using a Style

All UI controls contain a Style property (from the FrameworkElement base class). To use a style, we simply set the property of the element. Let's add a second button that uses our Style:

```
<!--Button Panel-->
<StackPanel Grid.Row="1" Orientation="Horizontal"
            HorizontalAlignment="Center" Margin="5">
    <!--Button with explicit Content tags-->
    <Button x:Name="startButton"
            FontSize="16"
            Margin="3"
            Padding="7,3,7,3">
        <Button.Content>Start</Button.Content>
    </Button>
    <!--Button with Style and implicit Content-->
    <Button x:Name="stopButton"
            Style="{StaticResource ButtonStyle}">Stop</Button>
</StackPanel>
```

You can see that the `Style` property is set to "`{StaticResource ButtonStyle}`". The curly braces note that we are referencing another element; this is known as a Markup Extension. "`StaticResource`" declares that we are referencing a Resource, and "`ButtonStyle`" is the Key of that Resource. This will set the FontSize, Margin, and Padding properties that we specified above. (As a side note, I have left the properties in the first button just to show the different formats. In a real application, you would want all of your buttons to refer the styles that apply).

Another Way to Specify Content

This second button shows a second way of specifying the Content property. Anything that is contained between the Button tags (the opening and closing tags) is the "Content". This is true for any controls that have a Content property. You can see that the text "Stop" is simply between the tags without any

additional identifiers. Most controls have a default property; you can find this in the on-line help for the control.

You can also put complex content with multiple elements between the tags (as we noted above if we were to create an icon/text button).

And a Final Way to Specify (Simple) Content

There is one further way of specifying Content. Let's look at our third button:

```
<StackPanel Grid.Row="1" Orientation="Horizontal"
            HorizontalAlignment="Center" Margin="5">
    <!--Button with explicit Content tags-->
    <Button x:Name="startButton"
            FontSize="16"
            Margin="3"
            Padding="7,3,7,3">
        <Button.Content>Start</Button.Content>
    </Button>
    <!--Button with Style and implicit Content-->
    <Button x:Name="stopButton"
            Style="{StaticResource ButtonStyle}">Stop</Button>
    <!--Button with Style and Content attribute-->
    <Button x:Name="clearButton"
            Style="{StaticResource ButtonStyle}"
            Content="Clear"/>
</StackPanel>
```

In this example, you'll see that we have specified the `Content` as an attribute: the text "Clear". This works well for buttons when we simply want to include text on the button. Since we do not have anything to put between the Button tags, we can use the "single tag" XML syntax which includes the closing slash at the end of the tag: `<Button.../>`

The limitation with this syntax is that we can only use this for simple text. If we want to do a more complex type (custom content or layout), then we would need to use one of the other two methods of specifying Content.

One More Complex Property

To finalize our layout, we'll add one more complex property: a gradient background for the grid. Here's the completed markup:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1" >
            <GradientStop Offset="0" Color="AliceBlue" />
            <GradientStop Offset="0.7" Color="SteelBlue" />
        </LinearGradientBrush>
    </Grid.Background>
...
</Grid>
```

Here we have defined the `<Grid.Background>` property and populated it with a `LinearGradientBrush`. We won't go into the details of this here; just note that it creates a diagonal gradient for our background. As a side note, even if you are hand-coding your XAML, it is often easier to create a gradient such as this in Expression Blend, and then copy the resulting XAML over to your file.

Here's our final layout:

Adding a Namespace

Now that we have our layout, we need to add the functionality. As noted when we started, we already have a "Ticker.cs" class that contains our Stop Watch behavior. We just have to figure out how to hook this into our application. Before we can use our custom class in the XAML, we have to add the namespace.

Fortunately, Visual Studio IntelliSense makes this easy for us. Up in our Window tag, we'll add another attribute. If we start typing "`xmlns:local=`", IntelliSense will give us a list of namespaces to choose from. These are the namespaces that are part of the assemblies in the "References" section of the application. If you look through the list, you will see an option for "JeremyBytes.StopWatch in assembly WPFStopWatch" ("JeremyBytes.StopWatch" is the namespace inside our "WPFStopWatch" project). This results in the following markup:

```
<Window x:Class="WPFStopWatch.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:JeremyBytes.StopWatch"
    Title="WPF Stop Watch" Height="150" Width="250" Topmost="True">
```

Notice that the syntax "clr-namespace:JeremyBytes.StopWatch" is put in for us by IntelliSense based on our selection. I chose the alias "local" since this is a convention when referencing namespaces that are part of the current project code.

Adding a Custom Class as a Static Resource

Next, we'll add the "Ticker" class as a Static Resource so that we can access it in our XAML. Here's the markup:

```
<Window.Resources>
    <local:Ticker x:Key="localTicker"/>

    <Style TargetType="Button" x:Key="ButtonStyle">
    ...
    </Style>
</Window.Resources>
```

Notice that we simply use the "local:" alias and our "Ticker" class name (which we get code-completion on). Then we give it a Key so that we can reference it (just like we did for the Style).

Data Binding

Data binding is well beyond the scope of this introduction. We'll just cover the specifics of how we are using it in this application. The data binding model in WPF (and Silverlight) is quite powerful, and you will definitely want to do further research in this area.

The Ticker class has a "DisplayInterval" property that shows the amount of time elapsed in a "mm:ss" format. We can bind the Text property of our TextBlock to this property of the Ticker class. Here's the markup:

```
<!--Output Display-->
<TextBlock Grid.Row="0"
           FontSize="48"
           HorizontalAlignment="Center" VerticalAlignment="Center"
           DataContext="{StaticResource localTicker}"
           Text="{Binding DisplayInterval}" />
```

The first thing we did was add a DataContext. The DataContext is where we hook up the object we want to bind to. In this case, we are connecting to the StaticResource that we just created and reference it by the Key. You'll note that this is a Markup Extension, and the syntax is exactly the same as we used for the Style properties of our buttons.

Next, we updated the Text property. Previously, it was hard-coded to "00:00". Now it is data bound to the "DisplayInterval" property in our DataContext (the Ticker class). As noted above, the complete data binding syntax is something you will want to look into further.

Event Handlers

Now that we have our data binding hooked up, the last step is to create the Event Handlers for our Buttons. The implementation of these will simply call methods from our Ticker class.

Specifying an Event Handler in XAML is just the same as setting any other Property as an attribute. For our buttons, we want to hook into the "`Click`" event. Visual Studio IntelliSense helps us out here, too. Inside the Button tag, type "`Click=`" and Visual Studio will give you an option to create a "New Event Handler". If you select this, then Visual Studio will automatically create the event handler in the code behind, give it a name, and hook it up in the XAML. Here's the markup for our "Start" button with the Click event:

```
<!--Button with explicit Content tags-->
<Button x:Name="startButton"
        FontSize="16"
        Margin="3"
        Padding="7,3,7,3"
        Click="startButton_Click">
    <Button.Content>Start</Button.Content>
</Button>
```

This is why we populated the `x:Name` property of our button: the Name is used in the automatic naming of the Event Handler.

Here are our other two buttons with the Click event:

```
<!--Button with a Style implicit Content-->
<Button x:Name="stopButton"
        Style="{StaticResource ButtonStyle}"
        Click="stopButton_Click">Stop</Button>
<!--Button with Content attribute-->
<Button x:Name="clearButton"
        Style="{StaticResource ButtonStyle}"
        Content="Clear"
        Click="clearButton_Click"/>
```

Completing the Application

The final step in our application is to implement the Event Handlers. This is very non-XAML as it all takes place in the code-behind. We'll run through this pretty quickly to get our application operational.

Here is the completed code-behind in the "MainWindow.xaml.cs" file:

```
using JeremyBytes.StopWatch;
...
public partial class MainWindow : Window
{
    Ticker _localTicker;
```

```
    public MainWindow()
    {
        InitializeComponent();
        _localTicker = (Ticker)FindResource("localTicker");
    }

    private void startButton_Click(object sender, RoutedEventArgs e)
    {
        _localTicker.Start();
    }

    private void stopButton_Click(object sender, RoutedEventArgs e)
    {
        _localTicker.Stop();
    }

    private void clearButton_Click(object sender, RoutedEventArgs e)
    {
        _localTicker.Clear();
    }
}
```

The first thing we do is create a class level variable of our Ticker class called "`_localTicker`". Next, we need to get a reference to the Ticker class that we are using in the XAML. We do this with the "`FindResource`" method of our Window and give it the Key of our resource. We then cast this to the "`Ticker`" type and assign it to our local variable.

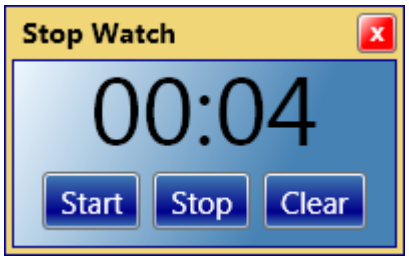
> *Silverlight 4 Difference: Silverlight 4 does not have a "FindResource" method. Instead, there is a "Resources" collection to reference. So the assignment looks like this in Silverlight:*
>
> ```
> _localTicker = (Ticker)Resources["localTicker"];
> ```

The rest of the methods are the implementations of our Click event handlers. You can see that we are simply passing the commands through to our `_localTicker`.

Here's the application in action:

A More Complex Stop Watch

In addition to the WPFStopWatch, the downloadable code also includes a WPFStyledStopWatch. This application has more complex styling and layout. Here's a screenshot of the output:



This application has a couple of interesting features. First, it uses a custom window (not using the standard Windows title bar or buttons), including a "Close" button with custom styling. One of the advantages of this is that this application will look the same regardless of whether it is run under Windows XP, Windows Vista, or Windows 7. As another feature, when it is inactive, it goes semi-transparent, yet still stays on top. This makes it useful if you want to time processes of other applications. You can look at the XAML and use it as a reference when exploring WPF further.

Wrap Up

We've taken a look at the basics of XAML, including namespaces, elements, properties, events, and attached properties. With this as an introduction, we can now hand edit our XAML when necessary. This will allow us to tweak the output of Expression Blend, or even hand-code an entire UI if we are more comfortable with that.

XAML is a key to creating UIs in WPF and Silverlight. It is powerful and flexible and gives us full control over how our users interact with our applications. Now all that's left is for us to go out and create some great user experiences.

Happy coding!