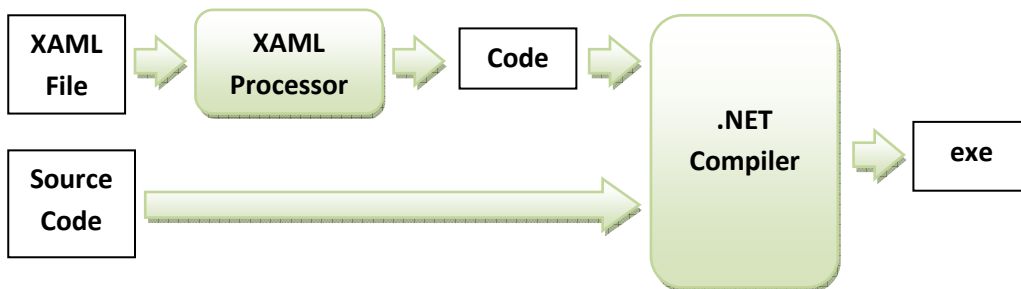




WPF XAML SYNTAX 101

This post provides the basic key points in order to understand the WPF XAML syntax and what is going on behind the scenes in the XAML Processor. Although you may be the kind of guy/girl who prefers to learn through writing/reading code, I recommend reading this article since it may save you a lot of time when trying to figure out how to combine the XAML Elements together.

First things first, we need to define what the XAML processor does. Well, the XAML processor as the following figure suggests, receives XAML XML language constructs and translates them to code that will be compiled to run in the .NET Framework.



So the user writes the XAML File and some source code.

Behind the scenes the XAML File is translated to code by the XAML Processor. The translated code is combined with the source code and the merged product is compiled to create the desired WPF application.

The programmer used in programming Windows Forms, apart from learning the new language constructs, he/she also needs to learn the XAML syntax and how it is translated to code. **This is the main topic of this article.** So we will be interested in this translation:



Elements and Attributes

First of all, in any XML document we have **XML elements and XML attributes**. For the XAML processor Elements are Class Instantiations and XML Attribute-Value pairs are Class Property-Value assignments.

For example the following table shows equivalence for the instantiation of a class of type Button:

XAML Code	C# Code
<code><Button x:Name="button1" IsEnabled="True"/></code>	<code>Button button1=new Button(); button1.IsEnabled=true;</code>

Namespaces

But as you know classes reside in namespaces. So how can you define a namespace to locate the classes that you need?

At the beginning of a XAML file there are two definitions:



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

The first one is the actual declaration of the WPF namespace and since it is not followed by a ":<name>" it is the default namespace. Since "Button" does not have a "<name>:" prefix its definition will be expected to be in there.

On the contrary the next namespace defines the XAML syntax and will carry the "x" name throughout the document. Actually the x namespace carries some special directives to the XAML Processor and does not necessarily follow the typical Attributes to Properties mapping. For example *the "x:Name" attribute in the Button declaration instructs the XAML Processor to create a named instance for the Button class because we want to be able to access it through code.*

So if you want to use in your XAML File your own classes and controls you need to define your namespace in XAML. The following example code snippet defines two namespaces the one (Entities) that will be named tools and resides in Entities.dll and the other (Force) that resides in the same dll with the XAML file.

```
xmlns:tools="clr-namespace:Entities;assembly=Entities"  
xmlns:local="clr-namespace:Force"
```

Moreover the definition:

```
x:Class="Force.MainClass"
```

states that there will be a code-behind class MainClass which will be used in conjunction with the XAML File which also carries code for the MainClass class.

Type Converters

But wait a second, in the previous example the property IsEnabled is Boolean and is assigned in XAML the value of "True" which is a String. And what will happen if the property is of type "Int"?

Well, when the programmer writes in XAML IsEnabled="True" the XAML Processor does the following:

It locates the property and finds out its type. It searches to find whether there exists a special class named "Type Converter" defined for the type. The type converter is a class that is fed by the XAML processor with the attribute value (eg "True") and returns a mapping of that value to the actual type (eg the Boolean value True). So for the Boolean type there is a Type Converter class that transforms String literals to Boolean values. You can create your own type converters for your own types (classes) by following the instructions in the following nice articles:

[How to: Implement a Type Converter](#)

[Walkthrough: Creating a Type Converter for the WPF Designer](#)

Nesting

Ok so we know how to instantiate classes in XAML and assign values to properties through the use of type converters.

Now we need to define how "Nesting" is interpreted by the XAML Processor. Nesting is the process of containing an XML tagged block within another. In XAML WPF the way nesting is handled is defined by the class that is instantiated.



For example nesting in the Button class is equivalent to assigning a value to the property "Content".

XAML Code	C# Code
<pre><Button x:Name="button1" IsEnabled="True"> Click Me! </Button></pre>	<pre>Button button1=new Button(); button1.IsEnabled=true; button1.Content="Click Me!"</pre>

Or in the ViewPort3D class nesting is equivalent to adding items to the property Children which is of a list type.

XAML Code	C# Code
<pre><ViewPort3D x:Name="view1"> <ModelVisual3D/> <ModelVisual3D/> </ViewPort3D></pre>	<pre>ViewPort3D view1=new ViewPort3D(); view1.Children.Add(new ModelVisual3D()); view1.Children.Add(new ModelVisual3D());</pre>

In general nesting either gives value to a specific "default" property or adds another element in a property that defines some kind of list.

A specific nesting type answers the previous question of giving values to properties that have to do with instantiating other classes. If a nested block is defined by properties of the class written in the dot syntax, then the contents are setting the value for this property:

To summarize up to now we can fully understand the following XAML files:

```
<Window x:Class="WpfApplication1.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">
  <Button x:Name="button1" IsEnabled="True">Test</Button>
</Window>
```

A button class is defined with its Content property equal to Test. The class is named button1. The property IsEnabled is assigned the Boolean value true through the use of a type converter that converts string values to Boolean. The property Content of the Window top-level class has the value of the instantiated button.

```
<Window x:Class="WpfApplication1.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">
  <Button x:Name="button1" Content="Test" IsEnabled="True" />
</Window>
```

Same as before without using the nesting.

Another way of setting the value to a property is by using the nesting as a property setting mechanism. For example, to set the Content property of the previous example we could:

```
<Window x:Class="WpfApplication1.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">
```



```
<Button x:Name="button1" IsEnabled="True">
  <Button.Content>Test</Button.Content>
</Button/>
</Window>
```

This way is especially useful when we want to instantiate whole new classes to be values for properties of objects. In the previous example the window class sets by default the content property through nesting which will be assigned a reference to the Button class. It is equivalent to:

```
<Window x:Class="WpfApplication1.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">
  <Window.Content>
    <Button x:Name="button1">
      <Button.Content>Test</Button.Content>
    </Button>
  </Window.Content>
</Window>
```

**Markup
Extensions**

But there is another way for setting values to properties through the use of Markup Extensions. Markup extensions are either defined by curly braces or as nested values for setting properties. For example:

```
<GridViewColumn Width="Auto" Header="Ερωτημα"
  DisplayMemberBinding="{Binding Path=Logo}"/>
```

Is equivalent to:

```
<GridViewColumn Width="Auto" Header="Ερωτημα">
  <GridViewColumn.DisplayMemberBinding>
    <Binding Path="Logo"/>
  </GridViewColumn.DisplayMemberBinding>
</GridViewColumn>
```

Markup extension classes tell the XAML Processor to create an instance of the defined class (in our case the Binding class), set some properties (in our case the Path property is set to "Logo") of the class and run the constructor with some parameters (in our case no parameters are passed) and call the ProvideValue member of the extension class to get the value that will be assigned to the property. The general syntax is as follows:

```
{MarkupExtensionClass
  ConstructorArg1, ConstructorArg2,...,ConstructorArgN,
  Property1=Value1,Property2=Value2,...,PropertyN=ValueN}
```

There is already a whole bunch of predefined markup extensions. Their summary can be found in the following:

[Markup Extensions and XAML](#)

This summarizes a brief introduction to the WPF XAML syntax.

Happy coding.