XAML From the Ground up

Instantiating object graphs using XML

THIS DOCUMENT COVERS

- Instantiating object graphs using XML
- Different Kinds of Element and Attribute

XAML is an XML-based declarative language which provides a concise way to construct and initialise object graphs. The best-known use of XAML is to specify the logical tree of controls and other user interface elements that constitute a user interface in WPF. There is, however, no reason why XAML cannot be used to construct object graphs in other situations. In this introductory article I will show how to initialise object graphs in XAML. We will start from something very simple and gradually add in more complexity to show how XAML supports very basic use cases while also enabling more complex scenarios.

HOW TO ADD A SIMPLE OBJECT TO THE OBJECT GRAPH

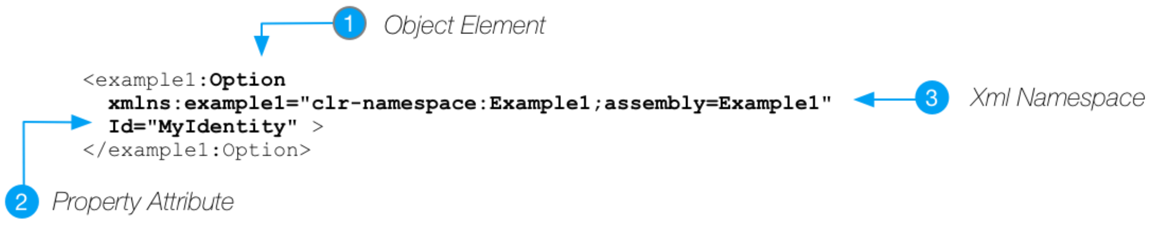
We define a type that has a single string property called `Id`.

Listing 1C# Option Type

```
namespace Example1
{
    public class Option
    {
        public string Id {get; set;}
    }


}
```

We then use XAML to declare that we want to instantiate an instance of Option as the root of an object graph and that we want to set the value of its `Id` property to the string `“MyIdentity”`

Listing 2 Declarative XAML



```
<example1:Option
  xmlns:example1="clr-namespace:Example1;assembly=Example1"
  Id="MyIdentity" >
</example1:Option>
```

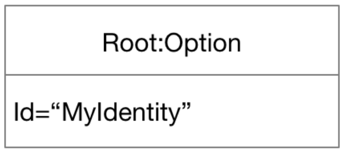
Finally, we use the **`XamlReader`** type from the `System.Windows.Markup` namespace to parse our XAML and instantiate the object graph as declared.

Listing 3 Parsing the XAML

```
private static void Main(string[] args)
{
    using (var fs = new FileStream("./ObjectGraph.xaml",
        FileMode.Open, FileAccess.Read))
    {
        Option option = (Option) XamlReader.Load(fs);
        Console.WriteLine(option.Id);
    }
}
```

Our runtime object graph is then as follows

Figure 1 Resulting Object Graph



How to set non-string properties via type converters

In the previous example our Id property was of type String, so it was easy to see how a string value in the XAML could be used to set a string property on the Option instance. What happens however if the property is on a non-string type? To highlight how this works we add a double property called `Strike` to our Option type

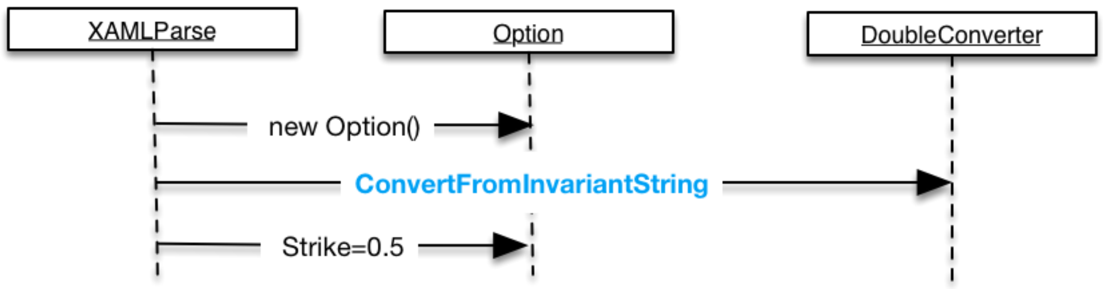
```
public class Option
{
    public string Id { get; set; }
    public double Strike { get; set; }
}
```

Our Xaml then becomes

```
<example1:Option
  xmlns:example1="clr-namespace:Example2;assembly=Example2"
  Id="MyIdentity" Strike="0.5">
</example1:Option>
```

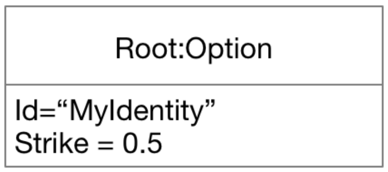
The XAML parser now needs to convert the string `"0.5"` into the double `0.5`. It does this via a subclass of the **TypeConverter**. The call sequence to instantiate and set our strike property is as follows.

Figure 2 Xaml Parsing with Type Conversion



Our final object graph then looks as follows

Figure 3Object Graph with Double Property



VALUECONVERTER VERSUS TYPECONVERTER

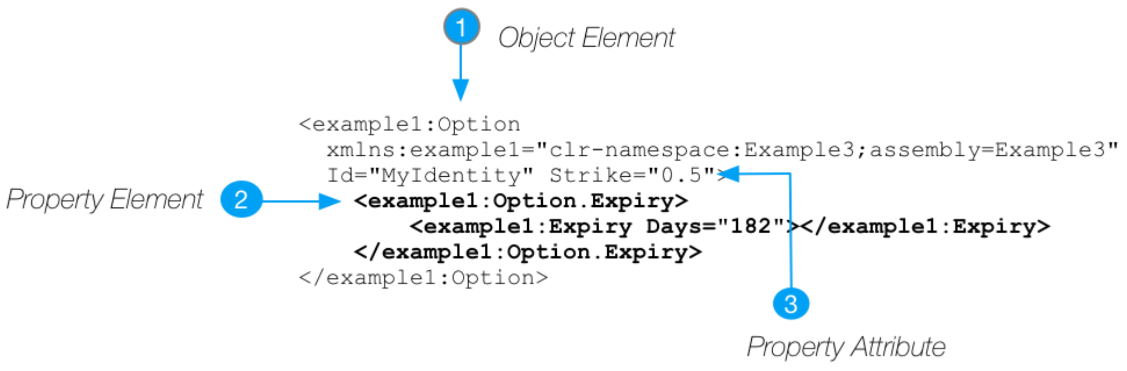
It is important to distinguish the `System.Windows.Data.IValueConverter` interface from the `Sytem.ComponentModel.TypeConverter` type. Instances of TypeConverter convert values to/from XAML strings. Instances of the IValueConverter interface are used in data-binding scenarios to allow properties of differing types to be bound together.

HOW TO SET COMPLEX OBJECTS AS PROPERTY VALUES

Setting simple properties such as doubles from XAML can be achieved easily via instances of TypeConverter as described in the previous section. Consider the scenario where the property we want to set is of a complex type consisting of multiple public properties which need to be set. To show how this works let us add a new non-primitive property to our Option type called `Expiry` of type `Expiry`. The Expiry type defines a simple single integer property called Days.

```
public class Option
{
    public string Id { get; set; }

    public double Strike { get; set; }


    public Expiry Expiry { get; set; }


}


public class Expiry
{
    public int Days { get; set; }
}
```

The way that XAML supports this is via **property elements.** Property elements are child elements that specify how to populate a property of an object element. The following XAML shows how this works



```
<example1:Option
  xmlns:example1="clr-namespace:Example3;assembly=Example3"
  Id="MyIdentity" Strike="0.5">
    <example1:Option.Expiry>
        <example1:Expiry Days="182"></example1:Expiry>
    </example1:Option.Expiry>
</example1:Option>
```

The XAML parser will then construct an instance of Expiry, set its Days property (using a IntegerConverter) and then set this as the value of the Options Expiry property.

How to create custom TypeConverters

We might want to define our own type converters that can convert from a string to an instance of the `Expiry` type. If we do this, we can contract our xaml to the following.

```
<example1:Option
  xmlns:example1="clr-namespace:Example4;assembly=Example4"
  Id="MyIdentity" Strike="0.5" Expiry="180">
</example1:Option>
```

In this case we write a class that can take a string representation of an expiry and use it to create a new instance of the **Expiry** type.

Listing 4Custom TypeConverter

```
    public class ExpiryConverter : TypeConverter
    {
        public override object ConvertFrom(ITypeDescriptorContext
context, CultureInfo culture, object value)
        {
            var exp = new Expiry();
            var daysToExpiry = 0;

            if (!string.IsNullOrEmpty(value as string) &&
                int.TryParse(value as string, out daysToExpiry))
            {
                exp.Days = daysToExpiry;
            }

            return exp;
        }

        public override object ConvertTo(ITypeDescriptorContext
context, CultureInfo culture, object value, destinationType)
        {
           Expiry exp = value as Expiry;
            if (exp != null)
                return exp.Days.ToString();

            return string.Empty;
        }

        public override bool CanConvertFrom(ITypeDescriptorContext
context, sourceType)
        {
            return sourceType == typeof (string);
        }

        public override bool CanConvertTo(ITypeDescriptorContext
context, Type destinationType)
        {
            return destinationType == typeof (string);
        }
    }
```

Finally, we need to mark the **`Expiry`** type with our newly created **`ExpiryConverter`** type thereby instructing the XAML parser to use it to map from instances of string to Expiry objects.

```
[TypeConverter(typeof(ExpiryConverter))]

public class Expiry
{
    public int Days { get; set; }
}
```

TYPECONVERTER AND IVALUECONVERTER

We need to be clear on when we would use System.ComponentModel.TypeConverter and when we would use System.Windows.Data.IvalueConverter. TypeConverter is used when mapping a string property value in XAML into a typed property value on the actual object. IvalueConverter is typically used when we are binding one value to another and need a conversion between them

Markup Extensions

If we extend our model types to include a compound object that has two Option object properties as follows.

```
public class TwoLegStrategy
{
    public Option LegOneOption { get; set; }

    public Option LegTwoOption { get; set; }
}
```

We could create our object graph as follows.

```
<TwoLegStrategy
  xmlns="clr-namespace:Example5;assembly=Example5" >
    <TwoLegStrategy.LegOneOption>
        <Option Strike="10.5" Expiry="180" ></Option>
    </TwoLegStrategy.LegOneOption>
    <TwoLegStrategy.LegTwoOption>
        <Option Strike="12" Expiry="180" ></Option>
    </TwoLegStrategy.LegTwoOption>
</TwoLegStrategy>
```

But this starts to get a little unwieldy. We ant to make the legs a bit more compact. To do this we use a markup extension. We do this as follows.

```
<TwoLegStrategy
    xmlns="clr-namespace:Example5;assembly=Example5"
    LegOneOption="{OptionMarkupExtension firstLeg,90,0.5}"
    LegTwoOption="{OptionMarkupExtension secondLeg,120,0.5}" />
```

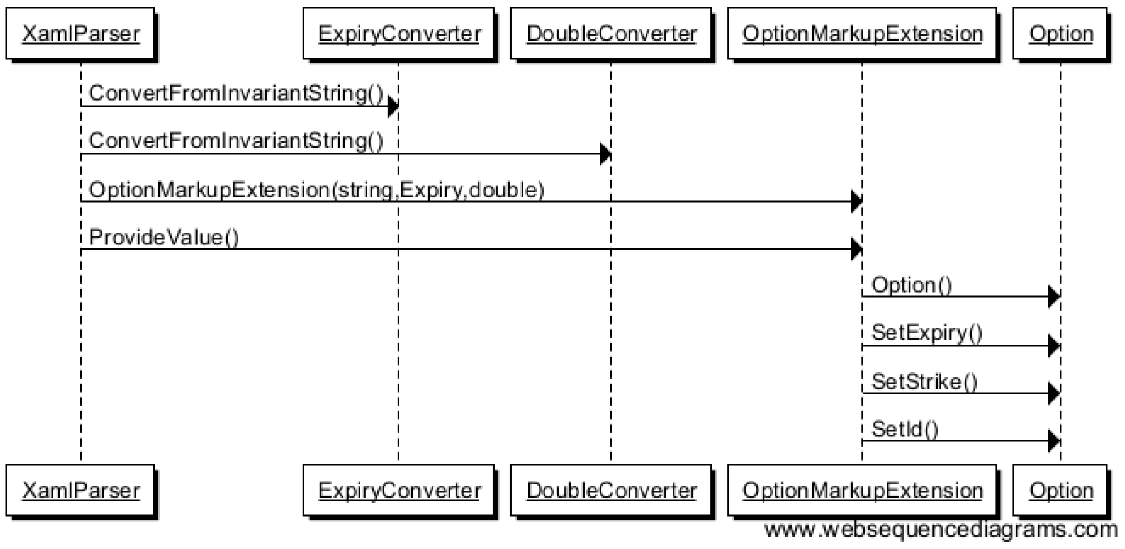
How do these magic markup extensions work? We need to create a subclass of the type MarkupExtension. This type will have a default constructor and also a constructor which takes one argument for each positional parameter that can be specified `="{OptionMarkupExtension firstLeg,90,0.5}` In this case a string id, integer number of days to expiry and a double strike.

```
public class OptionMarkupExtension : MarkupExtension
    {
        public OptionMarkupExtension(string id, Expiry expiry, double k)
        {
            Id = id;
            Expiry = expiry;
            K = k;
        }
```

We also need a method called **ProvideValue** which is the factory method that takes our **MarkupExtension** object and constructs an instance of the Option type.

```
        public override object ProvideValue(IServiceProvider serviceProvider)
        {
            var op = new Option {Expiry = Expiry, Strike = K, Id = Id};
            return op;
        }
```

Notice that in this case the Expiry field is of type Expiry, so the string in the XAML file must first be converted to an Expiry type using the ExpiryConverter and the string for the strike must be converted to a double using the **DoubleConverter** before the **OptionMarkupExtension** constructor is invoked.


XamlParser
ExpiryConverter
DoubleConverter
OptionMarkupExtension
Option
ConvertFromInvariantString()
ConvertFromInvariantString()
OptionMarkupExtension(string,Expiry,double)
ProvideValue()
Option()
SetExpiry()
SetStrike()
SetId()
www.websequencediagrams.com

We can also specify named parameters in our Xaml file in which case the OptionMarkupExtension's default constructor will be invoked followed by the named properties and then finally the **ProvideValue** method will be invoked.

```
[ConstructorArgument("Id")]
public string Id { get; set; }

[ConstructorArgument("Expiry")]
public Expiry Expiry { get; set; }

[ConstructorArgument("K")]
public double K { get; set; }
```

Our Xaml then becomes

```
<TwoLegStrategy
    xmlns="clr-namespace:Example5;assembly=Example5"
    LegOneOption="{OptionMarkupExtension Id=firstLeg,Expiry=180, K=0.5}"
    LegTwoOption="{OptionMarkupExtension Id=secondLeg,Expiry=120,K=0.5}" />
```

Because markup extensions are classes with default constructors we can use then as normal property element syntax. In the following the first leg is constructed using markup syntax and the second uses property element syntax.

```
<TwoLegStrategy
    xmlns="clr-namespace:Example5;assembly=Example5" >
    <TwoLegStrategy.LegOneOption>
        <OptionMarkup>
            <OptionMarkupExtension.Id>FirstLeg</OptionMarkupExtension.Id>
            <OptionMarkupExtension.Expiry>230</OptionMarkupExtension.Expiry>
            <OptionMarkupExtension.K>105</OptionMarkupExtension.K>
        </OptionMarkup>
    </TwoLegStrategy.LegOneOption>
    <TwoLegStrategy.LegTwoOption>
        <Option Id="SecondLeg" Expiry="250" Strike="110" />
    </TwoLegStrategy.LegTwoOption>
</TwoLegStrategy>
```

Child Items – building the object graph

We need to make the distinction between a property element and an object’s child elements. The distinction can be made as follows.

- A child element specifies a nodes child nodes in the object graph
- A property element specified the value of a particular property on a given node.

```
<Window x:Class="Example10PropertyAndChildNodes.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        SizeToContent="WidthAndHeight">
    <StackPanel>
        <Rectangle Fill="Red">
            <Rectangle.Height>60</Rectangle.Height>
            <Rectangle.Width>50</Rectangle.Width>
        </Rectangle>
    </StackPanel>
</Window>
```

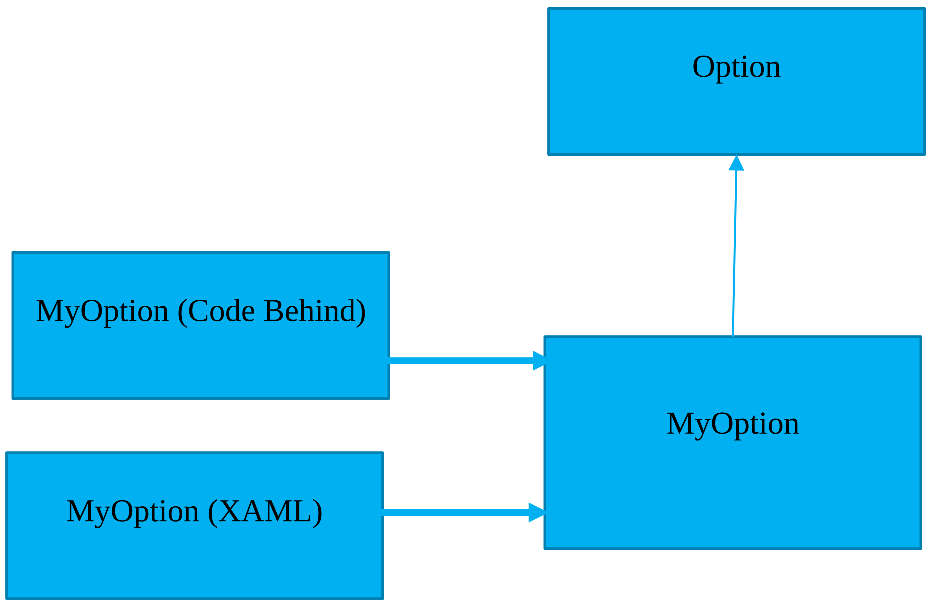
The green highlighted Rectangle element specifies a child node of the StackPanel. The blue highlighted Rectangle.Height and Rectangle.Width nodes specify how to construct objects that are set as the values of properties on a given node

Backing XAML with executable code

So far we have used XAML to define object graphs. Every such object graph has a single instance at the root of the XAML and the root of the object graph. We can provide executable code to back the object graph by creating a separate partial class which extends the type of the root element.

```
<example7:Option
    xmlns:example7="clr-namespace:Example7"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Example7.MyOption"
    Strike="102.5" Id="CodeBehing" Expiry="180"
    >
</example7:Option>
```

```
    public partial class MyOption : Option
    {
        public MyOption()
        {
            InitializeComponent();
        }

        public double CalculateIntrinsicValue(double spot)
        {
            return this.Strike;
        }
    }
```



Namespaces

The WPF assemblies map all the core .NET namespaces such as System.Windows, System.Windows.Controls etc to the xml namespace https://schemas.microsoft.com/winfx/xaml/presentation using a number of hardcoded XmlnsDefinitionAttribute custom attributes. As multiple .NET namespaces are mapping to a single xml namespace Microsoft have to be careful to ensure there are no name clashes in any of the relevant WPF namespaces.

The root element in any XAML file must contain at least one namespace. Consider the following simplest piece of XAML imaginable. It consists of one single element which is also the root element. Note in this case we have given this namespace a prefix of wpf that is then used on the identifiers from this namespace.

```
<wpf:Button
        xmlns:wpf="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  >
    <wpf:Button.Content>
        <wpf:Rectangle Height="40" Width="40" Fill="Black"></wpf:Rectangle>
    </wpf:Button.Content>
</wpf:Button>
```

Each XAML file can have a single primary namespace which has no prefix and that implicitly qualifies any identifiers not explicitly prefixed with another namespace

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  >
    <Button.Content>
        <Rectangle Height="40" Width="40" Fill="Black"></Rectangle>
    </Button.Content>
</Button>
```

In addition to `xmlns:wpf=http://schemas.microsoft.com/winfx/2006/xaml/presentation` most XAML files also contain the XAML language namespace `http://schemas.microsoft.com/winfx/2006/xaml`

Construct	**Use**
Object Element	Instantiate object note in graph

Property Attribute	Set object property using property value
Property Element	Set object property using element value
XML Namespace	Specify namespace of type to be instantiated

Mapping .NET types to XAML

Questions

Compare the use of the interface IValueConverter and the type TypeConverter

Subclasses of TypeConverter are used to convert between strings and objects, typically in XAML. For instance, a piece of XAML might specify a property attribute such as Strike=”0.5” and then the XAML parser will use a DoubleConverter to convert from the string “0.5” to the double 0.5.

Instances of IValueConverter are typically used in WPF data binding scenarios to convert between values of different types on the binding source and binding target

What is XAML

AnXML declarative language for constructing and initializing .NET object graphs

A precise way of describing object hierarchies

Why is XAML useful for building user interfaces?

The interface is a tree of elements

Compare and contrast child node elements and property node elements?

A child element specifies a sub-node in the graph

A property element specifies the value of a particular property on a given node.

How can we distinguish between property elements and object elements?

Property elements always contained inside a typename object element

They can never have attributes of their own

There element name is always of the form TyepName.PropertyName

How is a string property value in XAML converted to a non-string property?

By using a type converter (System.ComponentModek.TypeConverter)

What is a markup extension?

Enables one to create an object using shorthand

How do we create one?

Subclass System.Windows.Markup.MarkupExtention

What must a type provide to be useable from XAML?

Public default constructor and useful instance properties

What is the purpose of the x:Class in a xaml file?

Associates the xaml with the code-behind file

What is http://schemas.microsoft.com/winfx/xaml/presentation

The main WPF namespace

Contains types from many .NET namespaces such as System.Windows

What is http://schemas/microsoft.com/winfx/xaml/2006/xaml

The XAML language namespace

Maps types in System.Windows.Markup language

Defines some special directives for the XAML compiler or parser

How are the core WPF types mapped to https://schemas.microsoft.com/winfx/xaml/presentation?

By hardcoded XmlnsDefinition attribute in the .NET assemblies