1. WPF - XAML

This document describes the features of the XAML language and demonstrates how you can use XAML to write Windows Presentation Foundation (WPF) apps. This document specifically describes XAML as implemented by WPF. XAML itself is a larger language concept than WPF. It is, for example, also used in the MAUI framework that can be used to build cross-platform (Windows, Android, iOs) applications [1].

Contents

1.1 What is WPF?

Windows Presentation Foundation (WPF) is a UI framework to build Windows **desktop applications**. It is resolution-independent and uses a vector-based rendering engine, built to take advantage of modern graphics hardware. WPF provides a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2D and 3D graphics, animation, styles, templates, documents, media, text, and typography.

1.2 Markup and code-behind

WPF lets you develop an application using both *markup* and *code-behind*, an experience with which ASP.NET developers should be familiar. You generally use **XAML** markup to implement the appearance of an application while using **C#** (code-behind) to implement its behavior. This separation of appearance and behavior has the following benefits:

- Development and maintenance costs are reduced because appearance-specific markup isn't tightly coupled with behavior-specific code.
- Development is more efficient because designers can implement an application's appearance simultaneously with developers who are implementing the application's behavior.

1.2.1 Markup

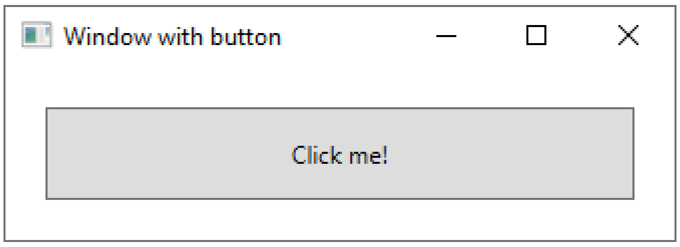
XAML is an XML-based markup language that implements an application's appearance declaratively. You typically use it to define windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics.

The following example uses XAML to implement the appearance of a window that contains a single button:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button">Click Me!</Button>

</Window>
```

Specifically, this XAML defines a window and a button by using the `Window` and `Button` elements. Each element is configured with attributes, such as the `Window` element's `Title` attribute to specify the window's title-bar text. At run time, **WPF converts the elements and attributes that are defined in markup to instances of WPF classes**. For example, the `Window` element is converted to an instance of the Window class whose Title property is the value of the `Title` attribute.

The following figure shows the user interface (UI) that is defined by the XAML in the previous example:



Since XAML is XML-based, the UI that you compose with it's assembled in a hierarchy of nested elements that is known as an element tree. The element tree provides a logical and intuitive way to create and manage UIs.

1.2.2 Code-behind

The main behavior of an application is to implement the **functionality** that responds to user interactions. For example clicking a menu or button, and calling business logic and data access logic in response. In WPF, this behavior is implemented in code that is associated with markup. This type of code is known as code-behind. The following example shows the updated markup from the previous example and the code-behind:

```xml
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

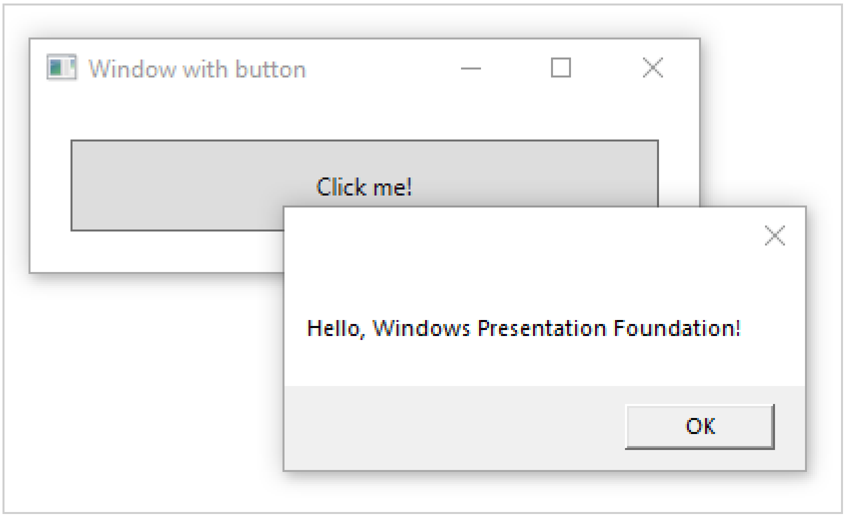
The updated markup defines the `xmlns:x` namespace and maps it to the schema that adds support for the code-behind types. The `x:Class` attribute is used to associate a code-behind class to this specific XAML markup. Considering this attribute is declared on the `<Window>` element, the code-behind class must inherit from the `Window` class.

```
using System.Windows;

namespace SDKSample
{
    public partial class AWindow : Window
    {
        public AWindow()
        {
            // InitializeComponent call is required to merge the UI
            // that is defined in markup with this class, including
            // setting properties and registering event handlers
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}
```

`InitializeComponent` is called from the code-behind class's constructor to merge the UI that is defined in markup with the code-behind class. (`InitializeComponent` is generated for you when your application is built, which is why you don't need to implement it manually.) The combination of `x:Class` and `InitializeComponent` ensure that your implementation is correctly initialized whenever it's created.

Notice that in the markup the `<Button>` element defined a value of `button_click` for the `Click` attribute. With the markup and code-behind initialized and working together, the Click event for the button is automatically mapped to the `button_click` method. When the button is clicked, the event handler is invoked and a message box is displayed by calling the System.Windows.MessageBox.Show method.

The following figure shows the result when the button is clicked:



Hello, Windows Presentation Foundation!

1.3 What is XAML?

XAML is a declarative markup language. XAML simplifies creating a UI for a .NET Core app. You can create visible UI elements in the declarative XAML markup, and then separate the UI definition from the run-time logic by using code-behind files that are joined to the markup through partial class definitions.

When represented as text, XAML files are XML files that generally have the `.xaml` extension.

1.3.1 XAML object elements

An object element typically declares an **instance of a type**.

Object element syntax always starts with an opening angle bracket (`<`). This is followed by the name of the type where you want to create an instance. (The name can include a prefix, a concept that will be explained later.) After this, you can optionally declare attributes on the object element. To complete the object element tag, end with a closing angle bracket (`>`). You can instead use a self-closing form that doesn't have any content, by completing the tag with a forward slash and closing angle bracket in succession (`/>`). For example, look at the following markup snippet:

```
<StackPanel>
    <Button Content="Click Me"/>
</StackPanel>
```

This specifies two object elements: `<StackPanel>` (with content, and a closing tag later), and `<Button .../>` (the self-closing form, with several attributes). The object elements `StackPanel` and `Button` each map to the name of a class that is defined by WPF and is part of the WPF assemblies.

When you specify an **object element tag**, you create an instruction for XAML processing to **create a new instance of the underlying type**. Each instance is created by calling the parameterless constructor of the underlying type when parsing and loading the XAML.

1.3.2 Attribute syntax

Properties of an object can often be expressed as attributes of the object element. **T**he attribute syntax names the **object property** that is being set, followed by the assignment operator (=). The **value** of an attribute is always **specified as a string** that is contained within quotation marks.

Attribute syntax is the most streamlined property setting syntax and is the most intuitive syntax to use for developers who have used markup languages in the past. For example, the following markup creates a button that has red text and a blue background with a display text of `Content`.

```
<Button Background="Blue" Foreground="Red" Content="This is a button"/>
```

1.3.3 Property element syntax

For some properties of an object element, attribute syntax isn't possible, because the object or information necessary to provide the property **value can't be adequately expressed within the quotation mark and string** restrictions of attribute syntax. For these cases, a different syntax known as property element syntax can be used.

The syntax for the property element start tag is **`<TypeName.PropertyName>`**. Generally, the **content of that tag is an object element of the type that the property takes as its value**. After specifying the content, you must close the property element with an end tag. The syntax for the end tag is `</TypeName.PropertyName>`.

If an attribute syntax is possible, using the attribute syntax is typically more convenient and enables a more compact markup, but that is often just a matter of style, not a technical limitation. The following example shows the same properties being set as in the previous attribute syntax example, but this time by using property element syntax for all properties of the `Button`.

```
<Button>
    <Button.Background>
        <SolidColorBrush Color="Blue"/>
    </Button.Background>
    <Button.Foreground>
        <SolidColorBrush Color="Red"/>
    </Button.Foreground>
    <Button.Content>
        This is a button
    </Button.Content>
</Button>
```

1.3.4 Collection syntax

The XAML language includes some optimizations that produce more human-readable markup. One such optimization is that if a particular property takes a collection type, then items that you declare in markup as child elements within that property's value become part of the collection. In this case, a collection of child object elements is the value being set to the collection property.

The following example shows collection syntax for setting values of the GradientStops property.

```
<LinearGradientBrush>
    <LinearGradientBrush.GradientStops>
        <!-- no explicit new GradientStopCollection, parser knows how to find or create -->
        <GradientStop Offset="0.0" Color="Red" />
        <GradientStop Offset="1.0" Color="Blue" />
    </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

1.3.5 Content syntax

XAML specifies a language feature whereby a class can designate exactly one of its properties to be the XAML *content* property. Child elements of that object element are used to set the value of that content property. In other words, for the content property uniquely, you can omit a property element when setting that property in XAML markup and produce a more visible parent/child metaphor in the markup.

For example, Border specifies a *content* property of Child. The following two Border elements are treated identically. The first one takes advantage of the content property syntax and omits the `Border.Child` property element. The second one shows `Border.Child` explicitly.

```
<Border>
    <TextBox Width="300"/>
</Border>
<!--explicit equivalent-->
<Border>
    <Border.Child>
        <TextBox Width="300"/>
    </Border.Child>
</Border>
```

Note: The XAML parser knows that the XAML content property of Border is Child, because its base class, Decorator, has an attribute *ContentProperty* set:

```
[System.Windows.Localizability(System.Windows.LocalizationCategory.Ignore,
Readability=System.Windows.Readability.Unreadable)]
[System.Windows.Markup.ContentProperty("Child")]
public class Decorator : System.Windows.FrameworkElement, System.Windows.Markup.IAddChild
```

As a rule of the XAML language, the value of a XAML content property must be given either entirely before or entirely after any other property elements on that object element. For instance, the following markup doesn't compile.

```
<Button>I am a
  <Button.Background>Blue</Button.Background>
  blue button</Button>
```

1.3.6 Attribute syntax (events)

Attribute syntax can also be used for members that are events rather than properties. In this case, the attribute's name is the name of the event. In the WPF implementation of events for XAML, the attribute's value is the name of a handler that implements that event's delegate. For example, the following markup assigns a handler for the Click event to a Button created in markup:

```
<Button Click="Button_Click" >Click Me!</Button>
```

1.3.7 Markup extensions

Markup extensions are a XAML language concept. When used to provide the value of an attribute syntax, curly braces (`{` and `}`) indicate a markup extension usage. This usage directs the XAML processing to **escape from the general treatment of attribute values** as either a literal string or a string-convertible value.

The most common markup extensions used in WPF app programming are Binding, used for data binding expressions, and the resource references StaticResource. By using markup extensions, you can use attribute syntax to provide values for properties even if that property doesn't support an attribute syntax in general. Markup extensions often use intermediate expression types to enable features such as deferring values or referencing other objects that are only present at run-time.

For example, the following markup sets the value of the Style property using attribute syntax. The Style property takes an instance of the Style class, which by default could not be instantiated by an attribute syntax string. But in this case, the attribute references a particular markup extension, `StaticResource`. When that markup extension is processed, it returns a reference to a style that was previously instantiated as a keyed resource in a resource dictionary.

```xml
<Window x:Class="index.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="100" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
        <Style TargetType="Border" x:Key="PageBackground">
            <Setter Property="BorderBrush" Value="Blue"/>
            <Setter Property="BorderThickness" Value="5" />
        </Style>
    </Window.Resources>
    <Border Style="{StaticResource PageBackground}">
        <StackPanel>
            <TextBlock Text="Hello" />
        </StackPanel>
    </Border>
</Window>
```

For more information about markup extension concepts, see Markup Extensions and WPF XAML.

1.3.8 Type converters

In the attribute syntax section, it was stated that the attribute value must be able to be set by a string. The basic, native handling of how strings are converted into other object types or primitive values is based on the String type itself, along with native processing for certain types such as DateTime or Uri. But many WPF types or members of those types extend the basic string attribute processing behavior in such a way that instances of more complex object types can be specified as strings and attributes.

The Thickness structure is an example of a type that has a type conversion enabled for XAML usages. Thickness indicates measurements within a nested rectangle and is used as the value for properties such as Margin. By placing a type converter on Thickness, all properties that use a Thickness are easier to specify in XAML because they can be specified as attributes. The following example uses a type conversion and attribute syntax to provide a value for a Margin:

```
<Button Margin="10,20,10,30" Content="Click me"/>
```

The previous attribute syntax example is equivalent to the following more verbose syntax example, where the Margin is instead set through property element syntax containing a Thickness object element. The four key properties of Thickness are set as attributes on the new instance:

```
<Button Content="Click me">
    <Button.Margin>
        <Thickness Left="10" Top="20" Right="10" Bottom="30"/>
    </Button.Margin>
</Button>
```

1.3.9 Root elements and namespaces

A XAML file must have only one root element, to be both a well-formed XML file and a valid XAML file. For typical WPF scenarios, you use a root element that has a prominent meaning in the WPF app model (for example, Window or Page for a page, ResourceDictionary for an external dictionary, or Application for the app definition). The following example shows the root element of a typical XAML file for a WPF page, with the root element of Page.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:NumericUpDownCustomControl;assembly=CustomLibrary"
    >
  <StackPanel Name="LayoutRoot">
    <custom:NumericUpDown Name="numericCtrl1" Width="100" Height="60"/>
...
  </StackPanel>
</Page>
```

The root element also contains the attributes `xmlns` (**xml** **name**space) and `xmlns:x`. These attributes indicate to a XAML processor which XAML namespaces contain the type definitions for backing types that the markup will reference as elements. The `xmlns` attribute specifically indicates the default XAML namespace. Within the default XAML namespace, object elements in the markup can be specified without a prefix. For most WPF app scenarios, and for almost all of the examples given in the WPF sections of the SDK, the default XAML namespace is mapped to the WPF namespace `http://schemas.microsoft.com/winfx/2006/xaml/presentation`.

The `xmlns:x` attribute indicates an additional XAML namespace, which maps the XAML language namespace `http://schemas.microsoft.com/winfx/2006/xaml`.

The `xmlns` attributes are only strictly necessary on the root element of each XAML file. `xmlns` definitions will apply to all descendant elements of the root element. `xmlns` attributes are also permitted on other elements underneath the root, and would apply to any descendant elements of the defining element. However, frequent definition or redefinition of XAML namespaces can result in a XAML markup style that is difficult to read.

1.3.10 The x: prefix

In the previous root element example, the prefix `x:` was used to map the XAML namespace `http://schemas.microsoft.com/winfx/2006/xaml`, which is the dedicated XAML namespace that supports XAML language constructs. The XAML namespace for the XAML language contains several programming constructs that you will use frequently in your XAML. The following is a listing of the most common `x:` prefix programming constructs you will use:

- x:Key: Sets a unique key for each resource in a ResourceDictionary (or similar dictionary concepts in other frameworks). `x:Key` will probably account for 90 percent of the `x:` usages you will see in a typical WPF app's markup.
- x:Class: Specifies the CLR (common language runtime) namespace and class name for the class that provides code-behind for a XAML page. You must have such a class to support code-behind per the WPF programming model, and therefore you almost always see `x:` mapped, even if there are no resources.
- x:Name: Specifies a run-time object name for the instance that exists in run-time code after an object element is processed. In general, you will frequently use a WPF-defined equivalent property for x:Name. Such properties map specifically to a CLR backing property and are thus more convenient for app programming, where you frequently use run-time code to find the named elements from initialized XAML. The most common such property is FrameworkElement.Name. You might still use x:Name when the equivalent WPF framework-level Name property isn't

supported in a particular type. This occurs in certain animation scenarios.
- x:Static: Enables a reference that returns a static value that isn't otherwise a XAML-compatible property.
- x:Type: Constructs a Type reference based on a type name. This is used to specify attributes that take Type, such as Style.TargetType, although frequently the property has native string-to-Type conversion in such a way that the x:Type markup extension usage is optional.x:Type is the XAML equivalent of typeof in C#

1.3.11 Custom prefixes and custom types

For your own custom assemblies, or for assemblies outside the WPF core of PresentationCore, PresentationFramework and WindowsBase, you can specify the assembly as part of a custom `xmlns` mapping. You can then reference types from that assembly in your XAML, so long as that type is correctly implemented to support the XAML usages you are attempting.

The syntax takes the following possible named tokens and following values:

- `clr-namespace:` The CLR namespace declared within the assembly that contains the public types to expose as elements.
- `assembly=` The assembly that contains some or all of the referenced CLR namespace. This value is typically just the name of the assembly, not the path, and does not include the extension (such as .dll or .exe).

The following is a basic example of how custom prefixes work in XAML markup. The prefix `custom` is defined in the root element tag, and mapped to a specific assembly that is packaged and available with the app. This assembly contains a type `NumericUpDown`, which is implemented to support general XAML usage as well as using a class inheritance that permits its insertion at this particular point in a WPF XAML content model. An instance of this `NumericUpDown` control is declared as an object element, using the prefix so that a XAML parser knows which XAML namespace contains the type, and therefore where the backing assembly is that contains the type definition.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:NumericUpDownCustomControl;assembly=CustomLibrary"
    >
  <StackPanel Name="LayoutRoot">
    <custom:NumericUpDown Name="numericCtrl1" Width="100" Height="60"/>
...
  </StackPanel>
</Page>
```

1.3.12 Named elements

By default, the object instance that is created in an object graph by processing a XAML object element doesn't have a unique identifier or object reference. In contrast, if you call a constructor in code, you almost always use the constructor result to set a variable to the constructed instance, so that you can reference the instance later in your code. To provide standardized access to objects that were created through a markup definition, XAML defines the x:Name attribute. You can set the value of the `x:Name` attribute on any object element. In your code-behind, the identifier you choose is equivalent to an instance variable that refers to the constructed instance. In all respects, named elements function as if they were object instances (the name references that instance), and your code-behind can reference the named elements to handle run-time interactions within the app.

WPF framework-level XAML elements inherit a Name property, which is equivalent to the XAML defined `x:Name` attribute. Certain other classes also provide property-level equivalents for `x:Name`, which is also usually defined as a `Name` property. Generally speaking, if you can't find a `Name` property in the members table for your chosen element/type, use `x:Name` instead. The `x:Name` values will provide an identifier to a XAML element that can be used at run-time.

The following example sets Name on a StackPanel element. Then, a handler on a Button within that StackPanel references the StackPanel through its instance reference `buttonContainer` as set by Name.

```
<StackPanel Name="buttonContainer">
    <Button Click="RemoveThis_Click">Click to remove this button</Button>
</StackPanel>
```

```
private void RemoveThis_Click(object sender, RoutedEventArgs e)
{
    var element = (FrameworkElement)e.Source;

    if (buttonContainer.Children.Contains(element))
        buttonContainer.Children.Remove(element);
}
```

1.3.13 Attached properties

XAML specifies a language feature that enables certain **properties to be specified on any element, even if the property doesn't exists in the type's definitions for the element it's being set on**. This feature is called an attached property. Conceptually, you can think of attached properties as **global members that can be set on any XAML element/object instance**.

Attached properties in XAML are typically used through attribute syntax. In attribute syntax, you specify an attached property in the form **`ownerType.propertyName`**.

Superficially, this resembles a property element usage, but in this case the `ownerType` you specify is always a different type than the object element where the attached property is being set. `ownerType` is the type that provides the accessor methods that are required by a XAML processor to get or set the attached property value.

The most common scenario for attached properties is to enable child elements to report a property value to their parent element.

The following example illustrates the DockPanel.Dock attached property. The DockPanel class defines the accessors for DockPanel.Dock and owns the attached property. The DockPanel class also includes logic that iterates its child elements and specifically checks each element for a set value of DockPanel.Dock. If a value is found, that value is used during layout to position the child elements. Use of the DockPanel.Dock attached property and this positioning capability is in fact the motivating scenario for the DockPanel class.

```
<DockPanel>
    <Button DockPanel.Dock="Left" Width="100" Height="20">I am on the left</Button>
    <Button DockPanel.Dock="Right" Width="100" Height="20">I am on the right</Button>
</DockPanel>
```

1.3.14 Base types

Underlying WPF XAML and its XAML namespace is a collection of types that correspond to CLR (**c**ommon **l**anguage **r**untime) objects and markup elements for XAML. However, not all classes can be mapped to elements. Abstract classes, such as ButtonBase, and certain non-abstract base classes, are used for inheritance in the CLR objects model. Base classes, including abstract ones, are still important to XAML development because each of the concrete XAML elements inherits members from some base class in its hierarchy. Often these members include properties that can be set as attributes on the element, or events that can be handled.

FrameworkElement is the concrete base UI class of WPF at the WPF framework level. When designing UI, you will use various shape, panel, decorator, or control classes, which all derive from FrameworkElement.

1.4 Sources

[1] Microsoft, “Desktop Guide (WPF .NET)”, Available: https://docs.microsoft.com/nl-nl/dotnet/desktop/wpf