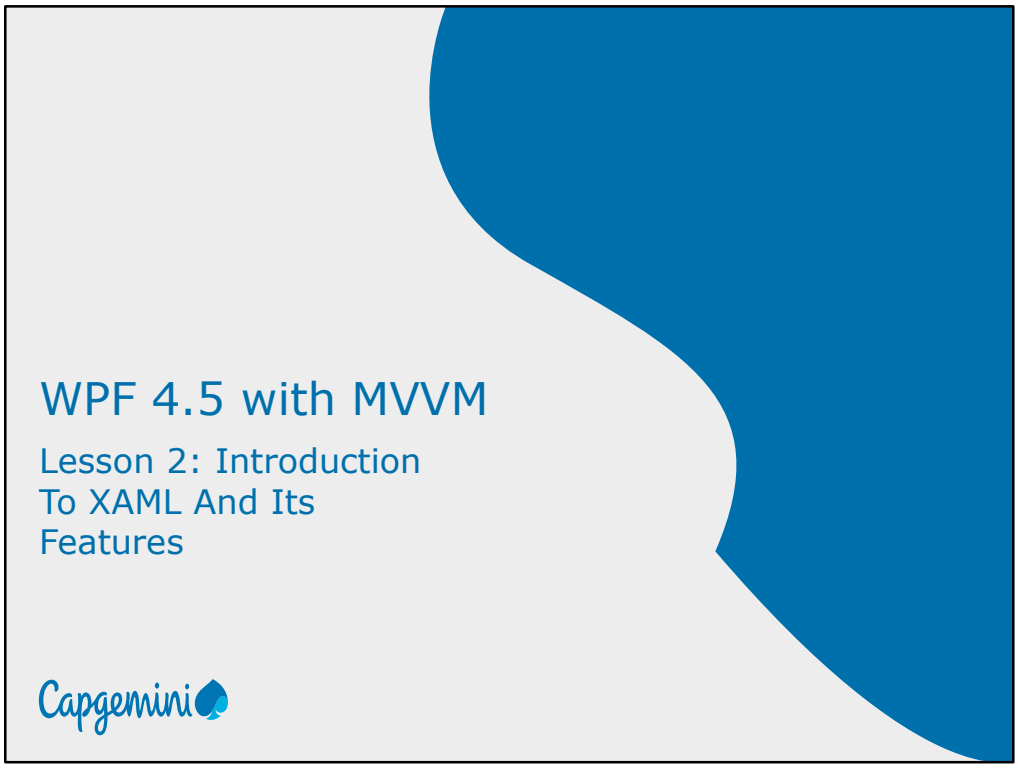
WPF 4.5 with MVVM
Lesson 2: Introduction
To XAML And Its
Features
Capgemini

Lesson Objectives

- In this lesson, you will learn:
  - Introduction to XAML
  - WPF Property System

2.1: Introduction To XAML

Overview

- XAML (Extensible Application Markup Language) is a markup language used to instantiate .NET objects
- Although XAML is a technology that can be applied to many different problem domains, its primary role in life is to construct WPF user interfaces
- In other words, XAML documents define the arrangement of panels, buttons, and controls that make up the windows in a WPF application

XAML:
XAML is a relatively simple and general-purpose declarative programming language suitable for constructing and initializing .NET objects.
The .NET Framework 3.0 includes a compiler and run-time parser for XAML, as well as a plug-in that enables you to view standalone WPF-based XAML files (sometimes called loose XAML pages) inside Internet Explorer.

2.1: Introduction To XAML

Overview

- XAML being reminiscent of WinForms, you will find many of the controls you used while building applications: Button, ComboBox, ListBox, and so on
- However, in WPF, UI design is represented in a completely different fashion
- Instead of using a designer generated code file or a resource file as the source of a UI definition, WPF uses XAML

2.1: Introduction To XAML

Overview

- XAML stands for eXtensible Application Markup Language
- You can think of XAML as HTML for Windows applications, however, it is a bit more expressive and powerful
- XAML is a relatively simple and general-purpose declarative programming language suitable for constructing and initializing .NET objects

XAML:
XAML stands for eXtensible Application Markup Language.
It is an all-purpose XML-based language used for declaring object graphs, which are instantiated at runtime. XAML is used by WPF developers to declare the layout of a user interface (UI), and the resources used in that UI.

2.1: Introduction To XAML

Basics

- Every element in a XAML document maps to an instance of a .NET class
- The name of the element matches the name of the class exactly
- Example: The element <Button>instructs WPF to create a Button object
- As with any XML document, you can nest one element inside another
- You can set properties of each class through attributes

XAML:

The XAML standard is quite straightforward once you understand a few ground rules:

Every element in a XAML document maps to an instance of a .NET class. The name of the element matches the name of the class exactly.

Example: The element <Button>instructs WPF to create a Button object.

As with any XML document, you can nest one element inside another. As you will see, XAML gives every class the flexibility to decide how it handles this situation. However, nesting is usually a way to express containment — in other words, if you find a Button element inside a Grid element, your user interface probably includes a Grid that contains a Button inside.

You can set the properties of each class through attributes. However, in some situations an attribute is not powerful enough to handle the job. In these cases, you will use nested tags with a special syntax.

2.1: Introduction To XAML

Basic Elements And Attributes

- The .NET Framework 3.0 includes a compiler and run-time parser for XAML. It also includes a plug-in that enables you to view standalone WPF-based XAML files (sometimes called loose XAML pages) inside Internet Explorer
- The XAML specification defines rules that map .NET namespaces, types, properties, and events into XML namespaces, elements, and attributes

2.1: Introduction To XAML

Basic Elements And Attributes

- When you compile an application that contains XAML files, the markup gets converted into BAML
- BAML is a tokenized, binary representation of XAML
- This binary representation is then stored inside the application's resources and loaded as needed by WPF during the execution of your program
- The main advantage of this approach is that you get a faster UI load time by reading a binary stream than through parsing XML

2.1: Introduction To XAML

Basic Elements And Attributes

- Let us see an example on XAML:

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

XAML:
Example of XAML:
This document includes only two elements:

- the top-level Window element, which represents the entire window,
- the Grid, in which you can place all your controls

Although you can use any top-level element, WPF applications rely on just a few:

- Window
- Page (which is similar to Window, but used for navigable applications)
- Application (which defines application resources and startup settings)

As in all XML documents, there can only be one top-level element. In the previous example, as soon as you close the Window element with the </Window> tag, you end the document. No more content can follow.

2.1: Introduction To XAML

Namespaces

- http://schemas.microsoft.com/winfx/2006/xaml/presentation is the core WPF namespace
  - It encompasses all the WPF classes, including the controls you use to build user interfaces
- http://schemas.microsoft.com/winfx/2006/xaml is the XAML namespace
  - It includes various XAML utility features that allow you to influence how your document is interpreted

XAML:
XAML Namespaces:
The most mysterious part about comparing the previous XAML example with the equivalent C# examples is how the XML namespace http://schemas.microsoft.com/winfx/2006/xaml/presentation maps to the .NET namespace System.Windows.Controls. Resultantly the mapping to XML and other WPF namespaces is hardcoded inside the WPF assemblies with several instances of an XmlnsDefinitionAttribute custom attribute.
The root object element in a XAML file must specify at least one XML namespace that is used to qualify itself and any child elements. You can declare additional XML namespaces (on the root or on children). However, each one must be given a distinct prefix to be used on any identifiers from that namespace.
Example: WPF XAML files typically use a second namespace with the prefix x (denoted by using xmlns:x instead of just xmlns):

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
This is the XAML language namespace, which maps to types in the System.Windows.Markup namespace but also defines some special directives for the XAML compiler or parser.

2.1: Introduction To XAML

Class Attribute

- XAML allows you to construct a user interface. However, in order to make a functioning application you need a way to connect the event handlers that have your application code
- XAML makes this easy using the Class attribute that is shown below:
- <Window x:Class="WindowsApplication1.Window1"

XAML:
The Code-Behind Class:
The x namespace prefix places the Class attribute in the XAML namespace. This is a more general part of the XAML language. In fact, the Class attribute tells the XAML parser to generate a new class with the specified name. That class derives from the class that is named by the XML element.
In other words, this example creates a new class named WindowsApplication1.Window1, which derives from the base Window class.

2.1: Introduction To XAML

.NET Object Creation

- XAML:

```
<Button
  xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
  Content="OK"/>
```

- C#:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
```

XAML:
.NET Object Creation:
Declaring an XML element in XAML (known as an object element) is equivalent to instantiating the corresponding .NET object (always via a default constructor).

2.1: Introduction To XAML

Setting Attributes Using XAML

- XAML:

```
<Button
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
Content="OK" Click="button_Click"/>
```

- C#:

```
System.Windows.Controls.Button b = new
System.Windows.Controls.Button();
    b.Click += new System.Windows.RoutedEventHandler(button_Click);
   b.Content = "OK" ;
```

XAML:
Setting Attributes using XAML:
Setting an attribute on the object element is equivalent to setting a property of the same name (called a property attribute) or hooking up a handler for an event of the same name (called an event attribute).
XAML is no more than a special type of CLR object serialization. It was intentionally architected to serialize CLR object graphs into an XML representation that is both verbose and human readable.

This makes it possible for developers to easily edit the markup by hand and enable the creation of powerful graphical tools that could generate markup for you behind the scenes.

2.1: Introduction To XAML

Content Property

- Any object can declare a default content property
  - <Button>Click Me!</Button>
  - "Click Me!" here is Content
- There is always a .NET class behind a XAML element
- With attributes and child elements, you set the value of properties and define handler methods for events

WPF Design Principles:
Properties:
Consider the Text attribute on the Window element:

```
<Window xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
x:Class="XamlProj.Window1" Text="Main Window">
```

Text: This attribute has no namespace qualifier. In XAML, unqualified attributes usually correspond to properties on the .NET object to which the element refers. (They can also refer to events, as we will see later). The Text attribute indicates that when an instance of this generated XamlProj.Window1 class is constructed, it should set its own Text property to "Main Window". This is equivalent to the following code:

```
myWindow.Text = "Main Window";
```

Before working with XAML, you need to know important characteristics of the XAML syntax. You can use XML attributes to specify the properties of classes. The following example shows the setting of the Content and Background properties of the Button class:

```
<Button Content="Click Me!" Background="LightGreen" />
```

2.1: Introduction To XAML

Content Property

- Properties as Attributes:
  - <Button Content="Click Me!" Background="LightGreen" />
- Properties as Elements:
  - Instead of using XML attributes, the properties can also be specified as child elements.
    - <Button>
    - <Button.Background>
    - …..
    - </Button>

WPF Design Principles:
Properties as Elements:
Instead of using XML attributes, the properties can also be specified as child elements. The value for the content can be directly set by specifying the child elements of the Button element. For all other properties of the Button, the name of the child element is defined by the name of the outer element, followed by the property name:

```
<Button>
  <Button.Background> LightGreen </Button.Background> Click
Me!
</Button>
```

In the previous example, it is not necessary to use child elements. By using XML attributes, the same result was achieved. However, using attributes is no longer possible if the value is more complex than a string. For example: Not only the background can be set to a simple color but also to a brush, for example, a linear gradient brush:

```
<Button>
   <Button.Background>
      <LinearGradientBrush StartPoint="0,0"
EndPoint="1,1">
          <GradientStop Color="Yellow" Offset="0.0" />
          <GradientStop Color="Orange" Offset="0.25" />
           <GradientStop Color="Red" Offset="0.75" />
           <GradientStop Color="Violet" Offset="1.0" />
      </LinearGradientBrush>
   </Button.Background>
   Click Me!
</Button>
```

2.2: WPF Design Principles

Attached Properties

- A WPF element can also get features from the parent element
  - Example: If the Button element is located inside a Canvas element, the button has Top and Left properties that are prefixed with the parent element's name. Such a property is known as attached property

2.2: WPF Design Principles

Attached Properties

- Let us see an example of Attached Properties:

```
<DockPanel>
  <TextBlock DockPanel.Dock="Top">My UI</TextBlock>
  <ListBox DockPanel.Dock="Right">
    <ListBoxItem>Item 1</ListBoxItem>
    <ListBoxItem>Item 2</ListBoxItem>
  </ListBox>
  <RichTextBox/>
</DockPanel>
```

WPF Design Principles:
Attached Properties:
Both TextBlock and ListBox have a DockPanel.Dock attribute on them.
If you examine these classes, you will find that neither has a property that looks like this. DockPanel declares a DependencyProperty called DockProperty. This special type of DependencyProperty, called an Attached Property, allows a parent control to store information with its children.
In this case, the DockPanel stores information it needs for layout with its child controls. In XAML this manifests itself as an attribute on the child element in the form: ParentName.AttachedPropertyName.

2.2: WPF Design Principles

Attached Properties

- Both TextBlock and ListBox have a DockPanel.Dock attribute on them
- If you examine these classes, you will find that neither has a property that looks like this
- DockPanel declares a DependencyProperty called DockProperty
- This special type of DependencyProperty, called an Attached Property, allows a parent control to store information with its children

2.2: WPF Design Principles

Attached Properties

- In this case, the DockPanel is storing information it needs for layout with its child controls
- In XAML this manifests itself as an attribute on the child element in the form: ParentName.AttachedPropertyName

2.3: WPF Application Class

Markup Extensions

- While setting values for elements, you can set the value directly. However, sometimes markup extensions are very helpful
- Markup extensions consist of curly brackets followed by a string token that defines the type of the markup extension
- Here is an example of a markup extension:
- <Style TargetType="{x:Type Button}">

WPF Design Principles:
Markup Extensions:
Type converters and property elements allows you to initialize most properties to constant values or fixed structures. But there are certain situations where a little more flexibility is required. For example, you might want to set a property to be equal to the value of some particular static property. However, we do not know at compile time what that value will be (for example, setting properties representing user-configured colors). XAML provides a powerful solution in the form of markup extensions. A markup extension is a class that decides what the value of a property should be.
The TargetType property of the Style has a value enclosed in braces. This indicates to the XAML compiler that a markup extension is being used. The first string inside the braces is the name of the markup extension class, and the remaining contents are passed to the markup extension during initialization.
We are using x:Type in the above example. There is no class called Type in any of the .NET namespaces represented by the XAML namespace. However, when the XAML compiler fails to find the markup-extension class, it tries appending Extension to the name. There is a TypeExtension class, so the XAML compiler will use that, passing the string Button to its constructor. Then it will call the extension's ProvideValue method, to obtain the real value to be used for the property. The TypeExtension will return the Type object for the Button class.

2.3: WPF Application Class

Markup Extensions

- Markup extensions allow you to compactly configure objects and reference other objects defined elsewhere in the application
- Markup extensions are used while setting a property on an object, either via an XML attribute or the property element syntax
- Markup extensions can be used in nested tags or in XML attributes, which is more common

WPF Design Principles:
Markup Extensions:
For most properties, the XAML property syntax works perfectly well.
However, in some cases, it just is not possible to hard-code the property value.
For example,

> You may want to set a property value to an object that already exists.
> Alternatively, you may want to set a property value dynamically, by binding it to a property in another control.

In both these cases, you need to use a markup extension — specialized syntax.

2.3: WPF Application Class

Markup Extensions

- When they are used in attributes, markup extensions are always bracketed by curly braces { }

```
<Grid>
  <Grid.Resources>
      <SolidColorBrush x:Key="fooBrush" Color="Blue"></SolidColorBrush>
</Grid.Resources>
  <Button Background="{StaticResource fooBrush}"  Name="myButton" />
</Grid>
```

WPF Design Principles:
Markup Extensions (contd.):
Whenever an attribute value is enclosed in curly braces ({}), the XAML compiler / parser treats it as a markup extension value rather than a literal string (or something that needs to be type-converted).
StaticResource returns the value of the specified resource. Note that StaticResource does not need to be qualified with an x: prefix. This is because resource management is a WPF feature, rather than a generic XAML feature. Hence the resource markup extensions are in the WPF namespace.
In the given example, the use of StaticResource in this markup is effectively equivalent to the following code:

```
myButton.Background = (Brush)
myButton.FindResource("fooBrush");
```

This is a one-shot resource lookup. The property value will be set to the resource value during initialization and then never changed. If the value associated with the resource name changes, then the property will not be updated automatically.

2.3: WPF Application Class

Demo

- Demo on how to create a WPF Application and use Properties and Markup Extensions