WPF and MVVM Study Guides

1. Introduction to WPF

https://msdn.microsoft.com/en-us/library/mt149842.aspx

2. Walkthrough: My First WPF Desktop Application

https://msdn.microsoft.com/en-us/library/ms752299(v=vs.110).aspx

3. Creating a UI by using XAML Designer in Visual Studio

https://msdn.microsoft.com/en-us/library/hh921077.aspx

4. XAML Syntax in brief

https://msdn.microsoft.com/en-us/library/ms752059(v=vs.100).aspx#xaml_syntax_in_brief

MVVM and WPF

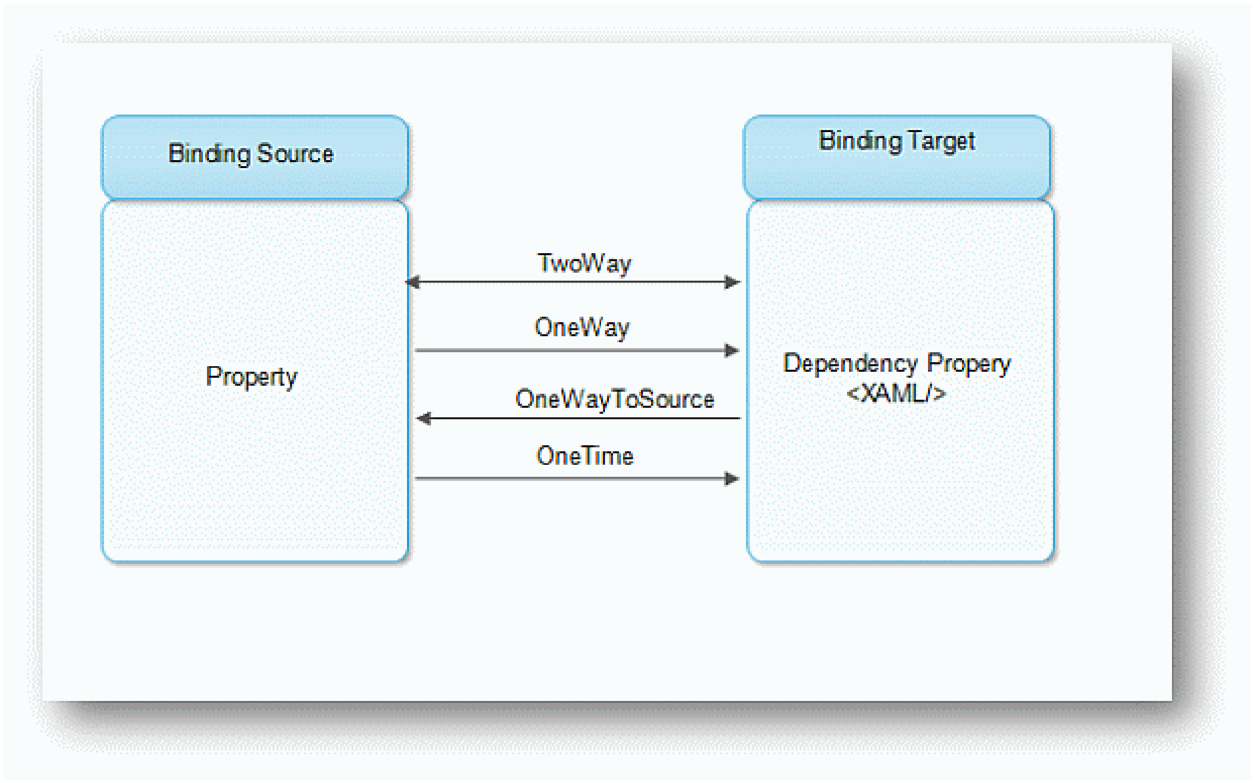
MVVM

- Model is a set of classes representing the data coming from the services or the database.
- View is the code corresponding to the visual representation of the data the way it is seen and interacted with by the user.
- ViewModel serves as the glue between the View and the Model. It wraps the data from the Model and makes it friendly for being presented and modified by the view. ViewModel also controls the View's interactions with the rest of the application (including any other Views).

Binding

Binding Modes in XAML(WPF,Silverlight,WP or Win8 App)

The Mode Property of Binding just changes the behaviors. The DataBinding mode defines the communication direction to the source or the direction of data flow from the source . In XAML (WPF, Silverlight, WP or Win8 App) there are five ways you can bind a data target object to a source.



The diagram above attempts to explain the databinding mode or communication way to communicate between the target to the source in XAML.

OneWay: Data moves only one direction, the source property automatically updates the target property but the source is not changed.

TwoWay: Data moves both directions, if you change it in the source or target it is automatically updated to the other.

OneWayToSource: Data moves from the target to the source changes to the target property to automatically update the source property but the target is not changed .

OneTime: Data is changed only one time and after that it is never set again, only the first time changes to the source property automatically update the target property but the source is not changed and subsequent changes do not affect the target property

View and ViewModels are connected thanks to binding. The ViewModel takes care of exposing the data to show in the View as properties, which will be connected to the controls that will display them using binding. Let's say, for example, that we have a page in the application that displays a list of products. The ViewModel will take care of retrieving this information (for example, from a local database) and store it into a specific property (like a collection of type **List<Order>**):

```
public List<Order> Orders { get; set; }
```

Let's say that your application has a page where it can create a new order and, consequently, it includes a TextBox control where to set the name of the product. This information needs to be handled by the

ViewModel, since it will take care of interacting with the model and adding the order to the database. In this case, we apply to the binding the Modeattribute and set it to TwoWay, so that everytime the user adds some text to the TextBox control, the connected property in the ViewModel will get the inserted value.

If, in the XAML, we have the following code, for example:

<TextBox Text="{Binding Path=ProductName, Mode=TwoWay}" />

It means that in the ViewModel we will have a property called ProductName, which will hold the text inserted by the user in the box.

with the MVVM pattern we connect properties in the ViewModel with controls in the UI using binding, like in the following sample:

```
<ListView ItemsSource="{Binding Path=Orders}" />
```

The Datacontext

You may be wondering how the View model is able to understand which is the ViewModel that populates its data. To understand it, we need to introduce the **DataContext**'s concept, which is a property offered by any XAML Control. The **DataContext**property defines the binding context: every time we set a class as a control's DataContext, we are able to access all of its public properties. Moreover, the DataContext is hierarchical: properties can be accessed not only by the control itself, but also all of the children controls will be able to access to them.

The core implementation of the MVVM pattern relies on this hierarachy: **the class that we create as ViewModel of a View is defined as DataContext of the entire page**. Consequently, every control we place in the XAML page will be able to access the ViewModel's properties and show or handle the various information. In an application developed with the MVVM pattern, usually, you end up having a page declaration like the following one:

```
<Page x:Class="Sample.MainPage"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      DataContext="{Binding Source={StaticResource MainViewModel}}"
      mc:Ignorable="d">
    <!-- page content goes here -->
</Page>
```

The DataContext property of the Page class has been connected to a new instance of theMainViewModel class.

The INotifyPropertyChanged interface

XAML offers the concept of dependency properties, which are special properties that can define a complex behavior and, under the hood, are able to send a notification to both sides of the binding channel every time its value changes. Most of the basic XAML controls use dependency properties (for example, the Text property of the TextBlock control is a dependency property). However, defining a new dependency property isn't very straightforward and, in most cases, it offers features which aren't needed for our MVVM scenario. Let's take the previous sample based on the ProductName property: we don't need to handle any special behavior or logic, we just need that every time the ProductName property changes, both sides of the binding channel receive a notification, so that the TextBlock control can update its visual layout to display the new value.

For these scenarios, XAML offers a specific interface called INotifyPropertyChanged, which we can implement in our ViewModels. This way, if we need to notify the UI when we change the value of a property, we don't need to create a complex dependency property, but we just need to implement this interface and invoke the related method every time the value of the property changes.

Here is how a property in a ViewModel looks:

```
private string _productName;
public string ProductName
{
    get { return _productName; }
    set
    {
        _productName = value;
        OnPropertyChanged();
    }
}
```

Now the property will work as expected; when we change its value, the **TextBlock** control in binding with it will change its appearance to display it.

Commands (or How to Handle Events in MVVM)

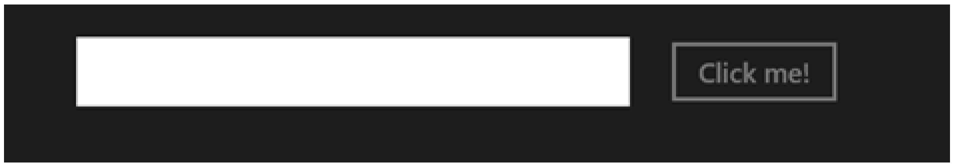
The XAML has introduced **commands**, which is a way to express a user interaction with a property instead of with an event handler. Since it's just a simple property, we can break the tight connection between the view and the event handler and also define it in an independent class, like a ViewModel.

The framework offers the **ICommand** interface to implement commands: with the standard approach, you end up having a separated class for each command. The following example shows how a command looks:

```

public class ClickCommand : ICommand
{
    public bool CanExecute(object parameter)
    {
    }
    public void Execute(object parameter)
    {
    }
    public event EventHandler CanExecuteChanged;
}
```

The **CanExecute()** method is one of the most interesting features provided by commands, since it can be used to handle the command's lifecycle when the app is running. For example, let's say that you have a page with a form to fill, with a button at the end of the page that the user has to press to send the form. Since all the fields are required, we want to disable the button until all the fields have been filled. If we handle the operation to send the form with a command, we are able to implement the **CanExecute()** method in a way that it will return **false** when there's at least one field still empty. This way, the **Button** control that we have linked to the command will automatically change his/her visual status: it will be disabled and the user will immediately understand that he won't be able to press it.



In the end, the command offers an event called CanExecuteChanged, which we can invoke inside the ViewModel every time the condition we want to monitor to handle the status of the command changes. For example, in the previous sample, we would call the CanExecuteChanged event every time the user fills one of the fields of the form.

Once we have defined a command, we can link it to the XAML thanks to the Command property, which is exposed by every control that are able to handle the interaction with the user (like Button, RadioButton, etc.)

```
<Button Content="Click me" Command="{Binding Path=ClickCommand}" />
```

As we're going to see in the next post, however, most of the toolkits and frameworks to implement the MVVM pattern offer an easier way to define a command, without forcing the developer to create a new class for each command of the application. For example, the popular MVVM Light toolkit offers a class called **RelayCommand**, which can be used to define a command in the following way:

```
private RelayCommand _sayHello;
public RelayCommand SayHello
{
    get
    {
        if (_sayHello == null)
        {
            _sayHello = new RelayCommand(() =>
            {
                Message = string.Format("Hello {0}", Name);
            }, () => !string.IsNullOrEmpty(Name));
        }
        return _sayHello;
    }
}
```

As you can see, we don't need to define a new class for each command, but by using anonymous methods, we can simply create a new RelayCommand object and pass, as parameters:

The code that we want to excecute when the command is invoked.
The code that evaluates if the command is enabled or not.

XAML Binding Basics

Referencing a control

Referencing a control in XAML is referencing a class. Declare it in XAML and the class' constructor is executed. Your control is now in the logical tree. Every control is part of a hierarchy, with the top-most control being the page element.

```
<Button></Button>
```

Adding content to a control

Not all XAML controls are content controls, but most are. Control controls allow you to put almost anything in them. That means you can put text in the content property, like this:

```
<Button>Hello World</Button>
```

The area between the button XAML tags defaults to its content property. So, you can put simple (like above) or complex (like below) content directly between the tags:

```
<Button>
    <Grid>
        <TextBlock>Hello World</TextBlock>
        <Button>Click me</Button>
    </Grid>
</Button>
```

In the code above, I place a grid inside my button and even another button. This is all possible because "content" in XAML allows for almost anything.

Setting a property

Not all properties are content properties. Like XML or HTML you can set property values simply by setting the attribute of the control, like this:

```
<Button Content="Hello World" />
```

In the code above, the content property interprets the string as a string. But some properties require integers, doubles, colors, and more. XAML automatically converts to these types for you.

Setting a complex property

Complex property values can't be jammed into an attribute string. When you have a complex value to set to a property, you do it like this:

```
<Button>
    <Button.Content>
        <Grid>
            <TextBlock>Hello World</TextBlock>
            <Button>Click me</Button>
        </Grid>
    </Button.Content>
</Button>
```

Binding a property

```
public class MyRecord
{
    public string ButtonText { get; set; }
}
```

Hard coding a property value may not be an option. For this reason, XAML supports binding. Say you have a class (above) that you set to the datacontext property of a control. Once set, that control and all of its children can bind to it – since properties propagate down through the hierarchy tree.

The three binding syntaxes (below) function identically:

```
<!-- one -->
<Button Content="{Binding Path=ButtonText}" />

<!-- two -->
<Button Content="{Binding ButtonText}" />

<!-- three -->
<Button>
    <Button.Content>
        <Binding Path="ButtonText" />
    </Button.Content>
</Button>
```

In the code above, the first (one) syntax explicitly calls the path binding property. This, however, is the default property so the second (two) syntax omits it. The third (three) syntax uses the expanded property syntax and sets the value with the fully qualified binding object. The three are identical in function and performance – only the syntax is different.

Binding with a converter

Sometimes the value you want to bind isn't the value you want to show. In the example above, I want to show the ButtonText value in all upper case. Since this is simply a UI requirement, we don't want to change class, we just want to convert the value being displayed.

```
public class ToUpperConverter: IValueConverter
{
    public object Convert(object value,
        Type targetType, object parameter, string language)
    {
        return (value as String).ToUpper();
    }
    public object ConvertBack(object value,
        Type targetType, object parameter, string language)
    {
        throw new NotImplementedException();
    }
}
```

The code above shows the implementation of a simple value converter for XAML binding. This converter takes the value and changes it to upper case. Since you write the converter, it means you can make it do whatever you want. Here's how you use them:

```
<Grid.Resources>
    <local:ToUpperConverter
        x:Name="ToUpperConverter" />
</Grid.Resources>

<Button Content="{Binding ButtonText,
    Converter={StaticResource ToUpperConverter}}" />
```

There are two parts to the code above. The first the resource. When I want to use a converter in my XAML, I need to reference my value converter class in the resources of any control higher in the control hierarchy, even the page. Then, I can references throughout my XAML.

The second part of the code shows the simple binding syntax we saw before. However, it adds the extra 'Converter' portion after the comma. This references the resource we added above the control by name. The syntax tells XAML to run the value through the converter before displaying it.

Using converter parameters

Converters can also be passed parameters from the bound control. This gives you even more control over the logic inside the converter. It is optional, but it is also a powerful option.The following syntaxes use a parameter – like above, these different techniques function identically:

```
<!-- one -->
<Button Content="{Binding ButtonText,
    Converter={StaticResource ToUpperConverter},
    ConverterParameter='SomeValue'}" />

<!-- two -->
<Button>
    <Button.Content>
        <Binding Path="ButtonText"
            Converter="{StaticResource ToUpperConverter}"
            ConverterParameter="SomeValue" />
    </Button.Content>
</Button>
```

Binding to an element

Sometimes the value of one control's property needs to match the value of another control's property. XAML allows this by referencing another control by name. The binding syntax continues to be the same, but allows you to enrich it more detail, like this:

```
<TextBox x:Name="MyTextBox"
         Text="Hello World" />

<Button x:Name="MyButton"
        Content="{Binding Text, ElementName=MyTextBox}" />
```

In the code above, this binding syntax sets the button's content to the textbox's text value. The complex interaction between controls is accomplished without any code behind. If you were to write the equivalent code behind it would look like this:

```
MyButton.Content = MyTextBox.Text;
```

Note: as the user changes the textbox, the content of the button is updated.

Binding modes

If you bind the property of any input control (a slider, a textbox, a radiobutton, et al) – basically anything the user interacts with – then the syntax would look something like this:

```
<CheckBox IsChecked="{Binding IsFavorite}" />
```

But this is a problem. This syntax only reads the value; it doesn't write the value back. Input controls typically want to write the value back. For this scenario, XAML provides binding modes: OneTime (most efficient), OneWay (default), and TwoWay (for input controls).

```
<CheckBox IsChecked="{Binding IsFavorite, Mode=TwoWay}" />
```

Inline invocation

As I mentioned above, whenever you declare a control in XAML you are instantiating a class. And, just like using a converter resource instantiates the converter class, you can instantiate any class you want to – this is very helpful with binding. Consider a class like this:

```
public class MyRecord
{
    public MyRecord()
    {
        ButtonText = "Hello World";
    }
    public string ButtonText { get; set; }
}
```

In the code above, the class has a constructor where its property value is set. But how do you use this in your XAML? Just like we made the converter a resource, we make our class a resources, like this:

```
<Grid.Resources>
    <local:MyRecord x:Name="MyRecord" />
</Grid.Resources>
```

Now we can reference this as a source for our binding. Source is slightly different. It is basically telling the XAML engine to ignore the datacontext of the control and look elsewhere instead. In this case, we will tell it to look at this MyRecord resource we just created, like this:

```
<TextBlock Text="{Binding ButtonText,
        Source={StaticResource MyRecord}}"  />
```

The datacontext

Referencing a source over and over is a lot of useless and redundant code. Instead, we want to set the datacontext of a control. The datacontext is the default source of a binding. What's nice is that we can set a control's datacontext or its parent or its parent's parent, and so on.

We can set it in code like this:

```
this.DataContext = new MyRecord();
```

We can also set it in XAML like this:

```
<Page.DataContext>
    <local:MyRecord />
</Page.DataContext>
```

Notice in the XAML above that we do not give a name to our MyRecord reference. This is because we will not be referencing it. Simply because we have set it as datacontext, it will flow down through the hierarchy. And, if it is not interrupted, our button can use it like this:

```
<TextBlock Text="{Binding ButtonText}"  />
```

Look familiar? This is the very first binding syntax we looked at up top. Once you set the datacontext you can set bind to it from any child element.

Deep binding

Our example is simple, but sometimes you want to bind to more than just the properties of a class. If a property is a complex type (a class) you might want to bind to the properties of that class. Is this kind "deep binding" possible? You bet. Look:

```
<TextBlock Text="{Binding ButtonText.Length}"  />
```

In the code above, we recognize that a string is actually a class. And the string class has a property of length. So, we bind to it with syntax very similar to how we access properties in C#. Let's pretend now that we bind to an array of strings, how would we access an item and its properties?

```
<TextBlock Text="{Binding ButtonText[0].Length}"  />
```

In the code above, I treat ButtonText like a string array (string[]) and access the first item (zero-based) and then a property of that first item. This is quite a complex operation if you think about it. But the XAML syntax keeps it simple, consistent, and compact.