WPF – Data binding and MVVM Commands

Windows Programming Course

Agenda

1. Dependency Properties
2. MVVM Pattern
3. Element/Data Binding
4. Commands

1.
Dependency Properties

The Dependency Property

Using dependency properties to plug into core WPF features such as animation, data binding, and styles.

Most of the properties that are exposed by WPF elements are dependency properties.

The purpose of dependency properties is to provide a way to compute the value of a property based on the value of other inputs.

The Advantages

Reduced memory footprint

- Dependency properties only stores modified properties in the instance. The default values are stored once within the dependency property.

Value inheritance

- The value is resolved by using a value resolution strategy. If no local value is set, the dependency property navigates up the logical tree until it finds a value.

Change notification

- Dependency properties have a built-in change notification mechanism. By registering a callback in the property metadata you get notified, when the value of the property has been changed. This is also used by the databinding.

Creating a Dependency Property

To create a DependencyProperty, add a static field of type DepdencyProperty to your type and call DependencyProperty.Register() to create an instance of a dependency property.

The name of the DependendyProperty must always end with **...Property**. This is a naming convention in WPF.

```
// Dependency Property
public static readonly DependencyProperty CurrentTimeProperty =
     DependencyProperty.Register( "CurrentTime", typeof(DateTime),
     typeof(MyClockControl), new FrameworkPropertyMetadata(DateTime.Now));
// .NET Property wrapper
public DateTime CurrentTime {
    get { return (DateTime)GetValue(CurrentTimeProperty); }
    set { SetValue(CurrentTimeProperty, value); }
}
```

Creating a Dependency Property – Value Changed Callback

Each DependencyProperty provides callbacks for change notification, value coercion and validation. These callbacks are registered on the dependency property.

```
new FrameworkPropertyMetadata(DateTime.Now,
                        OnCurrentTimePropertyChanged,
                        OnCoerceCurrentTimeProperty),
                        OnValidateCurrentTimeProperty);
```

Creating a Dependency Property – Value Changed Callback (cont.)

The change notification callback is a static method, that is called everytime when the value of the TimeProperty changes.

```
private static void OnCurrentTimePropertyChanged(DependencyObject
source, DependencyPropertyChangedEventArgs e)
{
    MyClockControl control = source as MyClockControl;
    DateTime time = (DateTime)e.NewValue;
    // Put some update logic here...
}
```

Creating a Dependency Property – Coerce Value Callback

The coerce callback allows you to adjust the value if its outside the boundaries without throwing an exception.

```
private static object OnCoerceTimeProperty(DependencyObject sender,
object data)
{
    if ((DateTime)data > DateTime.Now)
    {
        data = DateTime.Now;
    }
    return data;
}
```

Creating a Dependency Property – Validation Callback

In the validate callback you check if the set value is valid. If you return false, an ArgumentException will be thrown.

```
private static bool OnValidateTimeProperty(object data)
{
    return data is DateTime;
}
```

Attached Properties

Attached properties are a special kind of Dependency Property. They allow you to attach a value to an object that does not know anything about this value.

```
<Canvas>
    <Button Canvas.Top="20" Canvas.Left="20" Content="Click me!"/>
</Canvas>
```

Attached Properties – Creating

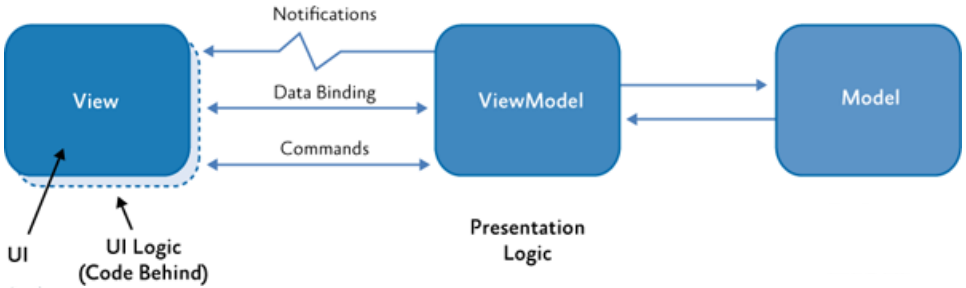
```
public static readonly DependencyProperty TopProperty =
    DependencyProperty.RegisterAttached("Top",
    typeof(double), typeof(Canvas),
    new FrameworkPropertyMetadata(0d,
        FrameworkPropertyMetadataOptions.Inherits));

public static void SetTop(UIElement element, double value) {
    element.SetValue(TopProperty, value);
}

public static double GetTop(UIElement element) {
    return (double)element.GetValue(TopProperty);
}
```

2.
MVVM Pattern

Model, View, View Model (MVVM Pattern)

It is a software architecture designed to organize and structure your code to write maintainable, testable and extensible applications.

MVVM pattern is ultimately the modern structure of the MVC pattern, so the main goal is still the same to provide a clear separation between domain logic and presentation layer.



MVVM – Model

The Model is the application's representation of the Data.

This can be either an object representation of your data, or a set of data calls (normally to a database or a web service) to retrieve your data.

Usually includes business rules about the data itself

- Required Fields
- Formatting Requirements
- Required associations to other data

No other logic is found in the Model. It is just Data.

MVVM – View

Also known as the “Presentation Layer”.

Displayed content visible to the end user.

Includes any logic about how to display any visual elements.

Views should be written under the assumption that any business logic is handled elsewhere.

MV**VM** – **V**iew **M**odel

ViewModel is responsible for exposing objects from the Model in such a way that the View can use these objects.

This is normally done through properties in the ViewModel that are then bound to in the View.

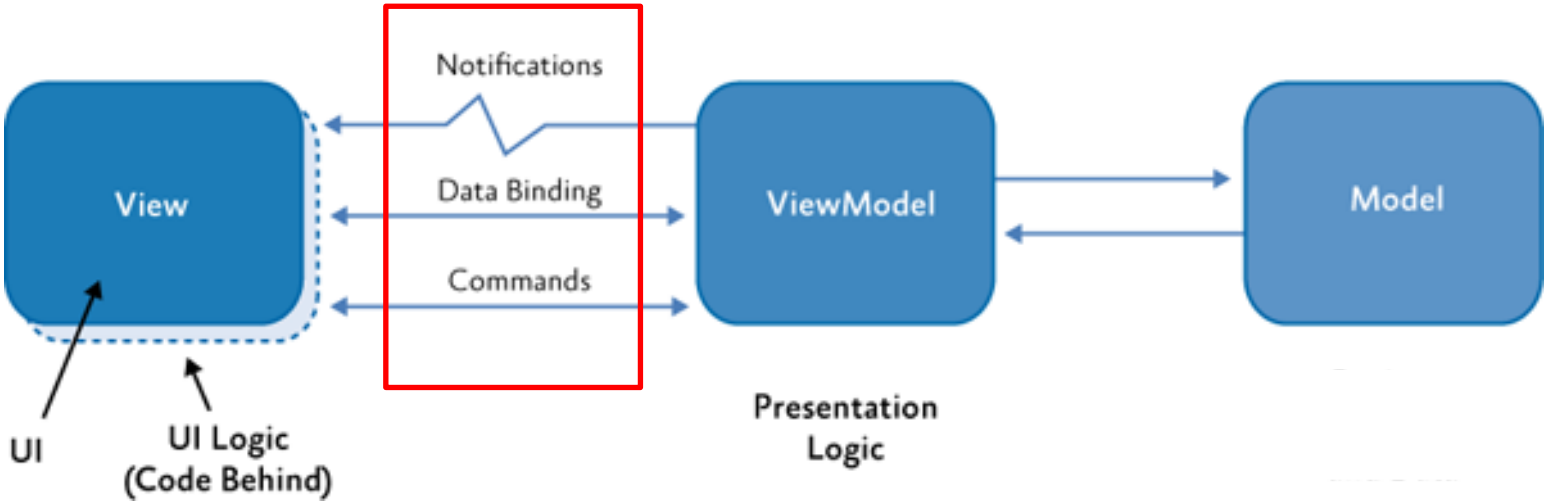
This is where the business logic of an application is found.

No longer need to write code like this:

```
TextBox1.Text = "New text from Database";
```

Can be viewed as the "State" of the data in the Model

Interaction



The Advantages

- Maintainability
  - A clean separation of different kinds of code should make it easier to go into one or several of those more granular and focused parts and make changes without worrying.
- Testability
- Extensibility
  - Having a better chance of making any of those parts more reusable.
  - It has also the ability to replace or add new pieces of code that do similar things into the right places in the architecture.

Hooking Up Views

View First Construction in XAML

```
<UserControl x:Class="MVVMDemo.Views.StudentView"
    xmlns:viewModel = "clr-namespace:MVVMDemo.ViewModel"
    mc:Ignorable = "d">
    <UserControl.DataContext>
        <viewModel:StudentViewModel/>
    </UserControl.DataContext>

    <Grid></Grid>
</UserControl>
```

Hooking Up Views (cont.)

View First Construction in Code-behind

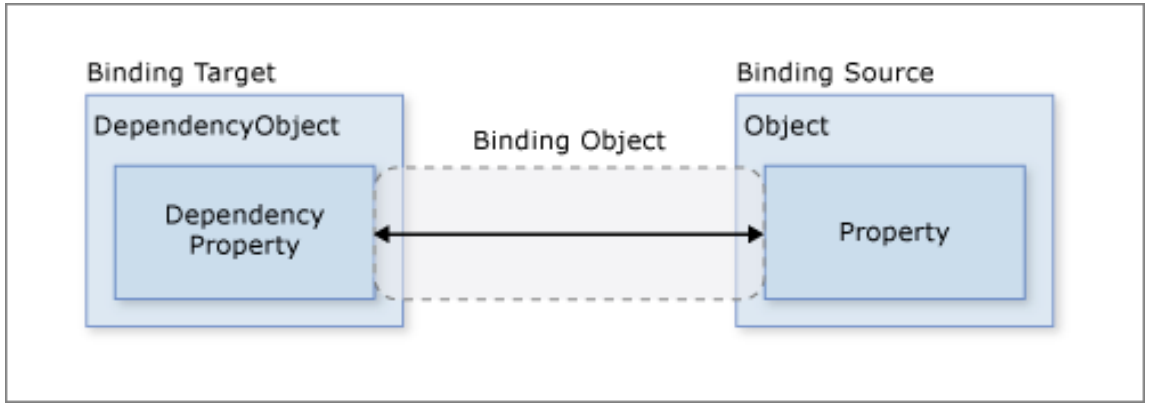
```
public partial class StudentView : UserControl {
  public StudentView() {
    InitializeComponent();
    this.DataContext = new StudentViewModel();
  }
}
```

2.
Element/Data Binding

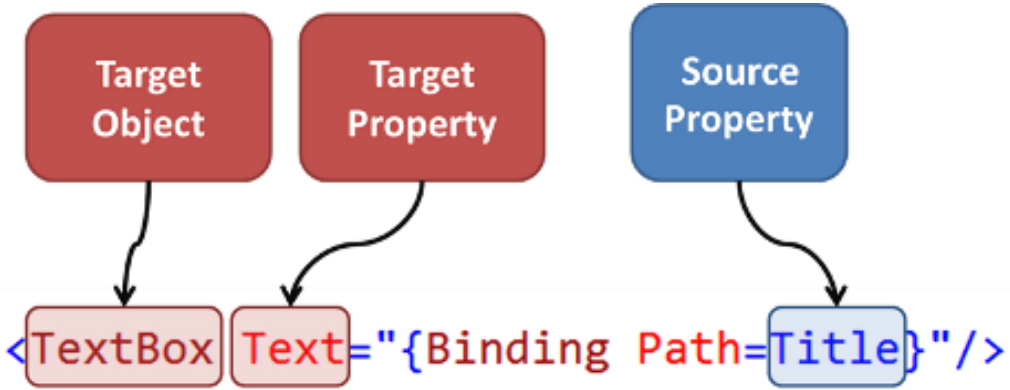
Element/Data Binding
Data Conversion
Validation

Element/Data Binding

Element/Data binding is the process that establishes a connection between the app UI and the data it displays.



Binding Expression



Element Binding Example

Element Binding: Source/Target properties is Dependency Properties of Dependency Objects (WPF Controls).

```
<Label Content="{Binding ElementName=txtInput, Path=Value}"></Label>
```

```
<TextBlock Text="Simple Text" Name="lblSampleText"
FontSize="{Binding ElementName=sliderFontSize, Path=Value}" >
</TextBlock>
```

Data Binding Example

Data Binding: Source properties is Properties of Objects (DataContext/Model).

```
<Label Content="{Binding Path=Value}"></Label>

<TextBox Text="{Binding Path=StudentName}"></TextBox >

<ComboBox ItemsSource="{Binding Path=Classes}"
          SelectedItem="{Binding Path=SelectedClass}"></ComboBox>
```

Change notification – INotifyPropertyChanged

In order to notify change to target property, the class (Model/ViewModel) containing the property should implement the **INotifyPropertyChanged** interface.

Content automatically is updated when using this interface.

Change notification – INotifyPropertyChanged (cont.)

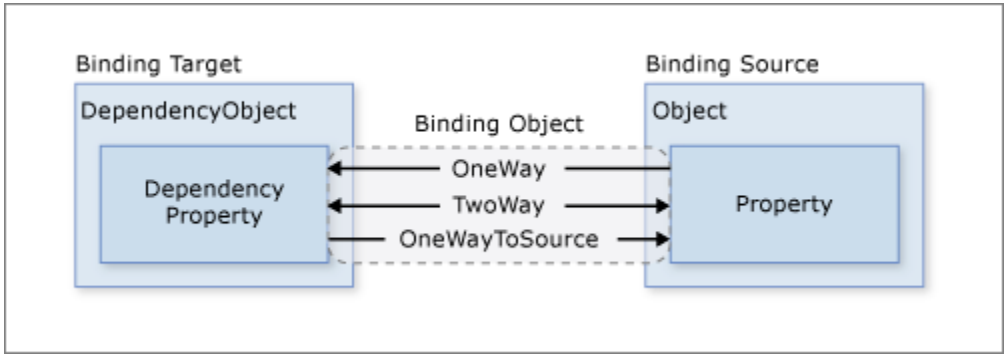
```
public class MainWindowViewModel : INotifyPropertyChanged {

  public event PropertyChangedEventHandler PropertyChanged;

  private void OnPropertyChanged(string propertyName)
  {
    if (PropertyChanged != null)
    {
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

Change notification – INotifyPropertyChanged (cont.)

```
public class MainWindowViewModel : INotifyPropertyChanged {
  private string m_hello;

  public string Hello {
    get => m_hello;
    set {
      m_hello = value;
      OnPropertyChanged("Hello"); // Replace "Hello" by nameof(Hello)
    }
  }
}
```

Binding Modes

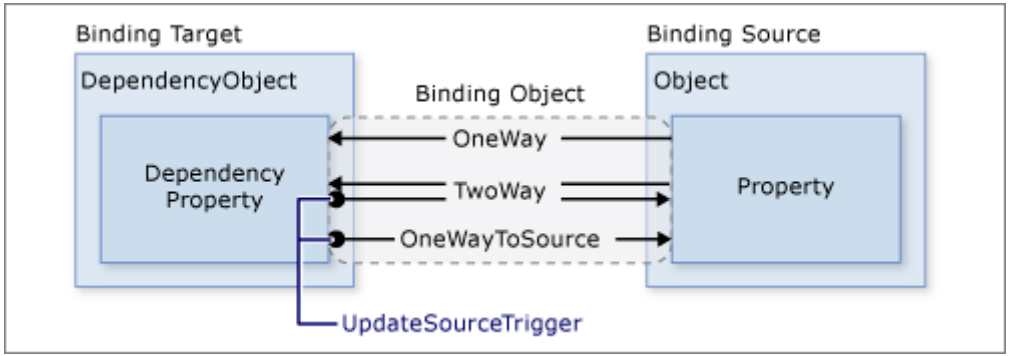
Name	Description
OneWay	The target property is updated when the source property changes.
TwoWay	It causes changes to either the source property or the target property to automatically update the other.
OneTime	The target property is set initially based on the source property value.
OneWayToSource	Similar to OneWay but in reverse. The source property is updated when the target property changes



Binding Updates

Name	Description
PropertyChanged	The source is updated immediately when the target property changes.
LostFocus	The source is updated when the target property changes and the target loses focus.
Explicit	The source is not updated unless you call the BindingExpression.UpdateSource() method.

```
<TextBox Text="{Binding ElementName=txtSampleText, Path=FontSize, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}" Name="txtFontSize"></TextBox>
```



Binding Errors

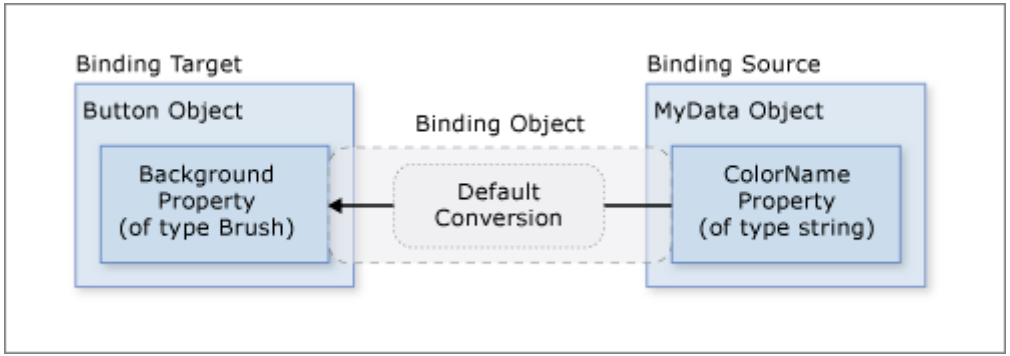
WPF doesn’t raise exceptions to notify you about data-binding problems.

WPF *does* output trace information that details binding failures in Visual Studio’s Output window, e.g.:

```
System.Windows.Data Error: 35 : BindingExpression path error:
'Tex' property not found on 'object' ''TextBox'
(Name='txtFontSize')'. BindingExpression:Path=Text;
DataItem='TextBox' (Name='txtFontSize'); target element is 'TextBox'
(Name='');

target property is 'Text' (type 'String')
```

Data conversion with IValueConverter

A Value Converter functions as a bridge between a target and a source and it is necessary when a target is bound with one source.



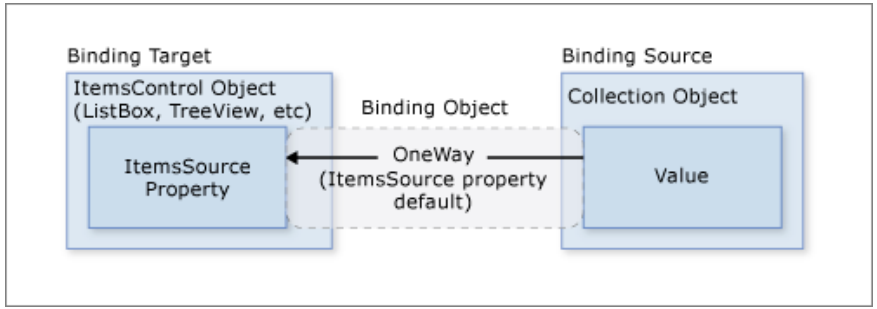
Data conversion with IValueConverter – Example

```
public class ValueConverter : IValueConverter  {
  public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture) {
    var text = value as String;
    return !string.IsNullOrEmpty(text) ? $"Hello {text}" : string.Empty;
  }

  public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture) {
    throw new NotImplementedException();
  }
}
```

Binding to collections

For Collection controls such as ItemsControl, ListBox, ListView, ComboBox…, the binding source should be a data collection (values or polymorphic objects).

Using **ObservableCollection<T>** class, which is a built-in implementation of a data collection that exposes the INotifyCollectionChanged interface



Binding to collections (cont.)

ViewModel

```
public ObservableCollection<string> Classes { get; set; }
      = new ObservableCollection<string>();
```

View

```
<ComboBox ItemsSource="{Binding Path=Classes}"
SelectedItem="{Binding Path=SelectedClass}"></ComboBox>
```

Data validation

There are several ways to validate data supported by WPF:

- Throwing exceptions on a property is set.
- Implementing the IDataErrorInfo interface.
- Implementing INotifyDataErrorInfo.
- Use WPF validation rules.

Data validation – IDataErrorInfo

```
public class Customer : IDataErrorInfo {
  public string Name { get; set; }
  public string Error {
    get { throw new NotImplementedException(); }
  }
  public string this[string columnName] {
    get {
      string result = string.Empty;
      if (columnName == "Name") {
        if (this.Name == "") result = "Name can not be empty";
      }
      return result;
    }
  }
}
```

Data validation – Show Error

Suppose we do have an error in text box, then what are all the things we want to notify to the user that something has gone wrong. (Basically tooltip, and changing background color of textbox).

```
<Style x:Key="TextErrorStyle" TargetType="{x:Type TextBox}">
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="True">
            <Setter Property="Background" Value="Red"/>
            <Setter Property="ToolTip" Value="{Binding RelativeSource={x:Static
RelativeSource.Self}, Path=(Validation.Errors)[0].ErrorContent}"></Setter>
        </Trigger>
    </Style.Triggers>
</Style>
```

Data validation – Show Error (cont.)

If suppose I want to draw a red color over the text box rather than changing background color.

```
<ControlTemplate x:Key="ErrorTemplate">
  <DockPanel LastChildFill="True">
    <Border BorderBrush="Red" BorderThickness="1">
      <AdornedElementPlaceholder />
    </Border>
  </DockPanel>
</ControlTemplate>

<TextBox Text="{Binding ValidateInputText, Mode=TwoWay}“
         Validation.ErrorTemplate="{StaticResource ErrorTemplate}"/>
```

3.
Commands

WPF Command Overview

Commanding is an input mechanism in Windows Presentation Foundation (WPF) which provides input handling at a more semantic level than device input.

Commands have several purposes:

- Separate the semantics and the object that invokes a command from the logic that executes the command.
- Indicate whether an action is available.

The ICommand Interface

The heart of the WPF command model is the `System.Windows.Input.ICommand` interface, which defines how commands work. This interface includes two methods and an event:

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

Sample RelayCommand

```
public class RelayCommand : ICommand {
  Action<object> execute;
  Func<object, bool> canExecute;

  public RelayCommand(Action<object> execute, Func<object, bool> canExecute =
null) {
    execute = execute;
    canExecute = canExecute;
  }
  public bool CanExecute(object parameter) {
    if (canExecute != null) {
      return canExecute(parameter);
    } else {
      return true;
    }
  }
```

Sample RelayCommand (cont.)

```
    public event EventHandler CanExecuteChanged {
      add {
        CommandManager.RequerySuggested += value;
      }
      remove {
        CommandManager.RequerySuggested -= value;
      }
    }
    public void Execute(object parameter) {
      execute(parameter);
    }
}
```

Use of ICommand

View

```
<StackPanel>
  <TextBox Text="{Binding MessageText}"/>
  <Button Command="{Binding ShowCommand}" />
</StackPanel>
```

ViewModel

```
private string messageText;
public string MessageText {
  get => messageText;
  set { messageText = value; OnPropertyChanges("MessageText");
}}
```

Use of ICommand (cont.)

ViewModel

```
private ICommand showCommand;
public ICommand ShowCommand {
  get {
    if (showCommand == null)
      showCommand = new RelayCommand(p => OnShow(),
                          () => !string.IsNullOrEmpty(MessageText));
      return showCommand;
    }
  }
private void OnShow() {
  MessageBox.Show("Hi... " + MessageText, "Message", MessageBoxButton.OK);
}
```

Thanks!

Any questions?

You can find me at:

tranminhphuoc@gmail.com