Mobile Computing

Xamarin
Data Binding
MVVM Pattern

Data Binding

- In simple apps
  - Get and set properties of controls is done explicitly in the code behind
    - entry1.Text = "Hello, world!";
- For complex or big apps
  - This code can become disorganized and error prone
  - XAML (or Page class constructors) allows us to link control properties with data classes in our app (known as data binding)
    - source class for data binding → is the ViewModel class
    - UI views can get their values from these classes
    - changing the data in the ViewModel can update the UI views
    - user input can update the data in the ViewModel classes

Target Views

- Views accepting bindings derive from
  - BindableObject class
    - Has a SetBinding() method to link some BindableProperty of the View to a source object and property (the ViewModel object)
    - Has a BindingContext property, where we can define the source object
- Sources can be described in a Binding object
  - Bindings allow automatic transfers between a source and a target (usually the target is a View and the source a ViewModel)
  - These transfers can be in one direction only or bidirectional
    - Mode: OneWay, OneWayToSource, TwoWay

XAML binding example

- Bindings between 2 Views can also be created
  - Example in XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="OpacityBindingXaml.OpacityBindingXamlPage"
             Padding="10, 0">
  <StackLayout>
    <Label Text="Opacity Binding Demo"
       FontSize="Large"
       VerticalOptions="CenterAndExpand"
       HorizontalOptions="Center"
       BindingContext="{x:Reference Name=slider}"
       Opacity="{Binding Path=Value}" />
    <Slider x:Name="slider"
       VerticalOptions="CenterAndExpand" />
  </StackLayout>
</ContentPage>
```

Source: The Slider Value property

Target: the Label View

The BindableProperty (on the target):
The Label Opacity property

Creating a Binding (code)

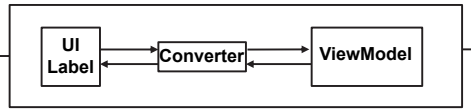
- The Binding class has a constructor specifying
  - path – the source property or path to a sub-property
  - mode – the direction of the binding
  - converter – an object implementing IValueConverter
    - methods Convert() and ConvertBack() to transform values and their types
  - convPar – an optional parameter to the converter
  - strFormat – to specify a string format if the target is a string
  - source – the source object (usually a ViewModel object)
- The source can also be specified as the
  - BindingContext property of the View
    - directly on the View or on a containing object (e.g. StackLayout, Grid or even Page)

Example

```
public class PersonViewModel … {
  public string Name { get; set; }
  public string Company { get; set; }
  …
}
```

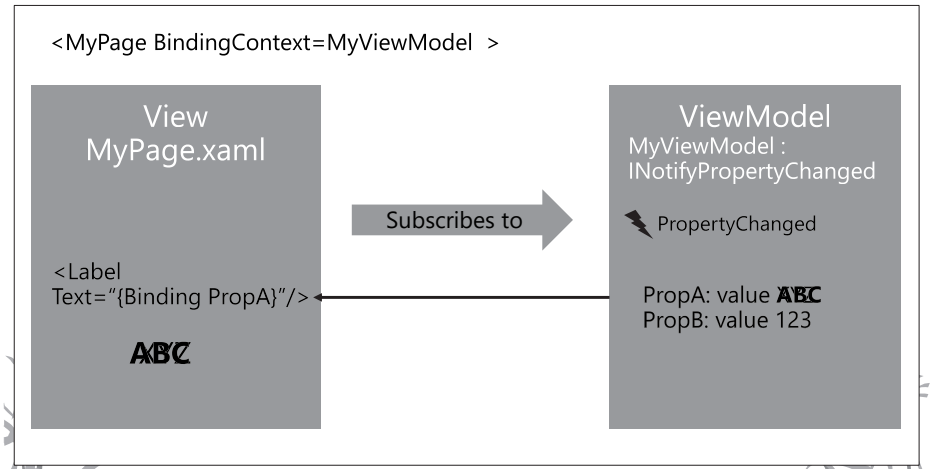
// needs more code to be a real ViewModel and a Binding source

```
public class ReverseConverter : IValueConverter {
  public object Convert (object value, Type targetType, object parameter, System.Globalization.CultureInfo culture) {
    var s = value as string;
    if (s == null)
      return value;
    return new string (s.Reverse ().ToArray ());
  }

  public object ConvertBack (object value, Type targetType, object parameter, System.Globalization.CultureInfo culture) {
    var s = value as string;
    if (s == null)
      return value;
    return new string (s.Reverse ().ToArray ());
  }
}
```




```
var label = new Label ();
PersonViewModel person;

label.BindingContext = person = new PersonViewModel { Name = "John Doe", Company= "Xamarin" };
label.SetBinding (Label.TextProperty, new Binding ("Name", mode: BindingMode.TwoWay, converter: new ReverseConverter ()));
// (label.Text) contains "eoD nhoJ".   ReverseConverter.Convert () is invoked in this case.
label.Text = "ooF";
// (person.Name) gets "Foo".  ReverseConverter.ConvertBack() is invoked in this case.
```


How data binding works



Implementing ViewModels

- Data objects bound as ViewModels must implement the interface INotifyPropertyChanged
  - It defines the PropertyChanged event

```
public class ItemViewModel : INotifyPropertyChanged
{
      public event PropertyChangedEventHandler PropertyChanged;
…
}
```

  - The object must fire the event whenever any object's property is changed
  - The name of the property, as a string, must be sent to the event handler
    - This is a frequent source of error, difficult to detect
  - The target object subscribes to this event

An example

```
public class DateTimeViewModel : INotifyPropertyChanged  {
   DateTime dateTime = DateTime.Now;
   public event PropertyChangedEventHandler PropertyChanged;

   public DateTimeViewModel() {
     Device.StartTimer(TimeSpan.FromMilliseconds(15), OnTimerTick);
   }
   bool OnTimerTick() {
     CurrentDateTime = DateTime.Now;
     return true;
   }

   public DateTime CurrentDateTime {
     private set {
       if (dateTime != value) {
         dateTime = value;        // Fire the event.
         PropertyChangedEventHandler handler = PropertyChanged;
         if (handler != null)
           handler(this, new PropertyChangedEventArgs("CurrentDateTime"));
       }
     }
     get {
       return dateTime;
     }
   }
}
```

The event should be fired only when the property is changed. Not when it is assigned (to prevent infinite two-way transfers).

The name of the changed property is included in the call to the event handler, as a string.
If there is a mistake the binding will not work, but the compiler does not detect the error.

Avoiding the name as string

Recent C# now have an attribute (or annotation) that can get the caller name as a string in a hidden parameter of the callee: it's the [CallerMemberName] attribute.

With that, the triggering of the event handler can be done in a method:

```
protected void OnPropertyChanged([CallerMemberName] string propertyName = null)  {
   PropertyChangedEventHandler handler = PropertyChanged;

   if (handler != null)
      handler(this, new PropertyChangedEventArgs(propertyName);
}
```

When we change a property of the ViewModel class, we just call the previous method

```
private double number = …;

public double Number {
   set {
      if (number != value) {
         number = value;
         OnPropertyChanged();
         …
      }
   }
   get { return number; }
}
```

The string "Number" is passed to the method.

Common code in a ViewModel class

The testing and triggering of the PropertyChanged event can even be put in a method:

```
protected bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null) {
   if (Object.Equals(storage, value))
      return false;
   storage = value;
   OnPropertyChanged(propertyName);
   return true;
}
```

And used in a property like the previous example:

```
private double number = …;

public double Number {
   set {
      if (SetProperty(ref number, value)) {
         // do something with the new value
      }
   }
   get { return number; }
}
```

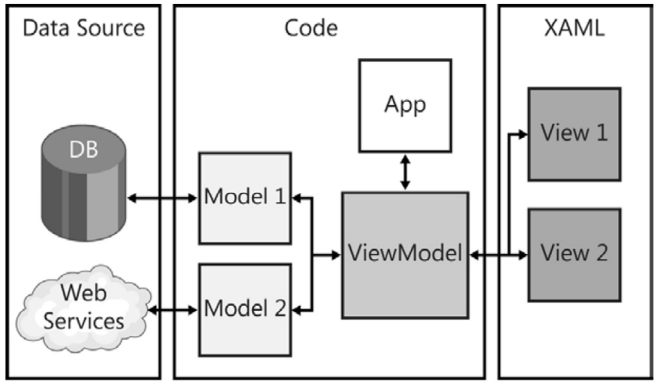
We can even write a base class:

```
public class ViewModelBase : INotifyPropertyChanged {
   public event PropertyChangedEventHandler PropertyChanged;

   protected bool SetProperty<T>(  ….
   protected void OnPropertyChanged( ….
}
```

If there is nothing to do in set (besides the event):

```
public int Exponent {
   set { SetProperty(ref exponent, value)) }
   get { return number; }
}
```


MVVM Pattern in Xamarin



Model – a class or classes that exposes the data of your application, either fetched from local data storage or externally such as a web service

ViewModel – a class or classes that has properties and methods that can be used to databind to a View

View – a class or classes that implement the presentation functionality of your application, displaying data and accepting user input. A View should contain no business logic, only logic pertaining to views such as that controlling the presentation. Views are bound to ViewModel classes.

Using the MVVM Pattern: MSDN http://msdn.microsoft.com/en-us/library/hh821028.aspx

Operation of MVVM

The ViewModel should contain the local data of the application and the related logic.
Persistent data and external service calls related to data should be in the Model block.
The ViewModel should be written independent of the View, and allow to change it without changing its code.

Data binding and command interface are powerful auxiliaries cleaning the border (separation) between View and ViewModel

The Command Interface

User interactions in the form of a click or tap gesture trigger an action as an event.
If those actions are not View related, but instead data related, they should be treated by the ViewModel, in replacement of the code behind of the View (or in addition).

Some Views have this kind of event, and allow the code to handle them in the ViewModel, using two of View properties: Command of type ICommand, and CommandParameter of type object.

Implementing classes: Button, MenuItem, ToolbarItem, SearchBar, TextCell, ImageCell, ListView, TapGestureRecognizer

When this interface event is triggered, the Execute() method of the ICommand object (in the ViewModel) bound to Command, is called, passing it the parameter assigned to CommandParameter.
Also, when we do the Command binding, and whenever the event of ICommand, CanExecuteChanged, is triggered, the View calls CanExecute() of the ICommand, and disables itself if it returns false.

The interface ICommand:

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


Implementing the Command Interface

The Xamarin framework include the classes Command and Command<T>, where T is the actual type of Execute() and CanExecute() parameter, in the implementation of ICommand.
We can use these classes in our ViewModel classes and bind them to the Views

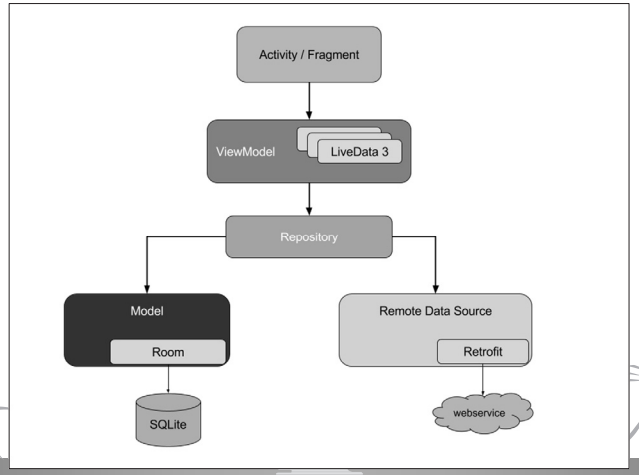
ViewModel:

```
public class MyViewModel: ViewModelBase {
   double exponent;
   ...
   public MyViewModel() {                    // constructor
      ...
      Exponent = 0;
      IncreaseExponentCommand = new Command(()=>{ Exponent += 1;});
   }

   public double Exponent {                  // data property
      private set { SetProperty(ref exponent, value); }
      get { return exponent; }
   }
                                             // command property
   public ICommand IncreaseExponentCommand { private set; get; }
   ...
}
```

```
MyViewModel myVM = new MyViewModel();
...
Button b = new Button() {
   ...
};
b.SetBinding(Button.CommandProperty,
    new Binding("IncreaseExponentCommand"));

Slider sl = new Slider(-5.0, 5.0, 0.0)  {
   ...
};
sl.SetBinding(Slider.ValueProperty,
    new Binding("Exponent"));

StackLayout stack = new StackLayout() {
   ...
   BindingContext = myVM,
   Children = { ..., b, sl, ,,, }
};
...
```


Android MVVM and Data Binding

- Recently Google introduced MVVM support in Android
  - Using the Architecture Components of the Jetpack support library



Showing Collections in Xamarin

The most versatile View for collections in Xamarin.Forms is the ListView.
It supports data binding, templates for each item, and also data binding for each displayed item.
The general data binding refreshes the display whenever a new item is added or one is removed. The displayed properties of each item can also be automatically refreshed.
Pickers can display a small number of items for choosing one, but that set is not bindable.
TableViews are also versatile and can be used for data items, forms, menus and toolbars.

ListView specify its collection of items in the ItemsSource property, and the item template in the ItemTemplate property.

The ItemsSource property can be set to any collection class that implements IEnumerable
- There are many in the framework, e.g. List or List<T>
- But for updating the UI with add/remove operations on the collection, the type must be an ObservableCollection<T> (also on the framework and implements the event INotifyCollectionChanged)
- Items inside the collection need to implement INotifyPropertyChanged in order to allow an automatic UI item updating

Collection and Template Example

Defining a template for a ListView

```
ItemTemplate = new DataTemplate( () => {
    // Create views with bindings for displaying each property.
    Label nameLabel = new Label();
    nameLabel.SetBinding(Label.TextProperty, "Name");
    Label birthdayLabel = new Label();
    birthdayLabel.SetBinding(Label.TextProperty,
            new Binding("Birthday", BindingMode.OneWay, null, null, "Born {0:d}"));
    BoxView boxView = new BoxView();
    boxView.SetBinding(BoxView.ColorProperty, "FavoriteColor");
    // Return an assembled ViewCell.
    return new ViewCell {
        View = new StackLayout {
            Padding = new Thickness(0, 5),
            Orientation = StackOrientation.Horizontal,
            Children = { boxView, new StackLayout {
                VerticalOptions = LayoutOptions.Center,
                Spacing = 0,
                Children = { nameLabel, birthdayLabel }
            } }
        }
    };
} );
```

The ViewModel and the collection

```
class PersonVM : ViewModelBase {
  public PersonVM(string name, DateTime birthday, Color favoriteColor) {
    ...
  }

  public string Name { ...
  public DateTime Birthday { ...
  public Color FavoriteColor { ...
};
```

```
ObservableCollection<PersonVM> people = new ObservableCollection<PersonVM> {
  new PersonVM("Abigail", new DateTime(1975, 1, 15), Color.Aqua),
  ...
}
```