



XAM320

# Model-View-ViewModel in Xamarin.Forms

Download class materials from  
[university.xamarin.com](http://university.xamarin.com)



**Xamarin** University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft or Xamarin may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft or Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

© 2014-2017 Xamarin Inc., Microsoft. All rights reserved.

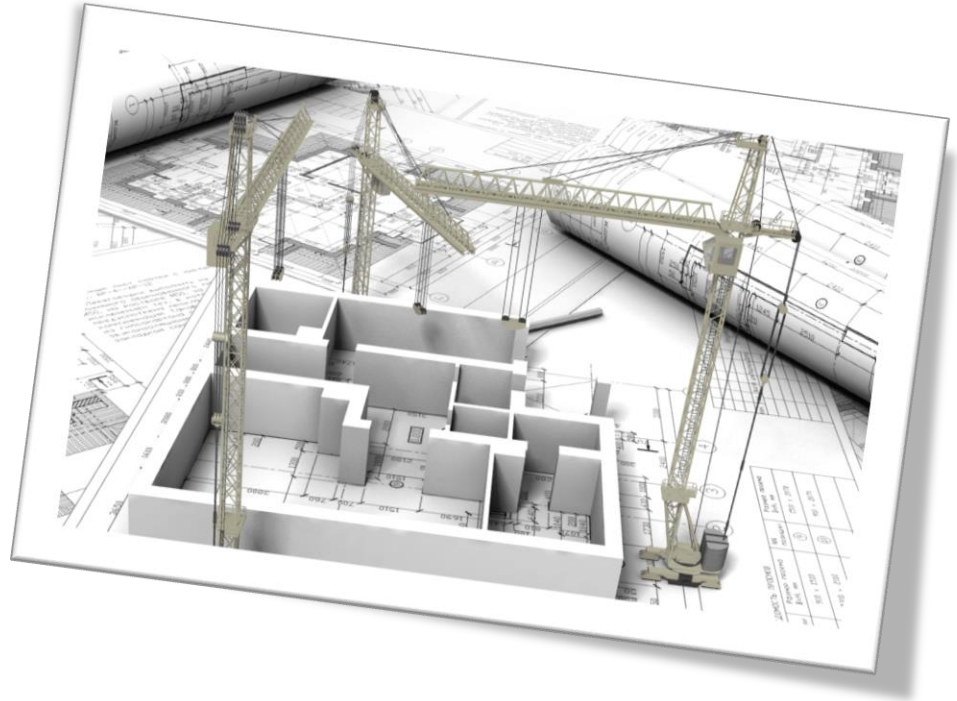
Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, Xamarin Studio, and Visual Studio are either registered trademarks or trademarks of Microsoft in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.



# Objectives

1. Model-View-ViewModel
2. Defining Visual Behavior
3. Commanding
4. Testing MVVM based apps



# Model-View-ViewModel

# Tasks

1. What is MVVM?
2. Define the Model and the View
3. Describe the ViewModel



# Separated Presentation

- ❖ Key to maximum code sharing is to *separate the presentation and domain layers*, this is referred to as the **Separated Presentation Pattern**

*"Ensure that any code that manipulates presentation only manipulates presentation, pushing all domain and data source logic into clearly separated areas of the program."* [1]

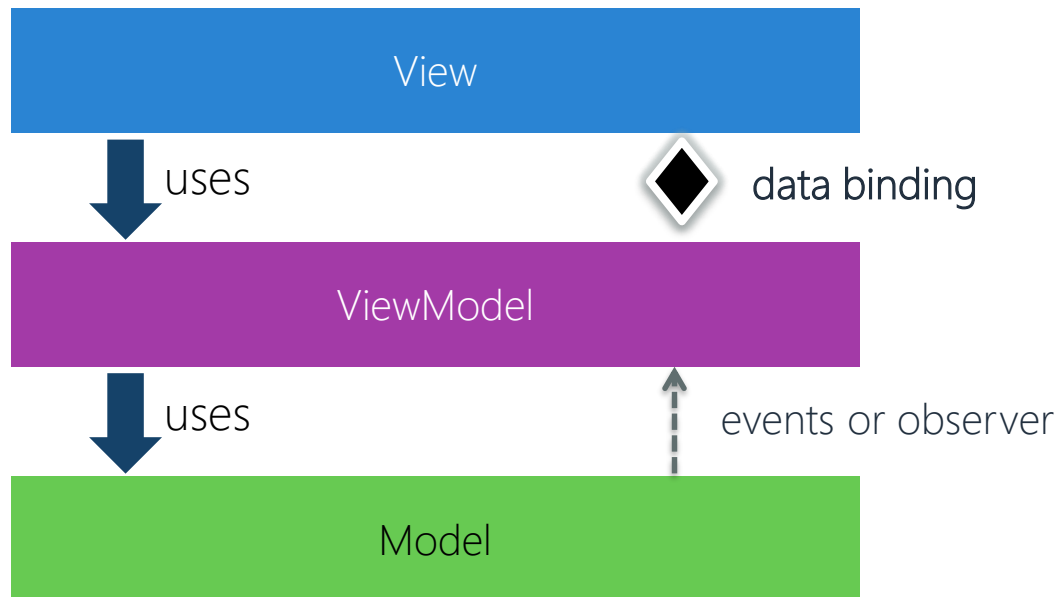
- Martin Fowler



Common examples of Separated Presentation Patterns include Model-View-Controller (MVC) and Model-View-Presenter (MVP), patterns you've likely used many times

# Model-View-ViewModel (MVVM)

- ❖ MVVM is a layered, **separated presentation pattern** made popular by XAML based UI where a data binding engine takes the place of the controller / presenter





# What is the Model?

- ❖ Models **manage the application data** and may include any combination of domain logic, persisted state and validation, not necessarily in one object

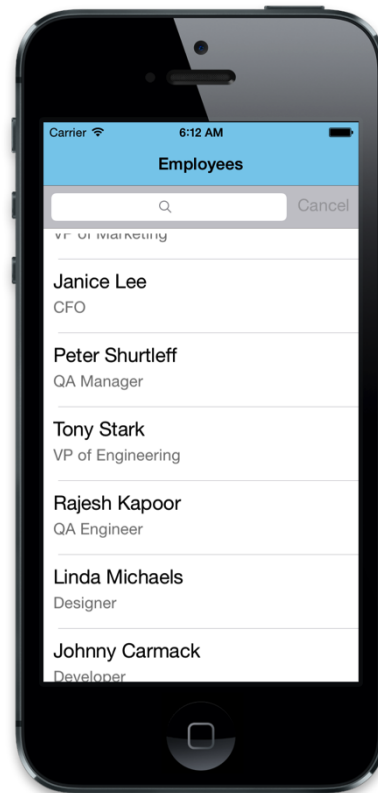
Models are intended to be shared across platforms and should not depend on platform-specific features

```
public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Title { get; set; }
    public DateTime HireDate { get; set; }
    public int Supervisor { get; set; }
    public static Employee GetById(int id);
    public static void UpdateRecord(Employee employee);
}
```



# What is the View?

- ❖ View presents the information to the user in a platform-specific fashion
- ❖ Should not contain code you want to unit test
- ❖ Everything *visual* should be managed here – fonts, colors, etc.



# What is the ViewModel?

- ❖ The ViewModel provides a **view-centric representation** of the data to display

exposes  
bindable  
properties and  
implements  
property change  
notification

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    private Employee model;

    public string Name {
        get { return model.Name; }
        set { model.Name = value; OnPropertyChanged("Name"); }
    }

    public EmployeeViewModel(Employee model) {
        model = model;
    }
    ...
}
```

often has a 1:1  
relationship with  
model

# What is the ViewModel?

- ❖ ... enables easier conversion / coercion of methods or model properties

```
partial class EmployeeViewModel
{
    ...
    public string DateHiredText {
        get { return model.HireDate.ToString("MMM d, yyyy"); }
    }

    public EmployeeViewModel Supervisor {
        get { return new EmployeeViewModel(
            Employee.GetById(this.supervisor)); }
    }
}
```

# What is the ViewModel?

❖ ... provides bindable way to access related data

```
partial class EmployeeViewModel
{
    ...
    public IEnumerable<string> ActiveProjects {
        get {
            return CompanyProjects.All
                .Where(p => p.Owner == model.Id
                    && p.IsActive)
                .Select(p => p.Name).ToList();
        }
    }
}
```

# What is the ViewModel?

- ❖ ... and provides a convenient place to put inconvenient logic for the UI
  - Perform input validation prior to storing it in the model
  - Perform visual calculations or runtime status values for the UI

```
partial class DownloaderViewModel {  
    private int percentComplete;  
    public int PercentComplete {  
        get { return percentComplete; }  
        set {  
            if (percentComplete != value) {  
                percentComplete = value;  
                OnPropertyChanged("PercentComplete");  
            }  
        }  
    }  
}
```

Can use property to drive a UI progress report

# Individual Exercise

Defining a ViewModel



**Xamarin**  
University

# Creating ViewModels

- ❖ Apps often have multiple view models – one for each "data-bindable" entity being displayed
- ❖ Views and ViewModels often have a 1:1 relationship, but VMs can be shared across views to provide UI synchronization

**MainViewModel**

**EmployeeViewModel**

**EmployeeViewModel**

**EmployeeViewModel**

**EmployeeViewModel**

**MainViewModel** might expose collection of **EmployeeViewModel** objects to bind to a **ListView**



# Connecting a View and ViewModel

- ❖ Main ViewModel is most often set as the **BindingContext** for the view in code behind, but can also be done in XAML if preferred

```
public partial class MainPage : ContentPage
{
    readonly MainViewModel viewModel = new MainViewModel();
    public MainPage ()
    {
        BindingContext = viewModel;
        InitializeComponent ();
    }
    ...
}
```

# MVVM Pros and Cons

- ❖ MVVM is well suited for platforms with a data binding infrastructure such as Xamarin.Forms and is the preferred architecture for non-trivial apps

Pros	Cons
<ul style="list-style-type: none"><li>■ Provides higher testable surface</li><li>■ Centralizes the visual &amp; business logic</li><li>■ Can reduce converter code used to tie models to UI</li><li>■ Takes advantage of binding infrastructure</li></ul>	<ul style="list-style-type: none"><li>■ Requires infrastructure, more for some platforms than others</li><li>■ Necessitates multiple layers which may not be worth it for smaller apps</li><li>■ Bindings can be hard to debug and may not be efficient for large data sets</li></ul>

# Flash Quiz

# Flash Quiz

- ① When using MVVM, the ViewModel should be platform-specific and created for each specific platform you want to support
- a) True
  - b) False

# Flash Quiz

- ① When using MVVM, the ViewModel should be platform-specific and created for each specific platform you want to support
- a) True
  - b) False

# Flash Quiz

- ② What are the members of the `INotifyPropertyChanged` interface?
- a) `PropertyChanged` event and `OnPropertyChanged` method
  - b) `OnPropertyChanged` method
  - c) `PropertyChanged` event

# Flash Quiz

- ② What are the members of the `INotifyPropertyChanged` interface?
- a) `PropertyChanged` event and `OnPropertyChanged` method
  - b) `OnPropertyChanged` method
  - c) `PropertyChanged` event



# Flash Quiz

- ③ Some of the disadvantages to MVVM are: (pick all that apply)
- a) Requires additional infrastructure
  - b) Reduces the testability of the logic
  - c) Can end up duplicating property definitions between model and VM
  - d) It can only be used with XAML

# Flash Quiz

- ③ Some of the disadvantages to MVVM are: (pick all that apply)
- a) Requires additional infrastructure
  - b) Reduces the testability of the logic
  - c) Can end up duplicating property definitions between model and VM
  - d) It can only be used with XAML



# Individual Exercise

Creating ViewModel Collections



**Xamarin**  
University

# Summary

1. What is MVVM?
2. Define the Model and the View
3. Describe the ViewModel





# Defining Visual Behavior



**Xamarin**  
University

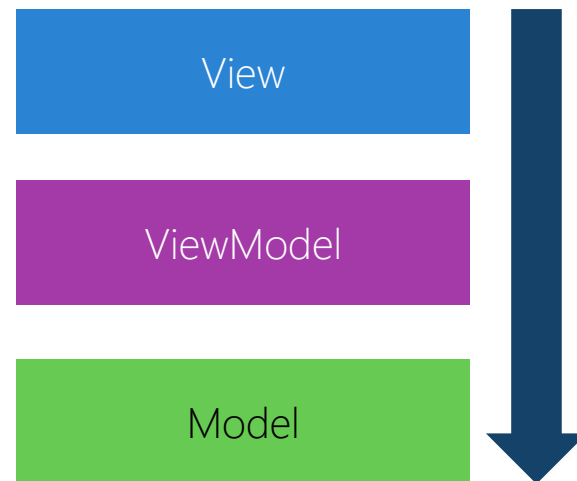
# Tasks

1. Compare the View vs. ViewModel
2. Control and activate events with selection
3. Utilize properties to define Visual Behavior
4. Employ Data Triggers
5. Other services



# View vs. ViewModel

- ❖ ViewModel is intentionally tied to the View, but should be written to be **UI-agnostic**
  - therefore, it should *not* have dependencies on anything in Xamarin.Forms



Each layer should only have direct knowledge about the layer below it



# Dealing with Selection

- ❖ Managing selection with MVVM provides a clean way to control and activate elements without dealing with events

```
<ListView ItemsSource="{Binding Employees}"  
          SelectedItem="{Binding SelectedEmployee, Mode=TwoWay}" />
```




Make sure to mark it *two-way* so ViewModel is notified when selection is altered by the UI

# Dealing with Selection

- ❖ Managing selection with MVVM provides a clean way to control and activate elements without dealing with events

```
public partial class MainViewModel : BaseViewModel
{
    ...
    private EmployeeViewModel selectedEmp;
    public EmployeeViewModel SelectedEmployee {
        get { return selectedEmp; }
        set { selectedEmp = value; RaisePropertyChanged("SelectedEmployee"); }
    }

    public MainViewModel() {
        SelectedEmployee = Employ
    }
}
```



Setter called when selection is changed

# Dealing with Selection

- ❖ Managing selection with MVVM provides a clean way to control and activate elements without dealing with events

```
public partial class MainViewModel : BaseViewModel
{
    ...
    private EmployeeViewModel selectedEmp;
    public EmployeeViewModel SelectedEmployee
    {
        get { return selectedEmp; }
        set { selectedEmp = value; RaisePropertyChanged(); }
    }

    public MainViewModel() {
        SelectedEmployee = Employees.FirstOrDefault();
    }
}
```

When UI supports "selection" vs. activation, view model can default or change selection based on runtime decisions, all in a unit-testable way

# Working with visual properties

- ❖ Assume a business requirement is to change the color of the employee's name in the UI if they are a supervisor

```
partial class EmployeeViewModel
{
    public Color NameColor { get; }
}
```

Avoid this! **Color** is a  
Xamarin.Forms specific type



... this is better but still not ideal –  
colors should be determined by the  
designer role and view code

```
partial class EmployeeViewModel
{
    public string NameColor { get; }
}
```

What we *really* want to do here is to have our UI change based on state properties such as **bool** or enumerations – we could do this with bindings and value converters

# Working with visual properties

- ❖ Assume a business requirement is to change the color of the employee's name in the UI if they are a supervisor

```
partial class EmployeeView
{
    public Color TitleColor { get; set; }
}
```

```
partial class EmployeeViewModel
{
    public bool IsSupervisor {
        get { ... }
        private set { ... }
    }
}
```

is a  
specific type

Let's expose a boolean property indicating whether the employee has subordinates ...

... this is better but still not ideal – colors should be determined by the designer role and view code

```
public string TitleColor { get; }
}
```

# Working with visual properties

- ❖ Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
  <Label.Triggers>
    <DataTrigger TargetType="Label"
                  Binding="{Binding IsSupervisor}"
                  Value="True">
      <Setter Property="TextColor" Value="Blue" />
    </DataTrigger>
  </Label.Triggers>
</Label>
```

# Visual Behavior through properties

- ❖ Data Triggers support dynamic UI property changes based on bindings with conditional tests

```

<Label Text="{Binding Name}" TextColor="Gray">
  <Label Text="Supervisor" />
  <DataTrigger Binding="{Binding IsSupervisor}"
    <Setter Property="TextColor" Value="Blue" />
  </DataTrigger>
</Label.Triggers>
</Label>

```

Assign default value – this is used when no trigger is matched



# Visual Behavior through properties

- ❖ Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
  <Label.Triggers>
    <DataTrigger TargetType="Label"
      Binding="{Binding IsSupervisor}"
      Value="True">
      <Setter Property="TextColor" Value="Blue" />
    </DataTrigger>
  </Label.Triggers>
</Label>
```

Can have zero or more *triggers* in the *triggers collection* exposed by the *Triggers* property

# Visual Behavior through properties

- ❖ Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
  <Label.Triggers>
    <DataTrigger TargetType="Label"
                  Binding="{Binding IsSupervisor}"
                  Value="True">
                        /="TextColor" Value="Blue" />
  </Label.Triggers>
</Label>
```

**DataTrigger** is used to change visual properties of an **Element** based on data binding

# Visual Behavior through properties

- ❖ Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
  <Label.Triggers>
    <DataTrigger TargetType="Label"
      Binding="{Binding IsSupervisor}"
      Value="True">
      <Setter Property="TextColor" Value="Blue" />
    </DataTrigger>
  </Label.Triggers>
</Label>
```

**Binding** property identifies the ViewModel property the Data Trigger is watching

# Visual Behavior through properties

- ❖ Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
  <Label.Triggers>
    <DataTrigger TargetType="Label"
      Binding="{Binding IsSupervisor}"
      Value="True">
      <Setter Property="TextColor" Value="Blue" />
    </DataTrigger>
  </Label.Triggers>
</Label>
```


... and a comparison test for that binding; e.g. when  
`IsSupervisor = true`

# Visual Behavior through properties

- ❖ Data Triggers support dynamic UI property changes based on bindings with conditional tests

Has one or more **setters** to change properties when the trigger condition is matched

```
<Label Text="{Binding Name}" TextColor="Red">
  <Label.Triggers>
    <DataTrigger TargetType="Label"
      Binding="{Binding IsSupervisor}"
      Value="True">
      <Setter Property="TextColor" Value="Blue" />
    </DataTrigger>
  </Label.Triggers>
</Label>
```



This is completely dynamic and is driven completely through the binding engine – so if the property changes at runtime, the trigger is re-evaluated and applied or removed!

# Value Converters

- ❖ Value Converters allow for *type mismatch* conversions – e.g. when the data does not match the UI requirements
- ❖ This conversion task is often taken up by the VM instead – reducing the need for value converters
- ❖ Still useful to have more primitive converters for bindings

**BooleanToColorConverter**

**ArrayToStringConverter**

**DoubleToIntegerConverter**

**NotBooleanConverter**

**IntegerToBooleanConverter**

# MVVM + other patterns

- ❖ MVVM is not the only design pattern needed, often need to utilize **other patterns** to provide necessary features through **abstractions**

Dependency  
Injection

Factory and  
Singleton

Command

Navigation

Alerts +  
Prompts

Messages

# Managing navigation

- ❖ Screen navigation can be handled in different ways – easiest is just to have an app-specific service that *knows* the screens which the VM uses

```
public enum AppScreen { Main, Detail, Edit, ... }

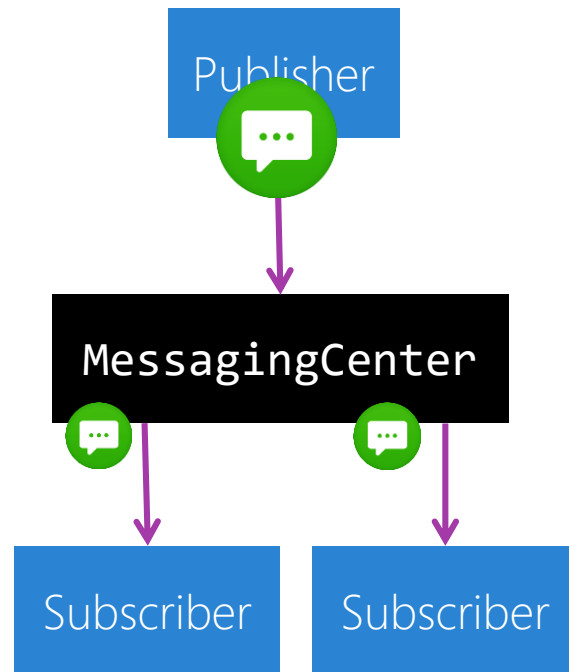
public class NavigationManager
{
    public Task<bool> GotoScreen(AppScreen screen) {...}
    public Task<bool> GoBack() { ... }
}
```

Enum defines the screens, and the class implements the navigation using the known app structure – master / detail, **NavigationPage**, etc.



# Loosely-coupled messages

- ❖ Another common requirement is communication between unrelated app components in a loosely-coupled fashion
  - VM to VM
  - service to VM
- ❖ This is easily solved with the built-in **MessagingCenter**



# Publishing a message

- ❖ Publisher passes message key and optional parameter

Publisher identifies sending type and parameter type through generic parameters



```
MessagingCenter.Send<MainViewModel, ItemViewModel>(
    this, "Select", selectedItem);
```

# Subscribing to a message

- ❖ Subscribers identify the message by the sender type and message key and provide a delegate callback to run when message is received

```
MessagingCenter.Subscribe<MainViewModel, ItemViewModel> (  
    this, "Select",  
    (mainVM, selectedItem) => {  
        // Action to run when "Select" is received  
        // from MainViewModel  
    });
```

Combination of the **sender type**, **string message**, and **parameter type** is the key for the message recipient – these must match between publisher and subscriber



# Individual Exercise

Driving behavior through properties



**Xamarin**  
University

# Summary

1. Compare the View vs. ViewModel
2. Control and activate events with selection
3. Utilize properties to define Visual Behavior
4. Employ Data Triggers
5. Other Services



# Commanding

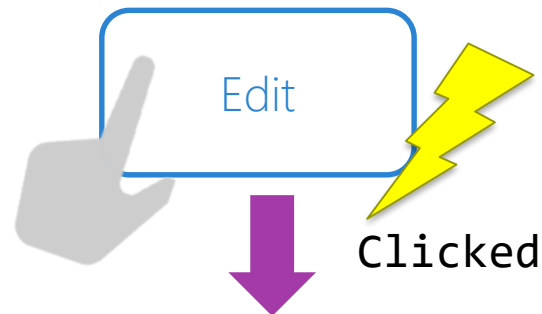
# Tasks

1. Identify the need for Commands
2. Illustrate the ICommand interface
3. Generalizing the command



# Event Handling

- ❖ UI raises events to notify code about user activity
  - **Clicked**
  - **ItemSelected**
  - ...
- ❖ The downside is that these events **must be handled** in the code behind file



```
public MainPage()
{
    ...
    Button editButton = ...;
    editButton.Clicked += OnClick;
}

void OnClick (object sender, EventArgs e)
{
    ...
}
```

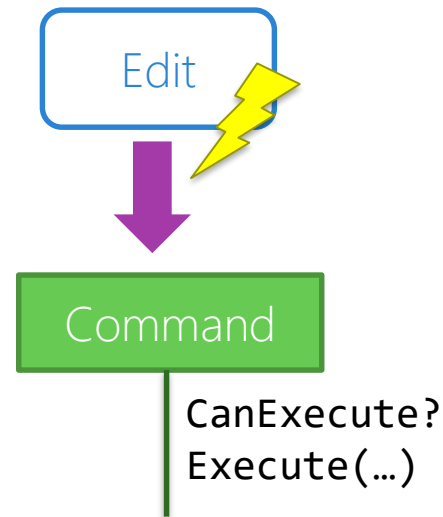


# Commands

- ❖ Microsoft defined the **ICommand** interface to provide a commanding abstraction for their XAML frameworks

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

Can provide an optional parameter (often **null**) for the command to work with for context



# Commands in Xamarin.Forms

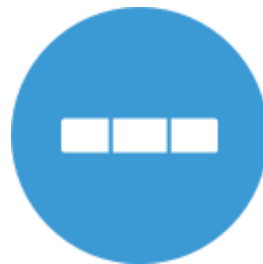
- ❖ A few Xamarin.Forms controls expose a **Command** property for the main action of a control



Button



Menu



ToolbarItem



TextCell

# Commands in Xamarin.Forms

- ❖ A few Xamarin.Forms controls expose a **Command** property for the main action of a control

```
public ICommand GiveBonus { get; }
```

```
<Button Text="Give Bonus"  
        Command="{Binding GiveBonus}" />
```



Can data bind a property of type **ICommand** to the **Command** property

# Gesture-based commands

- ❖ Xamarin.Forms also includes a **TapGestureRecognizer** which can provide a command interaction for other controls or visuals


```
<Image Source="IDareYouToTapMe.jpg">
  <Image.GestureRecognizers>
    <TapGestureRecognizer
      Command="{Binding BeBraveCommand}"
      CommandParameter="TheyTookTheDare!" />
  </Image.GestureRecognizers>
</Image>
```

**CommandParameter** property supplies the command's parameter – in this case as a **string**

# Implementing commands in the VM

- ❖ Command should be exposed as a public property from the ViewModel

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    public ICommand GiveBonus { get; private set; }
    ...
    public EmployeeViewModel(Employee model) {
        this.model = model;
        GiveBonus = new GiveBonusCommand(this);
    }
    ...
}
```




```
public class GiveBonusCommand : ICommand
```

# Implementing ICommand

❖ **ICommand** has three required members you must implement

**CanExecute** is called to determine whether the command is valid, this can enable / disable the control which is bound to the command



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

# Implementing ICommand

❖ **ICommand** has three required members you must implement

**Execute** is called to actually run the logic associated with the command when the control is activated – it will only be called if **CanExecute** returned **true**

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

# Implementing ICommand

❖ **ICommand** has three required members you must implement

## CanExecuteChanged

is an event which the binding will subscribe to, the ViewModel should raise this event when the validity of the command changes

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

The binding will then call **CanExecute** and enable / disable the UI in response



```
public partial class GiveBonusCommand : ICommand
{
    public event EventHandler CanExecuteChanged = delegate {};

    MainViewModel viewModel;
    public GiveBonusCommand(MainViewModel vm) {
        this.viewModel = vm;
    }

    public bool CanExecute(object parameter) {
        return this.viewModel.SelectedEmployee != null
            && (DateTime.Now - this.viewModel.SelectedEmployee.HireDate)
                .TotalHours > 8;
    }

    public void Execute(object parameter) {
        this.viewModel.SelectedEmployee.GiveBonus(1000);
    }

    public void RaiseCanExecuteChanged() {
        CanExecuteChanged(this, EventArgs.Empty);
    }
}
```

Command relies heavily  
on the data in the  
ViewModel ... could we  
move this logic?

# Implementing commands generically

- ❖ Can use built-in **Command** and **Command<T>** to forward command to VM

```
public class Command<T> : ICommand
{
    Action<T> _function;
    public void Execute(object parameter) {
        _function.Invoke((T) parameter);
    }

    public bool CanExecute(object parameter) {...}
    public event EventHandler CanExecuteChanged;
}
```

Initialize with delegates for each of the required methods – then you can define each command with logic in the ViewModel

# Using delegate commands

- ❖ **Command<T>** and **Command** provides mechanism to centralize the logic for the commands into the VM

```
public class EmployeeViewModel : INotifyPropertyChanged
{
    public ICommand GiveBonus { get; private set; }
    public EmployeeViewModel(Employee model) {
        GiveBonus = new Command(OnGiveBonus, OnCanGiveBonus);
    }

    void OnGiveBonus() { ... }
    bool OnCanGiveBonus() { return ... }
}
```

# Existing MVVM Libraries

- ❖ Easy to roll your own MVVM support, but there are several really good MVVM libraries available for cross platform development which include a lot of additional features
  - Prism [[pnpmvvm.codeplex.com](https://pnpmvvm.codeplex.com)]
  - MvvmCross [[github.com/MvvmCross](https://github.com/MvvmCross)]
  - MvvmLight [[codeplex.com/MvvmLight](https://codeplex.com/MvvmLight)]
  - ReactiveUI [[reactiveui.net](https://reactiveui.net)]
  - Caliburn.Micro [[github.com/Caliburn-Micro](https://github.com/Caliburn-Micro)]
  - MvvmHelpers [[codeplex.com/MvvmHelpers](https://codeplex.com/MvvmHelpers)]
  - [your favorite goes here] 😊

# Flash Quiz

# Flash Quiz

- ① Commands are *not* supported on which control?
- a) Button
  - b) Switch
  - c) MenuItem
  - d) Trick question - commands are supported on all of them!

# Flash Quiz

- ① Commands are *not* supported on which control?
- a) Button
  - b) Switch
  - c) MenuItem
  - d) Trick question - commands are supported on all of them!

# Flash Quiz

- ② Commands are described through \_\_\_\_\_.
- a) IDelegateCommand
  - b) DelegateCommand
  - c) ICommand
  - d) Command



# Flash Quiz

- ② Commands are described through \_\_\_\_\_.
- a) IDelegateCommand
  - b) DelegateCommand
  - c) ICommand
  - d) Command

# Group Exercise

Using commands to run behavior



**Xamarin**  
University

# Summary

1. Identify the need for Commands
2. Illustrate the ICommand interface
3. Generalizing the command





# Testing MVVM based apps

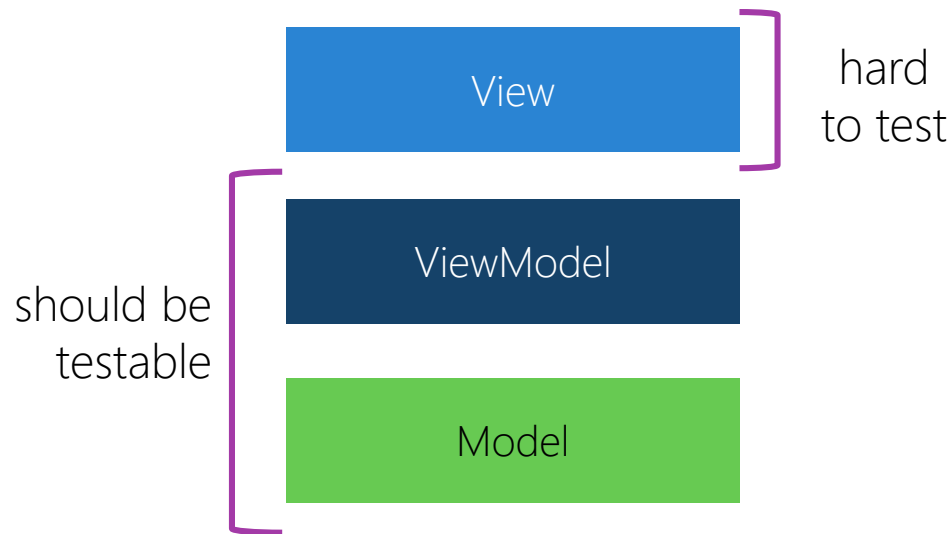
# Tasks

1. Testing the ViewModel



# Testing Surface

- ❖ Unit tests involve testing small, isolated pieces of our application independently; that's very hard to do for tightly coupled GUI applications
- ❖ Testable code is code which does not have dependencies on a UI being present



# Testing the ViewModel

- ❖ ViewModel can be tested independently of the UI / platform
- ❖ Allows for testing of business logic *and* visual logic
- ❖ Can use well-known unit testing frameworks such as NUnit or MSTest



# Testing the ViewModel

set properties  
and invoke  
command – just  
like UI would

```
[TestMethod]
void Employee_GiveBonus_Succeeds()
{
    var data = new Employee(...);
    var vm = new EmployeeViewModel(data);
    vm.GiveBonus.Execute("500");

    Assert.AreEqual(500,
                    data.GetNextPaycheckData().Extras);
}
```

... and then test the results to verify it does what you expect



# Demonstration

Adding unit tests for View Models

# Summary

## 1. Testing the ViewModel



# Thank You!

Please complete the class survey in your profile:  
[university.xamarin.com/profile](https://university.xamarin.com/profile)

