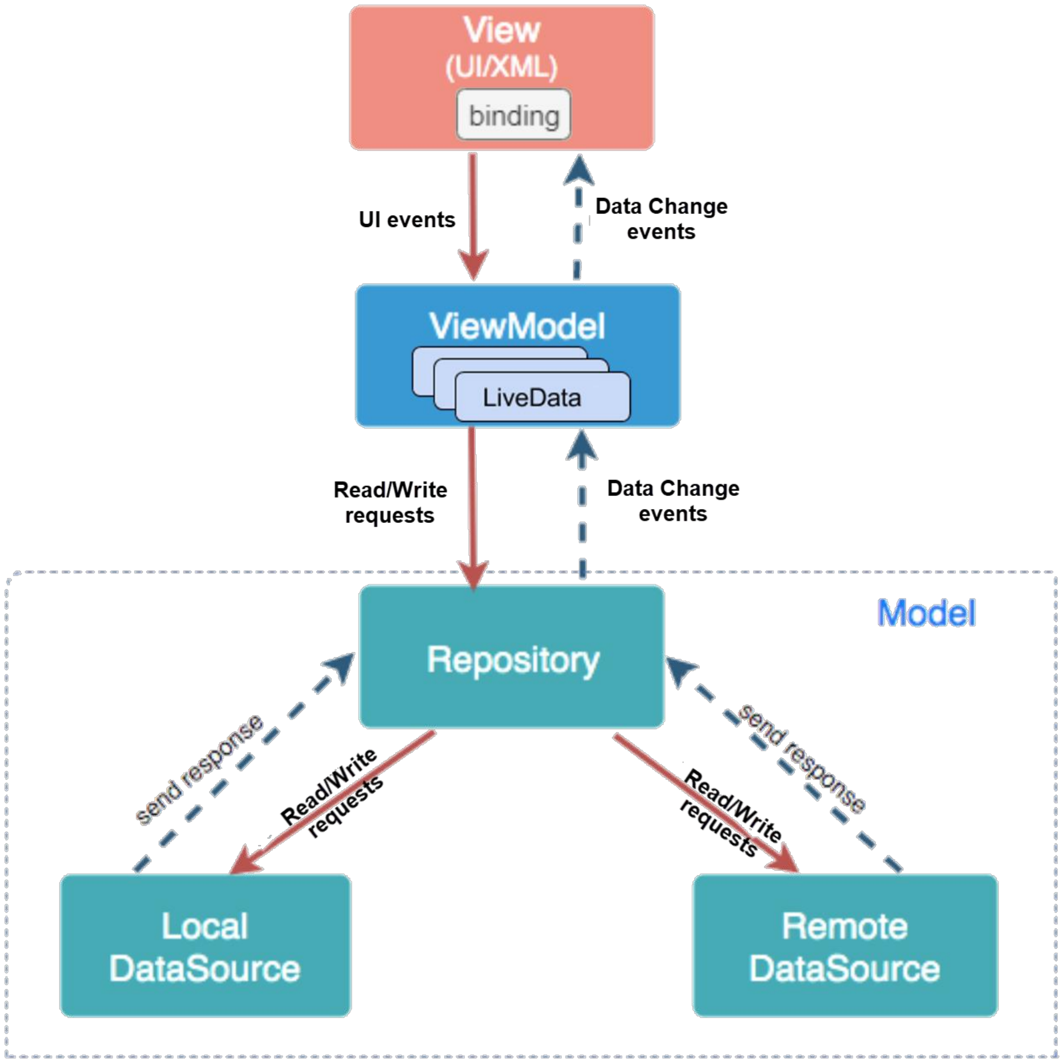
Outline

MVVM Architecture

Back

Model-View-ViewModel (MVVM) Architecture



View = UI to get input from the user.

It observes data changes from the ViewModel to update the UI accordingly

ViewModel

- Holds data needed for the UI
  - Interacts with the Model to read/write data based on user input
  - Notifies the view of data changes
- Implements logic / computation

Model - handles data operations

- Model has entities that represent app data
- Repositories read/write data from either a Local Database (using Room library) or a Remote Web API (using Retrofit library)
- Implements data-related logic / computation

MVVM Key Principles

- Separation of concerns:
  - View, ViewModel, and Model are **separate components** with distinct roles
- Loose coupling:
  - ViewModel has no direct reference to the View
  - View never accesses the model directly
  - Model unaware of the view
- Observer pattern:
  - View observes the ViewModel
  - ViewModel observes the Model
- Inversion of Control: not be covered in this course
  - Uses Dependency Injection instead of direct instantiation of objects

Advantages of MVVM

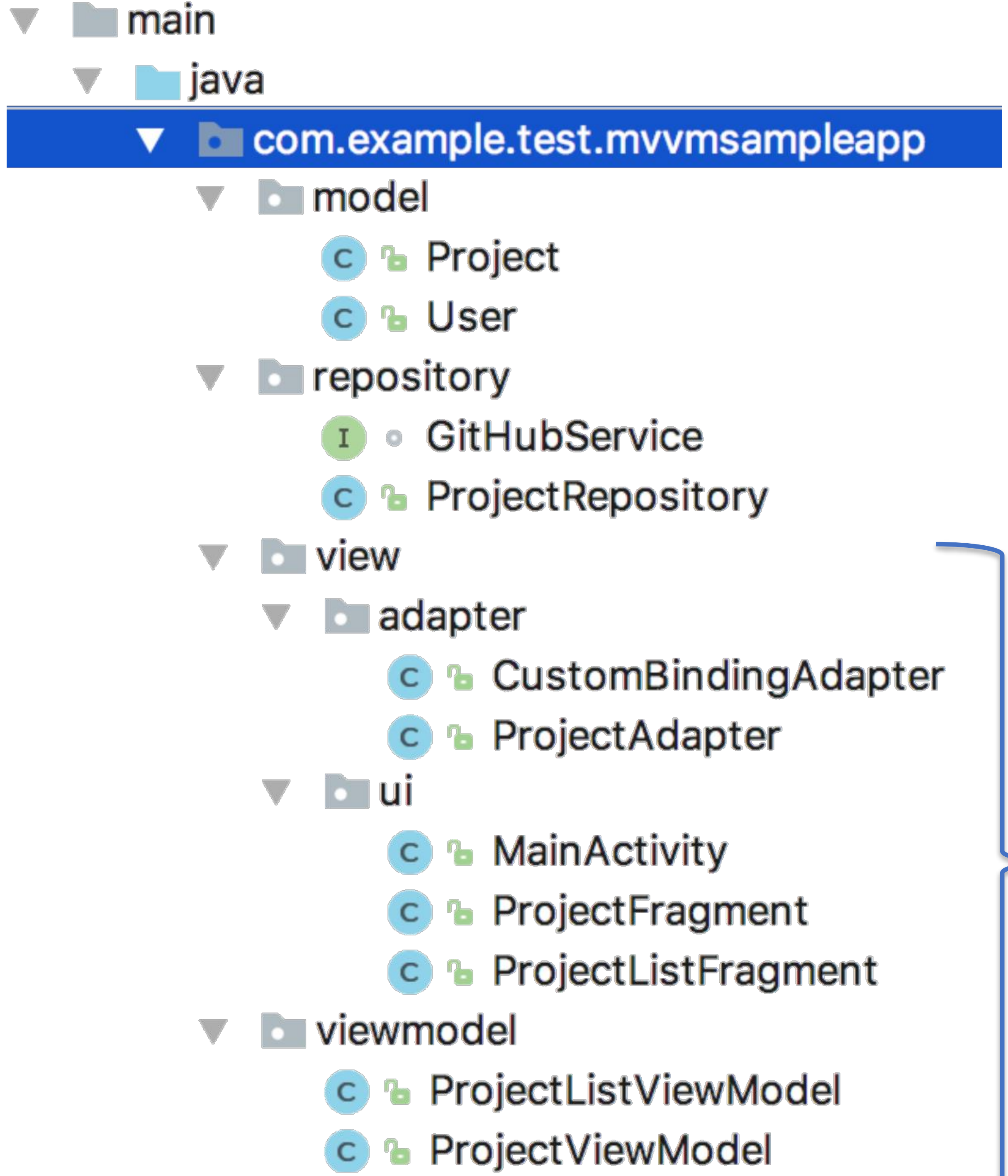
- ***Separation of concerns =*** **separate ui from app logic**
  - App logic is not intermixed with the UI. Consequently, code is cleaner, flexible and easier to understand and change
  - Allow changing a component without significantly disturbing the others (e.g., View can be completely changed without touching the model)
  - Easier **testing** of the App components

MVVM => Easily maintainable and testable app

Android Architecture Components

- Android architecture components are a collection of libraries to ease developing MVVM-based Apps
- Part of Android Jetpack They include:
  - ViewModel stores UI-related data that isn't destroyed on screen rotation
  - LiveData data holder that notifies the View when the underlying data changes
  - Data Binding of objects to UI components to trigger UI updates when the data changes
  - Room to read / write data to local SQLite database

Recommended Project Structure



```
main
  java
    com.example.test.mvvmsampleapp
      model
        Project
        User
      repository
        GitHubService
        ProjectRepository
      view
        adapter
          CustomBindingAdapter
          ProjectAdapter
        ui
          MainActivity
          ProjectFragment
          ProjectListFragment
      viewmodel
        ProjectListViewModel
        ProjectViewModel
```

You may organize the view by feature

ViewModel

Lifecycle Aware

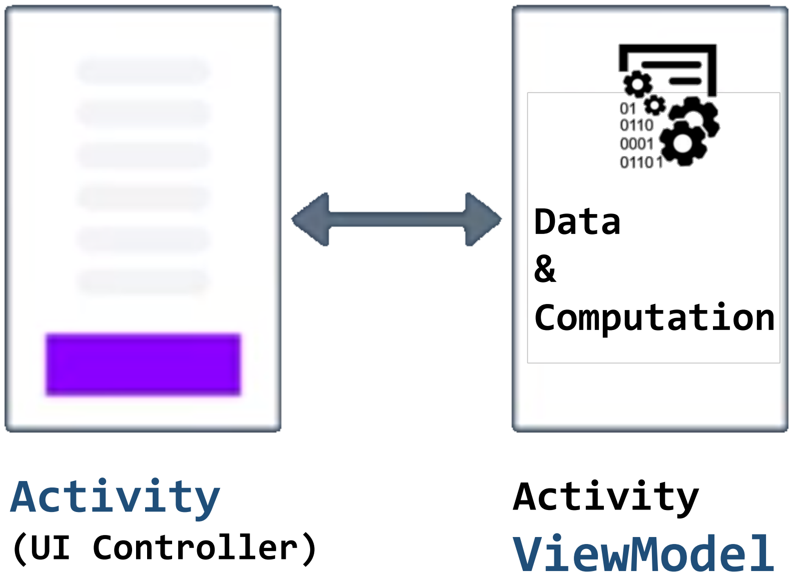
Survives Config Changes

Back

ViewModel

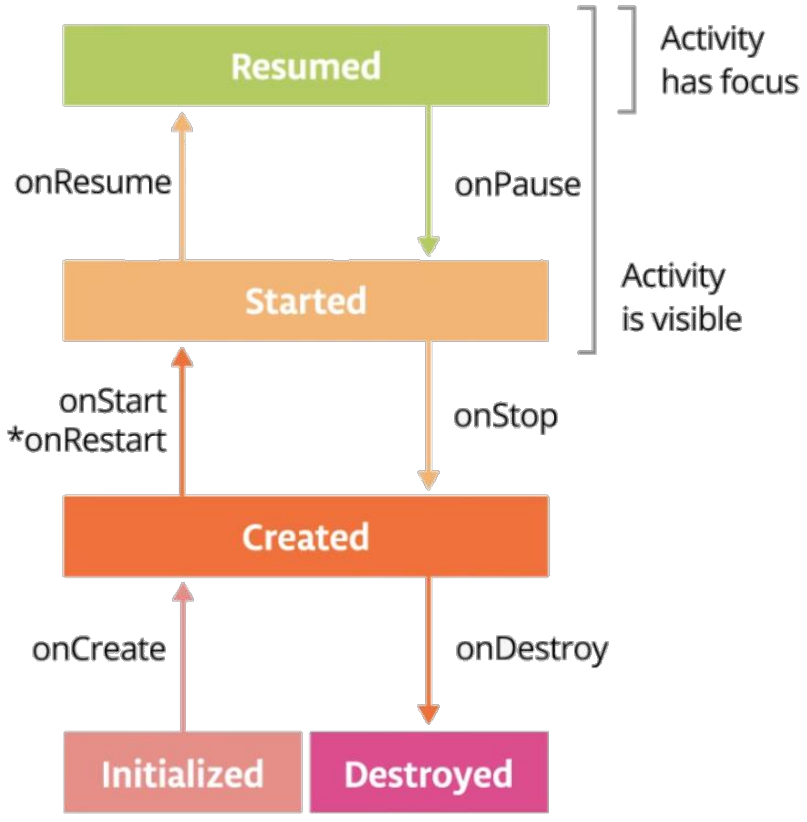
- ViewModel is used to **store and manage UI-related data**
  - in a lifecycle conscious way
  - allows data to survive device configuration changes such as *screen rotations* or *changing the device's language*

- If the system destroys or re-creates a UI Controller (e.g., when the screen rotates), any transient UI-related data you store in it is lost



User **ViewModel**:
- Store UI data
- Read/write data from a Repository

Activity Lifecycle

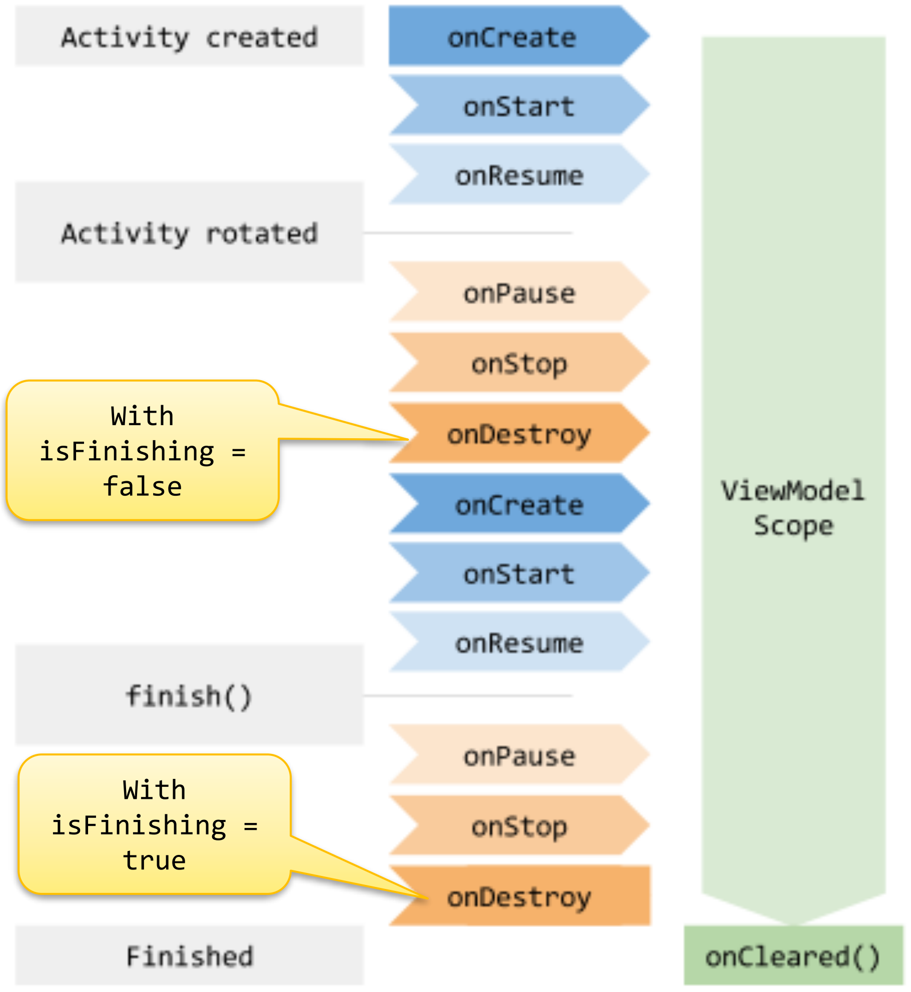


An activity has essentially **four states**:

- ***Resumed*** if the activity in the foreground of the screen (has focus)
- ***Started*** if the activity has lost focus but is still visible (e.g., beneath a dialog box).
  - When the user returns to the activity, it is **resumed**
- **Created** if the activity is completely obscured by another activity.
  - When the user navigates to the activity, it must be **restarted** and restored to its previous state.
- **Destroyed** when the user closes the app or if the activity is killed (when memory is needed or due to finish() being called on the activity)

ViewModel Lifecycle

- ViewModel object is scoped to the activity in which it is created
- However, it has a longer lifespan compared to the associated Activity which may undergo a rotation and get recreated
- It remains in memory until the activity is completely destroyed
    - When the activity is recreated (after a screen rotation) the associated ViewModel remains alive



ViewModel Example

```kotlin
class MainActivityViewModel : ViewModel() {
    var team1Score = 0
    fun incrementTeam1Score() = team1Score++
}
```

```kotlin
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        // Associate the Activity with the ViewModel
        val viewModel by viewModels<MainActivityViewModel>()
        team1ScoreTv.text = viewModel.team1Score.toString()
}
```

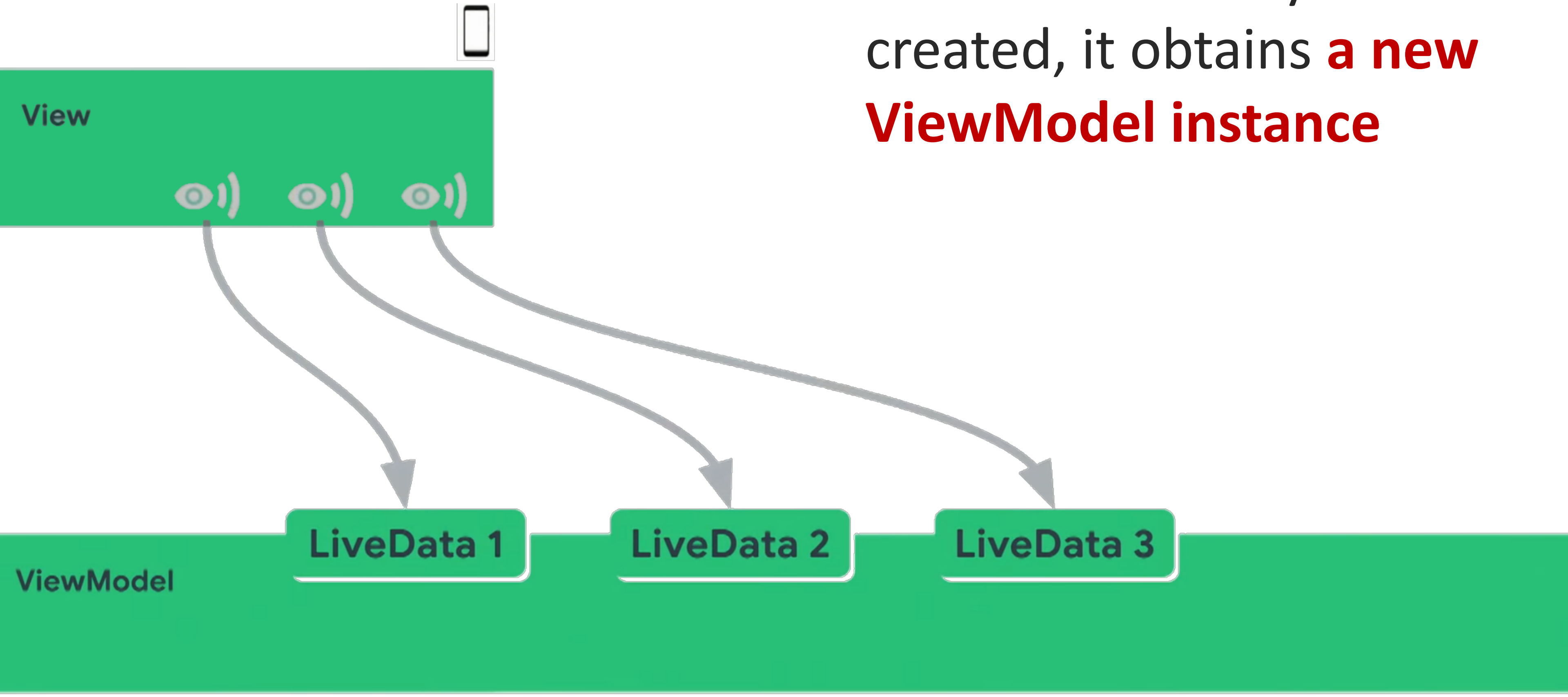
Associate the Activity and ViewModel

- The activity obtains an instance of the ViewModel using

```
val viewModel by viewModels<MainActivityViewModel>()
```

  - For the first call, this creates and returns a new ViewModel instance
  - For subsequent calls, which happens whenever onCreate is called, it will return the pre-existing ViewModel associated with the Activity (e.g., MainActivity)
    - ➢ This is what preserves the data and maintains the connection with the **same** ViewModel

When the Activity is first Created

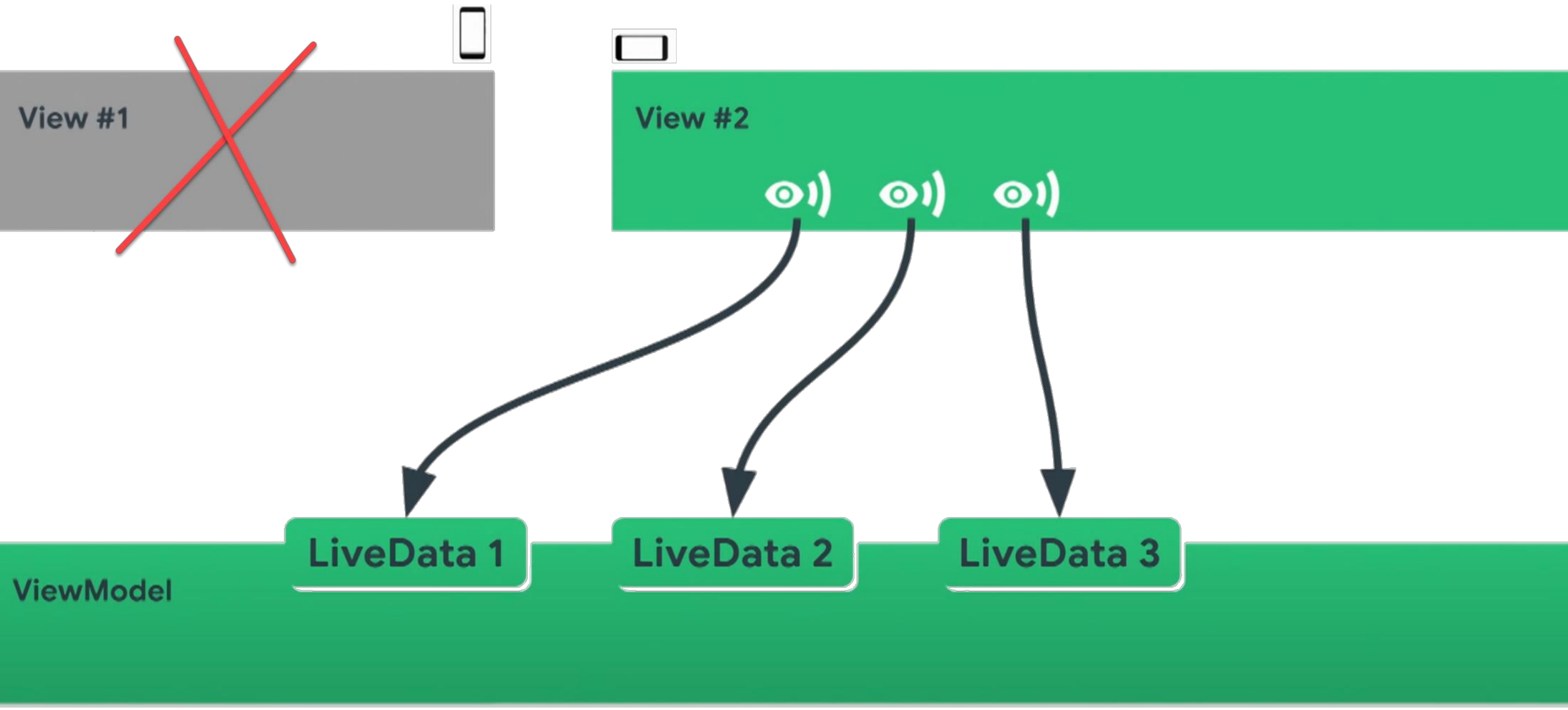


When the Activity is first created, it obtains **a new ViewModel instance**

Source: https://www.youtube.com/watch?v=B8ppnjGPAGE

OnConfig change (e.g., Screen Rotates)

OnConfig change, the Activity is destroyed, and a new instance of the Activity is created then it obtains **the same ViewModel instance used previously**

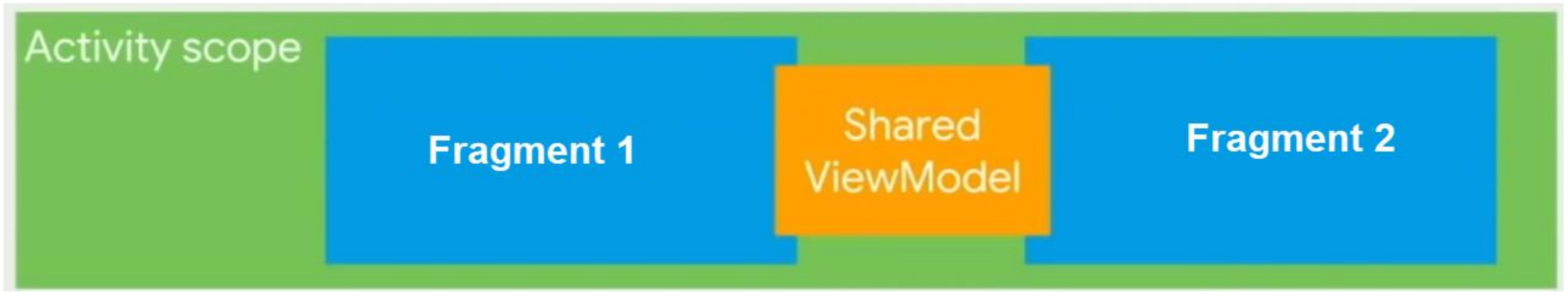


Source: https://www.youtube.com/watch?v=B8ppnjGPAGE

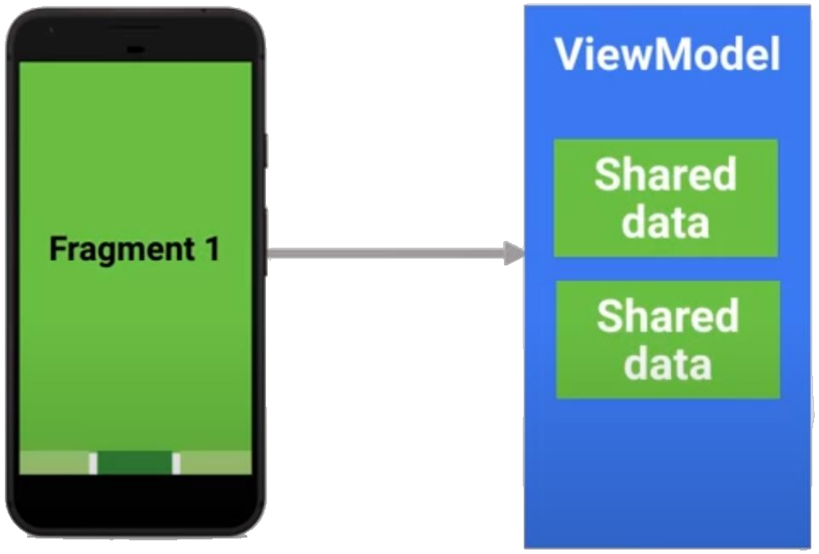
“no contexts in ViewModels” rule

- ViewModel should **not be aware of the View** who is interacting with
  => It should be decoupled from the View
  - ViewModel should not hold a reference to Activities, Fragments, or Views
  - Should not have any Android framework related code
  - As this defeats the purpose of separating the UI from the data
  - Can lead to memory leaks and crashes (due to null pointer exceptions) as the ViewModel outlives the View
    - if you rotate an Activity 3 times, 3 three different Activity instances will be created, but you only have one ViewModel instance

Share data between fragments



- Fragments can **share** a **ViewModel** associated with the activity



```
class DetailFragment : Fragment() {
    // Use the 'by activityViewModels()' to get a reference to the ViewModel
   // associated with the activity

    val viewmodel = by activityViewModels<SharedViewModel>()
}
```

Dependencies

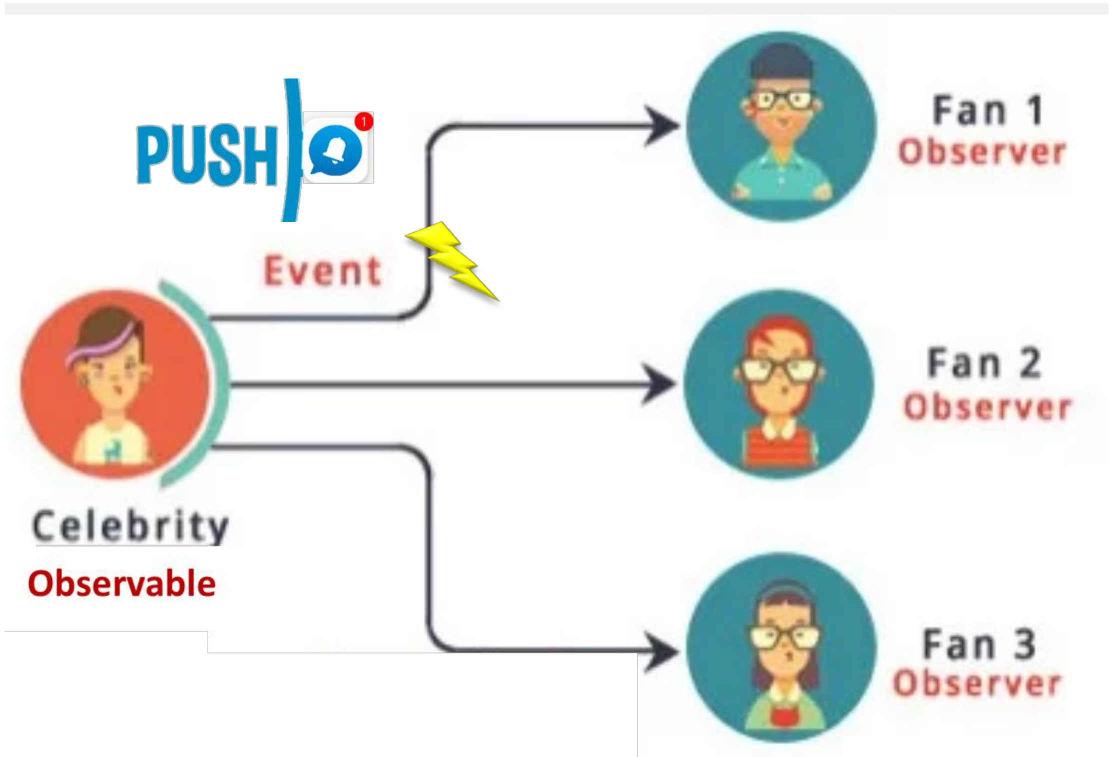
```
// Add to - Module:app build.gradle

def lifecycle_version = "2.2.0"
// ViewModel
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
// LiveData
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"

// Kotlin extensions - activity-ktx & fragment-ktx
def activity_version = "1.1.0"
implementation "androidx.activity:activity-ktx:$activity_version"
def fragment_version = "1.2.5"
implementation "androidx.fragment:fragment-ktx:$fragment_version"

// Configure using Java 8 - add Module:app/build.gradle under android { ...
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
kotlinOptions { jvmTarget = "1.8" }
```

LiveData

Back

Observable - Real-Life Example

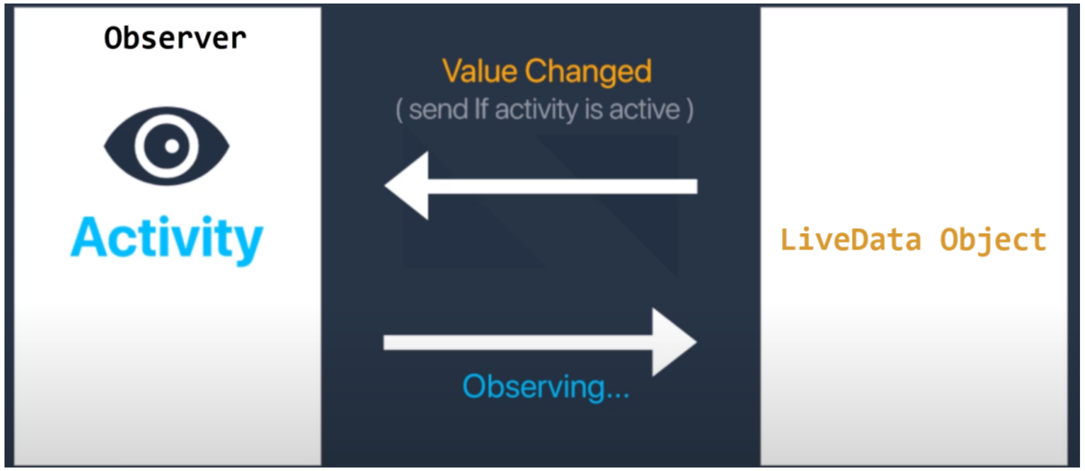
- A celebrity who has many fans on Instagram. Fans want to get all the latest updates (photos, videos, posts etc.). Here fans are **Observers** and celebrity is an **Observable** (called **LiveData** on Android)

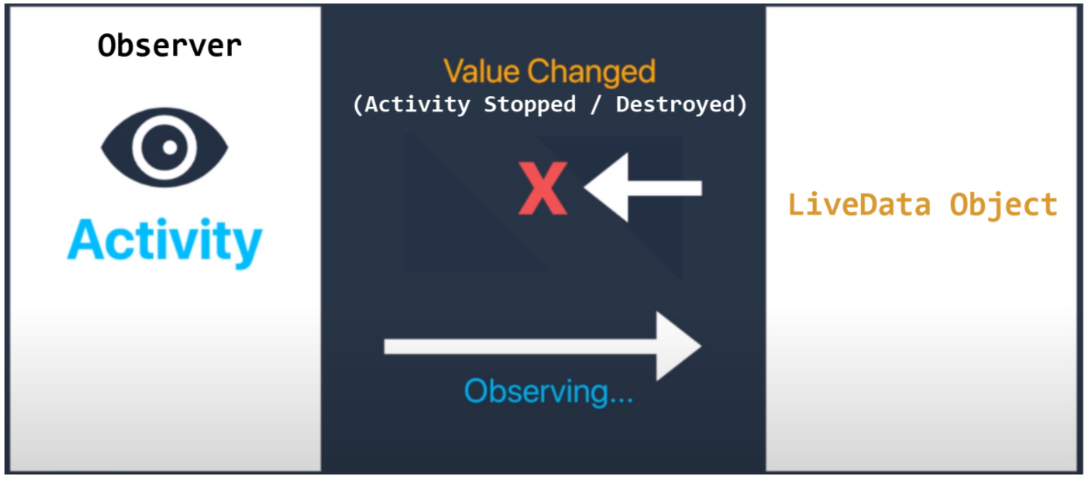


LiveData is lifecycle-aware

LiveData is aware of the Lifecycle of its Observer

- Notifies data changes to only **active** observers (Stopped/Destroyed activity/fragment will NOT receive updates)
- It automatically removes the subscription when the observer is destroyed so it will not get any updates

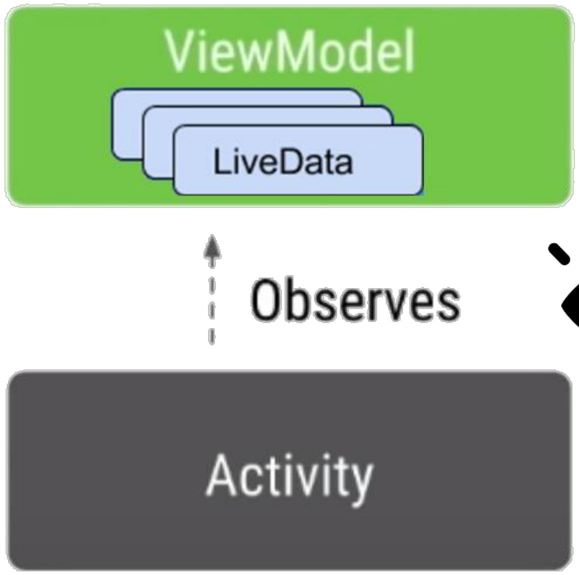




LiveData in Code

LiveData **warps around** an object and allows the view the **observe** it

LiveData
team1Score



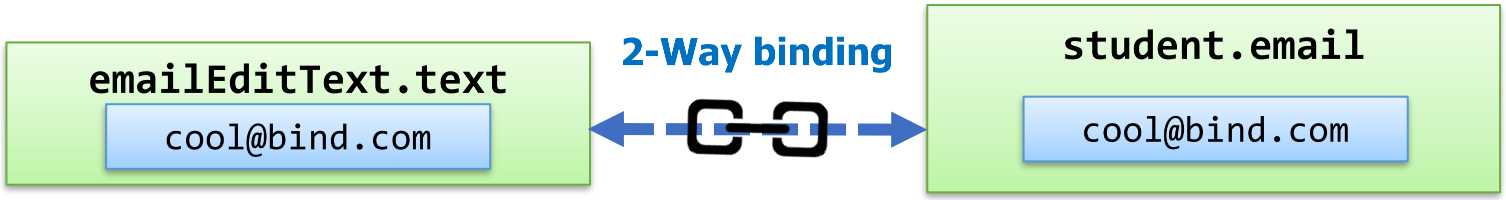
- ViewModel expose LiveData objects that the View can observe

```
class MainActivityViewModel : ViewModel() {
    private val _team1Score = MutableLiveData<Int>(0)

    // Expose read only LiveData that the View can observe or bind to
    val team1Score: LiveData<Int> get() = _team1Score

    fun incrementTeam1Score() {
       // call postValue to notify Observers
       _team1Score.postValue((_team1Score.value ?: 0) + 1)
    }
}
```

- View **observes** LiveData changes

```
class MainActivity : AppCompatActivity() {
    // onCreate
    // Associate the Activity with the ViewModel
    val viewModel by viewModels<MainActivityViewModel>()

    viewModel.team1Score.observe(this) {
            team1ScoreTv.text = it.toString()
    }
}
```

Data Binding



< Back

Excel is an excellent example …

Data Binding

- Data Binding allows **declarative** **binding** UI components -in the activity/fragment layouts- to a data source (typically a LiveData object in the ViewModel)
  - rather than programmatically assigning values to the UI components

- Declaratively **binding** the text property of the TextView with the userName property of the user object

```
<TextView android:id="@+id/userName"
    android:text="@{user.userName}" />
```

- Rather then programmatically assigning the values to UI components

```
userNameTv.text = user.userName
```

Enable Data Binding

- To enable data binding add to app / build.gradle

```
apply plugin: 'kotlin-kapt'
android {    ...
   buildTypes {  ...    }
   android.buildFeatures.dataBinding true
}
```

- To use data binding in a layout file, wrap the entire XML layout a in a **<layout>** tag. Then add layout variables.

Transforming a Standard XML Layout Into a Data Binding Layout

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">
    <data>
        <variable
            name="profile"
            type="qa.edu.cmps312.mvvm.model.Profile" />
    </data>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/firstName"
            ...
            android:text="@{profile.firstName}"
        />

    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Connecting the View with the ViewModel

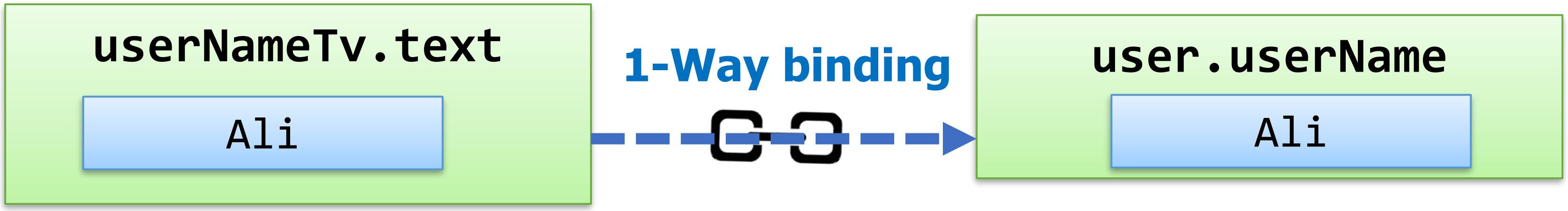
- onViewCreated obtain an instance of the view Binding Class (this class is auto-generated)
- Set the current fragment as the lifecycle owner of the obtained viewBinding
- Obtain a ViewModel instance using ***activityViewModels*** for shared viewModel scoped to the activity or using ***viewModels*** to get viewModel scoped to the fragment (if it is only used by the fragment)
- Bind the View with the viewModel

```
// onViewCreated(view: View, savedInstanceState: Bundle?)...
// Obtain an instance of the view binding class
val viewBinding = FragmentProfileBinding.bind(view)
// Specify the current fragment as the lifecycle owner
viewBinding.lifecycleOwner = this
// Obtain a ViewModel instance scoped to activity
val viewModel = by activityViewModels<ProfileViewModel>()
// Bind the View with the viewModel.profile
viewBinding.profile = viewModel.profile.value
```

Unidirectional Data Binding

- Data binding enables **synchronizing** UI with data source
  - The **target** listens for changes in the **source** and updates itself when the source changes
  - 1-Way binding syntax:

```
<TextView android:id="@+id/userName"
      android:text="@{user.userName}" />
```



Binding Adapter

- Provide custom binding logic for an attribute
  - E.g., Hide UI component is the value associated with it is 0

```
@BindingAdapter("app:hideIfZero")
fun View.hideIfZero(value: Int) {
  this.visibility = if (value == 0) View.GONE else View.VISIBLE
}
```

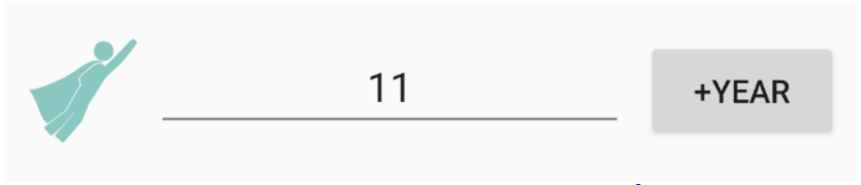
```
Seniority icon is hidden
if the number of
yearsExperience = 0
```

```
 <ImageView
     android:id="@+id/seniorityIv"
     ...
     app:hideIfZero="@{profile.yearsExperience}"/>
```

Binding Adapter

- Set icon and its color based on seniority

```
@BindingAdapter("app:seniorityIcon")
fun ImageView.popularityIcon(seniority: Seniority) {
    val color = getSeniorityColor(seniority, this.context)
    ImageViewCompat.setImageTintList(this, ColorStateList.valueOf(color))
    this.setImageDrawable(getSeniorityIcon(seniority, this.context))
}
```



```
<ImageView
    android:id="@+id/seniorityIv"
    ...
    app:seniorityIcon="@{profile.seniority}" />
```

Binding expressions

- You may also use Binding Expressions, but Binding Adapters are recommended to keep to view simpler. Also they are more reusable.

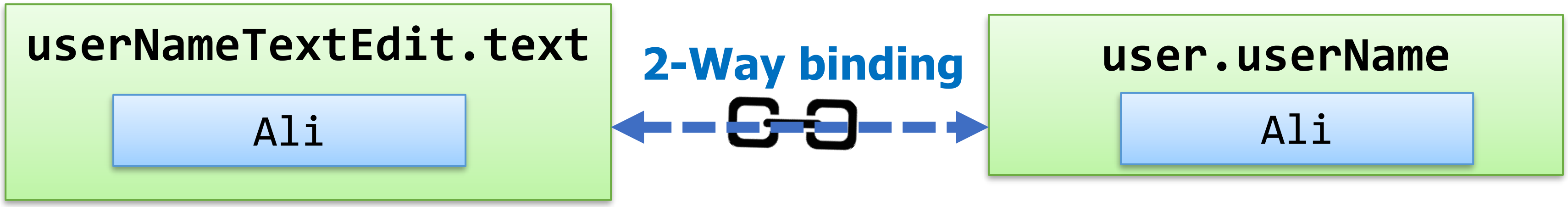
```
<TextView
  android:id="@+id/totalScoreTv"
  ...
  android:text="@{String.valueOf(viewModel.team1Score.intValue() + viewModel.team2Score.intValue())}"
  android:visibility="@{viewModel.team1Score.intValue() + viewModel.team2Score.intValue() > 0 ? View.VISIBLE : View.GONE}"
/>
```

Bidirectional Binding

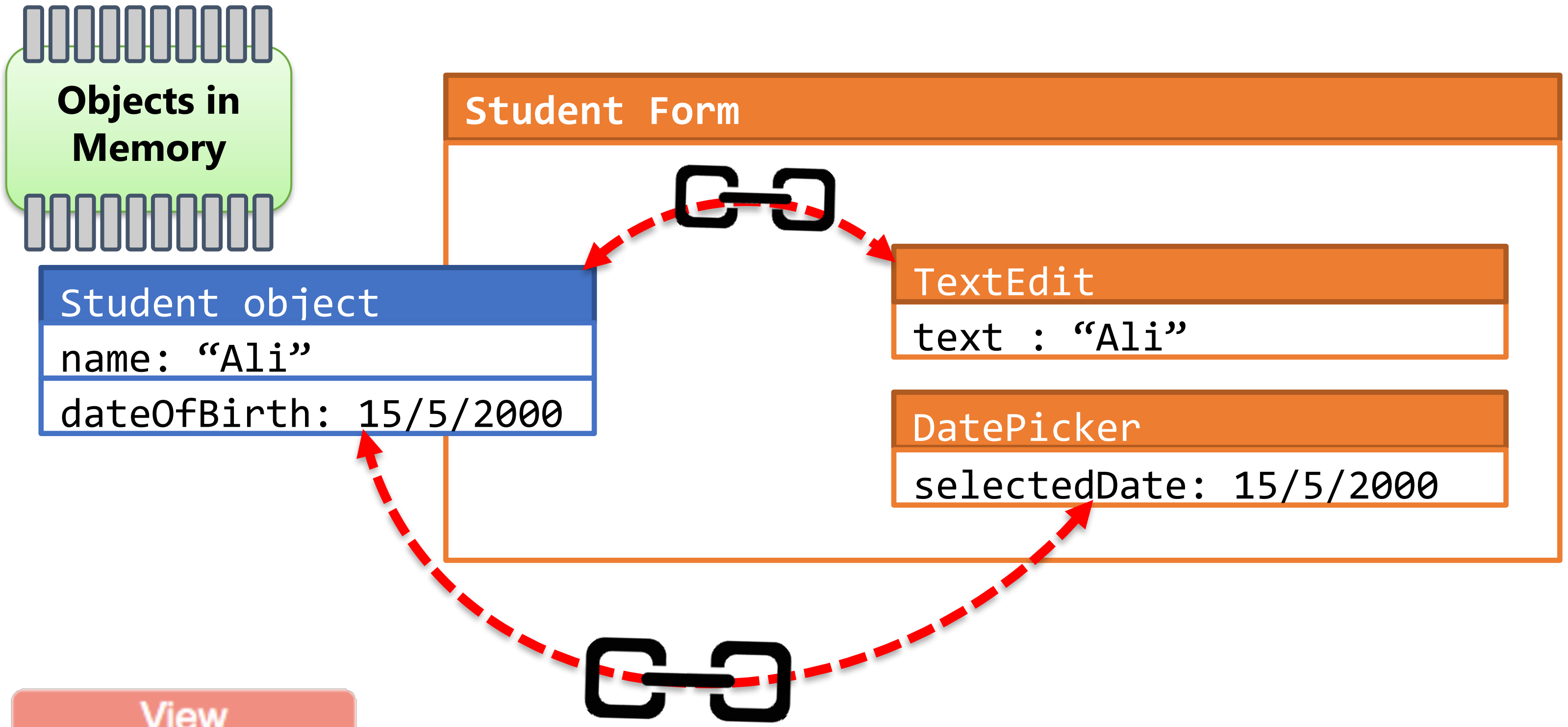
- Bidirectional (2-Way) Binding

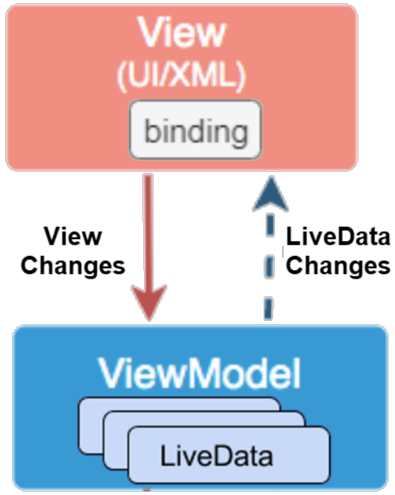
```
<TextEdit android:id="@+id/userName"
     android:text="@={user.userName}" />
```

  - Any changes of **`userNameTextEdit`** text or the **user.userName** property will be synchronized



Two-way Binding UI Components Properties with Object Properties





- Binding can be used to **synchronize** the UI components with the ViewModel
- Automatic propagation of data changes between the ViewModel and the View

2-Way Binding requires the model to implement BaseObservable

- The model class must implement **BaseObservable** to notify the observers when property values change

```
data class Profile(private var _firstName: String,
                  private var _lastName: String) : BaseObservable() {
    @get:Bindable
    var firstName
        get() = _firstName
        set(value) {
            _firstName = value
            notifyPropertyChanged(BR.firstName)
        }
... }
```

Resources

- MVVM
  - https://developer.android.com/jetpack/guide
  - https://medium.com/androiddevelopers/viewmodels-a-simple-example-ed5ac416317e
- Data Binding
  - https://developer.android.com/topic/libraries/data-binding
- Data Binding codelab
  - https://codelabs.developers.google.com/codelabs/android-databinding