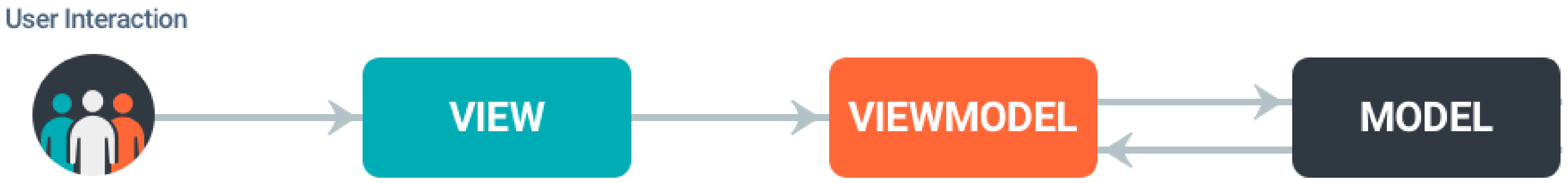
CMPS 312



Model-View-ViewModel (MVVM) Architecture

Dr. Abdelkarim Erradi

CSE@QU

Outline

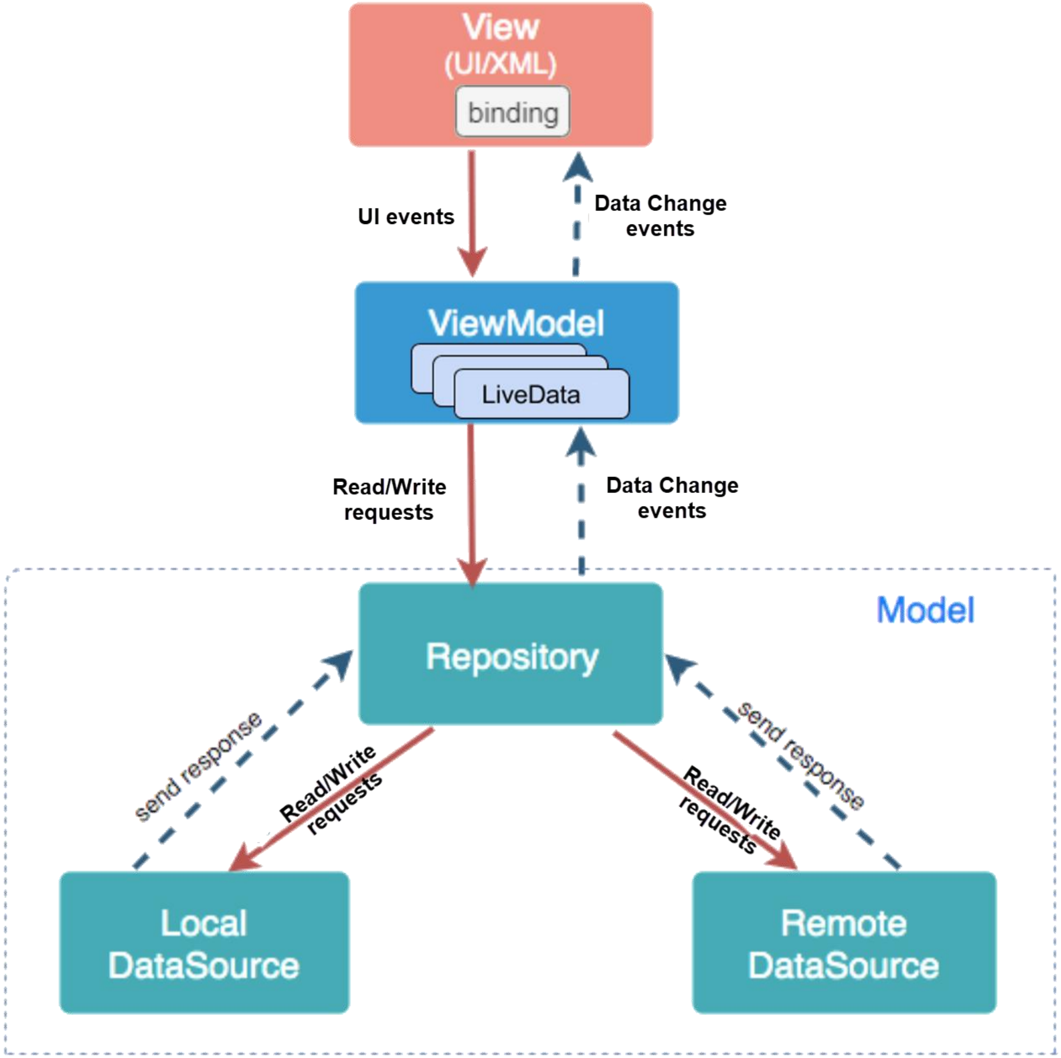
1. Model-View-ViewModel (MVVM)
2. ViewModel
3. LiveData
4. Data Binding

MVVM Architecture

< Back

Model-View-ViewModel (MVVM) Architecture

IMPORTANT



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)

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

IMPORTANT

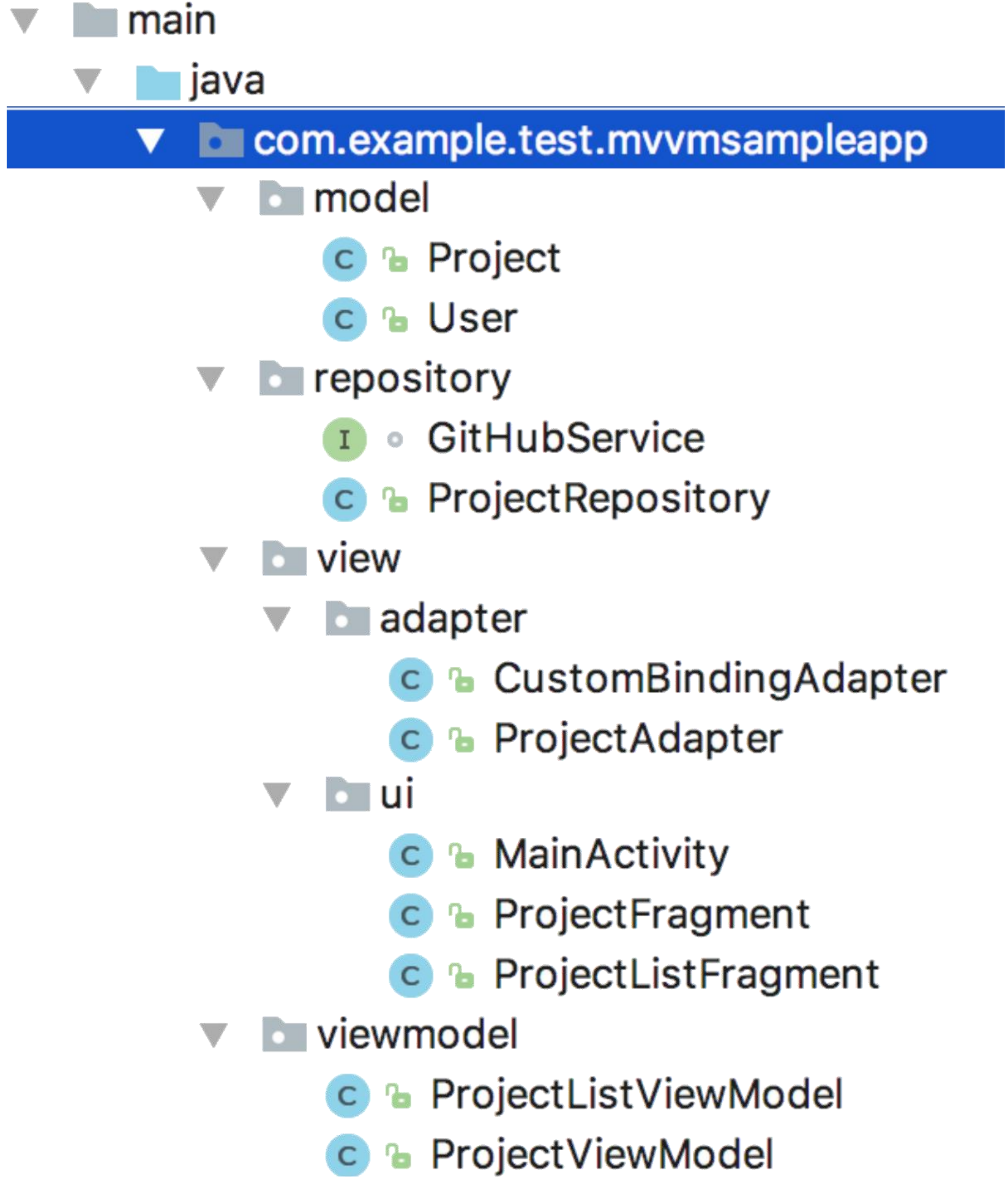
- ***Separation of concerns* = separate ui from app logic**
  - Computation is not intermixed with the UI. Consequently, code is cleaner, flexible and easier to understand and change.
  - Allow changing a component without significantly disturbing the others (e.g., UI can be completely changed without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

MVVM => Easily maintainable and testable app

Android Architecture Components

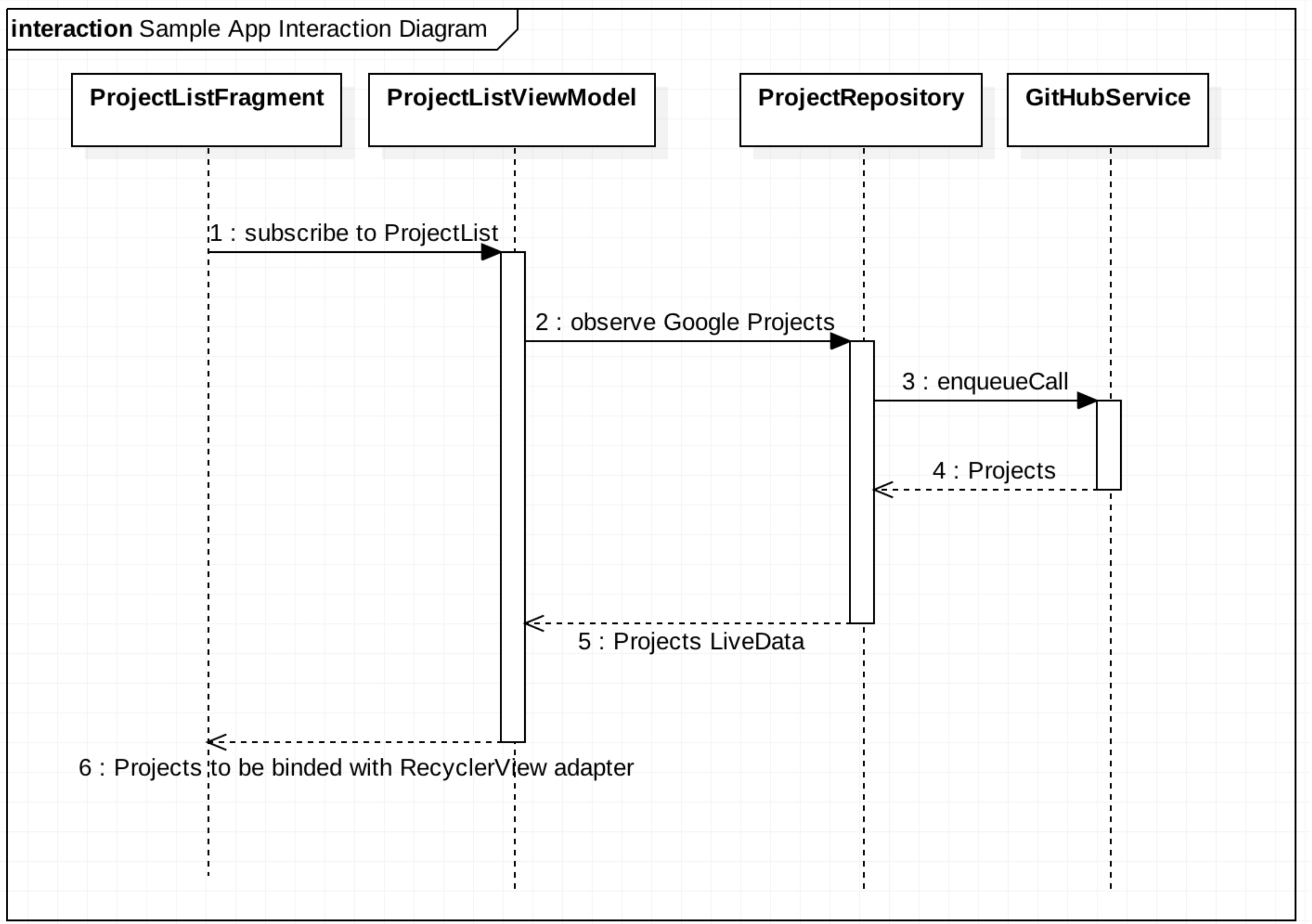
- Android architecture components are a collection of libraries to ease developing MVVM-based Apps
- Part of Android Jetpack They include:
  - ViewModel stores UI-related data that isn't destroyed on screen rotation
  - LiveData to create data objects that notify views when the underlying data changes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

Recommended Project Structure



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

Interaction diagram to retrieve Google GitHub projects


interaction Sample App Interaction Diagram
ProjectListFragment
ProjectListViewModel
ProjectRepository
GitHubService
1 : subscribe to ProjectList
2 : observe Google Projects
3 : enqueueCall
4 : Projects
5 : Projects LiveData
6 : Projects to be binded with RecyclerView adapter

Source: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1

ViewModel

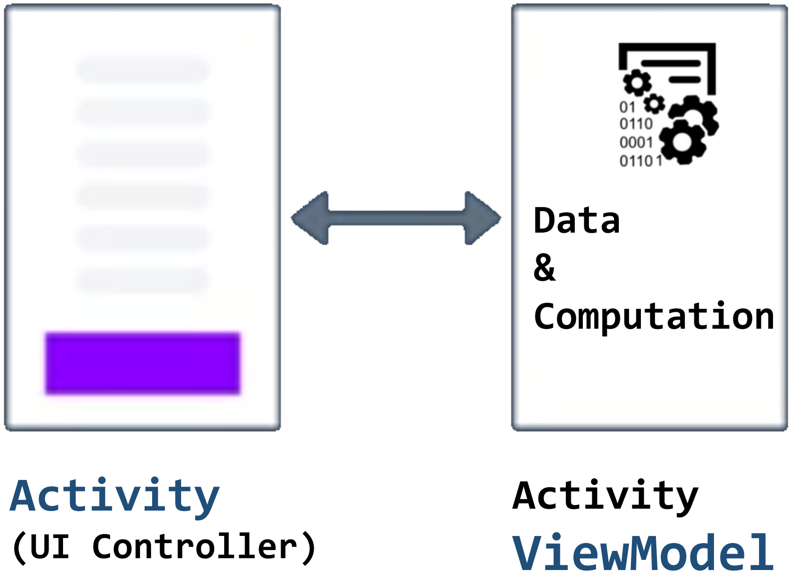
Lifecycle Aware

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*

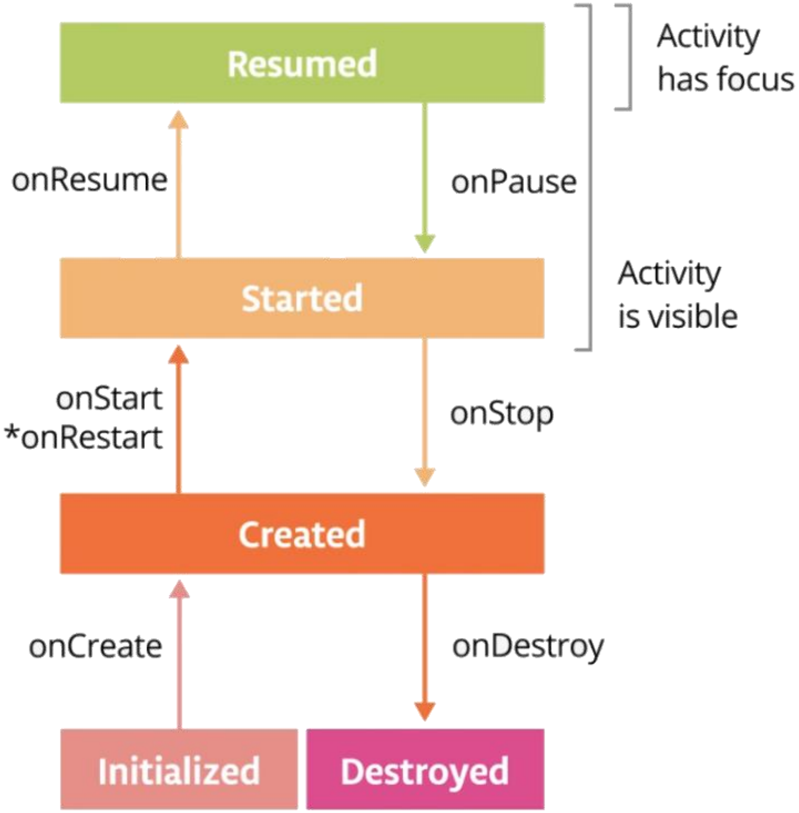


- If the system destroys or re-creates a UI Controller (e.g., when the screen rotates), any transient UI-related data you store in it is lost

User **ViewModel**:

- Store UI data
- Read/write data using Repository

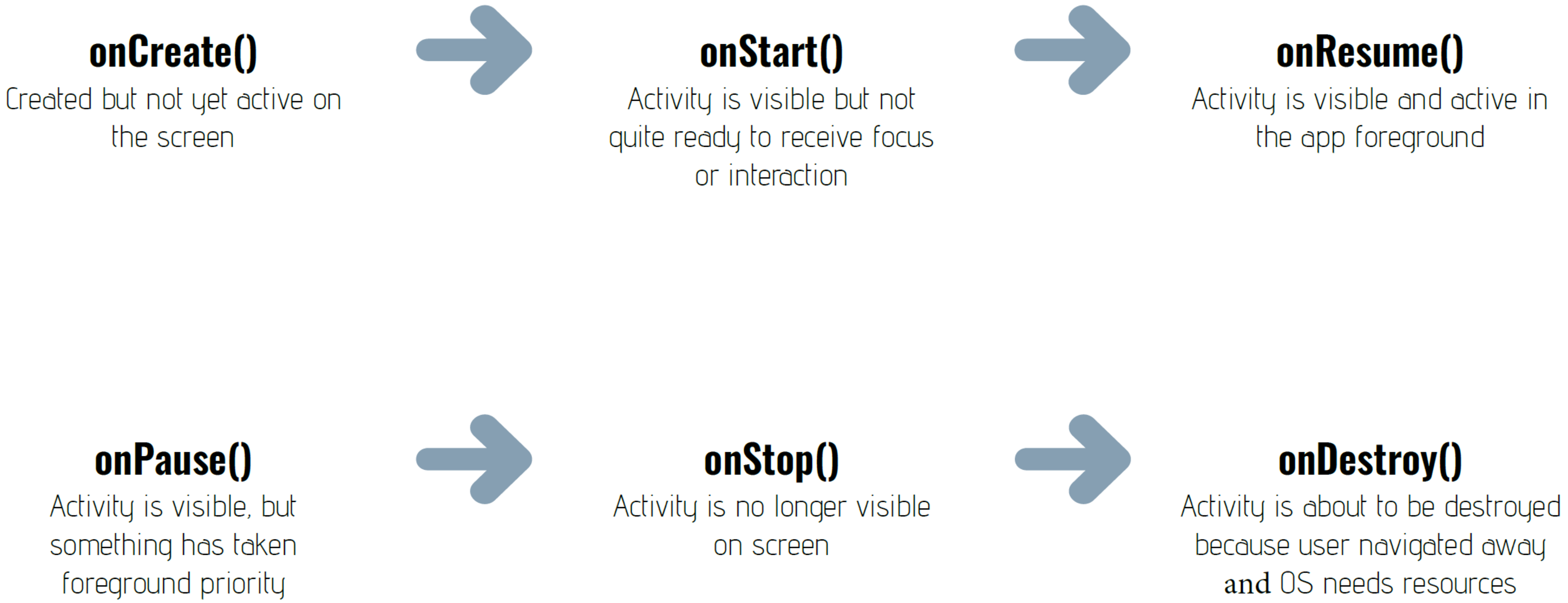
Activity Lifecycle



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)

Activity Lifecycle



onCreate()
Created but not yet active on the screen

onStart()
Activity is visible but not quite ready to receive focus or interaction

onResume()
Activity is visible and active in the app foreground

onPause()
Activity is visible, but something has taken foreground priority

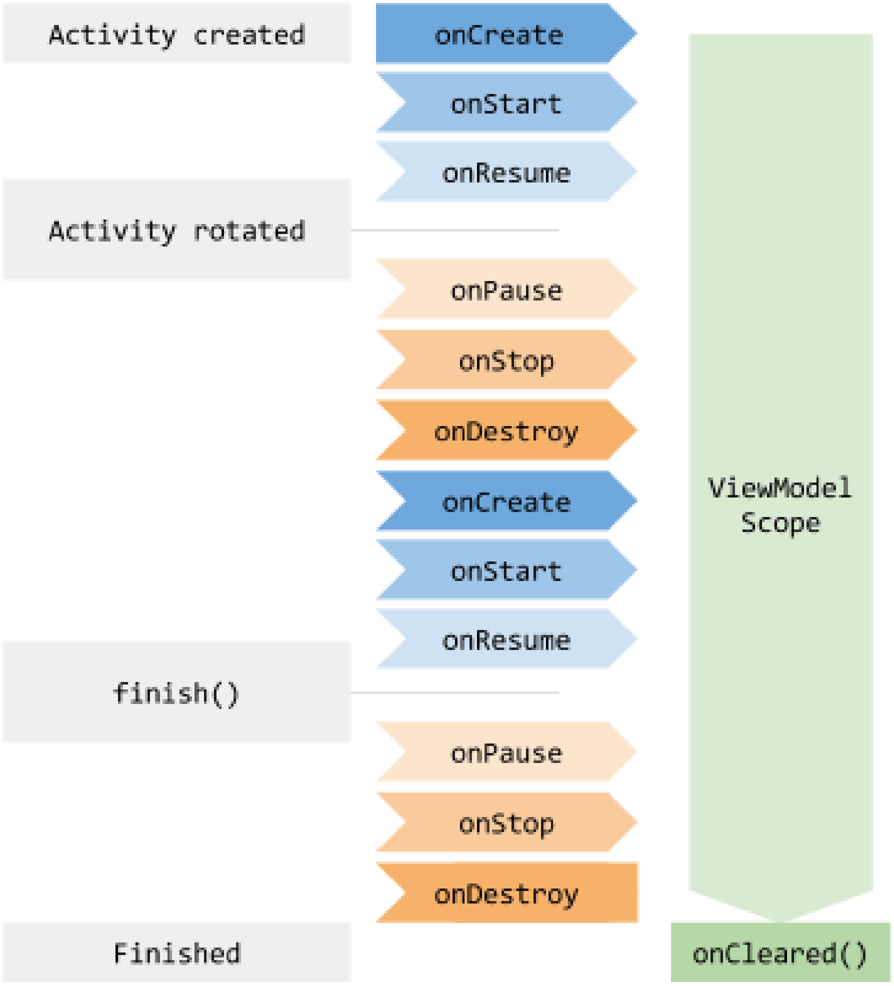
onStop()
Activity is no longer visible on screen

onDestroy()
Activity is about to be destroyed because user navigated away and OS needs resources

- Events handlers can be associated to these events
  - Android invokes them when the activity moves from one state to another
  - E.g., in onCreate() you inflate the layout and define click listeners

ViewModel Lifecycle

- Lifecycle of an Activity which undergoes a rotation and then is finally finished vs. ViewModel lifecycle
  - ViewModel object is scoped to activity in which it is created.
  - It remains in memory until the activity is completely destroyed



ViewModel Example

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

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

Associate the Activity and ViewModel

- The activity obtains an instance of the ViewModel using

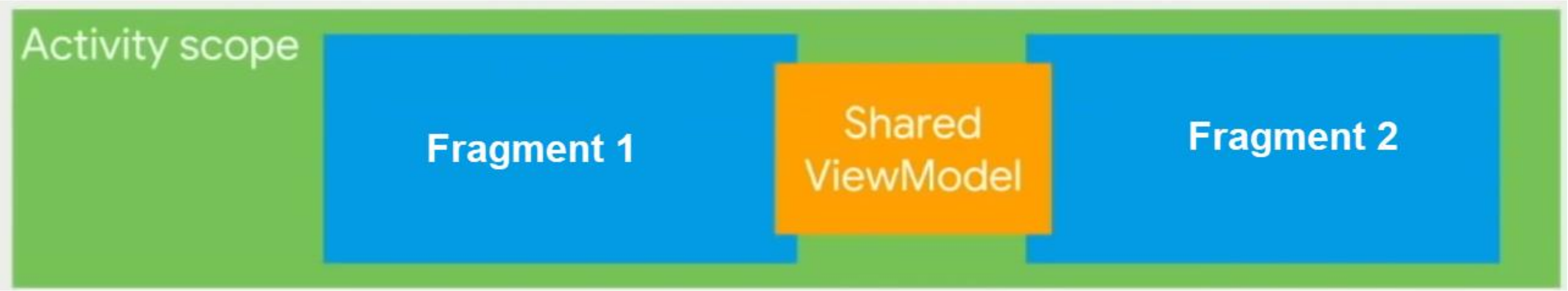
```
val viewModel by viewModels<MainActivityViewModel>()
```

  - For the first call, it creates a new ViewModel instance
  - For subsequent calls, which happens whenever onCreate is called, it will return the pre-existing ViewModel associated with the UI controller that is passed in as an argument (e.g., MainActivity)
  - This is what preserves the data and maintains the connection with the **same** ViewModel

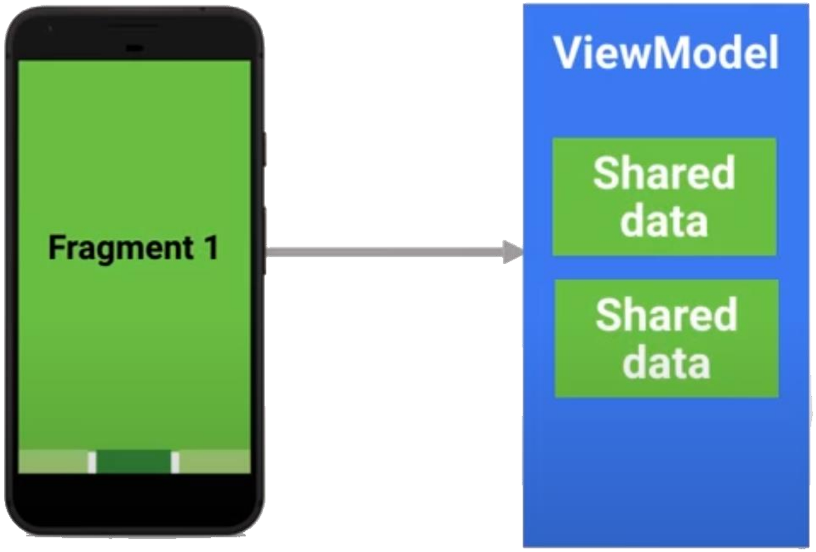
“no contexts in ViewModels” rule

- ViewModel should **not be aware of the view** who is interacting with => decouple it from the View
  - ViewModel should not hold a reference to Activities, Fragments, or Views
  - Defeats the purpose of separating the UI from the data and can lead to memory leaks
  - ViewModel outlives them
    - 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



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

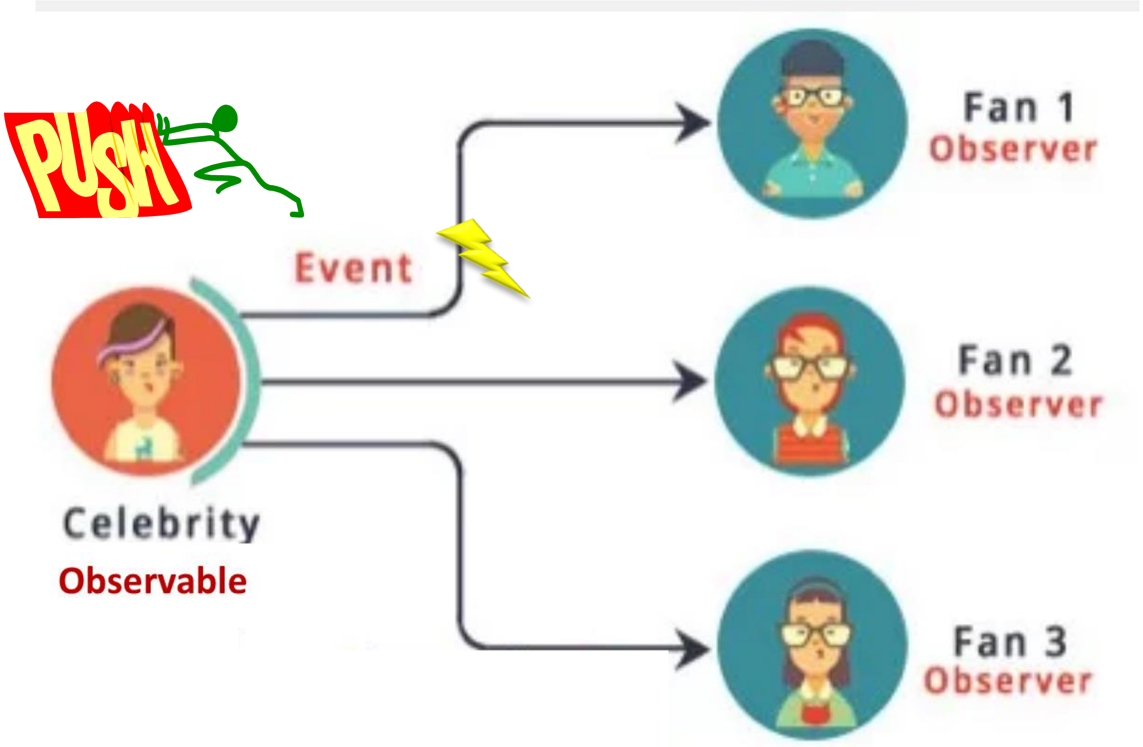
LiveData

- LiveData is an **observable data holder**: subscribers (e.g., UI) get notified when data change and can respond accordingly

- Other App components can observe LiveData objects for changes without creating **explicit and rigid dependency** between them
    - This decouples completely the LiveData object producer from the LiveData object consumer
    - E.g., ViewModel exposes its data using **LiveData** that the View can observe and update the UI accordingly

Observable - Real-Life Example

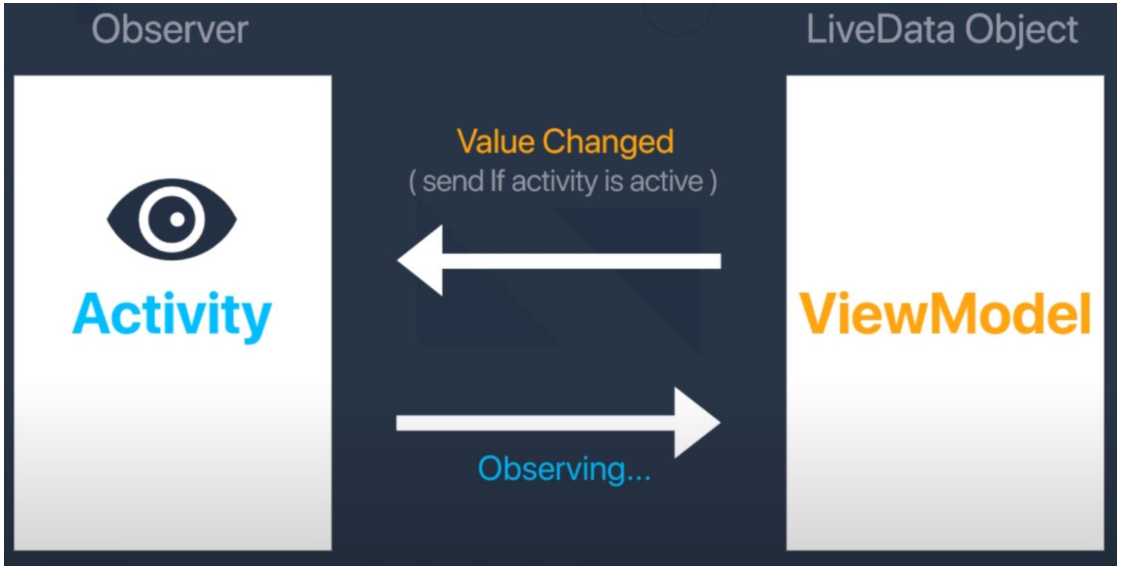
- A celebrity who has many fans on Instagram. Fans want to get all the latest updates (photos, videos, posts etc.). Here fans are **Observers** and celebrity is an **Observable** (called LiveData in Android)

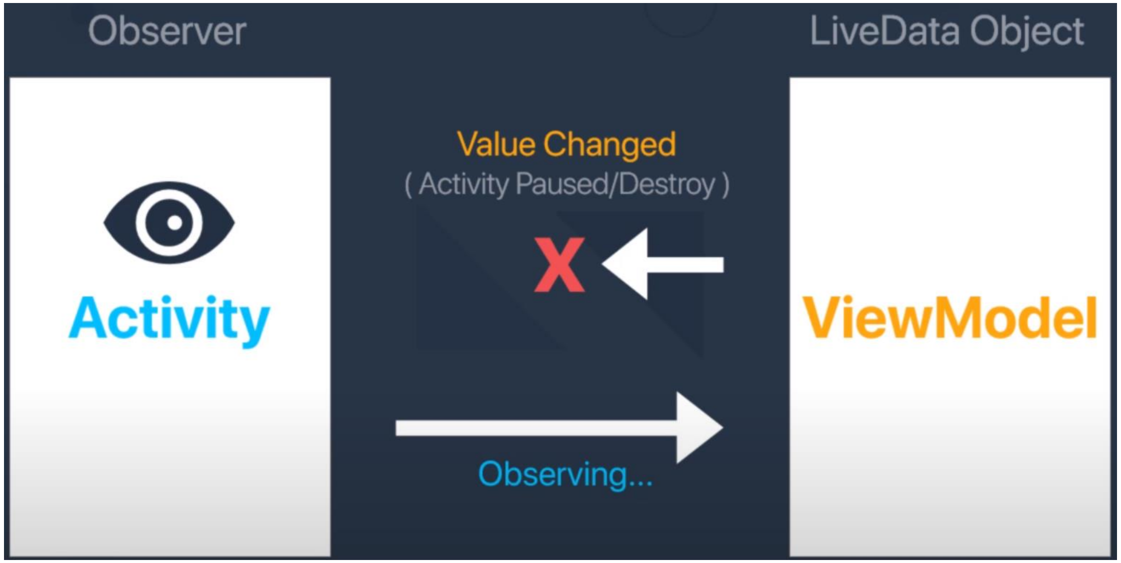


LiveData is lifecycle aware

LiveData is aware of the Lifecycle of its observer

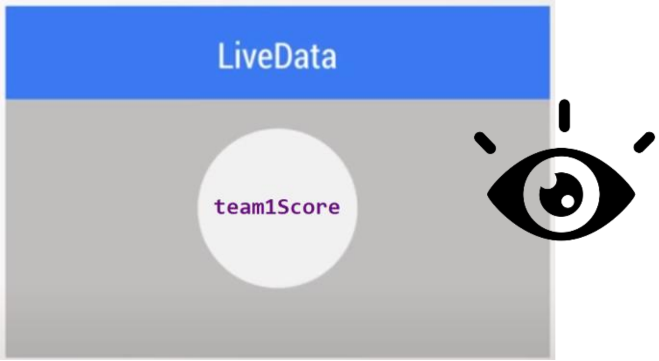
- Notifies data changes to only **active** observers (Paused/Destroyed activity/fragment will NOT receive updates)
- It automatically removes the subscription when the observer is destroyed





LiveData in Code

Warps around an object and allows the UI to automatically update whenever the properties of the wrapped object change



- Create LiveData object

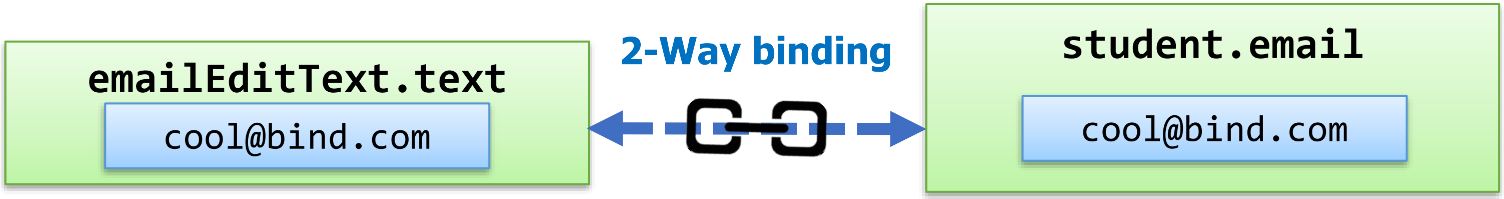
```
class MainActivityViewModel : ViewModel() {
    val team1Score = MutableLiveData<Int>(0)

    fun incrementTeam1Score() =
         team1Score.postValue(team1Score.value?.inc())
}
```

- Observe data changes

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

Data Binding



Excel is an excellent example …

< Back

Data Binding

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

- 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

- Enabling data binding (app/build.gradle)

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

- To use data binding in a layout file, you have to make the file a data binding layout file. You do that by wrapping the entire XML file in a **<layout>** tag.

Inflating Layout with Bindings

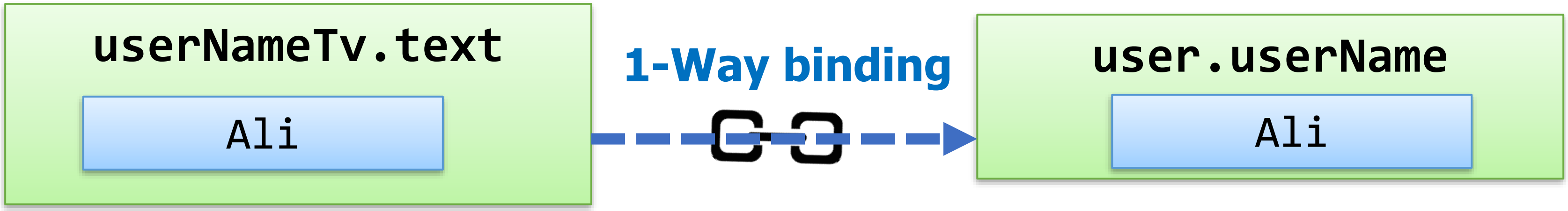
- onCreate in MainActivity use **DataBindingUtil** to inflate an instance of the generated binding class ActivityMainBinding

- Binding to LiveData can trigger UI updates when the data changes

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}" />
```

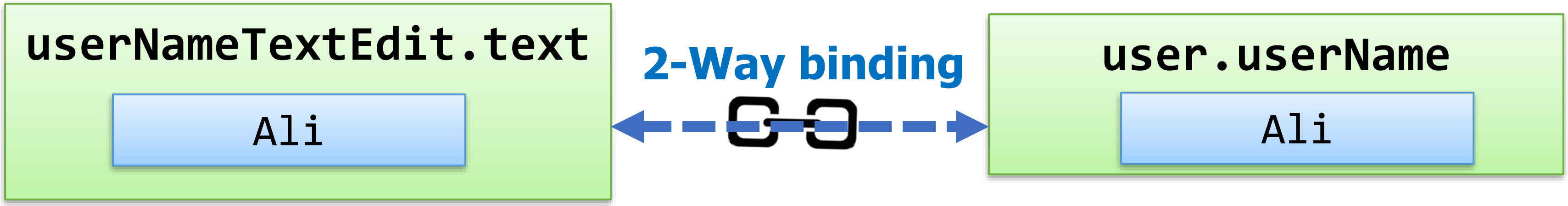


Bidirectional Binding

- Bidirectional (2-Way) Binding

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

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



userNameTextEdit.text

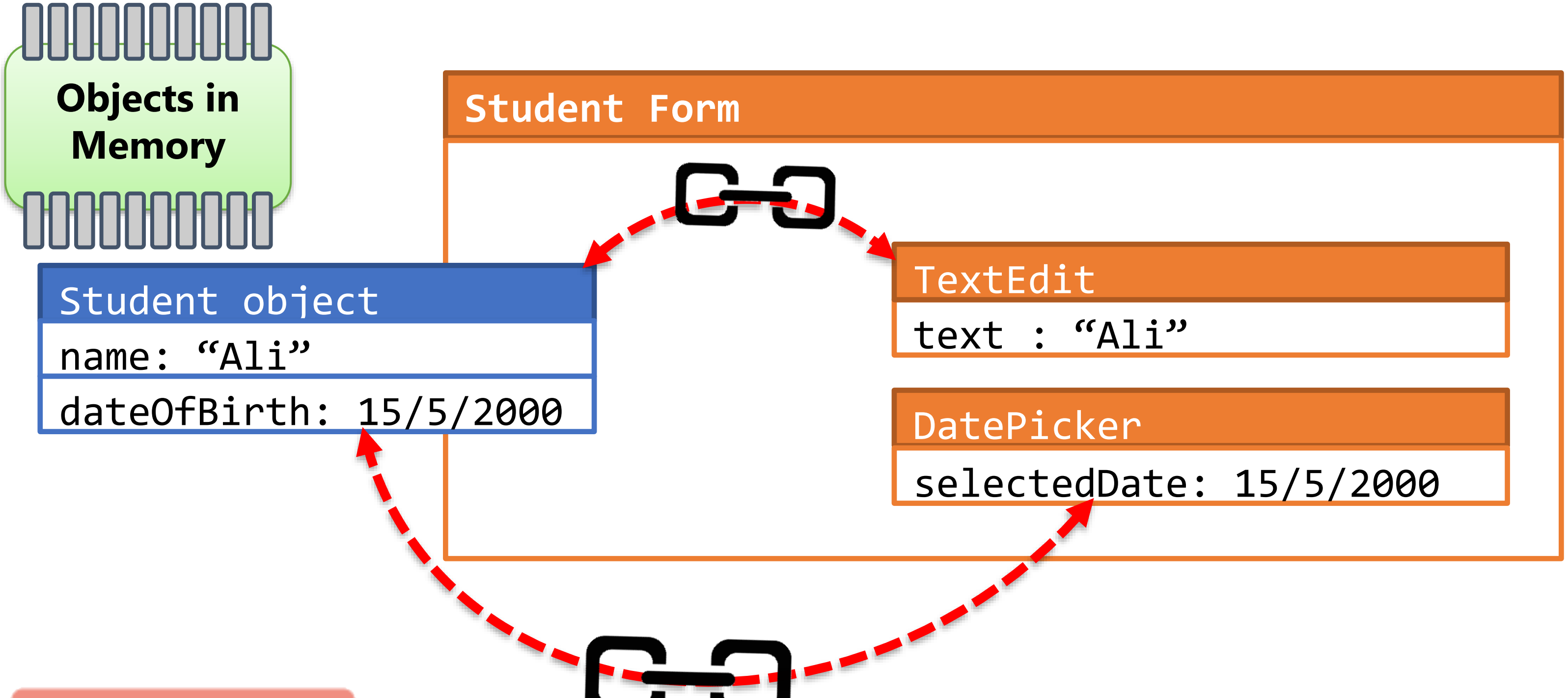
Ali

2-Way binding

user.userName

Ali

Two-way Binding UI Components Properties with Object Properties



Student object

name: “Ali”

dateOfBirth: 15/5/2000

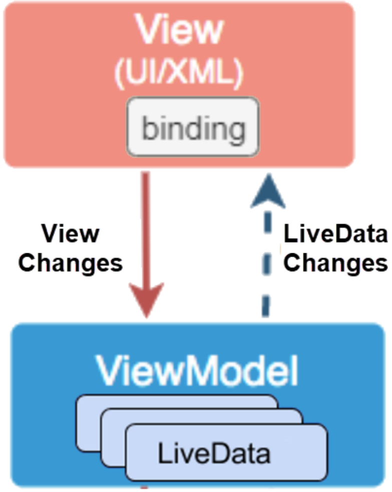
Student Form

TextEdit

text : “Ali”

DatePicker

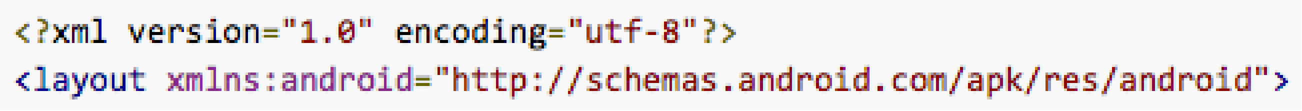
selectedDate: 15/5/2000



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

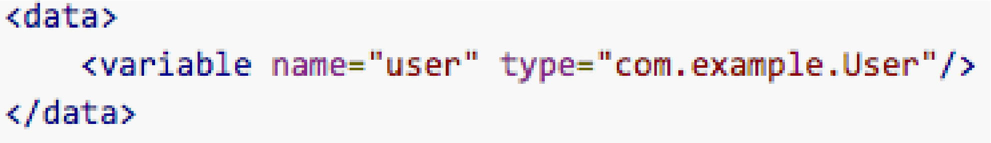
Data Binding – basic example

- Change your layout root



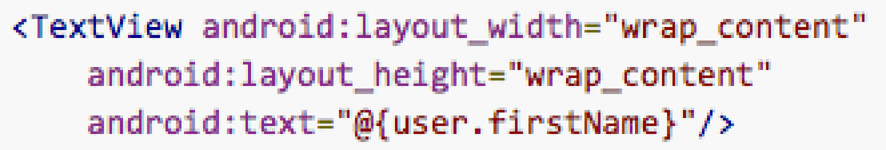
```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
```

- Introduce a *<data>* element



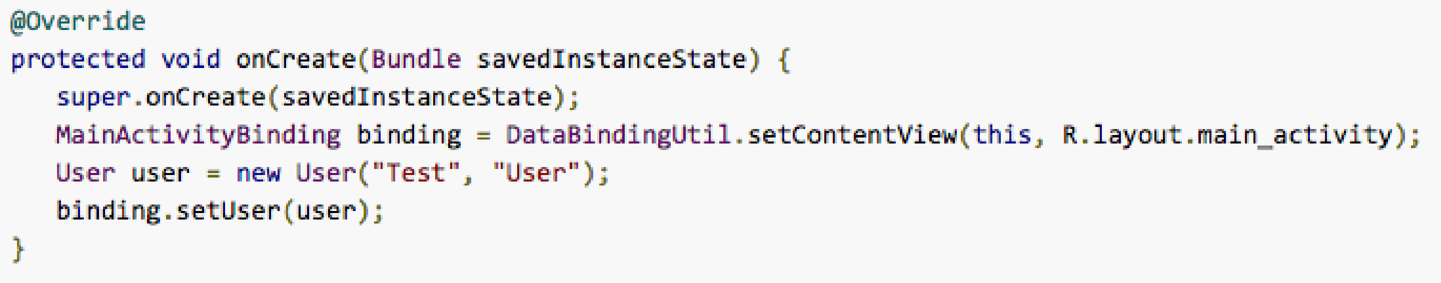
```
<data>
    <variable name="user" type="com.example.User"/>
</data>
```

- Connect your widgets to the data



```
<TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{user.firstName}"/>
```

- Connect the binding in the Activity



```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   MainActivityBinding binding = DataBindingUtil.setContentView(this, R.layout.main_activity);
   User user = new User("Test", "User");
   binding.setUser(user);
}
```

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