ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Lesson 8: App architecture (UI layer)

About this lesson

Lesson 8: App architecture (UI layer)

Android app architecture

Avoid short-term hacks

- External factors, such as tight deadlines, can lead to poor decisions about app design and structure.
- Decisions have consequences for future work (app can be harder to maintain long-term).
- Need to balance on-time delivery and future maintenance burden.

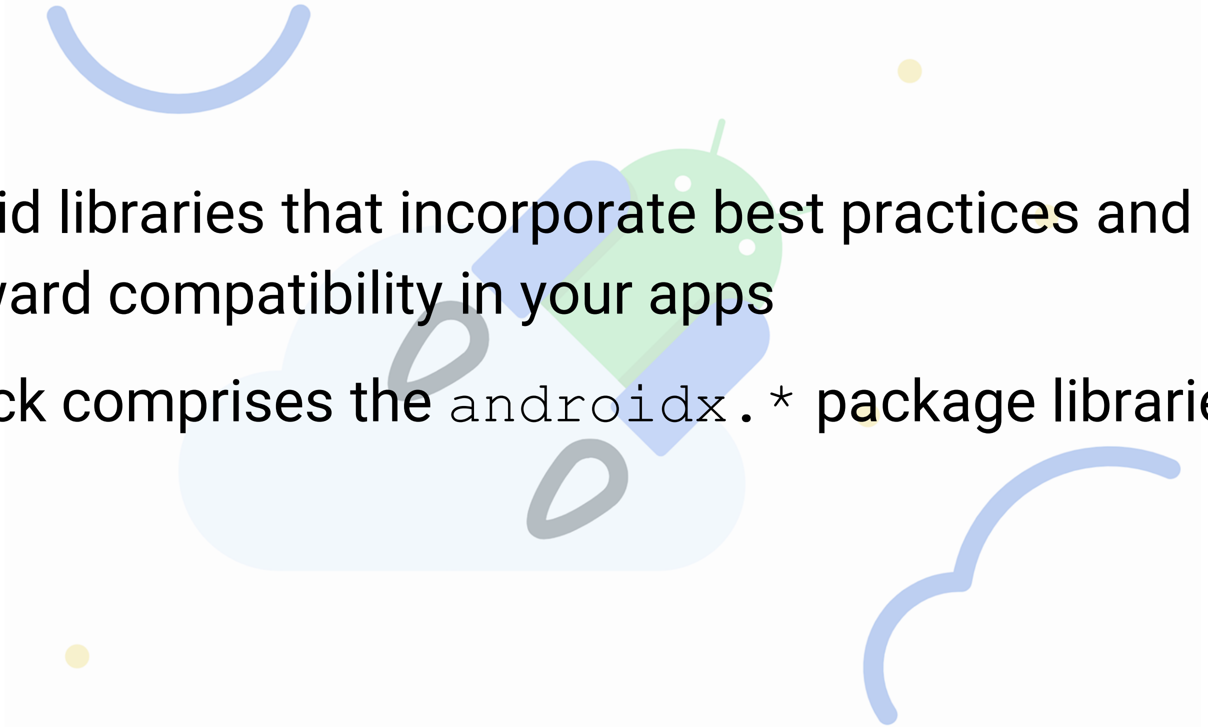
Examples of short-term hacks

- Tailoring your app to a specific device
- Blindly copying and pasting code into your files
- Placing all business logic in activity file
- Hardcoding user-facing strings in your code

Why you need good app architecture

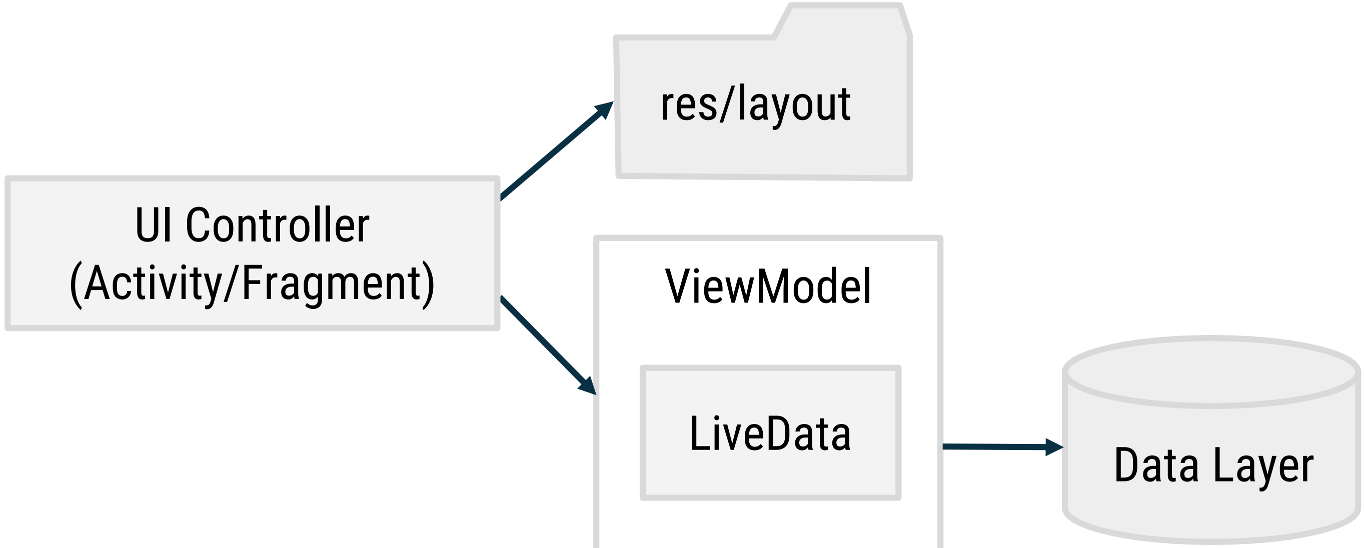
- Clearly defines where specific business logic belongs
- Makes it easier for developers to collaborate
- Makes your code easier to test
- Lets you benefit from already-solved problems
- Saves time and reduces technical debt as you extend your app

Android Jetpack



- Android libraries that incorporate best practices and provide backward compatibility in your apps
- Jetpack comprises the `androidx.*` package libraries

Separation of concerns



Architecture components

- Architecture design patterns, like MVVM and MVI, describe a loose template for what the structure of your app should be.
- Jetpack architecture components help you design robust, testable, and maintainable apps.

ViewModel

Gradle: lifecycle extensions

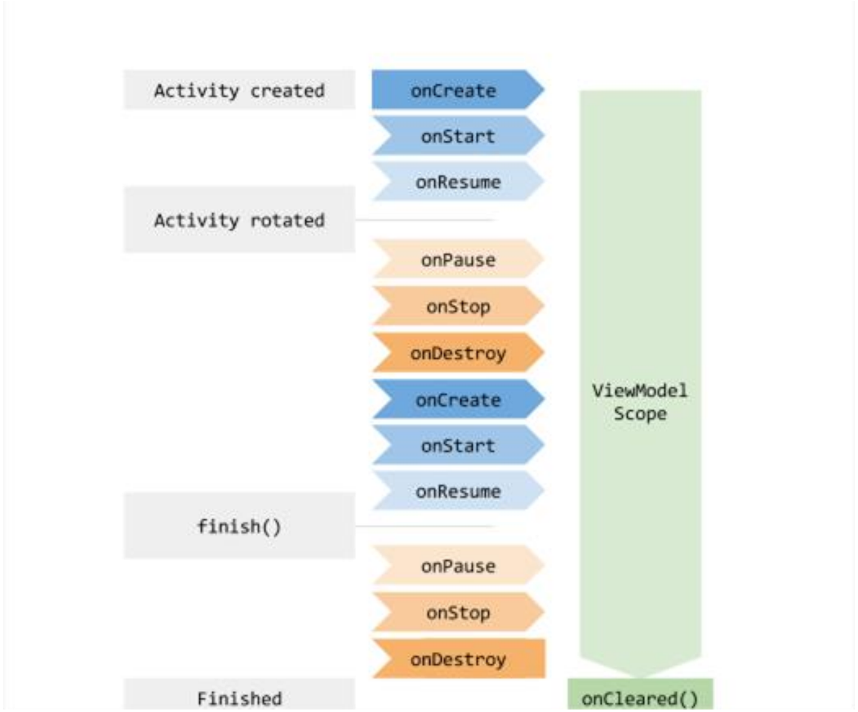
In `app/build.gradle` file:

```
dependencies {
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.activity:activity-ktx:$activity_version"
}
```

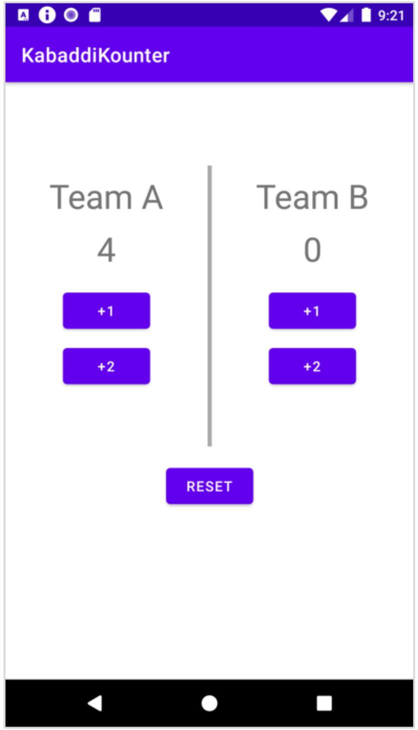
ViewModel

- Prepares data for the UI
- Must not reference activity, fragment, or views in view hierarchy
- Scoped to a lifecycle (which activity and fragment have)
- Enables data to survive configuration changes
- Survives as long as the scope is alive

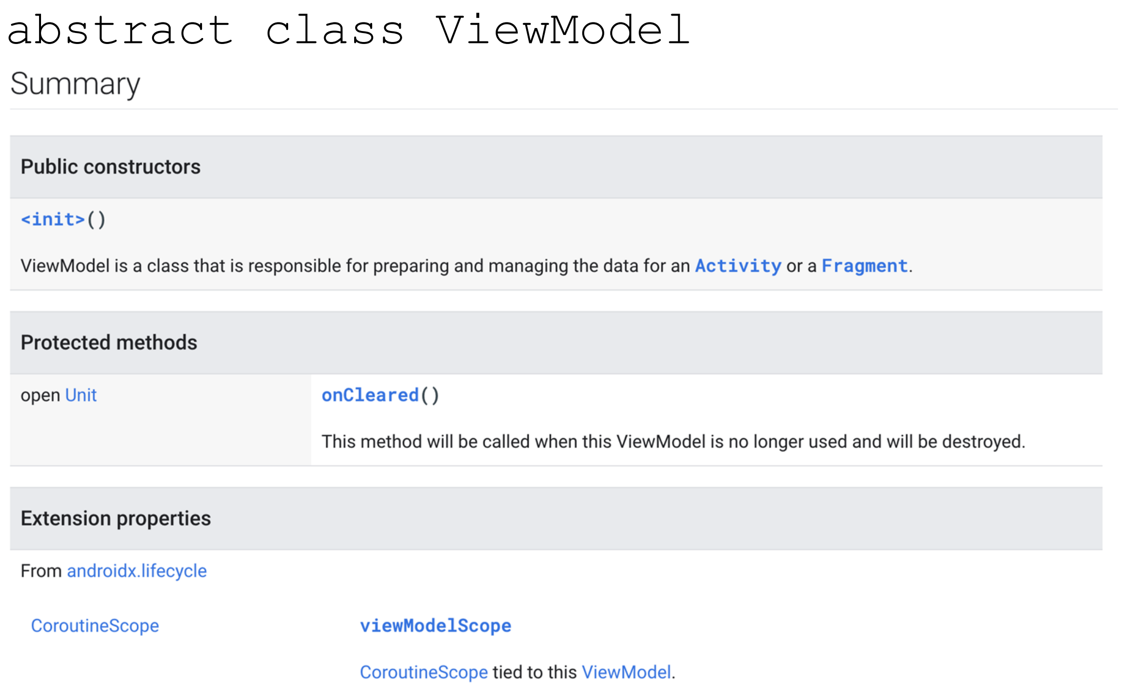
Lifetime of a ViewModel



Kabaddi Kounter



ViewModel class



```
abstract class ViewModel
```

Summary

Public constructors
`<init>()` ViewModel is a class that is responsible for preparing and managing the data for an `Activity` or a `Fragment`.

Protected methods	
open Unit	`onCleared()` This method will be called when this ViewModel is no longer used and will be destroyed.

Extension properties

From androidx.lifecycle

CoroutineScope	`viewModelScope` CoroutineScope tied to this ViewModel.

Implement a ViewModel

```kotlin
class ScoreViewModel : ViewModel() {
    var scoreA : Int = 0
    var scoreB : Int = 0
    fun incrementScore(isTeamA: Boolean) {
        if (isTeamA) {
            scoreA++
        }
        else {
            scoreB++
        }
    }
}
```

Load and use a ViewModel

```kotlin
class MainActivity : AppCompatActivity() {
    // Delegate provided by androidx.activity.viewModels
    val viewModel: ScoreViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        val scoreViewA: TextView = findViewById(R.id.scoreA)
        scoreViewA.text = viewModel.scoreA.toString()
    }
```

Using a ViewModel

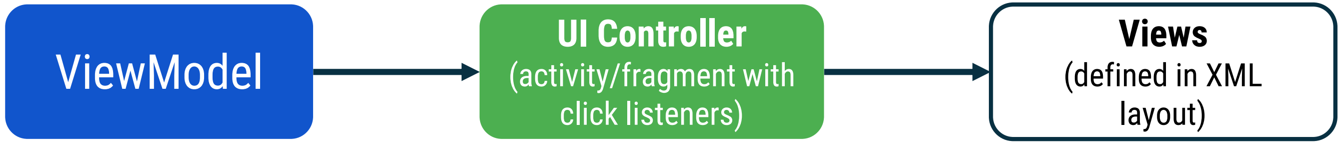
Within `MainActivity onCreate()`:

```
val scoreViewA: TextView = findViewById(R.id.scoreA)
val plusOneButtonA: Button = findViewById(R.id.plusOne_teamA)

plusOneButtonA.setOnClickListener {
    viewModel.incrementScore(true)
    scoreViewA.text = viewModel.scoreA.toString()
}
```

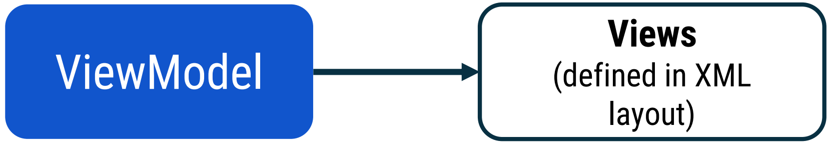
Data binding

ViewModels and data binding

- App architecture without data binding



- ViewModels can work in concert with data binding



Data binding in XML revisited

Specify ViewModels in the `data` tag of a binding.

```
<layout>
   <data>
       <variable>
           name="viewModel"
           type="com.example.kabaddikounter.ScoreViewModel" />
   </data>
   <ConstraintLayout ../>
</layout>
```

Attaching a ViewModel to a data binding

```
class MainActivity : AppCompatActivity() {

    val viewModel: ScoreViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding: ActivityMainBinding = DataBindingUtil.setContentView(this,
            R.layout.activity_main)

        binding.viewModel = viewModel

        ...
```

Using a ViewModel from a data binding

In `activity_main.xml`:

```
<TextView
    android:id="@+id/scoreViewA"
    android:text="@{viewModel.scoreA.toString()}" />

        ...
```

ViewModels and data binding

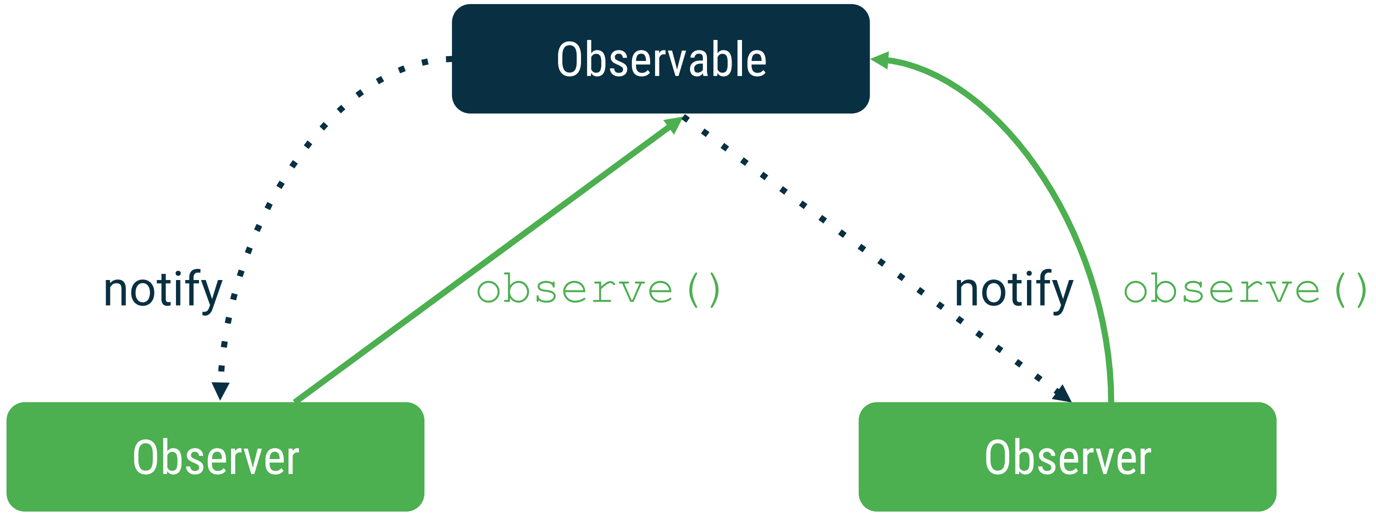
```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
   val binding: ActivityMainBinding = DataBindingUtil.setContentView(this,
        R.layout.activity_main)

   binding.plusOneButtonA.setOnClickListener {
       viewModel.incrementScore(true)
       binding.scoreViewA.text = viewModel.scoreA.toString()
   }
}
```

LiveData

Observer design pattern

- Subject maintains list of observers to notify when state changes.
- Observers receive state changes from subject and execute appropriate code.
- Observers can be added or removed at any time.

Observer design pattern diagram



LiveData

- A lifecycle-aware data holder that can be observed
- Wrapper that can be used with any data including lists (for example, `LiveData<Int>` holds an `Int`)
- Often used by ViewModels to hold individual data fields
- Observers (activity or fragment) can be added or removed
  - `observe(owner: LifecycleOwner, observer: Observer)`
    `removeObserver(observer: Observer)`

LiveData versus MutableLiveData

`LiveData<T>`	`MutableLiveData<T>`
• `getValue()`	• `getValue()` • `postValue(value: T)` • `setValue(value: T)`

`T` is the type of data that's stored in `LiveData` or `MutableLiveData`.

Use LiveData in ViewModel

```
class ScoreViewModel : ViewModel() {

    private val _scoreA = MutableLiveData<Int>(0)
    val scoreA: LiveData<Int>
        get() = _scoreA

    fun incrementScore(isTeamA: Boolean) {
        if (isTeamA) {
            _scoreA.value = _scoreA.value!! + 1
        }
        ...
```

Add an observer on LiveData

Set up click listener to increment `ViewModel` score:

```
binding.plusOneButtonA.setOnClickListener {
    viewModel.incrementScore(true)
}
```

Create observer to update team A score on screen:

```
val scoreA_Observer = Observer<Int> { newValue ->
    binding.scoreViewA.text = newValue.toString()
}
```

Add the observer onto `scoreA LiveData` in `ViewModel`:

```
viewModel.scoreA.observe(this, scoreA_Observer)
```

Two-way data binding

- We already have two-way binding with `ViewModel` and `LiveData`.
- Binding to `LiveData` in XML eliminates need for an observer in code.

Example layout XML

```
<layout>
   <data>
       <variable>
           name="viewModel"
           type="com.example.kabaddikounter.ScoreViewModel" />
   </data>

   <ConstraintLayout ..>
       <TextView ...
           android:id="@+id/scoreViewA"
           android:text="@{viewModel.scoreA.toString()}" />
       ...
   </ConstraintLayout>
</layout>
```

Example Activity

```
class MainActivity : AppCompatActivity() {
    val viewModel: ScoreViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val binding: ActivityMainBinding = DataBindingUtil
             .setContentView(this, R.layout.activity_main)

        binding.viewModel = viewModel
        binding.lifecycleOwner = this

        binding.plusOneButtonA.setOnClickListener {
            viewModel.incrementScore(true)
        }
        ...
```

Example ViewModel

```kotlin
class ScoreViewModel : ViewModel() {
    private val _scoreA = MutableLiveData<Int>(0)
    val scoreA : LiveData<Int>
        get() = _scoreA
    private val _scoreB = MutableLiveData<Int>(0)
    val scoreB : LiveData<Int>
        get() = _scoreB
    fun incrementScore(isTeamA: Boolean) {
        if (isTeamA) {
            _scoreA.value = _scoreA.value!! + 1
        } else {
            _scoreB.value = _scoreB.value!! + 1
        }
    }
}
```

Transform LiveData

Manipulating LiveData with transformations

`LiveData` can be transformed into a new `LiveData` object.

- `map()`
- `switchMap()`

Example LiveData with transformations

```kotlin
val result: LiveData<String> = Transformations.map(viewModel.scoreA) {
    x -> if (x > 10) "A Wins" else ""
}
```

Summary

Summary

In Lesson 8, you learned how to:

- Follow good app architecture design, and the separation-of-concerns principle to make apps more maintainable and reduce technical debt
- Create a `ViewModel` to hold data separately from a UI controller
- Use `ViewModel` with data binding to make a responsive UI with less code
- Use observers to automatically get updates from `LiveData`

Learn More

- Guide to app architecture
- Android Jetpack
- ViewModel Overview
- Android architecture sample app
- ViewModelProvider
- Lifecycle Aware Data Loading with Architecture Components
- ViewModels and LiveData: Patterns + AntiPatterns

Pathway

Practice what you've learned by completing the pathway:

[Lesson 8: App architecture (UI layer)]