

# Mobile Computing Managing State on Android

CC BY-SA 4.0, T. Amberg, FHNW; Based on  
CC BY 2.5, Android Open Source Project  
Slides: [tmb.gr/mc-mst](http://tmb.gr/mc-mst)



# Overview

These slides show how to manage state on Android.

How application UI state can be saved in Compose.

How to use a ViewModel to move data/logic away.

# Prerequisites

Have some basic knowledge of [writing Kotlin code](#).

Finish the lesson on [composing UIs for Android](#).

Bring your Android device or use the emulator.

# Compose is Kotlin

[.kt|.html](#)

Compose uses nested function calls and Kotlin idioms.

```
fun Row(  
    modifier: Modifier = Modifier, ...,  
    content: @Composable RowScope.() -> Unit)
```

```
Row(..., { ... }) // content param is a lambda  
Row(...) { ... } // trailing lambda, outside ()  
Row() { ... } // default params can be omitted  
Row { ... } // empty constructor, OK to omit ()
```

# Conditional UI

[.kt](#) | [.html](#)

Use conditionals (*if*, etc.), to show/hide UI elements.

```
if (newToThis) { Onboarding() } else { App() }
```

A multi-page UI could work like this, using *when*.

```
when (page) {  
    1 -> ScreenA(...) // calls page++  
    2 -> ScreenB(...) // calls page++ or page--  
    else ScreenC(...) }
```

# Button onClick event

[.html](#)

*Button* provides a *onClick* event, to plug in a lambda.

```
@Composable
```

```
fun MyCounter() {  
    var n = remember { mutableStateOf(0) } /*  
    Button(onClick = { n.value++ }) {  
        Text("${n.value}")  
    }  
}
```

\*Value stays around, like a *static* local variable in C.

# Mutable state

[.html](#)

Compose updates the UI, if underlying data changes, *mutableStateOf()* provides plumbing needed for this.

```
val state = x // does not notify on changes  
val state = mutableStateOf(x) // not stored
```

Functions can be (re)evaluated any time, in any order, *remember()* preserves the state across recomposition.

```
val state = remember { mutableStateOf(x) }
```

# Saveable state

[.html](#)

The *remember()* function works as long as the Activity.

```
val s = remember { mutableStateOf(x) }
```

On rotate\*, the Activity restarts and the state is lost.

Use *rememberSaveable()* instead, to persist state.

```
var s = rememberSaveable { mutableStateOf(x) }
```

\*Or when using dark mode or restarting the process.



# Hoisting state

[.kt](#) | [.html](#)

Hoisting is about where in the UI tree to place state.

Move state up to a common ancestor of who needs it, pass callbacks/lambdas down, to bubble events up.

```
Post(likeCount: Int, onLike: () -> Unit) {  
    Text(text = "$likeCount")  
    Button(onClick = onLike) { Text("+1") } }  
  
var likes ...; Post(likes, onLike = { likes++ })
```

# Hands-on, 10': State in Compose

Fix state and logic, *commit* and *push* changes.

- Update your private repository (see [these slides](#)).
- Open the *MyStatefulApp* in your repository /02, it implements a multi-page UI [as sketched](#) (p. 14)
- Use lambdas to update *page*, *onNext/onBack*.
- Make sure that *MultiPage* remembers its state.
- Try changing the screen orientation of the device.

# UI update loop

[.kt](#) | [.html](#)

Events notify the code that e.g. a click has *happened*.

Handling events leads to the UI state being *updated*.

Which leads to the updated UI state being *displayed*.

Upon seeing the updated UI state, the user reacts.\*

\*Events can be triggered by the system, as well.

# Lazy list components

[.kt](#) | [.html](#)

A *LazyColumn* or *LazyRow* allows 1000+ items, by creating visible items lazily, when scrolling the list.

```
import androidx.compose.foundation.lazy.items
```

```
LazyColumn(modifier = ...) { // or LazyRow
    items(items = names) { name ->
        Greeting(name = name)
    }
}
```

# Mutable observable lists

[.kt](#) | [.html](#)

*toMutableStateList()* makes mutable lists observable\*.

```
val list = listOf(...).toMutableStateList()
```

This works with *remember()*, but it's not ...*Saveable()*.

Also, large data objects should not be stored in the UI.

\*By the Compose framework, to update the UI.

# ViewModel overview

[.html](#)

A *ViewModel* exposes application state to the UI, it encapsulates business logic, caches and persists state. This relieves the UI from having to re-fetch data when navigating between activities, or rotating the screen. It stays around as long as its *ViewModelStoreOwner*, e.g. an activity, a UI fragment or a navigation graph.

# ViewModel implementation [.kt|.html](#)

To implement a *ViewModel* create a new subclass of it.

```
class MyViewModel : ViewModel() {  
    val people = listOf(Person("A"), ...) // data  
    fun remove(person: Person) { ... } // logic  
}
```

Usually the data itself is kept in a custom data class.

```
data class Person(val name: String)
```

# ViewModel instance

[.kts](#) | [.kt](#) | [.html](#)

The *viewModel()* call returns an existing instance of your *ViewModel* or creates a new one in that scope.

```
fun MyUI(myVM: MyViewModel = viewModel()) { ...
```

If you add this to *app/build.gradle.kts* dependencies.

```
implementation("androidx.lifecycle:lifecycle-  
viewmodel-compose:{latest_version}")
```



# ViewModel caveats

[.html](#)

The *ViewModel* is not a part of the UI composition.

Do not hold state created in composables, like a value to be remembered, as this could cause memory leaks.

Also, no references to lifecycle-related classes, such as the app *Context* or *Resources*, for the same reason.

# ViewModel best practice

[.html](#)

Use *ViewModels* as a screen level state holder, not for reusable UI parts, because instances are not reusable.

Keep the names of the methods a *ViewModel* exposes and those of the UI state fields as generic as possible.

Do not pass a *ViewModel* to other classes, functions or UI components to prevent access by lower-level code.

# ViewModel benefits

[.kt](#) | [.html](#)

TODO\*.

TODO( )

\*TODO.

# Hands-on, 10': ViewModel

Extend the code, *commit* and *push* changes.

- Open the *MyViewModelApp* in your repository /02 which implements a *ViewModel* as sketched before.
- Add a property surname to the *Person* data class.
- Make sure, the property is available in *Greeting*.
- If *show more* was clicked, display the full name.
- Add a *remove()* function to the *ViewModel* class.

# Summary

These are the basics of managing state on Android.

Remembering saveable UI state in the composition.

Moving state up (hoisting) and forwarding events.

Using a ViewModel to separate data/logic from UI.

Next: Storing Data on Android.

# ~~Challenge: Live data~~

Work through the [Jetpack Compose TODO codelab](#).

- Start from this [BasicTODOCodelab app project](#).
- Add the *project files* to your private repository.
- Make sure *not* to add the 3rd-party *repository*.
- Git *commit* and *push* your code to your repo.

Done? There are more [codelabs](#), e.g. [on TODO](#).

# Feedback or questions?

Write me on Teams or email

[thomas.amberg@fhnw.ch](mailto:thomas.amberg@fhnw.ch)

Thanks for your time.