Mobile Computing
Composing UIs for Android

Slides: tmb.gr/mc-uis

Source on GDocs

Overview

These slides introduce the *Compose* UI toolkit.

How to create a user interface from components.

How to write app-specific, composable functions.

Prerequisites

Have some basic knowledge of writing Kotlin code.

Finish the lesson on getting started with Android.

Bring your Android device or use the emulator.

Jetpack Compose .html

Jetpack Compose is a toolkit for UI development.

Specific UIs are composed in a declarative style.

The code describes what to achieve, not how*.

*It nevertheless is valid Kotlin code.

Material Design .html

Material Design is a design system made by Google.

A set of guidelines and components for good UI/UX.

To use them, import *androidx.compose.material3*.*

They are based on *androidx.compose.foundation*.

*There are multiple versions, M1, M2 and M3.

@Composable

.kt|.html

Annotate a function without return as *@Composable* to turn it into a custom, composable UI component.

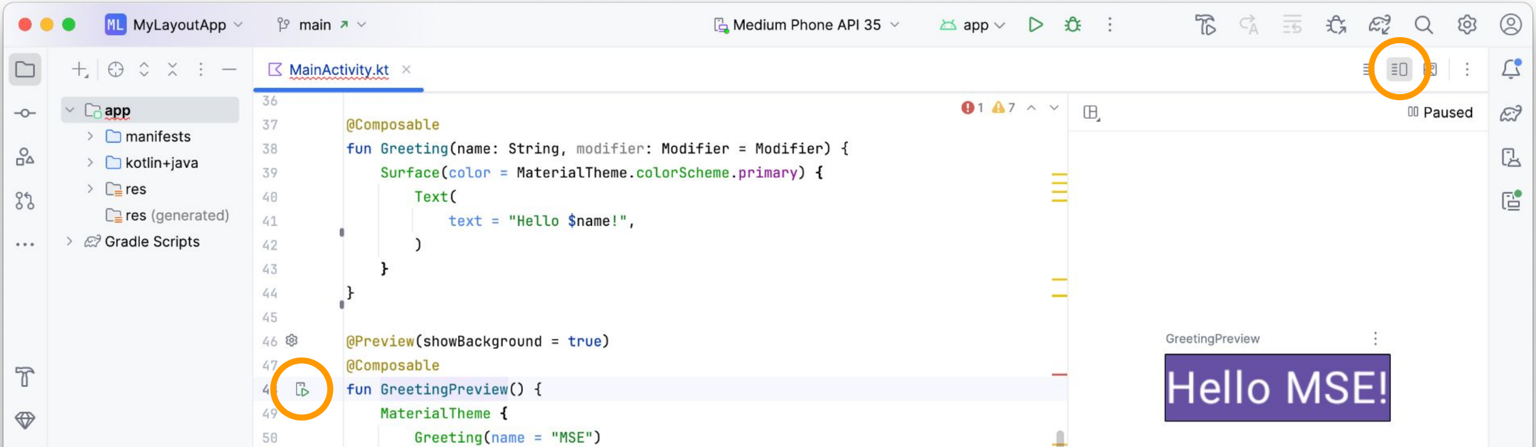
```
@Composable // functions are nouns, PascalCase
fun Greeting(name: String, …) {
  Surface(color = MaterialTheme.….primary) {
    Text(text = "Hello $name!")
  }
}
```

@Preview

.kt|.html

@Preview allows to render a specific component.

- Click the *run* icon next to a *@Preview* function.
- Make sure the *preview* (not emulator) is visible.



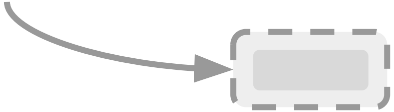
```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Surface(color = MaterialTheme.colorScheme.primary) {
        Text(
            text = "Hello $name!",
        )
    }
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MaterialTheme {
        Greeting(name = "MSE")
```

Modifiers

.kt|.html

Modifiers tell a UI element like *Text* or *Surface* how to lay out, display, or behave within its parent layout.

```
fun Greeting(…, m: Modifier = Modifier) { …
  Text(modifier = m.padding(24.dp)) { … }
}
```



Modifiers can be chained, the call order matters.

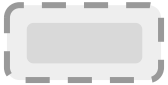
```
  modifier = m.padding(24.dp).fillMaxWidth()
```

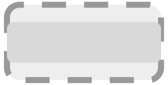
Padding

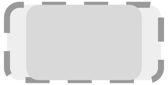
.kt|.html

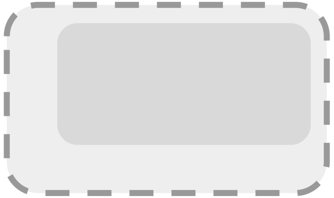
The padding modifier includes these variants.

```
Modifier.padding(all = 24.dp)          //
Modifier.padding(vertical = 24.dp)     //
Modifier.padding(horizontal = 24.dp) //
Modifier.padding(                    //
  start = 24.dp, top = 8.dp,         //
  end = 8.dp, bottom = 24.dp)        //
```







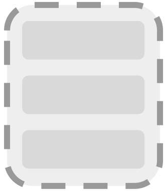
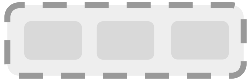
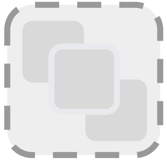


Columns and rows

.png|.kt|.html

Column, *Row*, *Box* are basic layouts, can be nested.

```
@Composable
fun Greetings(names: List<…>, m: Modifier) {
  Column(m) { // or Row(m) or Box(m)
    for (name in names) { // for, etc. is fine
      Greeting(name) // composable component
    }
  }
}
```

Col. , Row , Box 

Align- and arrangement .gif|.kt|.html

To set children component's positions within a *Row*, set *horizontalArrangement* and *verticalAlignment*.

```
horizontalArrangement = Arrangement.spaceBy(…),
verticalAlignment = Alignment.CenterVertically
```

For a *Column*, set *h…Alignment*/*v…Arrangement*.

```
h…Alignment = Alignment.CenterHorizontally,
v…Arrangement = Arrangement.spacedBy(8.dp)
```

Layout model .html

In the layout model, the UI tree is laid out in one pass.

```
state → composition → layout → drawing → UI
```

Parent elements/components measure themselves before, but are sized and placed after their children.

```
Column() {     // Column [1 measured][6 sized]
  Greeting()   //   Greeting [2 measured, 3 sized]
  Greeting() } //   Greeting [4 measured, 5 sized]
```

Hands-on, 10': Layout in Compose

Add composables, *commit* and *push* changes.

- Update your private repository (see these slides).
- Open the *MyLayoutApp* in your repository */02*.
- Check out the *TODOs*, and run/re-run the app.
- Create *GridGreetings* class and its *Preview*.
- Arrange Greetings in a full size 2 x 2 Grid*.

*Add a name that fits the others.

Using conditionals

.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 i = 0 // remember { mutableStateOf(0) }
  Button(onClick = { i.value++ }) {
    Text("${i.value}")
  } }
```

*Uncomment *remember*, etc. to make it work.

Managing state .html

Compose updates the UI, if underlying data changes, *mutableState()* provides the 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) }
```

Hoisting state

.kt|.html

Move state up to a common ancestor of who needs it.

Pass callbacks/lambdas down, to bubble events up.

```
@Composable
fun OnboardingScreen(
  onContinueClicked: () -> Unit, …) { …
  Button(onClick = onContinueClicked) {
    Text("Continue")
  } }
```

Persisting state

.html

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

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

On rotate*, the Acitivty restarts and the state is lost.

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

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

*Or when using dark mode or killing the process.

Hands-on, 10': State in Compose

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

- 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.

*Move state up, pass lambdas down.

Theming

.png|.kt|.html

Theming allows adapting color schemes, typography and shapes, to customise or personalise app design.

```
@Composable
fun MyAppTheme(…) { …
    MaterialTheme(colorScheme = …, // dynamic
        typography = Typography, // readable
        content = content // of composable
    )
}
```

Summary

These are the basics of using the Compose UI toolkit.

Creating a user interface from composable functions.

Using modifiers, a layout, state, logic and persistence.

Challenge: Implement a "real" design

Work through the Jetpack Compose Layouts codelab.

- Start from this BasicLayoutsCodelab 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 theming.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.