



Lab on apps development for tablets, smartphones and smartwatches

Week 2: State and interactivity

Giovanni Ansaloni

Rafael Medina, Hossein Taji, Yuxuan Wang
Qunyou Liu, Amirhossein Shahbazzinia, Christodoulos Kechris

School of Engineering (STI) – Institute of Electrical and Micro Engineering (IEM)



- **Handling events**
 - Buttons and Composables click events
- Managing State in Jetpack Compose
 - State hoisting
- Communicating with other applications
 - Listening for results

Class outline



A white silhouette of a person's head and shoulders against a light blue background.

Username

Password

Confirm Pick image

Kotlin Activity classes

- Activity classes manage the UI and the interaction with user actions
- Activity classes extend ComponentActivity / WearableActivity

simple Activity class

- OnCreate() callback

- call to super
- inflate layout
- inflate theme
- **Interactivity?**



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView {
            MaterialTheme {
                MyApp(modifier = Modifier.fillMaxSize())
            }
        }
    }
}
```

Interactive apps

- User generate **events** by interacting with the GUI → with composables
 - e.g. clicks/longClicks on buttons
- The apps should appropriately respond to user actions
- We specify the app behavior wrt events in Composable functions by specifying **function parameters**
 - Kotlin lambdas/functions attached to events



- Buttons have an `onClick` function parameter
 - defined inside the parentheses

```
Button(onClick = { /*TODO*/ }) {  
  
}
```



ButtonPreview

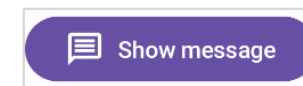


- The button appearance is defined inside of the curly brackets

```
Button(onClick = { /*TODO*/ }) {  
    Image(  
        imageView = Icons.Outlined.Message,  
        contentDescription = null,  
        colorFilter = ColorFilter.tint(Color.White),  
        modifier = Modifier  
            .padding(end = 8.dp)  
    )  
    Text(text = stringResource(R.string.button_text))  
}
```



ButtonPreview



- Implementing onClick (example: toast message)

```

ConstraintLayout(modifier = modifier.fillMaxSize()) { this: ConstraintLayoutScope
    val (button) = createRefs()
    val context = LocalContext.current

    Button(
        onClick = {
            Toast.makeText(
                context,
                context.getString(R.string.toast_message_text),
                Toast.LENGTH_LONG
            ).show()
        },
        modifier = Modifier.constrainAs(button) { this: ConstrainScope
            top.linkTo(parent.top)
            bottom.linkTo(parent.bottom)
            start.linkTo(parent.start)
            end.linkTo(parent.end)
        }
    ) { this: RowScope

```





Making a composable Clickable

- Click event can be added to any composable through modifiers
 - Clickable
 - CombinedClickable
 - also longpress, doublepress, ...

```
Row(  
    modifier  
        .fillMaxWidth()  
        .background(Color.Cyan)  
        .clickable {  
            Toast  
                .makeText(  
                    context,  
                    context.getString(R.string.toast_message_text)  
                    Toast.LENGTH_LONG  
                )  
                .show()  
        }  
)
```

Using a callback functions to handle events

- **Callback** → function called in response to an event
 - defined outside of the composable

@Composable

```
fun Message() {
```

```
    val context = LocalContext.current
```

```
    Row(modifier = Modifier.clickable {
```

```
        clickHandlerFunction(context)
```

```
    }) {
```

```
        ...
```

```
    }
```

```
}
```

Callback



```
fun clickHandlerFunction(context: Context) {
```

```
    Toast.makeText(context,
```

```
        context.getText(R.string.toast_message_text),
```

```
        Toast.LENGTH_LONG)
```

```
    .show()
```

```
}
```



- Handling events
 - Buttons and Composables click events
- Managing State in Jetpack Compose
 - State hoisting
- Communicating with other applications
 - Listening for results

Class outline



A white silhouette of a person's head and shoulders against a light blue background.

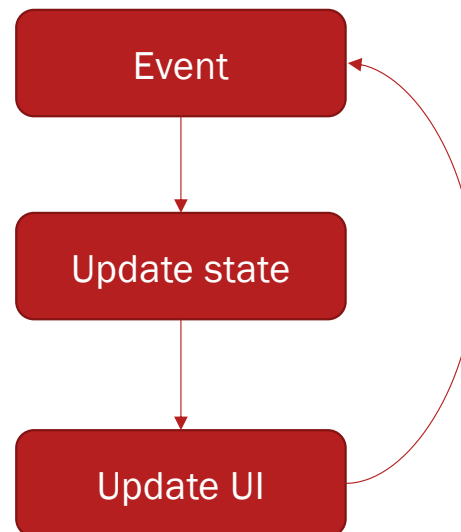
Username

Password

Confirm Pick image

- State in an app is any value that can change over time
 - State determines what is shown in the UI at any particular time
 - State changes trigger **recomposition**
→ redrawing of UI

- Recomposition flow:





MutableState and remember

- **MutableState** types make state observable variables

- can wrap any other type
- changes in observable variables trigger recomposition
- often, state must be **remembered** across recomposition

- Initializing a state object:

```
var count: MutableState<Int> = remember { mutableStateOf( value: 0) }
```

- Updating a state object:

```
Button(onClick = { count.value++ })
```

- Reading a state object:

```
Text(text = "Counter: ${count.value}")
```



Delegated properties

- Using **by** keyword when initializing the state
 - asks the remember API to initialize a mutableState variable

- Initializing a state object:

```
var count by remember { mutableStateOf( value: 0) }
```

- Updating a state object:

```
Button(onClick = { count++ }) {
```

- Reading a state object:

```
Text(text = "Counter: ${count}")
```

more about delegated properties:

kotlinlang.org/docs/delegated-properties.html

*no need to write
".value" when
using the
variable*



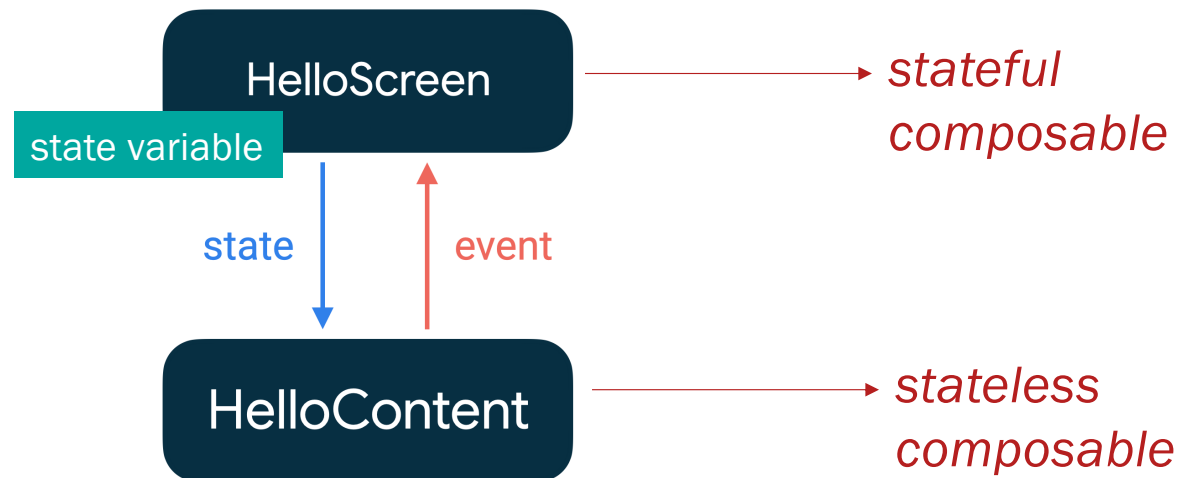
rememberSaveable

- **Remember** retains state across recomposition, not across configuration changes
 - Example: Screen rotation from portrait to landscape
- **rememberSaveable** retains state across recomposition, and across configuration changes
 - Large objects may affect performance
 - Viewmodels are a better alternative in those cases (next lectures)

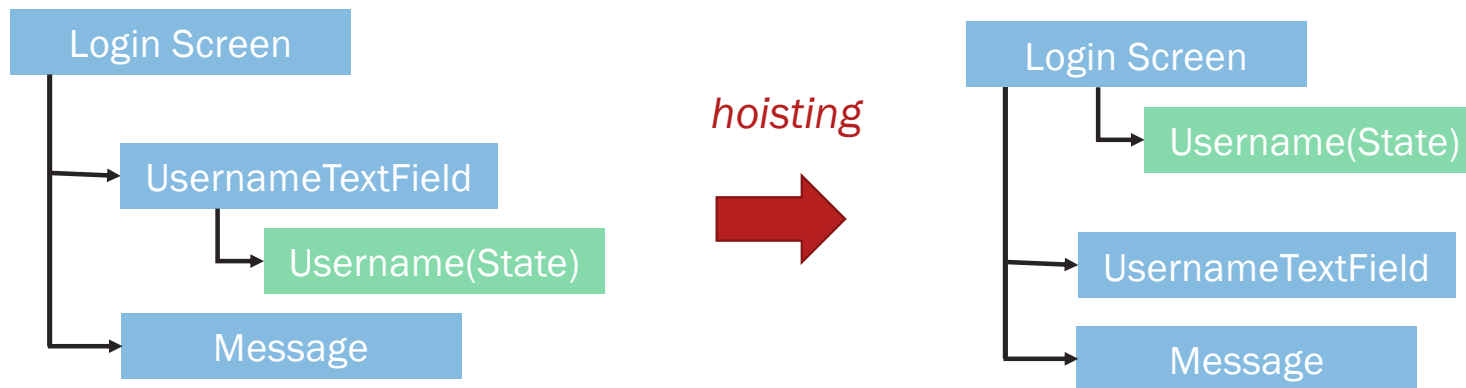
```
var count by rememberSaveable { mutableStateOf( value: 0) }
```

Values and events

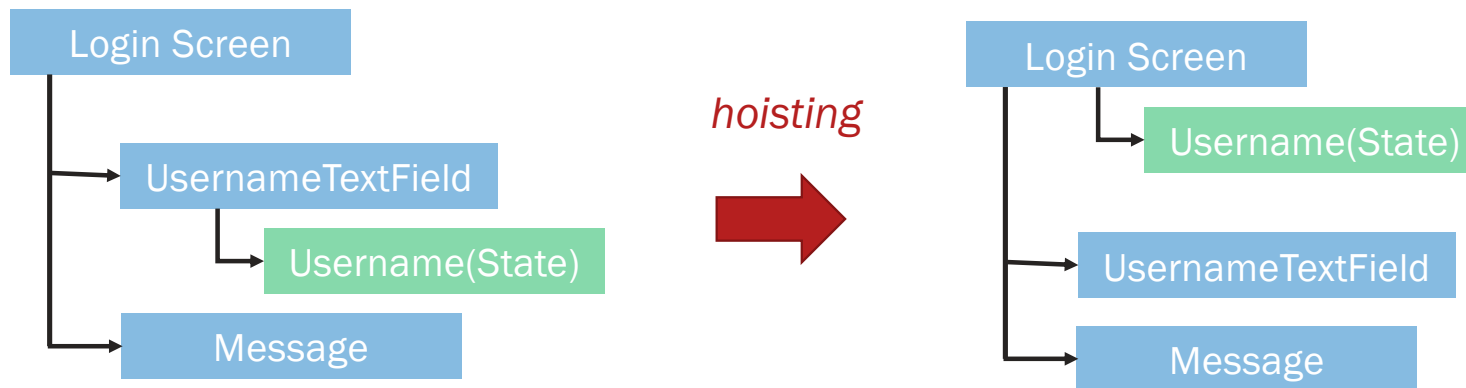
- In Composables hierarchies
 - State **values** are propagated downward
 - State **events** are propagated upward
 - events: everything that may change the value of a state variable



- A State variable should be held by the lowest common ancestor between all composable that write and/or read it
- State hoisting elevates the state variable declarations in a composable hierarchy



- Hoisting replaces the state variable with two parameters in the composable function:
 - **Value** → the current value to display
 - **onEvent()** → function to update the value when an event occurs





■ Example1

```
@Composable
fun LoginScreen() {
    var username by rememberSaveable { mutableStateOf("") }

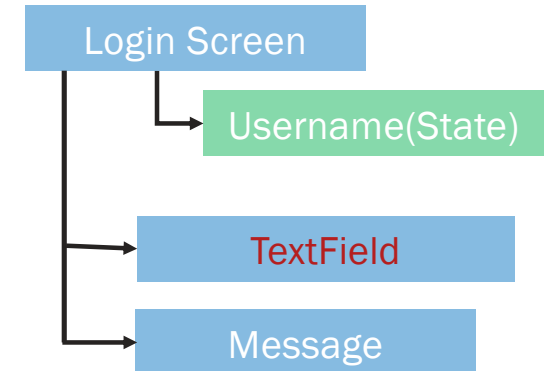
    Column {
        TextField(value = username,
                  onValueChange = { newValue -> username = newValue })
        Text(text = "Picked username: ${username}")
    }
}
```

hosted state variable

value parameter

event parameter

State hoisting





■ Example2

```
@Composable
fun LoginScreen() {
    var username by rememberSaveable { mutableStateOf("") }
```

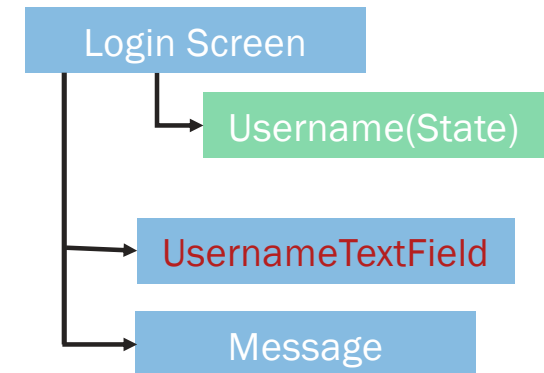
hosted state variable

```
Column {
    UsernameTextField(
        username = username,
        onUsernameChanged = { username = it }
    )
    Text(text = "Picked username: ${username}")
}
```

```
@Composable
fun UsernameTextField(username: String,
    onUsernameChanged: (String) -> Unit) {
    TextField(value = username,
        onValueChange = { newValue ->
            onUsernameChanged(newValue)})
}
```

or: { onUsernameChanged(it) }
or: onUsernameChanged

State hoisting





- Handling events
 - Buttons and Composables click events
- Managing State in Jetpack Compose
 - State hoisting
- **Communicating with other applications**
 - Listening for results

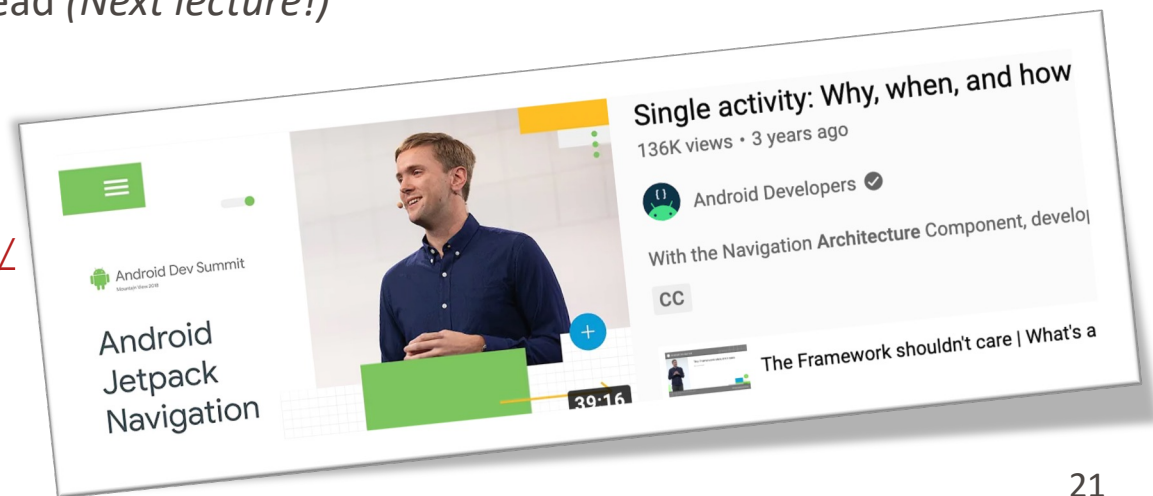
Class outline

A UI mockup for a user profile or login screen. It features a large blue square placeholder for a profile picture at the top. Below this, there are two light purple input fields: the first is labeled 'Username' and the second is labeled 'Password'. At the bottom, there are two rounded purple buttons: 'Confirm' on the left and 'Pick image' on the right.

Interfacing with other applications

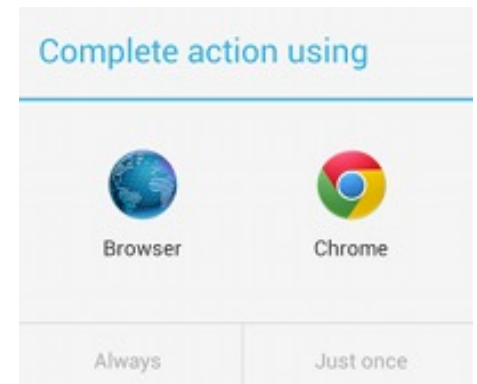
- You can use functionalities from other activities using **Intents**
- Intents: message objects among Android components
- Explicit Intent: message to a specific target
 - Usually employed for communicating among different activities in an App
 - Multi-activity app now discouraged
 - we will use Navigation instead (*Next lecture!*)

<https://www.youtube.com/watch?v=2k8x8V77CrU>



Interfacing with other applications

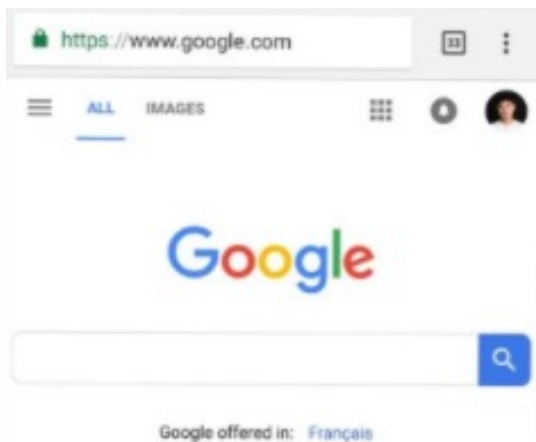
- You can use functionalities from other activities using **Intents**
- Intents: message objects among Android components
- Explicit Intent: message to a specific target
- **Implicit** Intent: the app does not know the recipient
 - only what is must be accomplished:
open a website, start a call, etc...
 - User presented with a chooser if multiple applications can handle a task
- (Much) more on Intents:
<https://developer.android.com/guide/components/intents-filters>



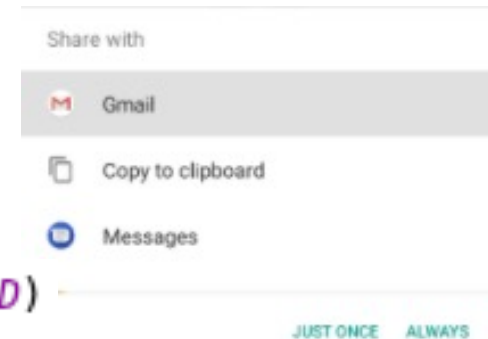
Defining an implicit Intent

- Implicit intent must provide **Action** to be done
 - ACTION_VIEW, ACTION_DIAL, ACTION_SEND
- Depending on the Action, they can also provide
 - A single Data content, specified either as URI or a data Type
 - Multiple EXTRAs specified as key-value pairs

```
val webIntent: Intent = Intent(Intent.ACTION_VIEW)
webIntent.setData(Uri.parse("https://www.google.com"))
```



```
val shareIntent = Intent(ACTION_SEND)
shareIntent.setType("text/plain")
shareIntent.putExtra("KEY", "VALUE")
```



Intents examples

- Make a phone call

```
val callIntent: Intent = Intent(Intent.ACTION_DIAL)
callIntent.setData(Uri.parse("tel:5551234"))
```



- Send an e-mail

```
val mailIntent: Intent = Intent(Intent.ACTION_SEND)
mailIntent.setType("text/plain")
mailIntent.putExtra(Intent.EXTRA_EMAIL, "jan@example.com")
mailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject")
mailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text")
mailIntent.putExtra(Intent.EXTRA_STREAM,
Uri.parse("content://path/to/email/attachment"))
```



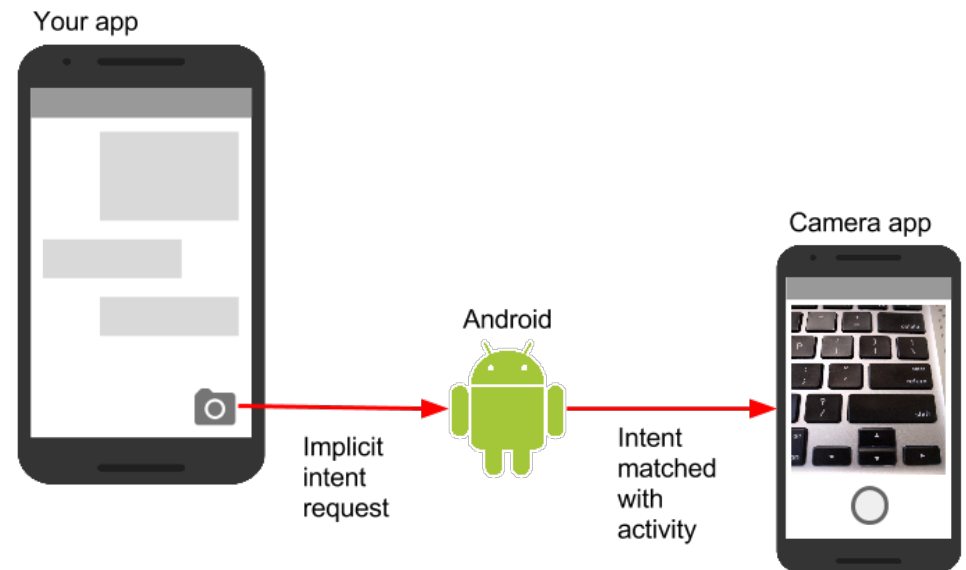
- complete list of Actions:

<https://developer.android.com/reference/kotlin/android/content/Intent>

Starting the external activity

- In all cases, the Intent message is `startActivity(myIntent)`

- myIntent → intent message
 - describing Action, Data, Extras etc...
- the app that executes startActivity() goes in background
- the one that manages the intent (or a chooser) comes in foreground



Receiving implicit intents

- The receiving app advertises which intents can process
 - Intent Filters in Android Manifest

```
<activity android:name="ShareActivity">  
  <intent-filter>  
    <action android:name="android.intent.action.SEND" />  
    <category android:name="android.intent.category.DEFAULT" />  
    <data android:mimeType="text/plain" />  
  </intent-filter>  
</activity>
```

- Retrieves the intent content in onCreate()

```
val bundle: Bundle? = intent.extras  
val name: String? = bundle?.getString("key")
```



Request data from other apps

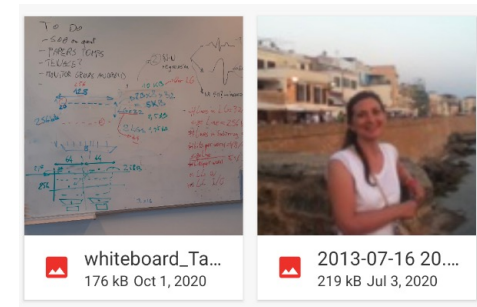
- Interfacing external apps not always a one-way street
 - sometimes, we expect Intents to provide data back
 - examples
 - select a picture from gallery (→ today's Lab!)
 - select a contact
- Starting the external activity and getting the result are decoupled
 - Requesting Apps may be destroyed/recreated while waiting for results
- Enabled by adding proper dependencies to gradle file
 - And update it to the latest version

```
implementation 'androidx.activity:activity-compose:1.8.0'
```

Request data from other apps

1. External apps are still invoked via Intents
 - Created and started e.g. in the event listener of a Composable
 - With the appropriate Action, type etc...
 - `launch()` instead of `startActivity()`

```
HomeContent(imageUri, onClicked = {
    val intent = Intent(Intent.ACTION_GET_CONTENT)
    intent.setType("image/*")
    imagePicker.launch(intent)
})
```



SELECT IMAGE

Request data from other apps

2. The callback is defined as property of a launcher class variable

- defined in the class, outside methods
- initialized with `rememberLauncherForActivityResult()`, with:
 - Contract
 - lambda function implementing callback

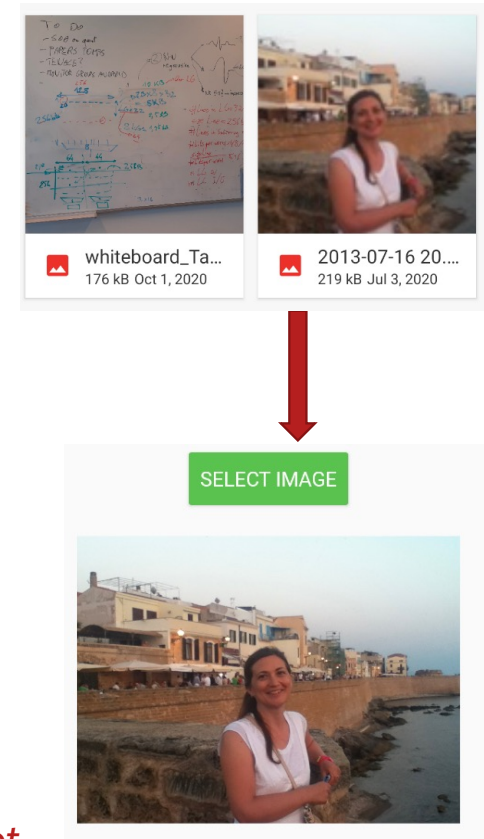
```
var imageUri by remember { mutableStateOf<Uri?>( value: null) }
```

```
val imagePicker = rememberLauncherForActivityResult(  
    contract = ActivityResultContracts.StartActivityForResult(),  
    onResult = { result ->  
        if (result.resultCode == Activity.RESULT_OK) {  
            val uri = result.data?.data  
            imageUri = uri  
        }  
    }  
)
```

© ESL-EPFL

Callback updating the state

non-specific contract





Request data from other apps

- Specialized contracts allow to simplify code
 - getContent() contract directly returns the Uri of the image
 - does not require to explicitly write Intent
 - action is always ACTION_GET_CONTENT
 - data type is passed as argument to launch()

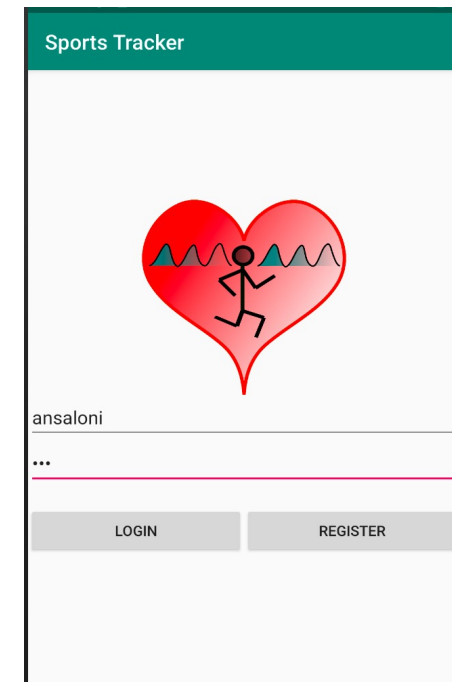
```
1. HomeContent(imageUri, onButtonClicked = { imagePicker.launch( input: "image/*") })

2. val imagePicker = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.GetContent(),
    onResult = { uri ->
        imageUri = uri
    }
)
```

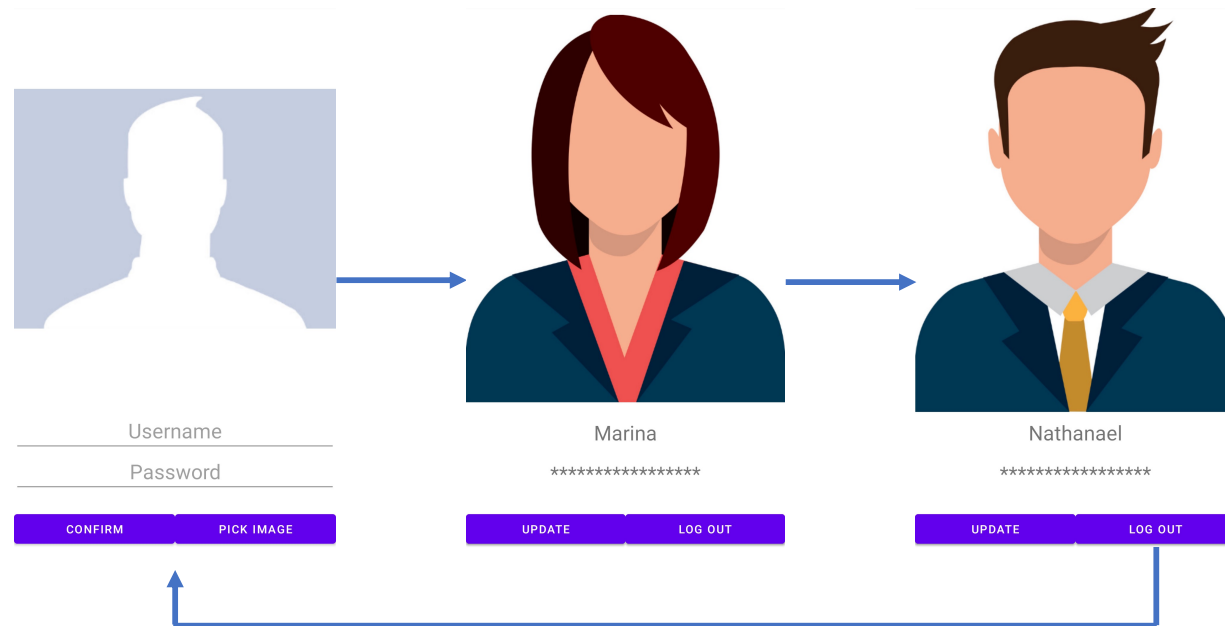
- Complete list of Android Contracts:
<https://developer.android.com/reference/androidx/activity/result/contract/ActivityResultContracts>

Class outline

- Handling events
 - Kotlin and XML callbacks
- Accessing Resources in Kotlin
 - View Binding
- Communicating with other applications
 - Listening for results
- **Lab of Today**



- Tablet application that reacts to the users' actions
- Login credentials and user picture



Announcements

- Groups and material
 - Form your group if you haven't already done so
 - Each group gets a tablet, smartwatch and HR sensor

- Solutions to the labs on Moodle
 - We make them available before each lecture

- Projects:
 - Groups can request projects using the googleForm (link on Moodle)
 - First assignment of students to projects on Moodle this week
 - Information on gitLab on Moodle





Midterm Exam: Information

- **Individual** assessment → 35% of the course evaluation
- Covers all content of the course from week 0 to week 8

- Where: MED 2 2419, MED 2 2519
 - You will be randomly assigned to one of the two classes
- When: **Tuesday November 19st at 14.15pm**, 90 mins duration
 - Exit the class when you finish the exam



Mid-term Exam: Logistics

- Exam with Android Studio and the Android emulator
 - You can do the exam with your laptop, or with the computers in the lab
 - No real devices needed

- Two parts
 1. Replying to short questions in Moodle
 2. Mini app: we will give you a template app, you need to implement missing features



Mid-term Exam: Material

- Only the following online material will be allowed during the mid-term:
 - Moodle lecture slides, labs, etc.
 - Android developers website
 - You can also bring
 - lectures/labs printed
 - your own notes

- Use of mobile phones and AI assistants is **not allowed** in any case



Part1: Questions on Moodle

- ~4 questions
- one attempt

Question **1**

Not yet
answered

Marked out of
0.50

Flag question

 [Edit
question](#)

What is/are the purpose(s) of using a `strings.xml` resource file in an Android project ?

Select one or more:

- ☐ Avoid repeating the text in different java & layout files
- ☐ Automatically correct typos
- ☐ Provide multiple translations
- ☐ Prevent other apps to access private content
- ☐ Allow developers to use emojis

Quiz navigation

1	2	3	4
---	---	---	---

[Finish attempt ...](#)

[Start a new preview](#)

[Next page](#)



Part2: Mini-app

- You will be given a draft of an App project (as we do with labs)
- You will be tasked to implement some features

- Examples:
 1. App crashes, fix the issue explaining how you did it
 2. Something should be performed in response to a button press

- At the end of each task, you will call us to verify that the functionality is working
 - When you raise your hand, the emulator must be already started and the app launched

Questions?

