Android programming

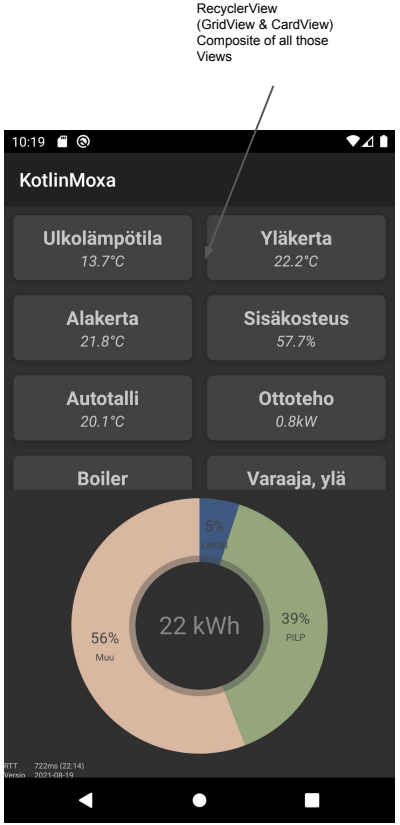
TX00CK66 Sensor Based Mobile Applications

Lecture w1d2 RecyclerView, Fragments and Compose

Jarkko.Vuori@metropolia.fi

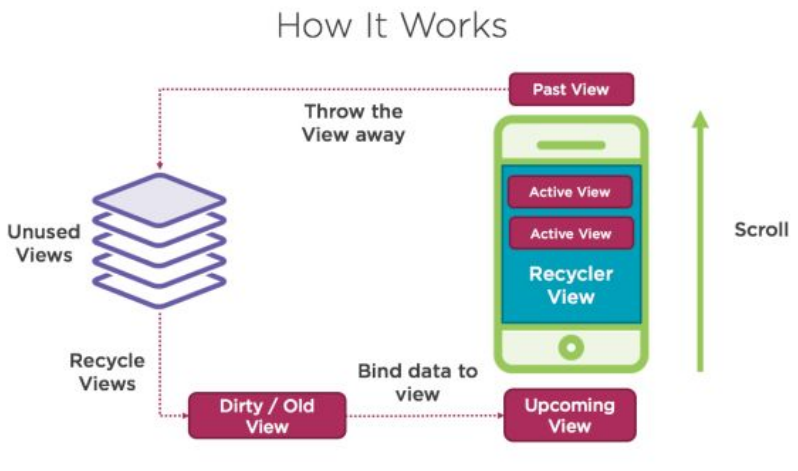
Composite UI Views

- Often a UI element is expected to show together a number of related views
  - address book, photo album, browser history, course list
- Composite UI element may add some functionality that makes it easier to browse the data
  - scrollbars, flipping pages etc.
- Composite UI element also makes it possible to select an item to be acted on (and select the action)
  - show/play etc the item, edit the item, delete the item

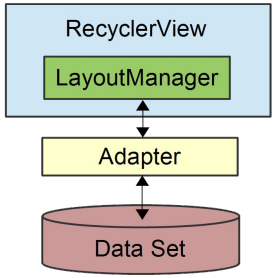


RecyclerView

- RecyclerView is a composite UI View
- The RecyclerView class is a more advanced and flexible version of ListView
  - It is a container for displaying large data sets that can be scrolled very efficiently by maintaining a limited number of views
  - Reduces battery drainage (does not need to redraw views every time user scrolls the list, important if the view is complex)
- RecyclerView has an adapter to fit to various data sources



Adapter architecture



Data to be displayed

- is adapted by the Adapter class for display
- adapter class imposes an order for the data (RecyclerView is ordered, and adapter needs to provide content for a view addressed by position)
- typically SQLite database, or an array

RecyclerView

- manages display of multiple views, uses layout manager for that:
  - `LinearLayout`
  - `GridLayout`
  - `Your own layoutmanager`

1
`RecyclerView` needs to draw one of its subviews, calls adapter method `onBindViewHolder(holder, position)`

2
`Adapter` reads the data to be shown at a given `position`

3
`Adapter` creates a `View` based on data read at step 2

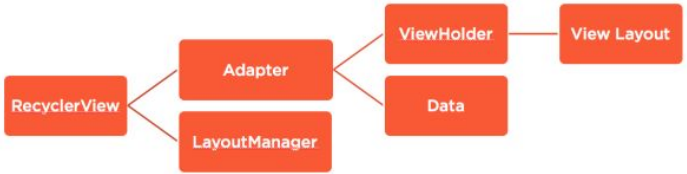
4
return a `View` to be drawn in given position

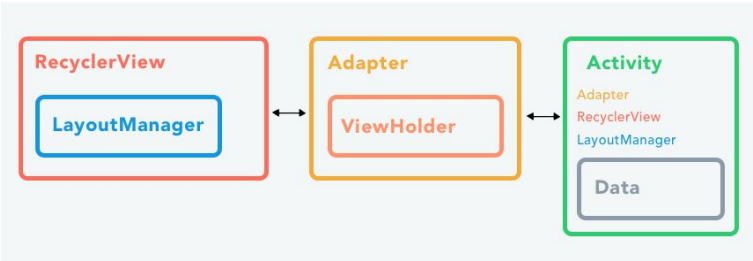
Adapter

- provides views for the UI element to display, functions include:
  - `onCreateViewHolder`
  - `getItemCount`
  - `onBindViewHolder`

RecyclerView

- Classes and elements in RecyclerView:
- Data
  - You will use the data
- A RecyclerView
  - The scrolling list that contains the list items
- Layout for one item of data
  - All list items look the same
- A layout manager
  - The layout manager handles the organization (layout) of user interface components in a view
  - RecyclerView requires an explicit layout manager to manage the arrangement of list items contained within it
    - This layout could be vertical, horizontal, or a grid
- An adapter
  - The adapter connects your data to the RecyclerView
  - It prepares the data in a view holder
- A view holder
  - Inside the adapter, there is a ViewHolder class that contains the view information for displaying one item from the item's layout





RecyclerView example

In this case we are in a fragment

```
inner class ItemHolder(view: View): RecyclerView.ViewHolder(view) {
    var textField = view.findViewById<TextView>(android.R.id.text1)
}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    val rvitems = view.findViewById<RecyclerView>(R.id.idRecycleList);
    rvitems.layoutManager = LinearLayoutManager( context)
    rvitems.adapter = object : RecyclerView.Adapter<ItemHolder>() {
        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemHolder {
            return ItemHolder(LayoutInflater.from(parent.context).inflate(android.R.layout.simple_list_item_1, parent,
false))
        }

        override fun getItemCount(): Int {
            return GlobalModel.presidents.size
        }

        override fun onBindViewHolder(holder: ItemHolder, position: Int) {
            holder.textField.text = GlobalModel.presidents[position].name
        }
    }
```

This guy holds the view (in textField variable)

We use LinearLayout (vertical orientation is default)

Here the view is populated by the data

Here we use an RecyclerView.Adapter as the adapter. It implements needed methods.

GlobalModel is our Singleton class which holds presidents list containing President class

React to item selection

```
override fun onBindViewHolder(holder: ItemHolder, position: Int) {
   holder.textField.text = GlobalModel.presidents[position].name
   holder.textField.setOnClickListener {
       Log.d("USR", "Clicked $position")
           .
           .
           .
   }
}
```

Jetpack Compose List

- Jetpack Compose has an easy way to show a list
  - this is like ListView widget in Layout design
  - not efficient if the list is large
    - because all the items will be composed and laid out whether or not they are visible

```
class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       val boring = List<String>(100) { "List line #$it" }

       super.onCreate(savedInstanceState)
       setContent {
           MessageList(boring)
       }
   }
}

@Composable
fun MessageList(messages: List<String>) {
   Column(modifier = Modifier
       .verticalScroll(rememberScrollState())) {
       messages.forEach { message ->
           Text(message)
       }
   }
}
```

Jetpack Compose List - react to user selection

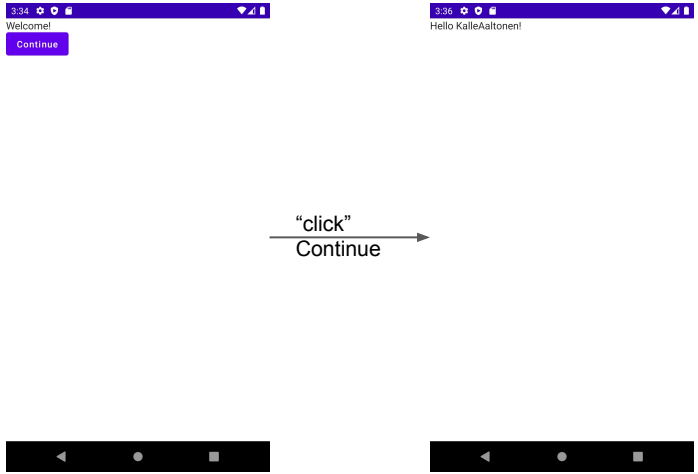
- The trick is to modify the Text() element to be selectable

```
@Composable
fun MessageList(messages: List<String>) {
    Column(modifier = Modifier
        .verticalScroll(rememberScrollState())) {
        messages.forEach { message ->
            Text(text = message,
                 modifier = Modifier
                     .selectable(
                         selected = true,
                         onClick = { Log.i("DBG", "Button clicked! ($message)")
})
            )
        }
    }
}
```

Jetpack Compose - multiple screens

- Using navController it is possible to have multiple screens in an application and be able to switch between them easily
  - https://developer.android.com/jetpack/compose/navigation#groovy

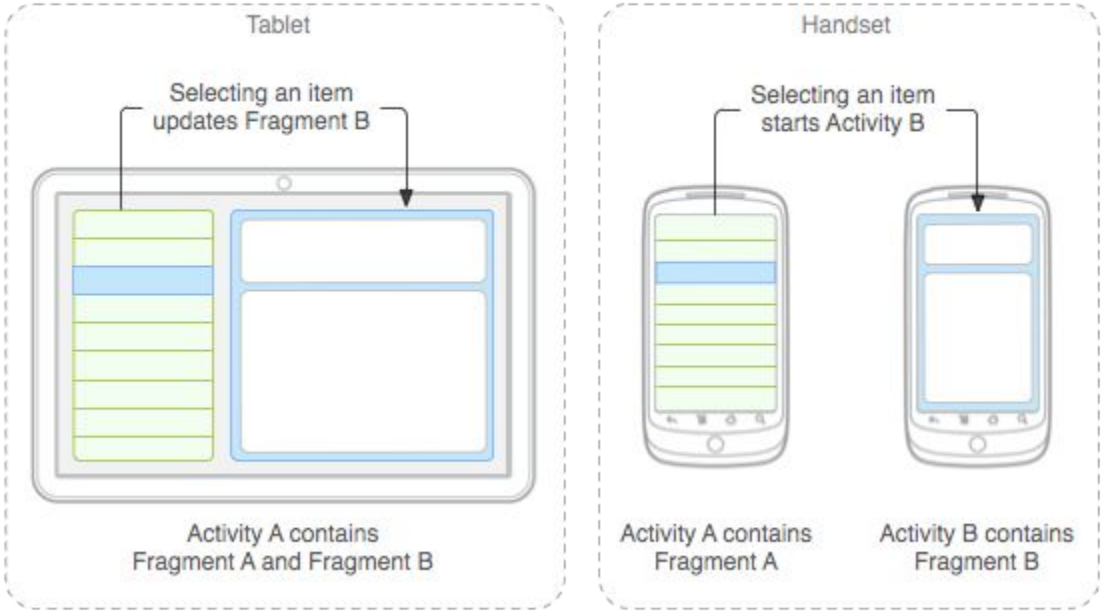
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val navController = rememberNavController()
            AndroidsensorlabsTheme {
                // A surface container using the 'background' color from the theme
                Surface(color = MaterialTheme.colors.background) {
                    NavHost(navController, startDestination = "greeting") {
                        composable("greeting") { Greeting(navController = navController) }
                        composable("greetingName/{userName}") { navBackStackEntry ->

                GreetingWithName(navBackStackEntry.arguments?.getString("userName")!!) }
                    }
                }
            }
        }
    }
}

@Composable
fun Greeting(navController: NavController) {
    Column {
        Text(text = "Welcome!")
        Button(onClick = { navController.navigate("greetingName/KalleAaltonen") }) {
            Text(text = "Continue")
        }
    }
}

@Composable
fun GreetingWithName(userName: String) {
    Text(text = "Hello $userName!")
}
```



Activity and Fragment

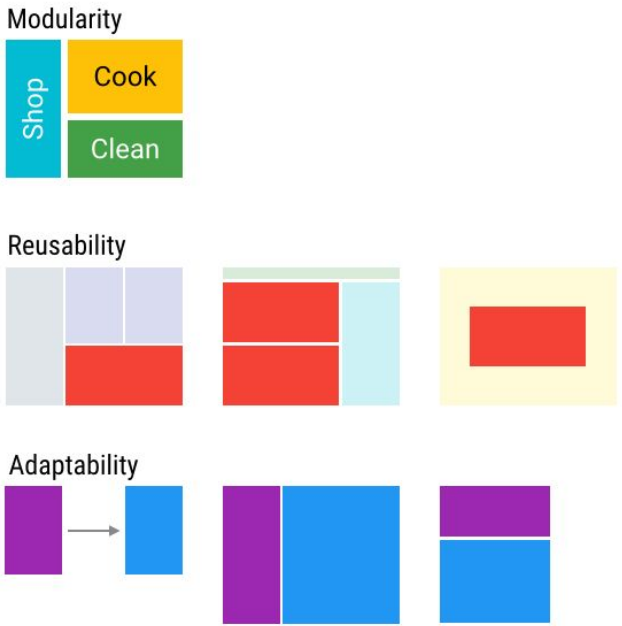
- In traditional (up to mid-size screen) smartphones each of the views of the application (controlled by a subclass of `Activity`) typically occupy the whole screen
- In large-screen devices (such as tablets) this makes the UI ugly and non-attractive, and also non-optimal from usability point of view
- Solution: include in an `Activity` one or more independent “parts” of the UI - **Fragment** objects
  - In a large-screen device multiple `Fragment` objects may be visible at the same time
  - In a small-screen device fragments may be displayed like `Activities` - one at a time, each occupying the whole screen

Activity and Fragment



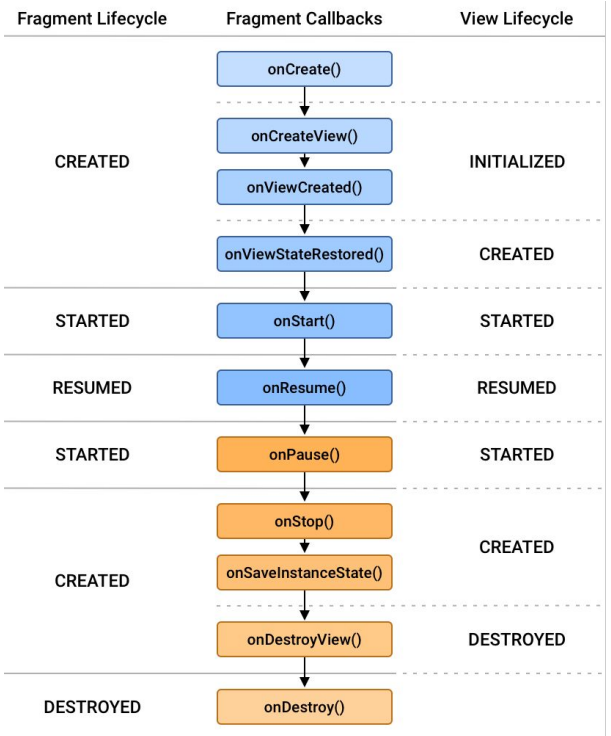
Activity and Fragment

- There are other reasons (other than the previously mentioned adaptability) for the fragment
  - Modularity: Dividing complex activity code across fragments for better organization and maintenance
  - Reusability: Placing behavior or UI parts into fragments that can be shared across multiple activities



Fragment lifecycle

- `Fragment` has lifecycle quite similar to `Activity` lifecycle
- Key callback methods:
  - `onCreate()` - fragment-specific initializations, but do not create view here
  - `onCreateView()` - should return the root view of fragment. Note: fragment might not have UI at all - return null in that case
  - `onViewCreated()`- now you can use findViewById() function call
  - `onResume()/onPause()` - view got the user focus/view looses the user focus (but remains, at least partially, on the screen)
  - `onStop()/onStart()` - view loose the screen/view acquire the screen. Persist the data fragment wants to preserve for next time it is used



Tasks and Back Stack

- Chain of `Activity` invocations executed when the user performs an action is a task
  - `Activities` in a task are arranged into a back stack
  - When a new `Activity` is started, it is pushed into the back stack
  - When user presses back button, `Activity` is popped from the back stack
  - `Activity` maintains a back stack for `Fragments` - allowing user to navigate backwards in `Fragment` chain

Fragments

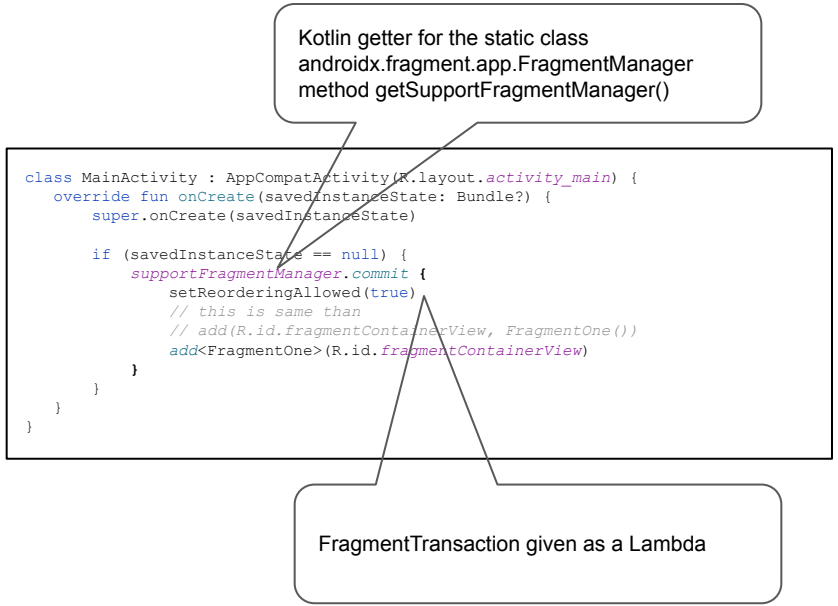
- Starting from Android API 28, Fragment implementation is changed a little
  - standard Fragment Manager is deprecated
  - Now (API 28 and later) Fragments require a dependency on the AndroidX Fragment library
    - build.gradle file need to be modified in order to include this AndroidX Fragment library dependency

Build cradle for the **Module**, not the Project:

```
dependencies {
   def fragment_version = "1.3.6"
   implementation "androidx.fragment:fragment-ktx:$fragment_version"
        .
        .
        .
}
```

Fragment transactions - add

- Changes in activity's fragments are performed using `FragmentTransaction` object
  - `add()`, `remove()`, `replace()`, etc
  - if transaction should be placed into fragment back stack, call `addToBackStack()` - user can navigate back to transaction with back button
- finally `commit()`
- transaction is scheduled for execution in the UI thread of the activity (more on treads later)
- Android KTX (Android Kotlin Extensions) makes it easier to use many Android features (like FragmentManager)
  - e.g. beginTransaction(), endTransaction() calls are automated in FragmentManager



Fragment transactions - replace

- It is possible to give arguments to the newly created Fragment using bundle
  - Bundle is an Android way to pack variable to one container
  - bundleOf() is an Android KTX extension creating a bundle with <String, Int> pair
  - replace() extension allows sending this bundle as an args

```
override fun onButtonClick(position: Int) {
    Log.d("USR", "MainActivity vastaanotti $position")

    val bundle = bundleOf("pos" to position)
    supportFragmentManager.commit {
        setReorderingAllowed(true)
        replace<FragmentTwo>(R.id.fragmentContainerView, args = bundle)
        addToBackStack(null)
    }
}
```

```
class FragmentTwo : Fragment(R.layout.fragment_two) {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        val position = requireArguments().getInt("pos")
            .
            .
            .
    }
}
```

Reading list

- Jetpack Compose
  - https://developer.android.com/jetpack/compose/mental-model
  - https://developer.android.com/jetpack/compose/lists
  - https://developer.android.com/jetpack/compose/navigation#groovy
- RecyclerView
  - https://developer.android.com/guide/topics/ui/layout/recyclerview
  - https://www.raywenderlich.com/1560485-android-recyclerview-tutorial-with-kotlin
- Fragment
  - https://developer.android.com/guide/fragments
  - Notice that most of the stuff found from the net is for the pre API 28 way of using Fragments, like this (but it contains some useful overall information)
    - https://www.raywenderlich.com/1364094-android-fragments-tutorial-an-introduction-with-kotlin
- Android Kotlin Extensions (good to know, not essential)
  - https://developer.android.com/kotlin/ktx