5COSC023W - MOBILE APPLICATION DEVELOPMENT
Lecture 2: Android Programming: Jetpack Compose - Activities, Intents

Dr Dimitris C. Dracopoulos

Android Programming

Jetpack Compose - The Story

- Since the second half of 2023, Jetpack Compose is the recommended way for the creation of the UI of Android applications.
- Completely different way of doing things, not just simply the UI.
- The whole code for an Android application needs to be changed, not just the UI.

Jetpack Compose - The Story

- Since the second half of 2023, Jetpack Compose is the recommended way for the creation of the UI of Android applications.
- Completely different way of doing things, not just simply the UI.
- The whole code for an Android application needs to be changed, not just the UI.

Jetpack Compose - The Story

- Since the second half of 2023, Jetpack Compose is the recommended way for the creation of the UI of Android applications.
- Completely different way of doing things, not just simply the UI.
- The whole code for an Android application needs to be changed, not just the UI.

The Views Way of Implementation

Still supported but *Compose* is the new recommended way.

- Every widget (e.g. `Button`, `TextView`) is a subclass of the `View` class.
- All widgets are objects which are manipulated by calling their methods and change their state (e.g. call the set method to change the text of a `TextView`)
- The widget objects are passed around the program as references in order to access and modify them in other parts of the application.
- Classic way of object oriented programming: object are passed to different parts of the code so they can be accessed and call methods on them.

The Views Way of Implementation

Still supported but *Compose* is the new recommended way.

- Every widget (e.g. `Button, TextView`) is a subclass of the `View` class.
- All widgets are objects which are manipulated by calling their methods and change their state (e.g. call the set method to change the text of a `TextView`)
- The widget objects are passed around the program as references in order to access and modify them in other parts of the application.
- Classic way of object oriented programming: object are passed to different parts of the code so they can be accessed and call methods on them.

The `Views` Way of Implementation

Still supported but *Compose* is the new recommended way.

- Every widget (e.g. `Button, TextView`) is a subclass of the `View` class.
- All widgets are objects which are manipulated by calling their methods and change their state (e.g. call the `set` method to change the text of a `TextView`)
- The widget objects are passed around the program as references in order to access and modify them in other parts of the application.
- Classic way of object oriented programming: object are passed to different parts of the code so they can be accessed and call methods on them.

The Views Way of Implementation

Still supported but *Compose* is the new recommended way.

- Every widget (e.g. `Button, TextView`) is a subclass of the `View` class.
- All widgets are objects which are manipulated by calling their methods and change their state (e.g. call the `set` method to change the text of a `TextView`)
- The widget objects are passed around the program as references in order to access and modify them in other parts of the application.
- Classic way of object oriented programming: object are passed to different parts of the code so they can be accessed and call methods on them.

The `Views` Way of Implementation

Still supported but *Compose* is the new recommended way.

- Every widget (e.g. `Button, TextView`) is a subclass of the `View` class.
- All widgets are objects which are manipulated by calling their methods and change their state (e.g. call the `set` method to change the text of a `TextView`)
- The widget objects are passed around the program as references in order to access and modify them in other parts of the application.
- Classic way of object oriented programming: object are passed to different parts of the code so they can be accessed and call methods on them.

Views vs Jetpack Compose

- **Views approach**: Widgets (e.g. buttons) are objects and we need to call their methods to change what they display.
- **Compose approach**: Everything is based on `Composable` functions which are responsible to emit one or more UI components.
- **Compose** describe *WHAT* to draw, while **View** describe *HOW* to draw UI elements.
- Declarative (Compose) UI definition vs Object Oriented Programming way.

Views vs Jetpack Compose

- **Views approach**: Widgets (e.g. buttons) are objects and we need to call their methods to change what they display.
- **Compose approach**: Everything is based on `Composable` functions which are responsible to emit one or more UI components.
- **Compose** describe *WHAT* to draw, while **View** describe *HOW* to draw UI elements.
- Declarative (Compose) UI definition vs Object Oriented Programming way.

Views vs Jetpack Compose

- **Views approach**: Widgets (e.g. buttons) are objects and we need to call their methods to change what they display.
- **Compose approach**: Everything is based on `Composable` functions which are responsible to emit one or more UI components.
- **Compose** describe *WHAT* to draw, while **View** describe *HOW* to draw UI elements.
- Declarative (Compose) UI definition vs Object Oriented Programming way.

Views vs Jetpack Compose

- **Views approach**: Widgets (e.g. buttons) are objects and we need to call their methods to change what they display.
- **Compose approach**: Everything is based on `Composable` functions which are responsible to emit one or more UI components.
- **Compose** describe *WHAT* to draw, while **View** describe *HOW* to draw UI elements.
- Declarative (Compose) UI definition vs Object Oriented Programming way.

The `Compose` Way of Implementation

- `Composable` functions are describing what UI elements to draw.
- The functions are automatically called again (recomposition) when they need to be redrawn because the state of the UI elements model has changed (although UI elements are stateless in theory)
- Many imports of classes (compared with Views) but more efficient and less bug prone.
- A new way of thinking about implementation in Android applications if you had previous experience.
- UI elements are functions, NOT objects (unlike `Views`)
- No XML is required for UI creation. All the UI is implemented in code.

The Compose Way of Implementation

- Composable functions are describing what UI elements to draw.
- The functions are automatically called again (recomposition) when they need to be redrawn because the state of the UI elements model has changed (although UI elements are stateless in theory)
- Many imports of classes (compared with Views) but more efficient and less bug prone.
- A new way of thinking about implementation in Android applications if you had previous experience.
- UI elements are functions, NOT objects (unlike Views)
- No XML is required for UI creation. All the UI is implemented in code.

The `Compose` Way of Implementation

- `Composable` functions are describing what UI elements to draw.
- The functions are automatically called again (recomposition) when they need to be redrawn because the state of the UI elements model has changed (although UI elements are stateless in theory)
- Many imports of classes (compared with Views) but more efficient and less bug prone.
- A new way of thinking about implementation in Android applications if you had previous experience.
- UI elements are functions, NOT objects (unlike `Views`)
- No XML is required for UI creation. All the UI is implemented in code.

The `Compose` Way of Implementation

- `Composable` functions are describing what UI elements to draw.
- The functions are automatically called again (recomposition) when they need to be redrawn because the state of the UI elements model has changed (although UI elements are stateless in theory)
- Many imports of classes (compared with Views) but more efficient and less bug prone.
- A new way of thinking about implementation in Android applications if you had previous experience.
- UI elements are functions, NOT objects (unlike `Views`)
- No XML is required for UI creation. All the UI is implemented in code.

The `Compose` Way of Implementation

- `Composable` functions are describing what UI elements to draw.
- The functions are automatically called again (recomposition) when they need to be redrawn because the state of the UI elements model has changed (although UI elements are stateless in theory)
- Many imports of classes (compared with Views) but more efficient and less bug prone.
- A new way of thinking about implementation in Android applications if you had previous experience.
- UI elements are functions, NOT objects (unlike `Views`)
- No XML is required for UI creation. All the UI is implemented in code.

The Compose Way of Implementation

- Composable functions are describing what UI elements to draw.
- The functions are automatically called again (recomposition) when they need to be redrawn because the state of the UI elements model has changed (although UI elements are stateless in theory)
- Many imports of classes (compared with Views) but more efficient and less bug prone.
- A new way of thinking about implementation in Android applications if you had previous experience.
- UI elements are functions, NOT objects (unlike Views)
- No XML is required for UI creation. All the UI is implemented in code.

Characteristics of Composable Functions

- Composable functions can only be called by other composable functions.
- The exception to the above rule is the `setContent` function which is the starting point of calling composable functions.
- Composable functions can call non-composable functions.
- Composable functions can be rendered and seen within the editor by annotating them with `@Preview`.
- Composable functions annotated with `@Preview` should not accept any parameters.

Characteristics of Composable Functions

- Composable functions can only be called by other composable functions.
- The exception to the above rule is the `setContent` function which is the starting point of calling composable functions.
- Composable functions can call non-composable functions.
- Composable functions can be rendered and seen within the editor by annotating them with `@Preview`.
- Composable functions annotated with `@Preview` should not accept any parameters.

Characteristics of Composable Functions

- Composable functions can only be called by other composable functions.
- The exception to the above rule is the `setContent` function which is the starting point of calling composable functions.
- Composable functions can call non-composable functions.
- Composable functions can be rendered and seen within the editor by annotating them with `@Preview`.
- Composable functions annotated with `@Preview` should not accept any parameters.

Characteristics of Composable Functions

- Composable functions can only be called by other composable functions.
- The exception to the above rule is the `setContent` function which is the starting point of calling composable functions.
- Composable functions can call non-composable functions.
- Composable functions can be rendered and seen within the editor by annotating them with `@Preview`.
- Composable functions annotated with `@Preview` should not accept any parameters.

Characteristics of Composable Functions

- Composable functions can only be called by other composable functions.
- The exception to the above rule is the `setContent` function which is the starting point of calling composable functions.
- Composable functions can call non-composable functions.
- Composable functions can be rendered and seen within the editor by annotating them with `@Preview`.
- Composable functions annotated with `@Preview` should not accept any parameters.

Characteristics of Composable Functions

- Composable functions can only be called by other composable functions.
- The exception to the above rule is the `setContent` function which is the starting point of calling composable functions.
- Composable functions can call non-composable functions.
- Composable functions can be rendered and seen within the editor by annotating them with `@Preview`.
- Composable functions annotated with `@Preview` should not accept any parameters.

Default Values for Function Arguments

Function arguments can have an optional name and an optional default value.

- The order of arguments can be changed if their names is used.

```
fun colour(red: Int = 0, green: Int = 0, blue:Int = 0) {

}

fun main() {
    // default value for green is used, i.e. 0
    colour(blue = 255, red = 125)
}
```

Default Values for Function Arguments

Function arguments can have an optional name and an optional default value.

- The order of arguments can be changed if their names is used.

```
fun colour(red: Int = 0, green: Int = 0, blue:Int = 0) {

}

fun main() {
    // default value for green is used, i.e. 0
    colour(blue = 255, red = 125)
}
```

Default Values for Function Arguments

Function arguments can have an optional name and an optional default value.

- The order of arguments can be changed if their names is used.

```
fun colour(red: Int = 0, green: Int = 0, blue:Int = 0) {

}

fun main() {
    // default value for green is used, i.e. 0
    colour(blue = 255, red = 125)
}
```

Default Values for Function Arguments

Function arguments can have an optional name and an optional default value.

- The order of arguments can be changed if their names is used.

```
fun colour(red: Int = 0, green: Int = 0, blue:Int = 0) {

}

fun main() {
    // default value for green is used, i.e. 0
    colour(blue = 255, red = 125)
}
```

A Lottery Program

A lottery ticket consists of 6 unique numbers in the range between 1 and 59.

Write an Android application which calculates such a 6 lucky random unique numbers that the user can play in the next lottery. Every time a button is pressed a new set of unique numbers is generated.

A Lottery Program

A lottery ticket consists of 6 unique numbers in the range between 1 and 59.
Write an Android application which calculates such a 6 lucky random unique numbers that the user can play in the next lottery. Every time a button is pressed a new set of unique numbers is generated.

The Lottery Program with Jetpack Compose

- When you create the project in Android Studio choose the template “Empty Activity” and NOT “Empty Views Activity”.

The activity file:

```
package com.example.lotterycomposableapp
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import kotlin.random.Random
```

The Activity code (cont'd)

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
           displayNumbers()
        }
    }
}

@Composable
fun displayNumbers() {
    var results by remember{mutableStateOf("")}
    Column (
        Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center) {
        Text("results: " + results)
        Button(onClick = {results = calculate() }) {
            Text(text = "Generate")
        }
    }
}
```

The Activity code (cont'd)

```
fun calculate(): String {
    val numbers = mutableListOf<Int>()

    while (numbers.size < 6) {
        val new_number = 1 + Random.nextInt(59)
        if (new_number !in numbers)
            numbers.add(new_number)
    }

    var results = ""
    for (i in numbers)
        results += "" + i + " "

    return results
}
```

Adding State to a Composable

1. Example:

   ```
   var results by remember{mutableStateOf("")}
   ```

   to define a variable within a composable function (for example a variable `result`).
2. Change the value of the variable within the composable function, for example within the `onClick` method of a `Button` composable or within the `onChangeValue` of a `TextField` composable.

Activities

- An Android component representing a whole window (screen)
- One Class
- To display a different screen a new `Activity` can be created (or alternatively display new composables within the same activity).
- An activity needs to have a layout

Activities

- An Android component representing a whole window (screen)
- One Class
- To display a different screen a new `Activity` can be created (or alternatively display new composables within the same activity).
- An activity needs to have a layout (typically created in XML or using Jetpack Compose. Using `Views` it can also be created or modified dynamically as well, similarly with Jetpack Compose).
- **Important:** All of the created activities should be declared in the manifest of the application (file: `AndroidManifest.xml`).

Intents

- A description of an operation to be performed

Intents

- A description of an operation to be performed
- Can be used to start other activities

Intents

- A description of an operation to be performed
- Can be used to start other activities
- Can be *explicit* (starting a specific activity) or *implicit* (letting the system or the user to choose which activity to start)

Intents

- A description of an operation to be performed
- Can be used to start other activities
- Can be *explicit* (starting a specific activity) or *implicit* (letting the system or the user to choose which activity to start)

Example:

```
val in = Intent(this, NewActivity::class.java)
```

Intents

- A description of an operation to be performed
- Can be used to start other activities
- Can be *explicit* (starting a specific activity) or *implicit* (letting the system or the user to choose which activity to start)

Example:

```
val in = Intent(this, NewActivity::class.java)
startActivity(in)
```