Mobile Applications

2. User interfaces in Android

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

uc3m | Universidad Carlos III de Madrid

Table of contents

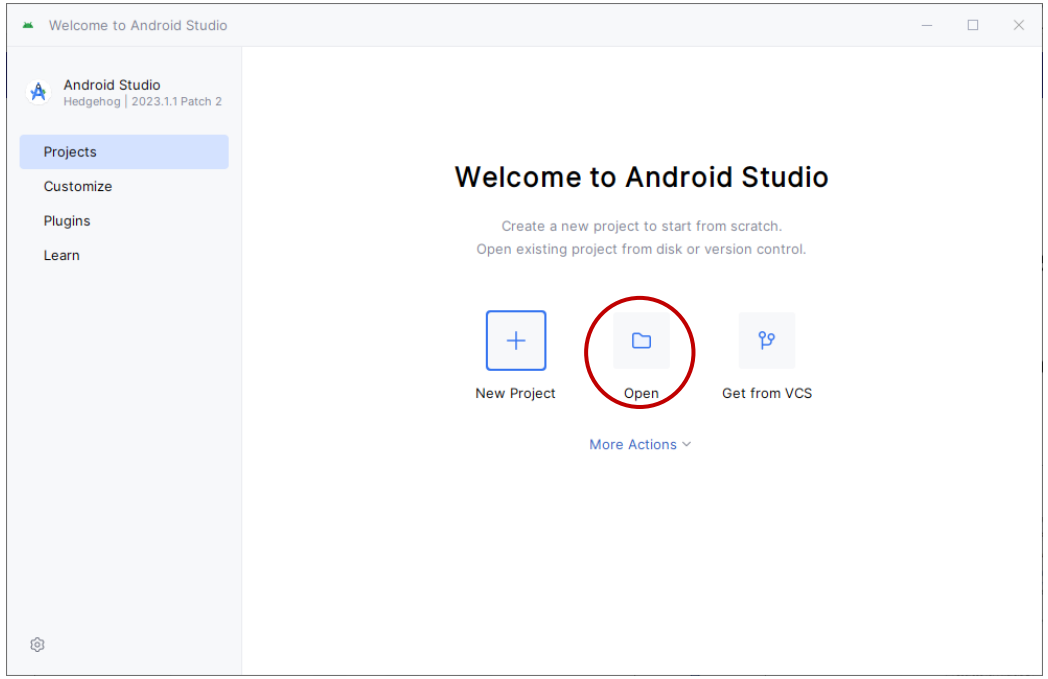
1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

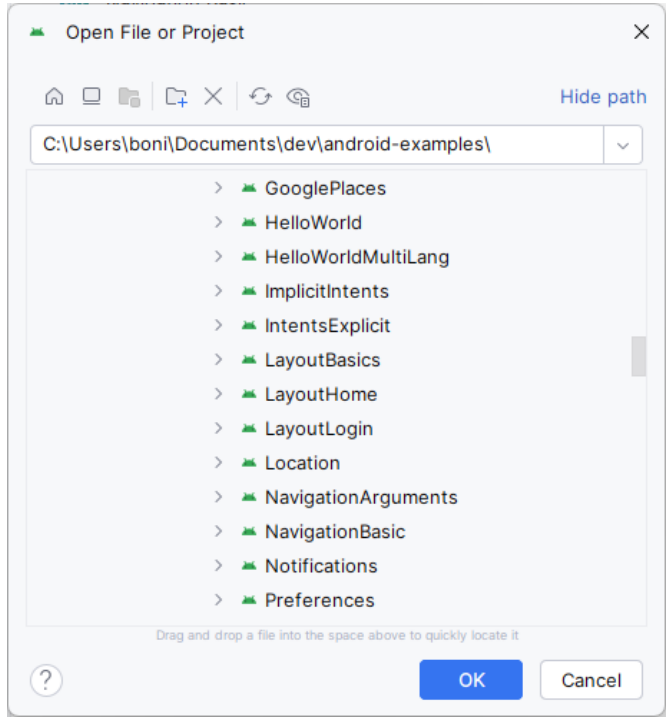
1. Introduction

- In the context of mobile apps, the **User Interface (UI)** refers to the visual and interactive elements of an app that users interact with to perform tasks or access information
- The UI is a critical of the overall user experience (UX) and includes:
  - Visual design: Layout, colors, typography, icons, and images
  - Interactive elements: Buttons, text fields, and other touchable components
  - Navigation: How users move between screens or sections of the app (e.g., tabs, menus, back buttons)
  - Responsiveness: How the UI adapts to different screen sizes, orientations, and devices (e.g., smartphones, tablets)
  - Accessibility: Features like larger text, voice commands, and screen readers to make the app usable for people with disabilities

1. Introduction

- To follow the master lectures, it is recommended to clone the **GitHub examples repository** and import each app in Android Studio to play with it





https://github.com/bonigarcia/android-examples/

Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

2. Activities

- As we have seen, an **Activity** is a type of app component which represents a single screen with a UI, where users can perform interactions and tasks
  - App components (like activities) are implemented using specific classes (Kotlin or Java) and are declared in the Android manifest

AndroidManifest.xml

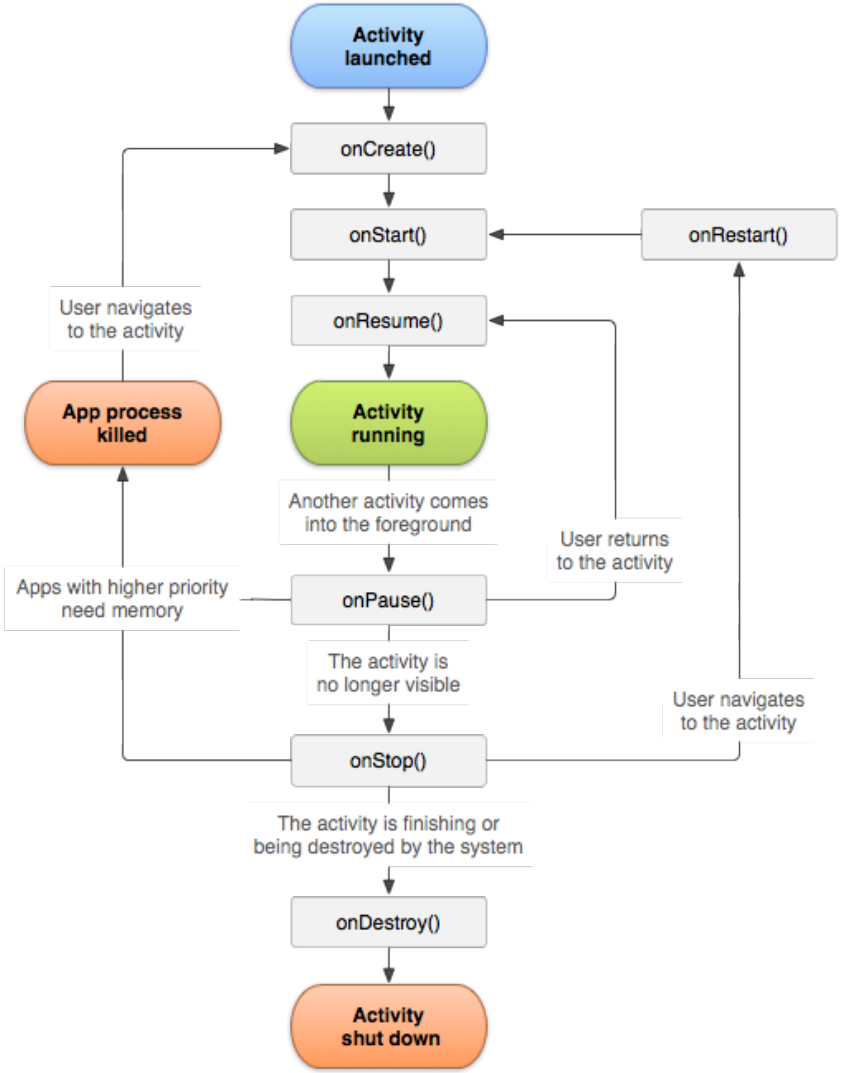
```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:theme="@style/Theme.HelloWorld">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

MainActivity.kt

```kotlin
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

2. Activities

- Activities are managed by the Android system, and they go through different states such as "*created*," "*started*," "*resumed*," "*paused*," "*stopped*," and "*destroyed*" based on the user's interactions and the lifecycle of the app
- This **lifecycle** follows a state machine as depicted in this picture
- Android invokes Java/Kotlin methods defined in the Activities (called *callbacks*) when an activity enters a new state



https://developer.android.com/guide/components/activities/activity-lifecycle

2. Activities

- Let's analyze the activity in the hello-world app:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

- `class MainActivity` → Kotlin class
- `ComponentActivity` → Parent class for activities (in Jetpack Compose)
- `override fun onCreate` → activity entry point
- `savedInstanceState: Bundle?` → parameter used to restore saved state when the activity is recreated. The operator ? allows this parameter to be null
- `super.onCreate(savedInstanceState)` → call to parent to ensure proper initialization

Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
   - Setup
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

3. Jetpack Compose

- We will use **Jetpack Compose** to create the UIs in Android
  - Jetpack Compose is a Google's modern toolkit for building native UI in Android applications, recommended in Modern Android Development (MAD)
  - It simplifies UI development by combining a declarative approach with a Kotlin-based syntax
  - Jetpack Compose is available since Android 8 (API level 21)
- Before Jetpack Compose, the traditional **Android View system** was imperative and based on XML to write UIs:
  - Views were inflated from XML layout files
  - Themes, styles, and value resources were also defined in XML files
  - For us to be able to access the views from XML files, we used view binding or data binding (e.g., `findViewById`)
  - This method of writing a UI required huge effort, requiring more boilerplate code and being error prone

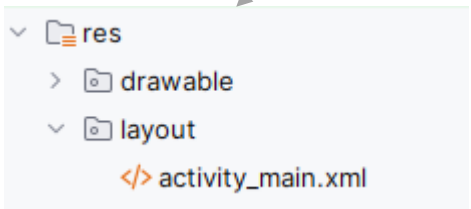
https://developer.android.com/compose

3. Jetpack Compose

- A basic example (Hello World) using Java and XML layout:

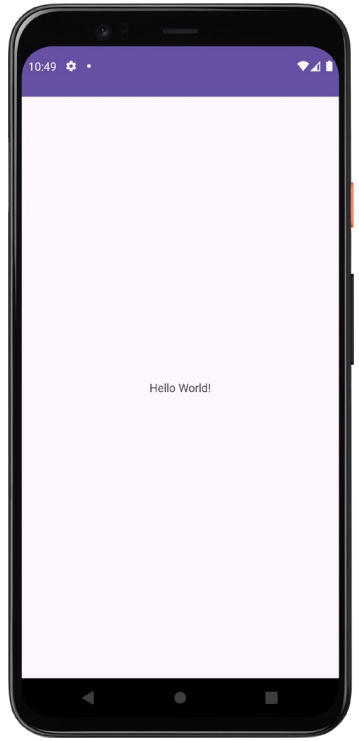
MainActivity.java

```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```



activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```



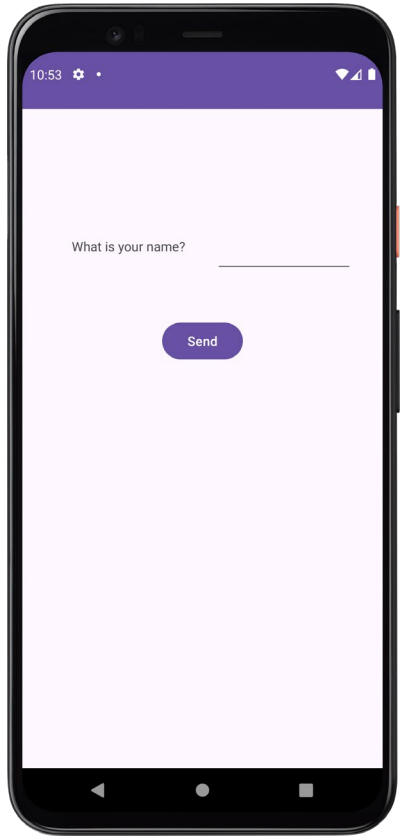
We don't study these XMLs in this course. These slides are only to illustrate how the legacy Android View system looked like

This layout contained only one visual element: a `TextView`, which displays some text to the user

3. Jetpack Compose

- Another example using XML:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        View button = findViewById(R.id.button);
        button.setOnClickListener(view -> {
            Intent intent = new Intent(getBaseContext(), SecondActivity.class);
            EditText nameText = findViewById(R.id.editText);
            intent.putExtra("name", nameText.getText().toString());
            startActivity(intent);
        });
    }

}
```



XML-based views are still supported alongside Jetpack Compose for backward compatibility and mixed use cases where apps have both XML layouts and Jetpack Compose

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/nameLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/edit_message"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.2"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.2" />

    <EditText
        android:id="@+id/editText"
        android:layout_width="150dp"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/nameLabel"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.2" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editText"
        app:layout_constraintVertical_bias="0.1" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

3. Jetpack Compose

- The advantages of Jetpack Compose are:
  - Kotlin-based
    - The code we write is only in Kotlin, rather than having it split between Java/Kotlin and XML
  - Less boilerplate
    - Compose allows us to do more with less code, compared to the XML view system
  - Declarative
    - Compose allows you to describe “what” the UI should look like instead of “how” to draw it. This makes the code cleaner and easier to read
  - Reactive
    - The UI is data-driven and automatically reacts to changes in state, eliminating the need for manually updating views with `findViewById` or `notifyDataSetChanged`
  - High performance
    - Jetpack Compose uses a highly optimized rendering engine, and the framework has built-in mechanisms for efficient UI updates
  - Rich tooling support
    - Live preview: Developers can preview UI changes directly in Android Studio without running the app on a device/emulator
    - Compose preview: Multiple preview modes (e.g., light/dark themes, device sizes) simplify design testing

3. Jetpack Compose - Setup

build.gradle.kts

```
android {
    buildFeatures {
        compose = true
    }
}
```

build.gradle.kts

```
dependencies {
    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.activity.compose)
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.ui)
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)
    debugImplementation(libs.androidx.ui.tooling)
    debugImplementation(libs.androidx.ui.test.manifest)
}
```

To use Jetpack Compose, we need to set up our Android project (using Gradle) with the necessary configuration and dependencies

libs.version.toml

```
[versions]
agp = "8.8.0"
kotlin = "2.1.10"
coreKtx = "1.15.0"
lifecycleRuntimeKtx = "2.8.7"
activityCompose = "1.10.0"
composeBom = "2025.01.01"

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx" }
androidx-lifecycle-runtime-ktx = { group = "androidx.lifecycle", name = "lifecycle-runtime-ktx", version.ref = "lifecycleRuntimeKtx" }
androidx-activity-compose = { group = "androidx.activity", name = "activity-compose", version.ref = "activityCompose" }
androidx-compose-bom = { group = "androidx.compose", name = "compose-bom", version.ref = "composeBom" }
androidx-ui = { group = "androidx.compose.ui", name = "ui" }
androidx-ui-graphics = { group = "androidx.compose.ui", name = "ui-graphics" }
androidx-ui-tooling = { group = "androidx.compose.ui", name = "ui-tooling" }
androidx-ui-tooling-preview = { group = "androidx.compose.ui", name = "ui-tooling-preview" }
androidx-ui-test-manifest = { group = "androidx.compose.ui", name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-test-junit4" }
androidx-material3 = { group = "androidx.compose.material3", name = "material3" }
```

Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

4. Compose functions

- A **composable** (or **compose function**) is a Kotlin function that defines a piece of UI
  - Composables are the building block of Jetpack Compose
  - Composables are annotated with `@Composable` and describes how a portion of the UI should look based on the current state
- Key characteristics:
  - Declarative: instead of imperatively defining how the UI should change (as in traditional Android Views), we describe what the UI should look like
  - Reusable: Compose functions can be called from other Compose functions, making them modular and reusable
  - State-driven: composables react to changes in state (i.e., the data that affects the UI). When the state changes, the composable function is recomposed (re-drawn) to reflect the new state

4. Compose functions

- Let's analyze how composables work in the hello-world app:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

- `@Composable` → Annotation to mark the function as a compose function
- `Greeting` → The name of the function. It takes a string parameter (`name: String`) and displays it in a `Text` composable
- `modifier: Modifier = Modifier` → A modifier is an object that allows us to decorate or modify a composable. This parameter has a default value, i.e., an empty modifier (`Modifier`) which means no modifications applied
- `Text` → A built-in composable that displays text on the screen

4. Compose functions

- Let's analyze how composables work in the hello-world app:

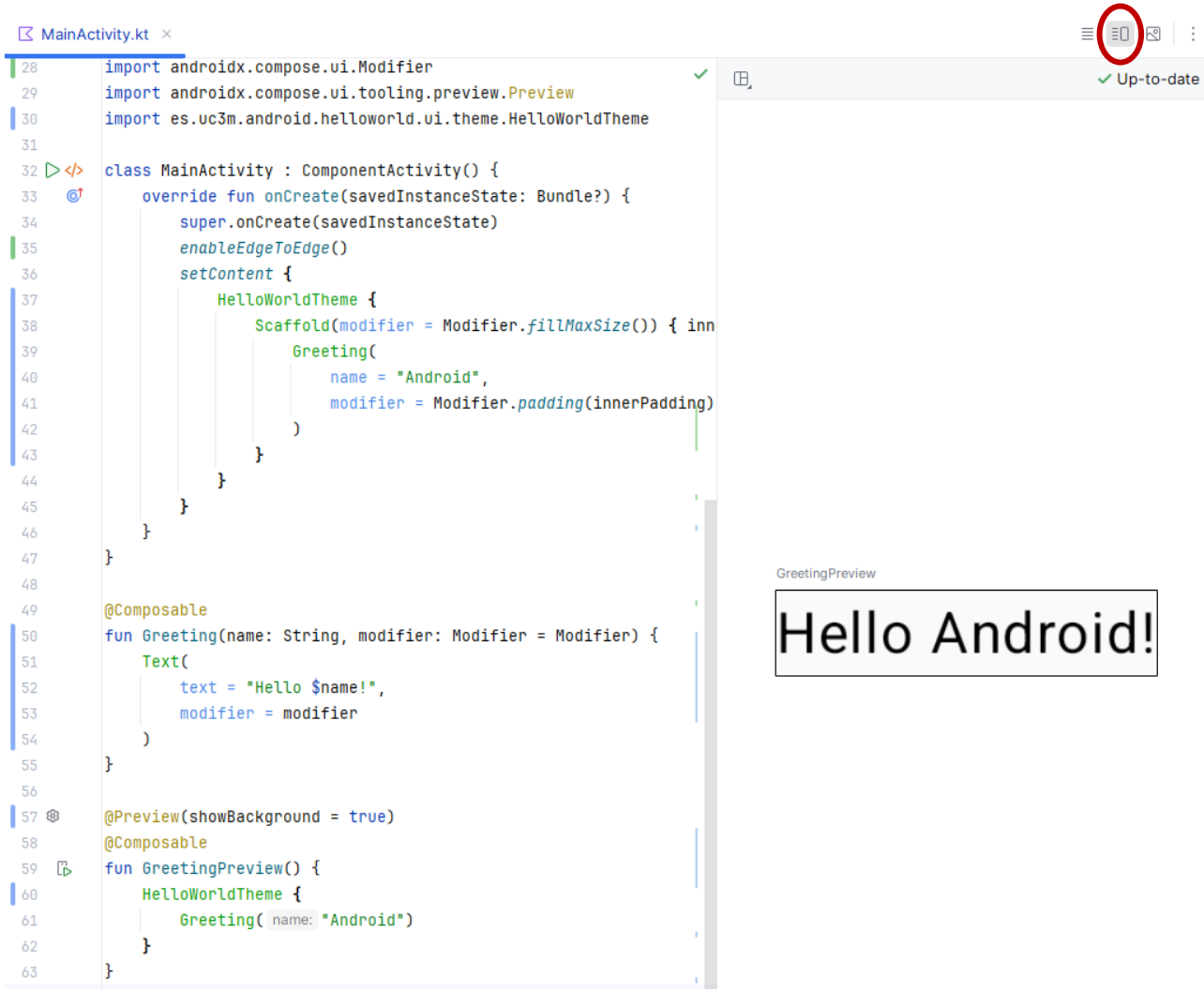
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloWorldTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

- *enableEdgeToEdge()* → draw app content behind the status bar
- *setContent* → method provided by the parent class that takes a composable function as argument and renders it as the root of the UI
- HelloWorldTheme → custom composable that applies a Material Design theme to our app (explained latter)
- Scaffold() → Recommended Material Design layout structure
- modifier = Modifier.*fillMaxSize*() → ensures the Scaffold expands to fill the entire screen
- innerPadding → parameter in lambda expression used to ensure that the content doesn't overlap with system UI elements (e.g., status bar, navigation bar)
- Greeting(...) → Call to our composable function

4. Compose functions


```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HelloWorldTheme {
        Greeting("Android")
    }
}
```

- @Preview → Annotation that tells Android Studio that this composable should be shown in the design view (it has no impact in runtime)



4. Compose functions

- A Compose UI is built as a hierarchy of compose functions. Each composable can contain others composables, forming a tree-like structure
  - In the hello-world example the composable hierarchy is:

```
HelloWorldTheme
└── MaterialTheme
    └── Scaffold
        └── Greeting
            └── Text
```

- When the `Greeting` function is called, Compose renders the `Text` composable with the provided `name`
- If the **state** (i.e., the data that can change over time and affects the UI –the variable `name` in this example–) changes, Compose automatically **recomposes** (redraws) the UI to reflect the new value

4. Compose functions

We study these composables in the following sections

- Jetpack Compose provides a rich set of built-in composable functions:

1. Basic components: used to create the UI, e.g., `Text`, `Button`, `Image`, `Icon`
2. Modifiers: used to customize the appearance and behavior of composables, e.g., `padding`, `fillMaxSize`, `background`, `clickable`
3. Layouts: used to define the structure of the UI, e.g., `Column`, `Row`, `Box`, `Spacer`, `Scaffold`, `Surface`
4. Theming: used to handle the styles for an app: `MaterialTheme`
5. State management: used to handle the UI state, e.g., `mutableStateOf`, `remember`, `rememberSaveable`
6. List and grids: used to display a group of elements, e.g., `LazyColumn`, `LazyRow`, `LazyVerticalGrid`, `LazyHorizontalGrid`
7. Navigation: used to change between different screens, e.g., `NavHost`
8. Animations: used to provide transitions to our apps, e.g., `animate*AsState`, `AnimatedVisibility`

Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
   - Modifiers
   - Unit of measurements
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

5. Basic components

- The `Text` Composable is used to display a text string
  - It can be customized with the attributes `fontSize`, `color`, `fontWeight`, etc.
    - Nevertheless, it is not recommended to change this attributes individually (instead, we will use global theme styles)

```
Text(
    text = "Hello, Compose!",
    fontSize = 20.sp,
    color = Color.Blue,
    fontWeight = FontWeight.Bold
)
```

- The `Button` Composable displays a clickable button with a text label
  - It handles clicks using the `onClick` lambda

```
Button(onClick = { /* Handle click */ }) {
    Text("Click Me")
}
```

5. Basic components

- The `Image` Composable is used to display an image
  - It can load images from resources, URIs, or bitmaps

```
Image(
    painter = painterResource(id = R.drawable.ic_launcher_foreground),
    contentDescription = "App Icon"
)
```

- The `Icon` Composable displays an icon
  - It handles clicks using the `onClick` lambda

```
Icon(
    painter = painterResource(R.drawable.baseline_directions_bus_24),
    contentDescription = "App Icon"
)
```

We study how to get graphics from the resources in the next section

https://developer.android.com/develop/ui/compose/graphics

5. Basic components - Modifiers

- Modifiers are compose functions that allow us to decorate or augment (i.e., style, position, and add behavior) another composables
- Common modifiers are:
  - *padding*: Adds space around the composable (`top`, `bottom`, `start`, `end`)
  - *fillMaxSize*: Makes the composable fill the available space in its parent layout, both in terms of width and height
  - *fillMaxWidth*: Makes the composable fill the maximum width given to it from its parent
  - *fillMaxHeight*: Makes the composable fill the maximum height given to it from its parent
  - *clickable*: Makes the composable respond to clicks
  - *background*: Sets the background color

https://developer.android.com/develop/ui/compose/modifiers

5. Basic components - Modifiers

- A basic example using some modifiers:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello, $name!",
        modifier = modifier
            .fillMaxWidth()
            .background(Color.LightGray)
    )
    Button(
        onClick = {
            println("TODO handle click")
        },
        modifier = modifier.padding(32.dp)
    ) {
        Text(
            text = "Click Me",
            fontSize = 24.sp,
            color = Color.Green
        )
    }
}

@Preview(showBackground = true)
@Composable
fun Preview() {
    MyAppTheme {
        Greeting("Android")
    }
}
```

5. Basic components - Units of measurement

- Handling measurements is a fundamental to ensures that our app's UI looks consistent and scales correctly across devices with different screen sizes and densities. Common units of measurement are:
  - Density-independent pixels (*dp*)
    - Purpose: used for layout dimensions (e.g., padding, margins, width, height)
    - Behavior: scales based on the screen's density
    - Use case: responsive designs
  - Scale-independent pixels (*sp*)
    - Purpose: used for text sizes
    - Behavior: scales based on the screen's density and the user's font size preferences
    - Use case: similar to dps, but adjusts for the user's preferred text size
  - Pixels (*px*)
    - Purpose: rarely used directly. Represents actual screen pixels
    - Behavior: does not scale with screen density
    - Use case: only use px for very specific cases (e.g., custom drawing or precise control)

5. Basic components - Units of measurement

- A pixel (px) is the smallest unit of display on a screen
  - Each pixel represents a single point of color
- Density-independent pixels (dp or dpi) refers to the number of pixels that fit into an inch
  - For example, 1 dp is equal to 1 pixel on a medium-density screen (160 dpi) and it automatically scales on screens with higher densities

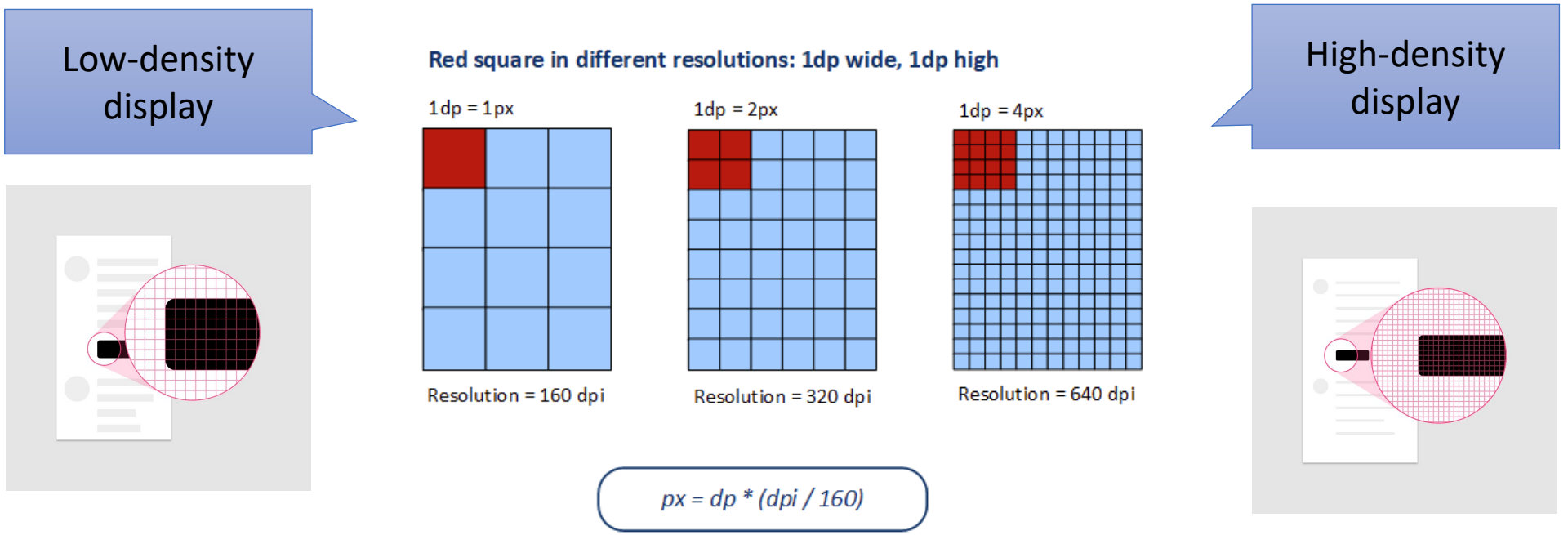
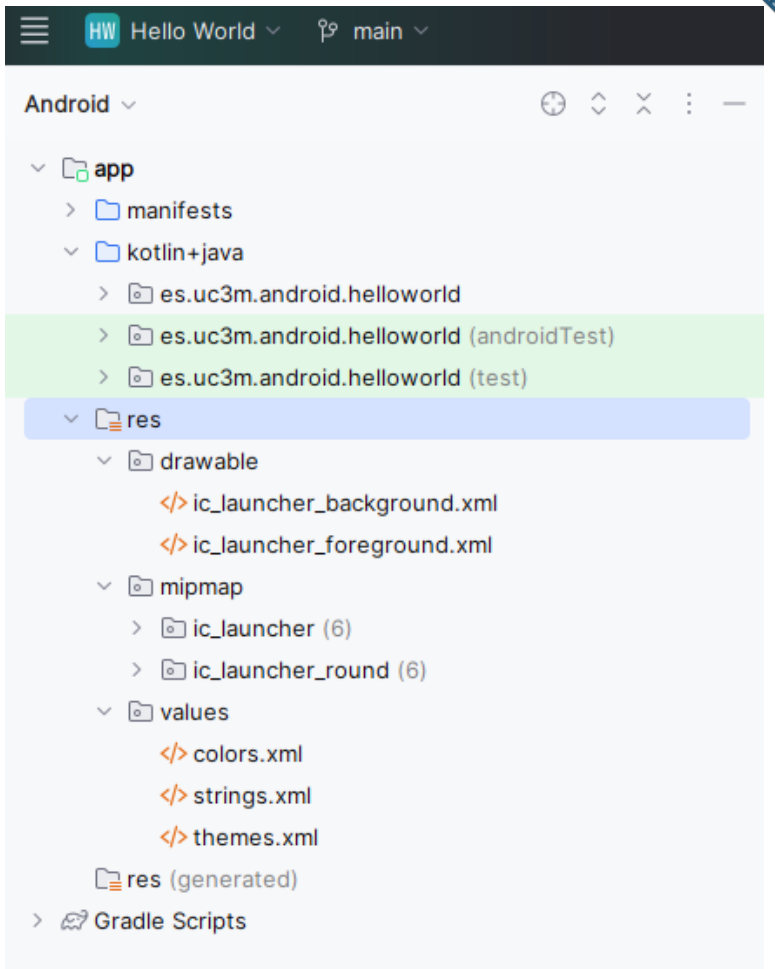


Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
   - Drawable
   - Mipmap
   - Qualifiers
   - Values
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

6. Resources

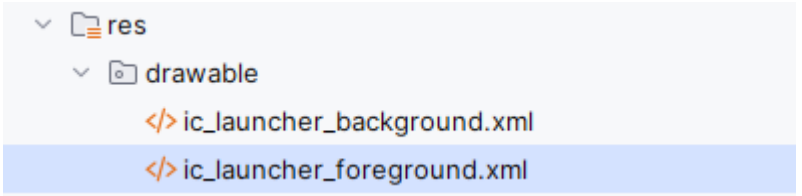
- Resources are the additional files and static content that your app uses, such as images, icons, or, strings
- These resources are located in a folder called `res` within the app module
- We can use these resources in the Kotlin code using the class **R** (a dynamically generated class created during build process to map all resources)

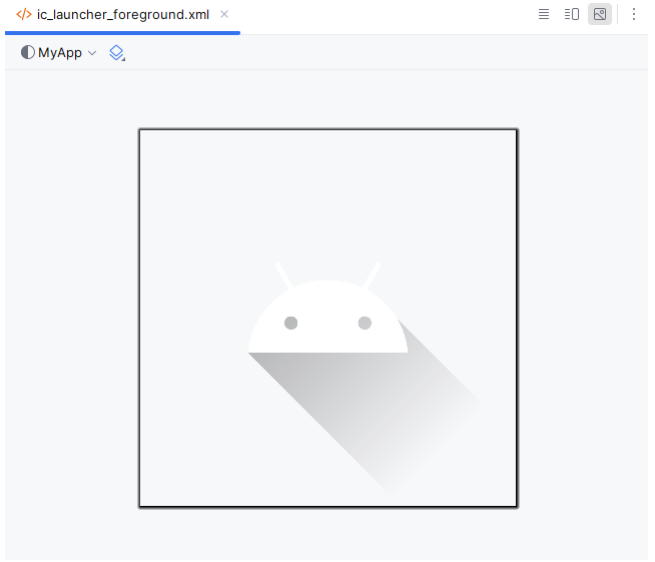


https://developer.android.com/guide/topics/resources/providing-resources

6. Resources - Drawable

- Drawable resources are pictures in the following formats:
  - Bitmap images in PNG, JPG, or other format
  - XML-based vector images
- Android Studio provides a couple of XML-based vector images used for the app icon

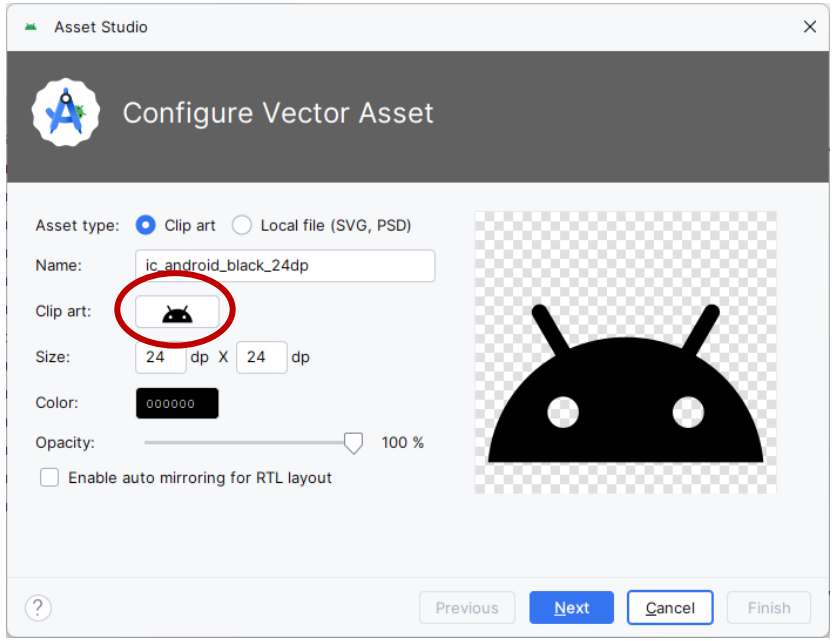


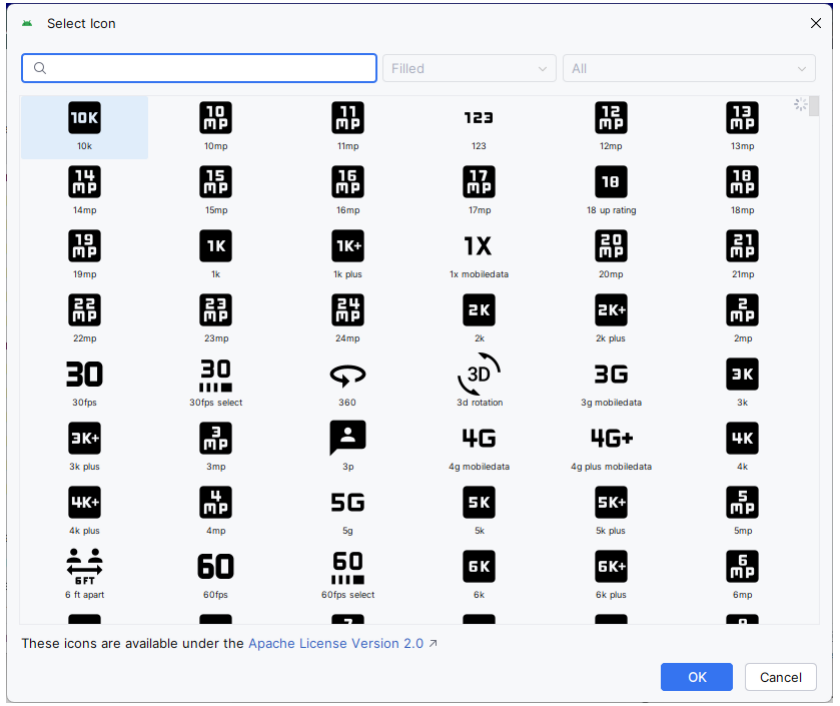


https://developer.android.com/guide/topics/resources/drawable-resource

6. Resources - Drawable

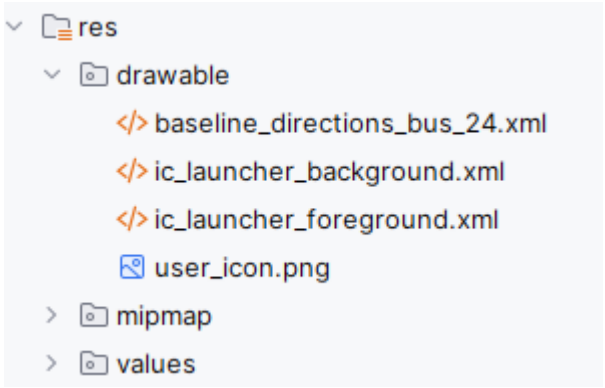
- Android Studio provides a graphical tool to include XML-based vector images for our app:
  - File→New→Vector Asset





6. Resources - Drawable

- We can include custom pictures in the drawable folder and use them in our composables:




```
Image(
    bitmap = ImageBitmap.imageResource(R.drawable.user_icon),
    contentDescription = "Developer image"
)
```

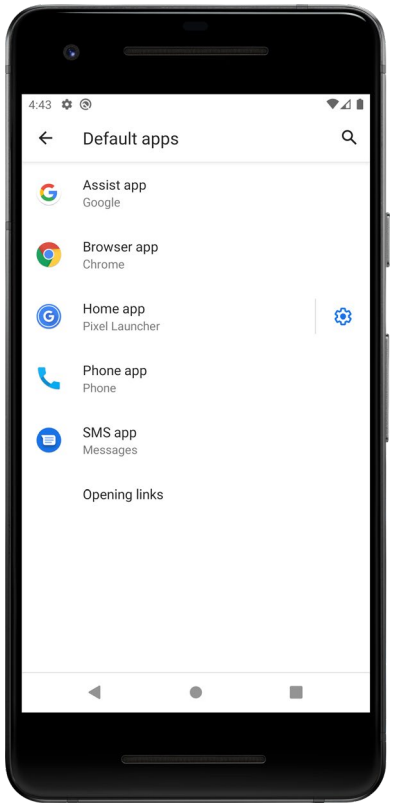
BoxPreview

```
Icon(
    painter = painterResource(R.drawable.baseline_directions_bus_24),
    contentDescription = "App Icon"
)
```

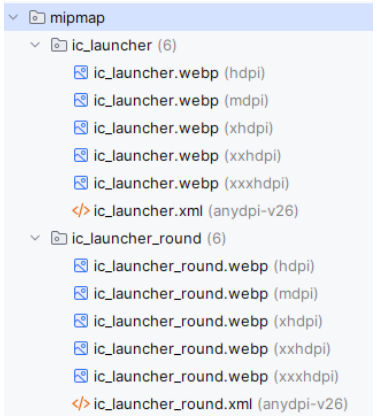
IconSample

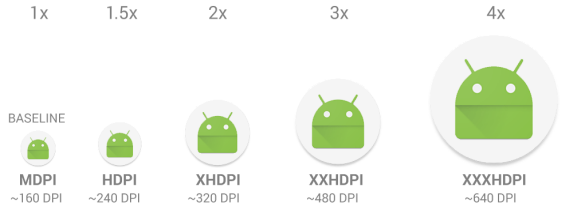
6. Resources - Mipmap

- The mipmap folder(s) provide contains the icons used by the **launcher**
  - The launcher in Android is an app that provides the home screen and overall navigation of the device, allowing to execute the rest of the apps
  - We can configure the Launcher app in Settings→Apps→Default apps



- Since Android runs on a variety of devices that have different screen sizes and pixel densities, it is a good practice to provide icons different pixel densities

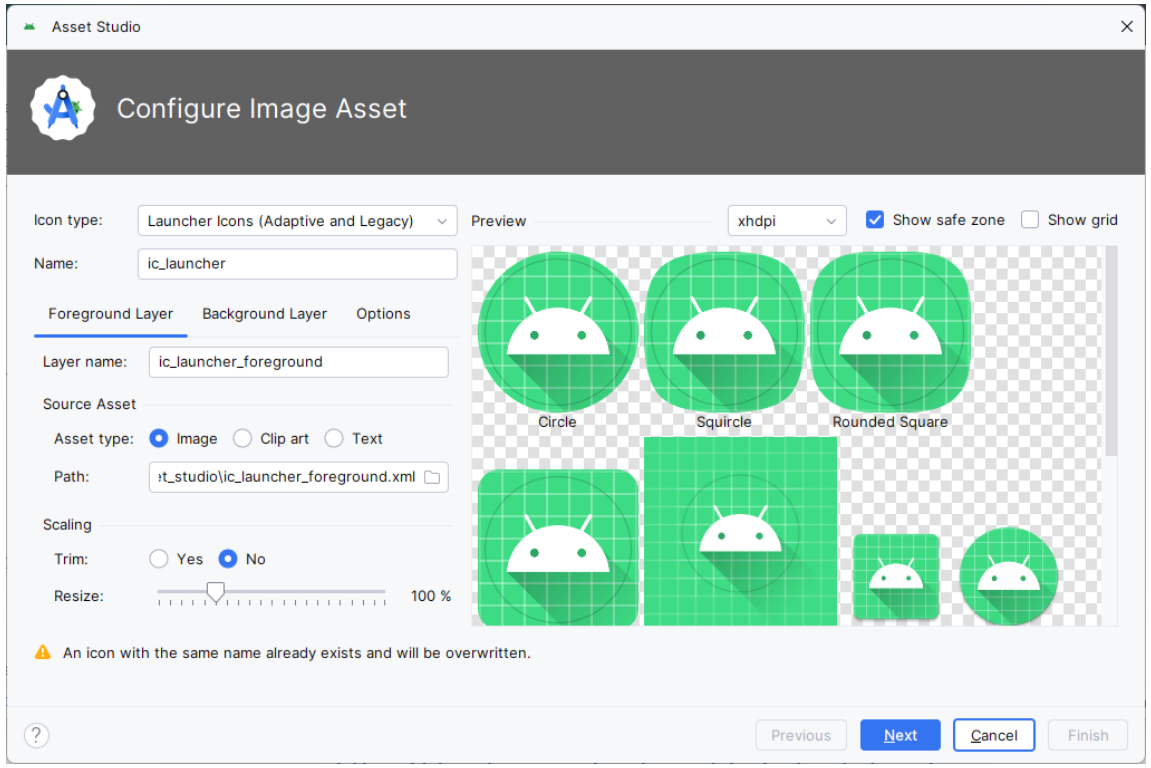




https://developer.android.com/training/multiscreen/screendensities

6. Resources - Mipmap

- Android Studio provides a graphical tools to create icons for our app:
  - File→New→Image Asset



https://developer.android.com/studio/write/create-app-icons

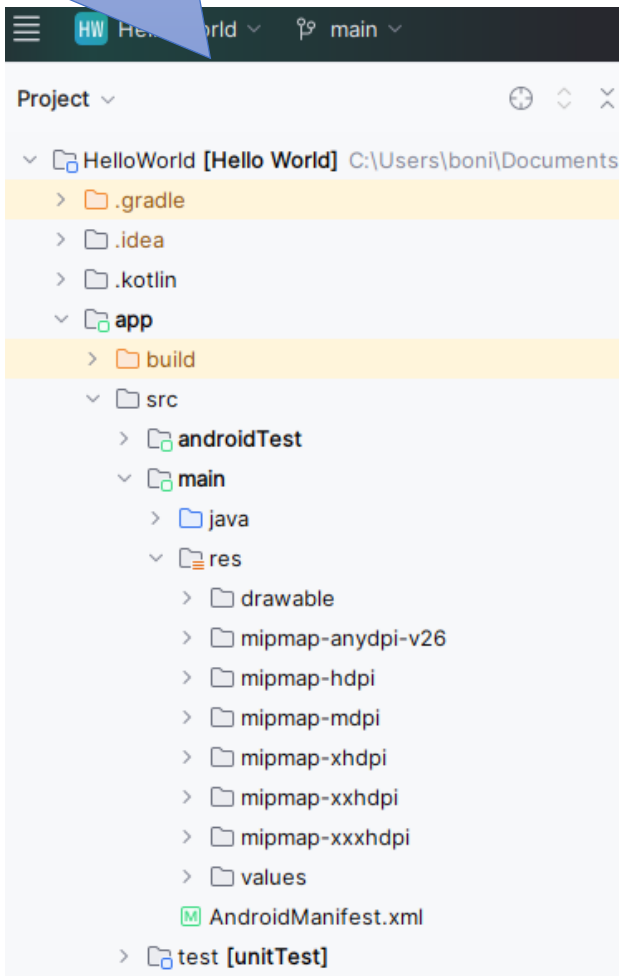
6. Resources - Mipmap

- The common screen densities are the following:

Density	Description
ldpi	Low-density (*ldpi*) screens (~120dpi)
mdpi	Medium-density (*mdpi*) screens (~160dpi)
hdpi	High-density (*hdpi*) screens (~240dpi)
xhdpi	Extra-high-density (*xhdpi*) screens (~320dpi)
xxhdpi	Extra-extra-high-density (*xxhdpi*) screens (~480dpi)
xxxhdpi	Extra-extra-extra-high-density (*xxxhdpi*) uses (~640dpi)
nodpi	These are density-independent resources
tvdpi	Smart TVs screens between mdpi and hdpi (~213dpi)

https://developer.android.com/guide/topics/resources/providing-resources

We can use these density labels as **resource qualifiers**, i.e., alternative resources based on screen densities.



6. Resources - Qualifiers

- Resource qualifiers in Android allows us to provide alternative resources (e.g., drawables, strings) tailored for different devices
- By using resource qualifiers, we can ensure that our app looks and behaves optimally across a wide range of devices and user settings
- Each version of the resource file is placed in a resource directory with a specific qualifier appended to its name
- At runtime, the Android system automatically selects the most appropriate resource file based on the device's current configuration

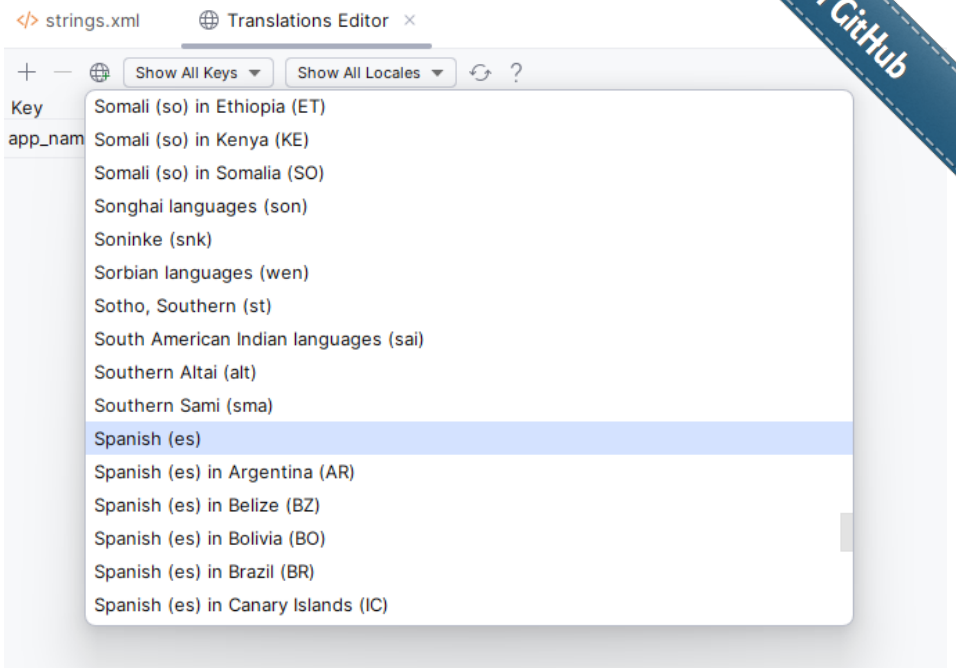
https://developer.android.com/studio/write/add-resources

6. Resources - Values

- The value folder contains XML files with simple values, such as strings or colors
- For creating alternative string resources for different languages (i.e., to create multi language apps), we need to use a locale qualifier (ISO 639-1 codes), for instance:
  - string-es.xml : For Spanish language
  - string-es-rES.xml : For Spanish language and Spain region
- A simple way to include these locale message (for multilanguage apps) is by using Android Studio

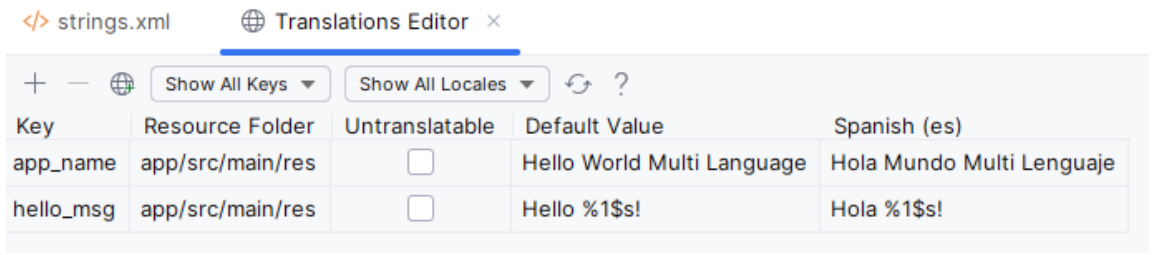
https://developer.android.com/guide/topics/resources/providing-resources

6. Resources - Values



Finally, we read the string values from our composables

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = stringResource(R.string.hello_msg, name),
        modifier = modifier
    )
}
```



6. Resources - Values

- If we change the system language (Settings→System→Language & input→Language), our app will use the locale messages

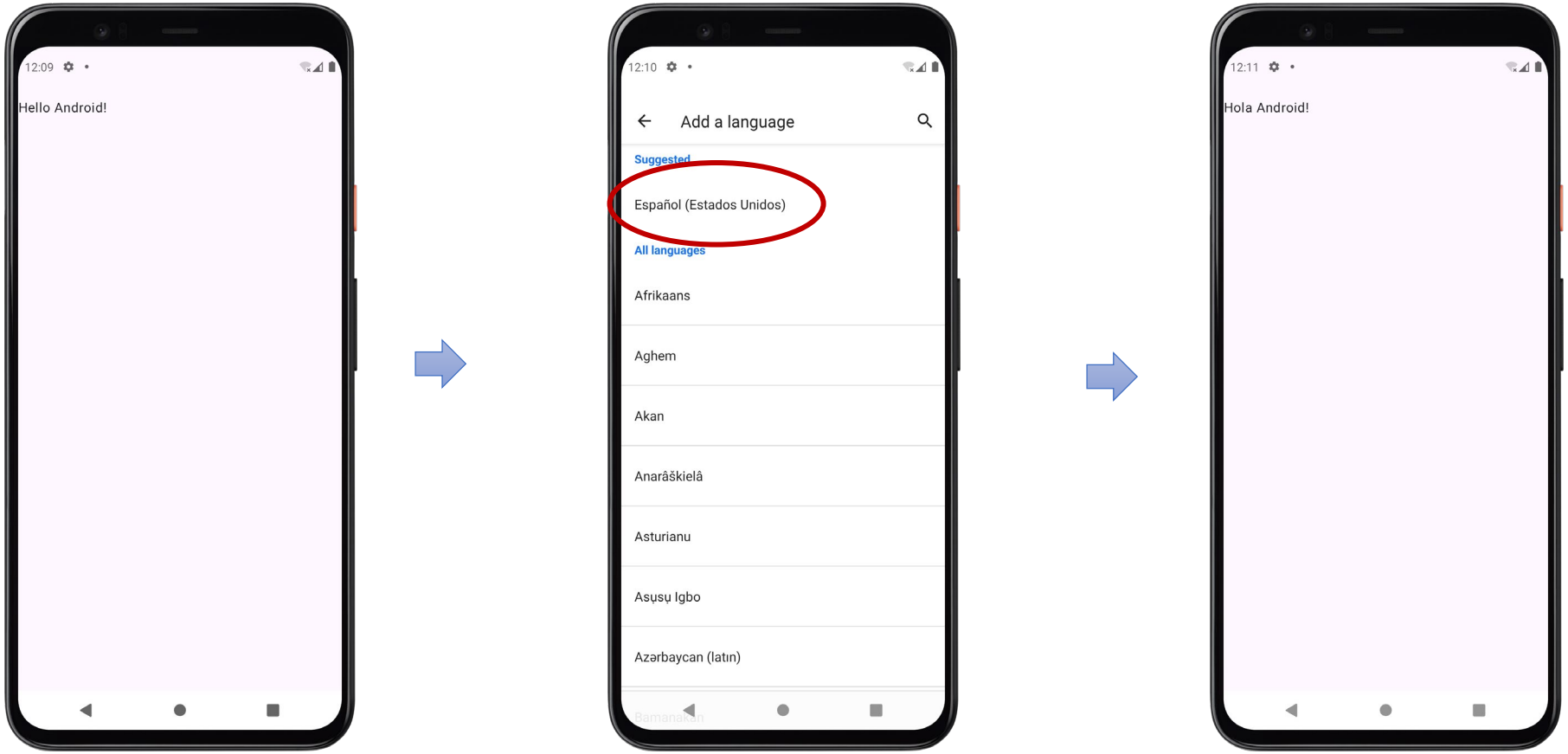


Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
   - Constraint layout
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

7. Layouts

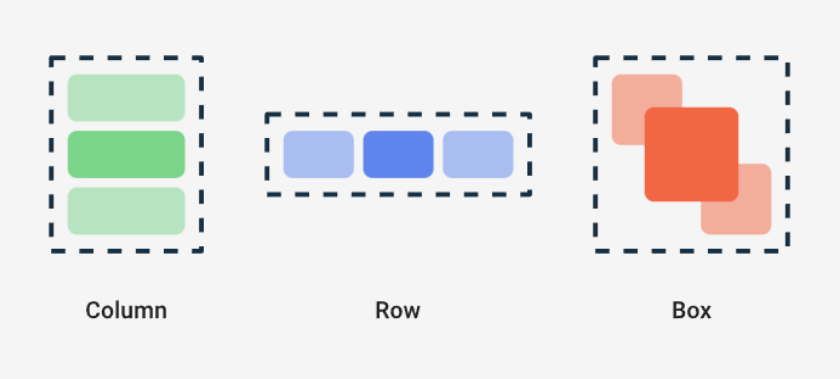
- If we don’t provide some guidance to organize our UI elements, they are arranged in the screen in a way we don’t like
  - To avoid this problem, we to organize the UI elements using a layout

```
@Composable
fun DevCard() {
    Text("John Doe")
    Text("Developer")
}
```

- A **layout** refers to the arrangement and organization of visual elements within a screen and determines how components are positioned relative to each other

7. Layouts

- Compose provides a collection of built-in **layouts** to help us arrange our UI elements. The basic composable layouts are:
  - `Column` to place items vertically on the screen
  - `Row` to place items horizontally on the screen
  - `Box` to put elements on top of another
  - `Spacer` to adds empty space between composables



https://developer.android.com/develop/ui/compose/layouts/basics

7. Layouts

Column layout

```
@Composable
fun DevCardColumn() {
    Column {
        Text("John Doe")
        Text("Developer")
    }
}
```

ColumnPreview

John Doe
Developer

Row layout

```
@Composable
fun DevCardRow() {
    Row {
        Text("John Doe")
        Text(" - ")
        Text("Developer")
    }
}
```

RowPreview

John Doe - Developer

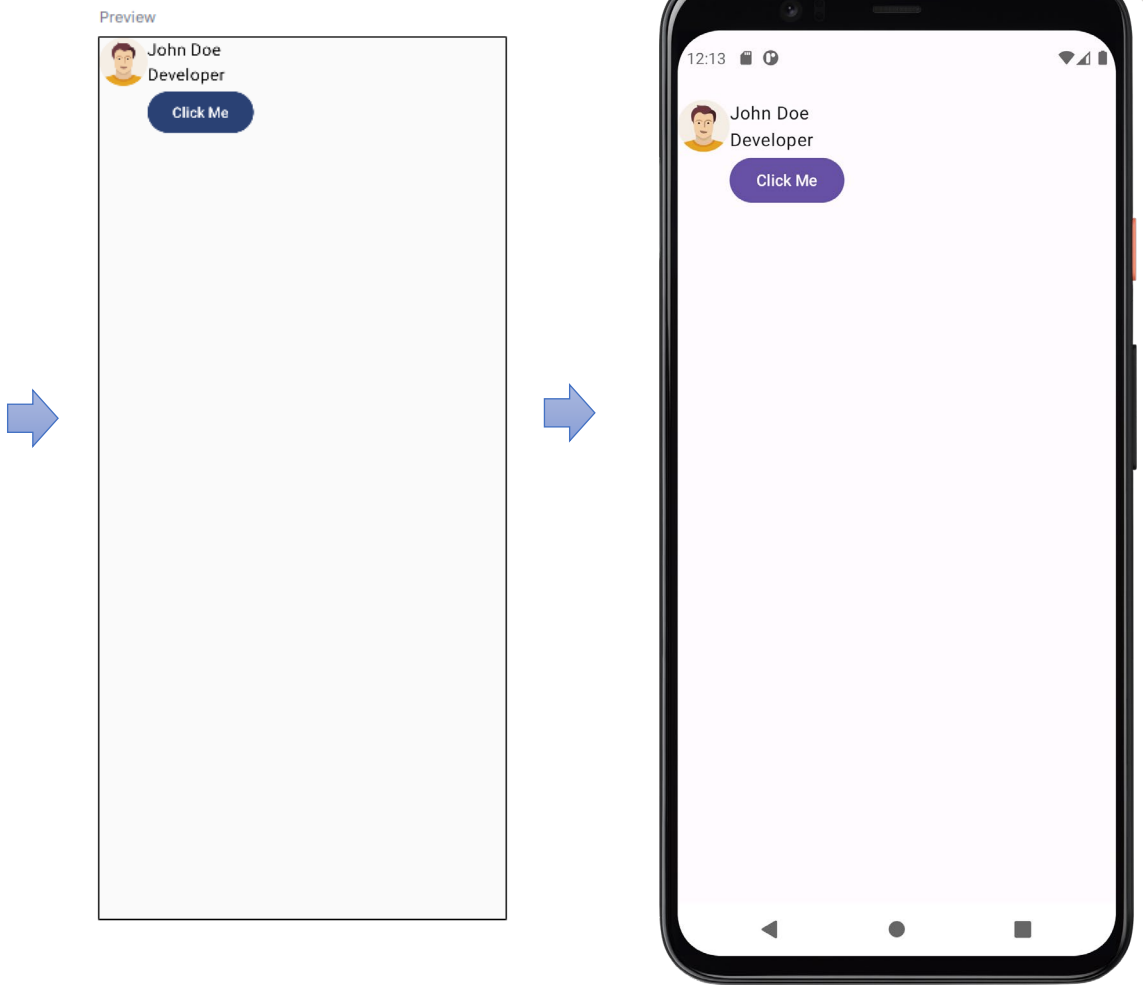
Box layout

```
@Composable
fun DevCardBox() {
    Box {
        Image(
            bitmap = ImageBitmap.imageResource(R.drawable.user_icon),
            contentDescription = "Developer image"
        )
        Icon(
            imageVector = Icons.Outlined.Edit,
            contentDescription = "Edit user"
        )
    }
}
```

BoxPreview

7. Layouts

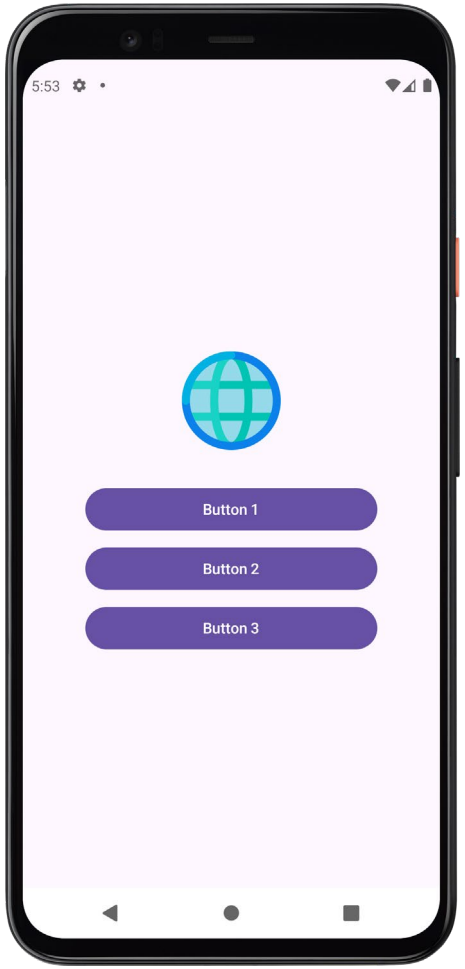

```
data class Developer(var name: String, var role: String)

@Composable
fun DevLayout(developer: Developer) {
    Row {
        Image(
            bitmap = ImageBitmap.imageResource(R.drawable.user_icon),
            contentDescription = "Developer image"
        )
        Column {
            Text(developer.name)
            Text(developer.role)
            Button(
                onClick = {
                    Log.d("MainActivity", "Button clicked")
                },
                content = {
                    Text("Click Me")
                }
            )
        }
    }
}


@Preview(showBackground = true)
@Composable
fun Preview() {
    MyAppTheme {
        DevLayout(Developer("John Doe", "Developer"))
    }
}
```



7. Layouts

- The following example shows a more elaborated layout using basic elements:

```
@Composable
fun MyLayout(modifier: Modifier = Modifier) {
    Row {
        // Left spacer (15% width)
        Spacer(
            modifier = modifier
                .weight(0.15f)
                .fillMaxHeight()
        )

        // Middle content (70% width)
        Box(
            modifier = modifier
                .weight(0.7f)
                .fillMaxHeight(),
            contentAlignment = Alignment.Center
        ) {
            Column(
                horizontalAlignment = Alignment.CenterHorizontally
            ) {
                // Logo and buttons
            }
        }

        // Right spacer (15% width)
        Spacer(
            modifier = modifier
                .weight(0.15f)
                .fillMaxHeight()
        )
    }
}
```



7. Layouts - Constraint layout

- `ConstraintLayout` is a layout that allows you to place composables relative to other composables on the screen
  - It is an alternative to using multiple nested `Row`, `Column`, and `Box`
  - `ConstraintLayout` is useful when implementing responsive layouts with complicated alignment requirements
- To use `ConstraintLayout`, first we need to include the following dependency in our project:

build.gradle.kts

```
dependencies {
    implementation(libs.androidx.constraintlayout.compose)
}
```

libs.version.toml

```
[versions]
constraintlayoutCompose = "1.1.0"

[libraries]
androidx-constraintlayout-compose = { module = "androidx.constraintlayout:constraintlayout-compose", version.ref = "constraintlayoutCompose" }
```

https://developer.android.com/develop/ui/compose/layouts/constraintlayout

7. Layouts - Constraint layout

- We define constraints (e.g., `start`, `end`, `top`, `bottom`) between composables to position them relative to each other or to the parent
  - It also allows us to create barriers (dynamic boundaries) and guidelines (fixed or percentage-based lines) for advanced layouts
- Let's see the following example:

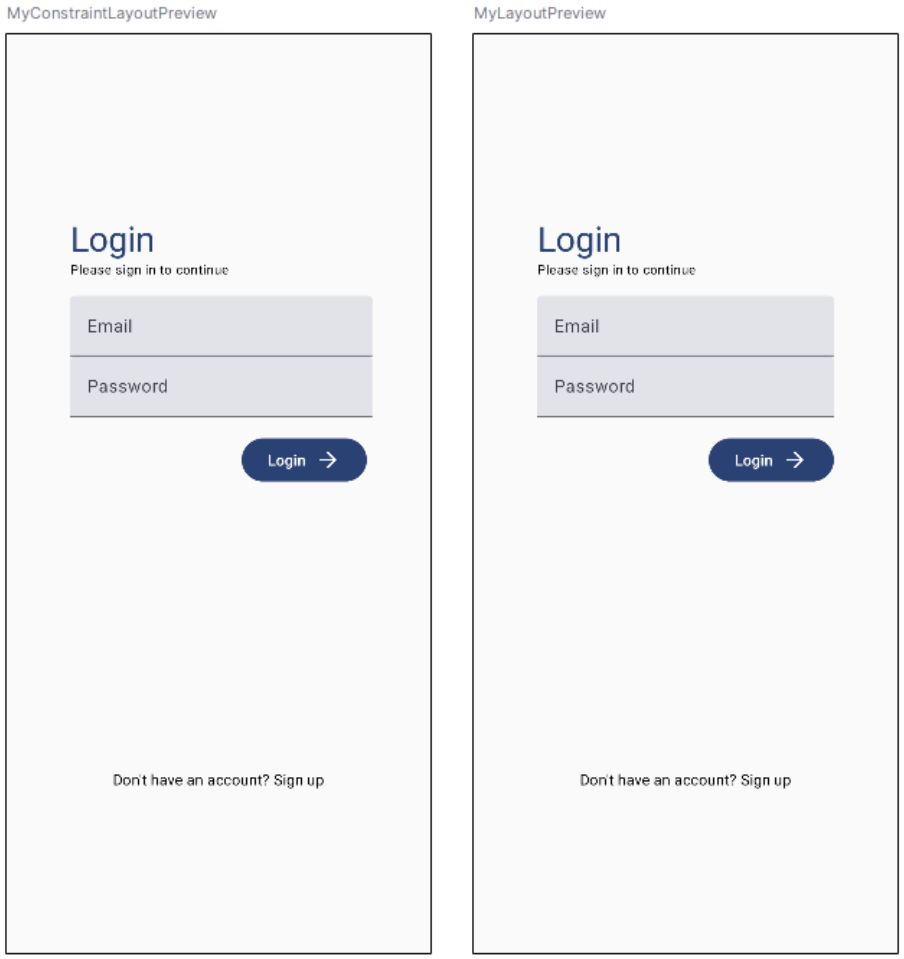
7. Layouts - Constraint layout


```
@Composable
fun MyConstraintLayout(modifier: Modifier = Modifier) {
    ConstraintLayout(
        modifier = modifier
    ) {
        // Create references for the components
        val (text, button) = createRefs()

        // Text component
        Text(
            text = stringResource(R.string.hello_msg),
            modifier = Modifier
                .constrainAs(text) {
                    top.linkTo(parent.top, margin = 16.dp)
                    start.linkTo(parent.start, margin = 16.dp)
                }
        )

        // Button component
        Button(
            onClick = {
                // TODO: Handle click
            },
            modifier = Modifier
                .constrainAs(button) {
                    top.linkTo(text.bottom, margin = 16.dp)
                    start.linkTo(parent.start, margin = 16.dp)
                }
        ) {
            Text(stringResource(R.string.button_msg))
        }
    }
}
```

7. Layouts - Constraint layout

- The following example contains a more elaborated layout using constraints:

This example also contains the equivalent layout but using the nested `Row`, `Column`, and `Box`



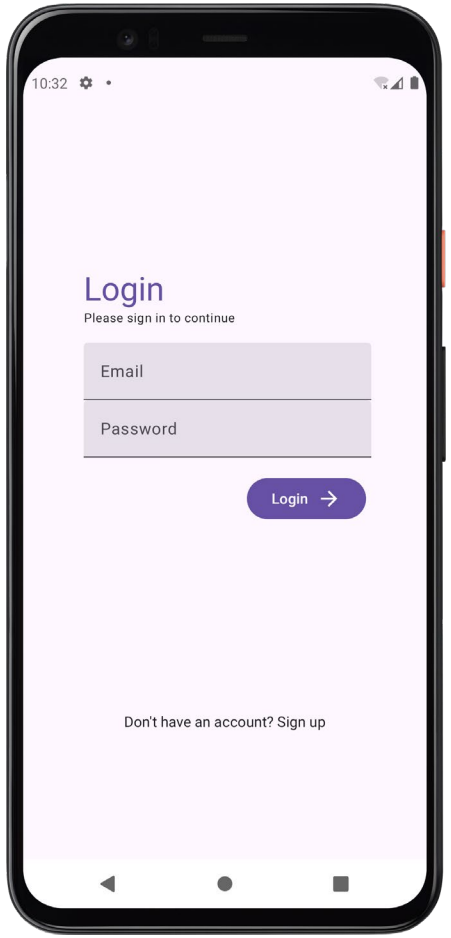


Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
   - Android View system
   - Jetpack Compose
   - Dark mode
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

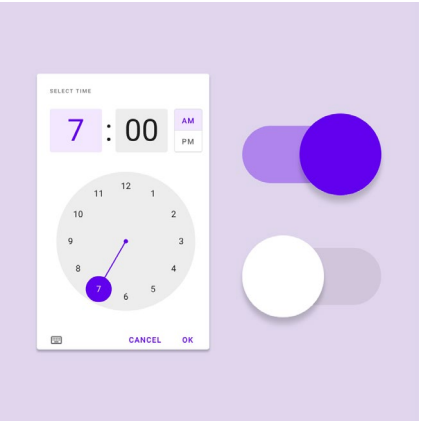
8. Theme

- A **theme** is a collection of **styles** (e.g., colors, fonts, dimensions) that define the overall look and feel of an app
- *Theming* refers to the process of defining and applying a consistent visual style across an app, and its purpose is:
  - Consistency: Ensures that all screens and components in the app look and behave consistently
  - Branding: Reflects the app's brand identity through colors, fonts, and shapes
  - Adaptability: Supports light and dark themes, as well as other device-specific configurations (e.g., screen size, orientation)
  - Maintainability: Centralizes style definitions, making it easier to update the app's appearance

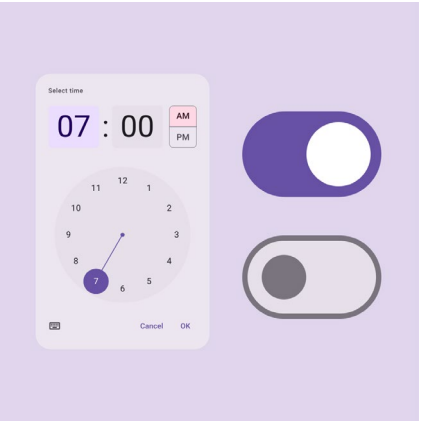
https://developer.android.com/design/ui/mobile/guides/styles/themes

8. Theme

- **Material Design** is a design language developed by Google that aims to create a unified and consistent look and feel across different platforms and devices
  - It was introduced in 2014 and has since become a standard for designing Android applications, web applications, and other digital products
- Material Design provides a set of guidelines, principles, and components to help designers and developers create visually appealing and user-friendly interfaces
  - Android Studio applies Material Design (3, currently) by default when creating Android projects (e.g., in the *hello world* project)



Material Design 2



Material Design 3
(a.k.a. Material 3 or M3)

https://material.io/

8. Theme

- In Android, theming can be applied in two main ways:

1. XML-based:
   - Used in the traditional Android View system
   - Theming is done using XML files in the res/values directory

https://developer.android.com/develop/ui/views/theming/themes

2. Programmatic:
   - Uses in Jetpack Compose, the modern UI toolkit
   - Theming is done programmatically using Kotlin

Even if we use Jetpack Compose for the UI, we also need to consider the traditional XML-based system

https://developer.android.com/develop/ui/compose/designsystems

8. Theme - Android View system

- In the traditional Android View system, theming is done using XML files in the `res/values` directory

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="Theme.MyApp" parent="android:Theme.Material.Light.NoActionBar" />
</resources>
```

Themes are declared in the file `res/values/themes.xml`

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```

Colors (in RGBA) are declared in the file `res/values/colors.xml`

8. Theme - Android View system

- The XML theme is declared in the manifest:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:theme="@style/Theme.MyApp">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

res/values/themes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="Theme.MyApp" parent="android:Theme.Material.Light.NoActionBar" />
</resources>
```

8. Theme - Android View system

- We can change the XML parent theme:
  - Theme.Material.Light
  - Theme.Material.Light.NoActionBar
  - Theme.Material.Light.NoActionBar.Fullscreen
  - Theme.Material.Dialog
  - Theme.Material.Settings
  - Theme.Material.InputMethod
  - Theme.Material.NoActionBar
  - Theme.Material.Panel
  - Theme.Material.Voice
  - Theme.Material.Wallpaper

This one is used by default in the projects created with Android Studio

For example, we can change the following in any app demo (e.g., in the hello-world)

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="Theme.MyApp" parent="android:Theme.Material.Light" />
</resources>
```

8. Theme - Android View system

- Also, we can customize the XML theme by overriding the default values for colors, typography, and shapes

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="Theme.MyApp" parent="android:Theme.Material.Light.NoActionBar">
        <!-- Customize theme here -->
        <item name="android:colorPrimary">@color/purple_200</item>
    </style>
</resources>
```

8. Theme - Jetpack Compose

- Jetpack Compose provides an implementation of Material Design 3 using a composable called **`MaterialTheme`**
  - This composable provides a consistent way to define and apply colors, typography, and shapes throughout our app
  - For that, it includes pre-defined components for:
    - Colors: Primary, secondary, background, surface, etc.
    - Typography: Headlines, body text, captions, etc.
    - Shapes: Small, medium, and large components

https://developer.android.com/develop/ui/compose/designsystems/material3

8. Theme - Jetpack Compose

- A common practice for theming with Jetpack Compose is to customize **`MaterialTheme`** by overriding its default values for colors and typography
- For that, we usually define a custom theme, color palette, and typography in separate Kotlin files:
  - **`Color.kt`**: to define the color palette for our app
    - Defined through global constants to ensure consistency and reusability
  - **`Type.kt`**: to define the typography for our app
    - It includes text styles for different UI elements (e.g., headlines, body text, captions)
  - **`Theme.kt`**: To define the theme for our app
    - Combines the color palette, typography, and shapes into a single theme that can be applied to the entire app
    - Typically supports light and dark color schemas by defining separate color schemes

8. Theme - Jetpack Compose

<namespace>/ui/theme/Color.kt

```
val Purple80 = Color(0xFFD0BCFF)
val PurpleGrey80 = Color(0xFFCCC2DC)
val Pink80 = Color(0xFFEFB8C8)

val Purple40 = Color(0xFF6650a4)
val PurpleGrey40 = Color(0xFF625b71)
val Pink40 = Color(0xFF7D5260)
```

<namespace>/ui/theme/Type.kt

```
val Typography = Typography(
    bodyLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp,
        lineHeight = 24.sp,
        letterSpacing = 0.5.sp
    )
)
```

Material Design 3 introduces dynamic color, which allows the app's theme to adapt to the user's wallpaper or system settings

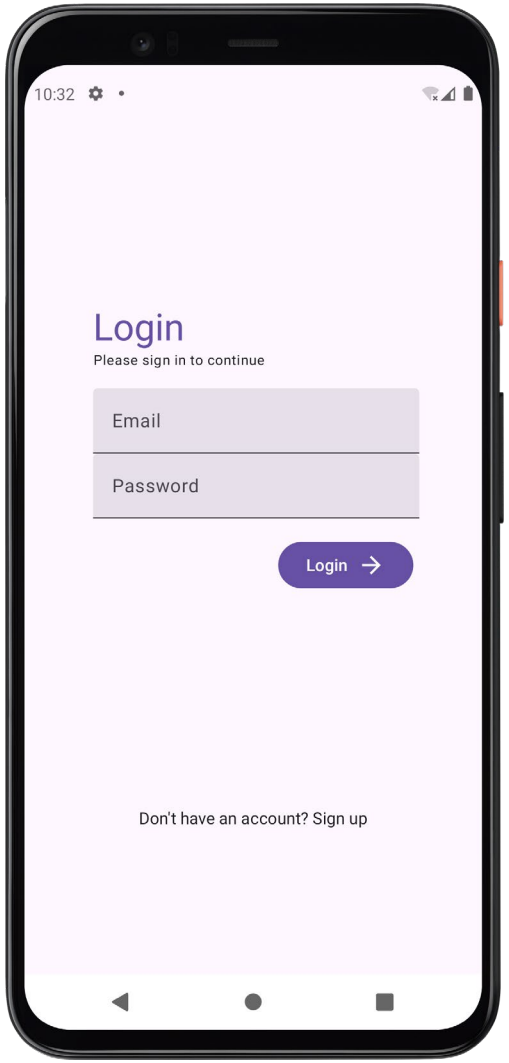
<namespace>/ui/theme/Theme.kt

```
private val DarkColorScheme = darkColorScheme(
    primary = Purple80,
    secondary = PurpleGrey80,
    tertiary = Pink80
)

private val LightColorScheme = lightColorScheme(
    primary = Purple40,
    secondary = PurpleGrey40,
    tertiary = Pink40
)

@Composable
fun MyAppTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    dynamicColor: Boolean = true,
    content: @Composable () -> Unit
) {
    val colorScheme = when {
        dynamicColor && Build.VERSION.SDK_INT >= Build.VERSION_CODES.S -> {
            val context = LocalContext.current
            if (darkTheme) dynamicDarkColorScheme(context) else dynamicLightColorScheme(context)
        }

        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }

    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography,
        content = content
    )
}
```

8. Theme - Jetpack Compose

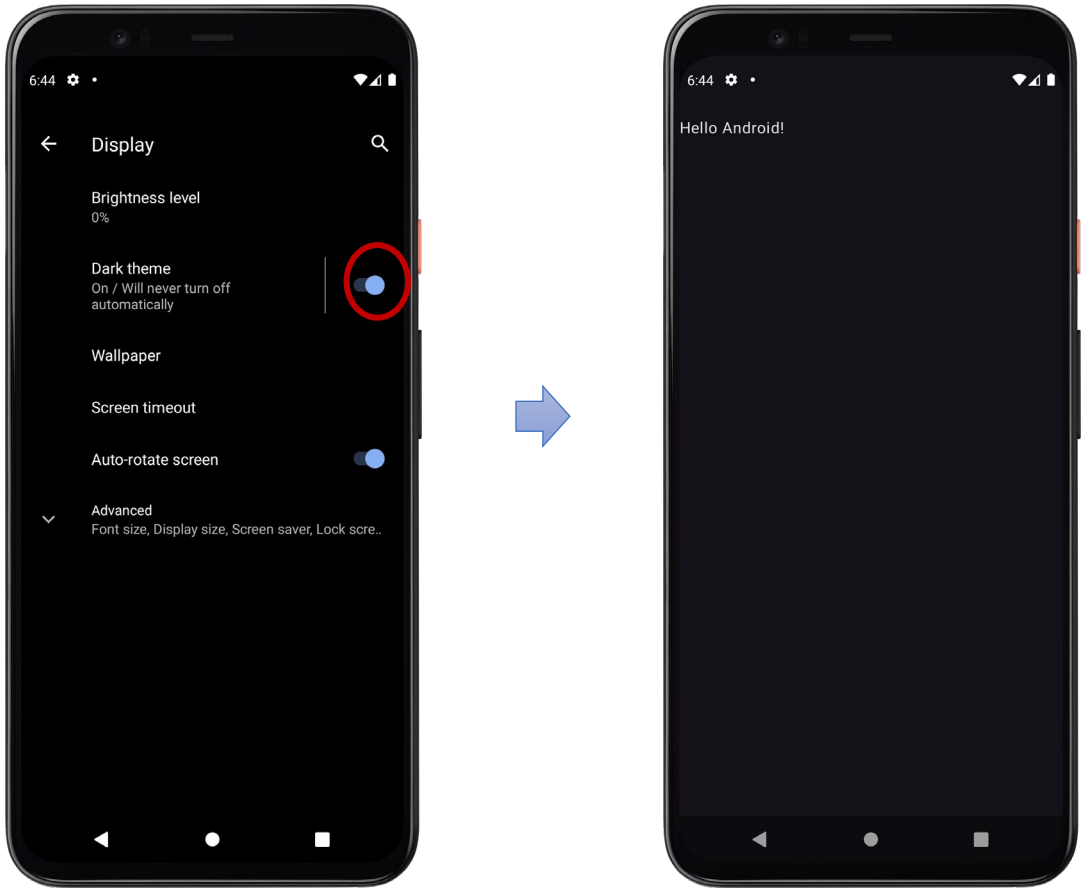
We can apply different `MaterialTheme` styles for example as follows:

```
Text(
    text = stringResource(R.string.login_label),
    style = MaterialTheme.typography.headlineLarge,
    color = MaterialTheme.colorScheme.primary
)
Text(
    text = stringResource(R.string.sign_in_to_continue),
    style = MaterialTheme.typography.bodySmall
)
```



8. Theme - Dark mode

- We can enable the dark mode In the Android configuration (Settings → Display)



In Jetpack Compose, themes are propagated through the composition tree. If we don't use a `Scaffold` or another Material component (e.g., `Surface`) that properly propagates the theme, the theme might not be applied correctly to our UI components

8. Theme - Dark mode

- The following example is a modified version of the hello-world app using `Surface` instead of `Scaffold`:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyAppTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting(name = "Android")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String) {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(text = "Hello $name!")
    }
}
```

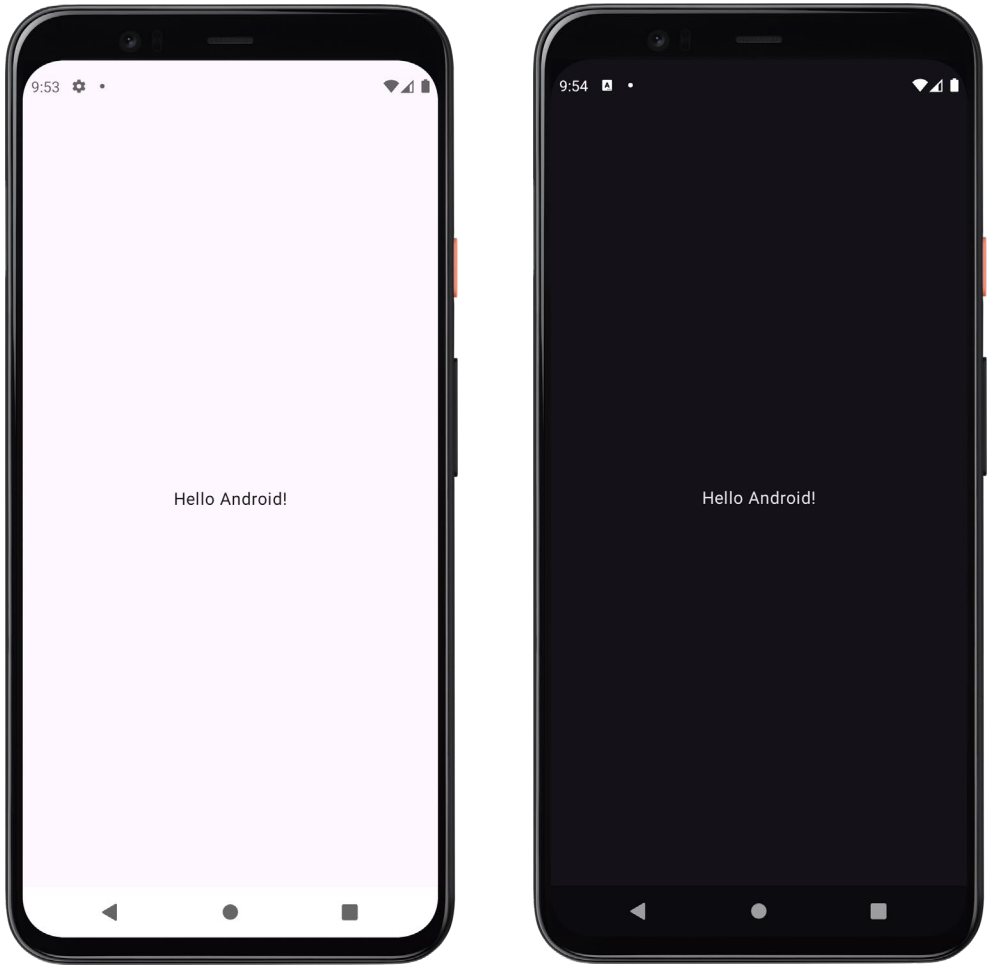


Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
   - Scaffold
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

9. Material components

- Jetpack Compose offers an implementation of the Material components a collection of composable functions
- The following slides reviews the most relevant material components
- If you need to use some of the following, it is recommended to check the official documentation for code examples:

  https://developer.android.com/develop/ui/compose/components

- Also, the following site provides a good reference of Material composables:

  https://composables.com/material/

9. Material components

- App bar:
  - Containers that provide the user access to key features and navigation items

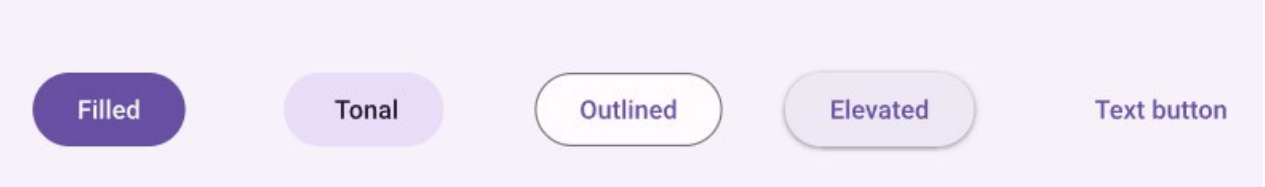
https://developer.android.com/develop/ui/compose/components/app-bars
https://composables.com/app-bars

- Badge:
  - Small visual element to denote status or a numeric value on another composable

https://developer.android.com/develop/ui/compose/components/badges
https://composables.com/badges

9. Material components

- Button:
  - Fundamental components that allow the user to trigger a defined action



https://developer.android.com/develop/ui/compose/components/button
https://composables.com/buttons

- Segmented button:
  - Let users choose from a set of options

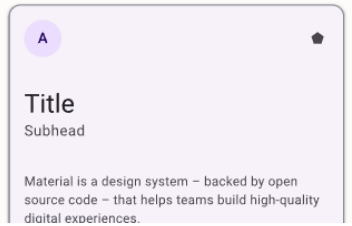
https://developer.android.com/develop/ui/compose/components/segmented-button

9. Material components

- Bottom sheet:
  - Supplementary content that are anchored to the bottom of the screen

https://developer.android.com/develop/ui/compose/components/bottom-sheets
https://composables.com/sheets

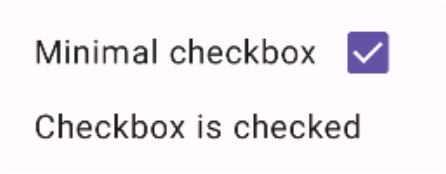
- Card:
  - Container for our UI. Cards typically present a single coherent piece of content

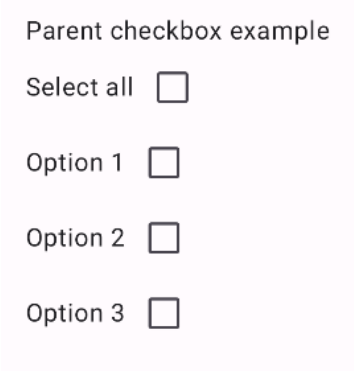


https://developer.android.com/develop/ui/compose/components/card
https://composables.com/cards

9. Material components

- Checkbox:
  - Elements that let users select one or more items from a list





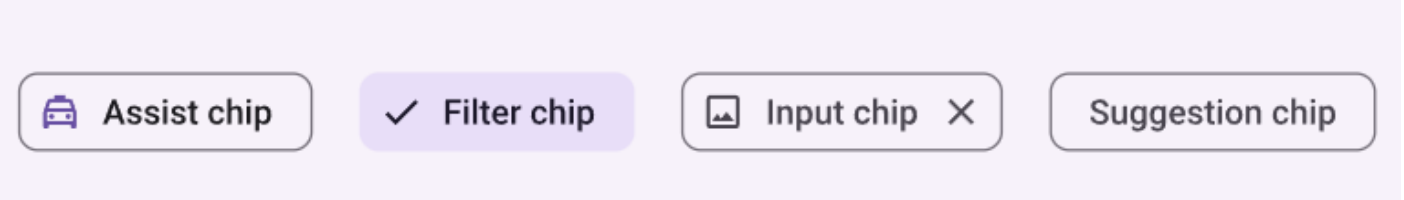
https://developer.android.com/develop/ui/compose/components/checkbox
https://composables.com/checkboxes

- Radio button:
  - To select only one option from a list

https://developer.android.com/develop/ui/compose/components/radio-button
https://composables.com/radio-buttons

9. Material components

- Chip:
  - Represents complex entities like a contact or tag, often with an icon and label



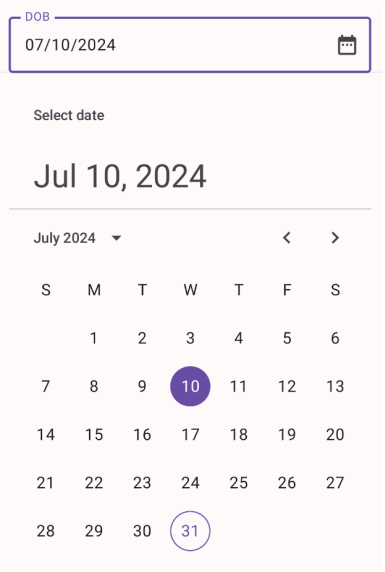
https://developer.android.com/develop/ui/compose/components/chip
https://composables.com/chips

- Switch:
  - To toggle between two states (checked and unchecked)

https://developer.android.com/develop/ui/compose/components/switch
https://composables.com/switches

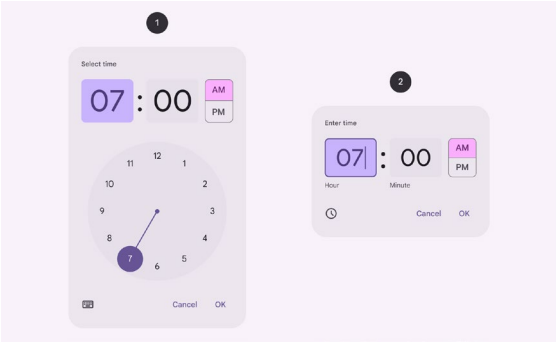
9. Material components

- Date picker:
  - To select a date, a date range, or both



https://developer.android.com/develop/ui/compose/components/datepickers
https://composables.com/date

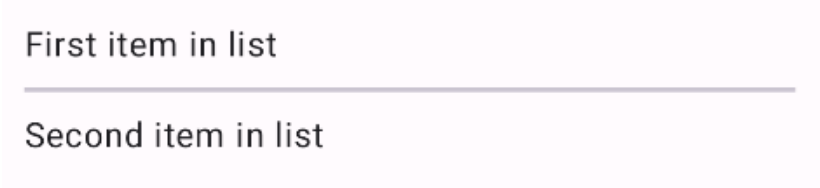
- Time picker:
  - To select a time



https://developer.android.com/develop/ui/compose/components/time-pickers
https://composables.com/time

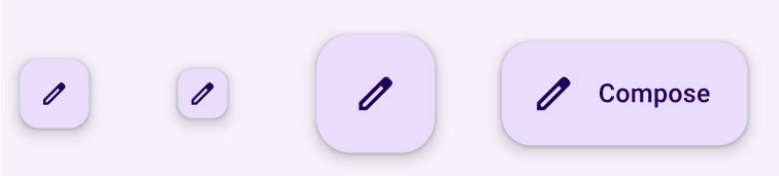
9. Material components

- Divider:
  - Thin lines that separate items in lists or other containers



https://developer.android.com/develop/ui/compose/components/divider
https://composables.com/dividers

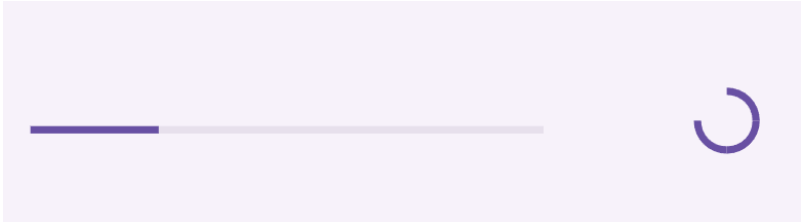
- Floating action button:
  - High-emphasis button that lets the user perform a primary action in an application



https://developer.android.com/develop/ui/compose/components/fab
https://composables.com/floating-action-buttons

9. Material components

- Progress indicator:
  - Visually represent the status of an operation



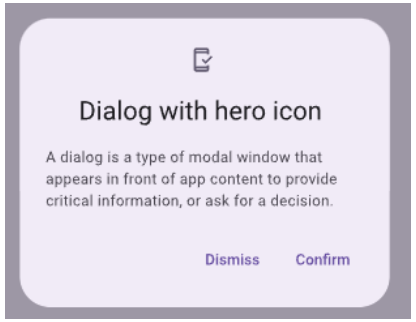
https://developer.android.com/develop/ui/compose/components/progress
https://composables.com/progress-indicators

- Slider:
  - To make selections from a range of values

https://developer.android.com/develop/ui/compose/components/slider
https://composables.com/sliders

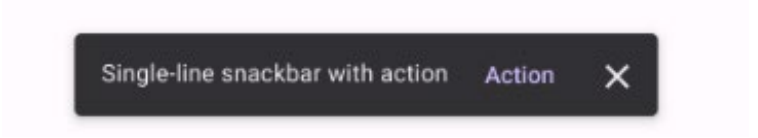
9. Material components

- Dialog:
  - Component that displays pop up messages or requests user input



https://developer.android.com/develop/ui/compose/components/dialog
https://composables.com/dialogs

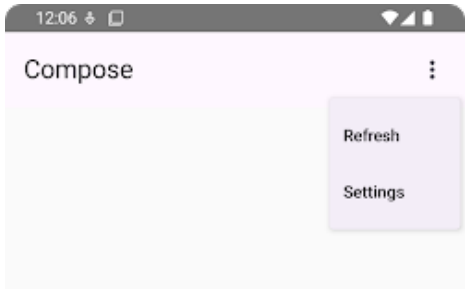
- Snackbar:
  - Brief notification that appears at the bottom of the screen



https://developer.android.com/develop/ui/compose/components/snackbar
https://composables.com/snackbars

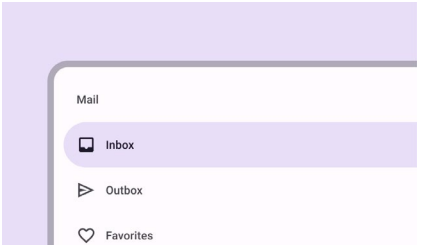
9. Material components

- Drop-down menus
  - Let users select from a list of options on a temporary surface



https://developer.android.com/develop/ui/compose/components/menu
https://composables.com/dropdown-menus

- Navigation drawer:
  - Slide-in menu that lets users navigate to various sections of your app



https://developer.android.com/develop/ui/compose/components/drawer
https://composables.com/drawers

9. Material components - Scaffold

- Scaffold is a pre-built composable In Jetpack Compose that provides a high-level structure for implementing Material Design layouts
  - It simplifies the process of creating common UI patterns by offering slots for components like top bars, bottom bars, floating action buttons, snackbars, and content
- Scaffold provides slots for the following components:
  - Top Bar: For app bars (e.g., `TopAppBar`)
  - Bottom Bar: For bottom navigation bars (e.g., `BottomAppBar`)
  - Floating Action Button: A prominent button for primary actions
  - Snackbar: For displaying short messages at the bottom of the screen
  - Content: The main content of the screen (e.g., a `Column`, `LazyColumn`, or other composables)
  - Drawer: For navigation drawers (e.g., `ModalDrawer`)

9. Material components - Scaffold


```
@Composable
fun MyScreen() {

    // ...

    ModalNavigationDrawer(
        drawerState = drawerState,
        drawerContent = {
            MyDrawerContent(items, scope, drawerState)
        },
    ) {
        Scaffold(
            topBar = {
                MyTopAppBar(scope, drawerState)
            },
            floatingActionButton = {
                MyFloatingActionButtons()
            },
            content = { innerPadding ->
                MyContent(Modifier.padding(innerPadding))
            },
            bottomBar = {
                MyNavigationBar(items)
            }
        )
    }
}
```

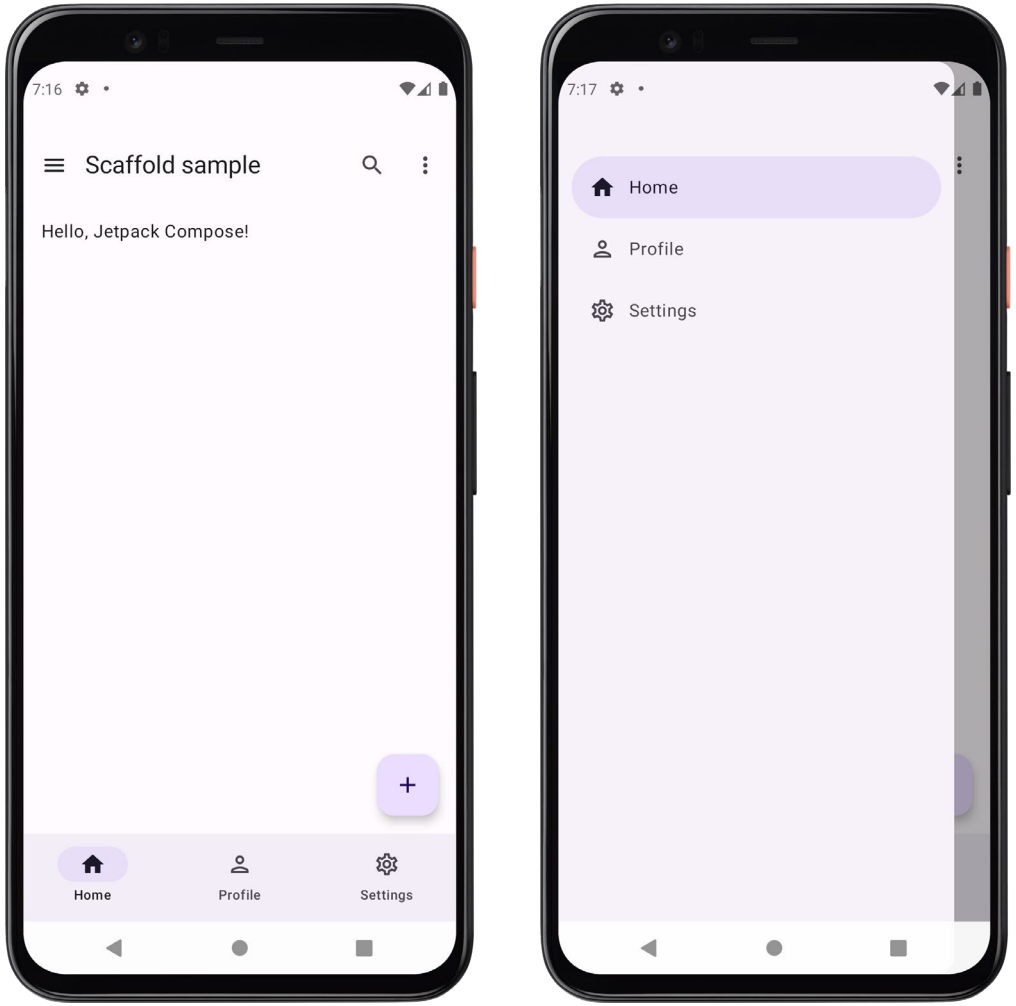


Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
    - Stateful
    - Stateless
    - State restoration
    - View model
    - Text fields
11. Navigation
12. List and grids
13. Animations
14. Takeaways

10. State management

- In Jetpack Compose, the **state** refers to any data that can change over time and affects the UI
  - When the state changes, Compose automatically recomposes (re-draws) the affected parts of the UI to reflect the new state
- To manage state in Jetpack Compose, we use the following functions:
  - `remember`: Retains state across recompositions
  - `mutableStateOf`: Creates a mutable state that triggers recomposition when changed
  - `rememberSaveable`: Retains state across configuration changes (e.g., screen rotation)
  - `ViewModel`: For managing complex state that survives configuration changes

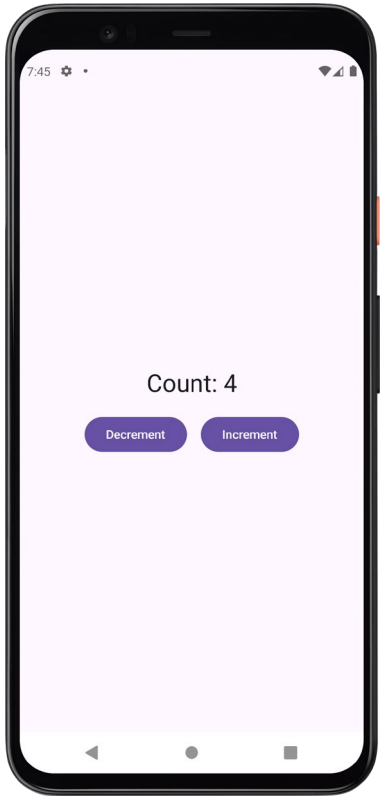
https://developer.android.com/develop/ui/compose/state

10. State management - Stateful

- A composable that uses `remember` to store an object creates internal state, making the composable **stateful**
  - Example: A counter app where the UI updates whenever the count changes

In this example, we use the specialized function for creating a mutable state for integers (*mutableIntStateOf*) to create a **state variable** called count

```
@Composable
fun Counter() {
    // State is managed internally within the composable
    var count by remember { mutableIntStateOf(0) }

    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            text = "Count: $count",
            style = MaterialTheme.typography.headlineMedium
        )
        Spacer(modifier = Modifier.height(16.dp))
        Row(
            horizontalArrangement = Arrangement.spacedBy(16.dp)
        ) {
            Button(onClick = { count++ }) {
                Text(stringResource(R.string.increment))
            }
            Button(onClick = { count-- }) {
                Text(stringResource(R.string.decrement))
            }
        }
    }
}
```



10. State management - Stateless

- A **stateless** composable is a composable that doesn't hold any state
  - An easy way to achieve stateless is by using state hoisting
- **State hoisting** is a pattern where the state is moved to a higher-level composable, making the composable stateless and more reusable
  - The state owner exposes to consumers the state and events to modify it
- The benefits of state hoisting are:
  - Improves reusability and testability
  - Centralizes state management

https://developer.android.com/develop/ui/compose/state#stateful-vs-stateless
https://developer.android.com/develop/ui/compose/state-hoisting

10. State management - Stateless

- The following example shows the stateless version of the previous counter:

```
@Composable
fun CounterApp() {
    // State is hoisted to the parent composable
    var count by remember { mutableIntStateOf(0) }

    // Stateless Counter composable
    Counter(
        count = count,
        onIncrement = { count++ },
        onDecrement = { count-- },
    )
}
```

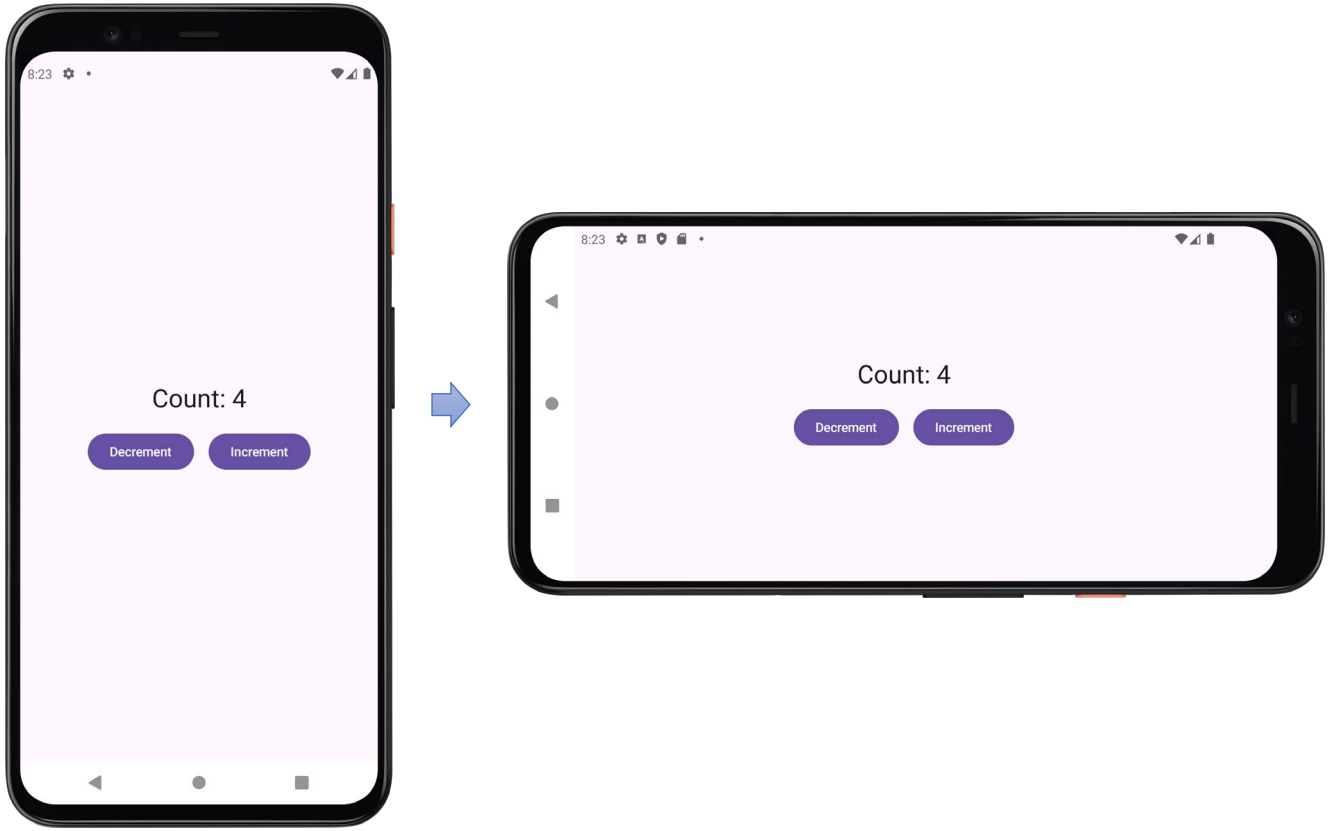
The parent composable exposes the state (count) and events to modify it (onIncrement and onDecrement)

```
@Composable
fun Counter(
    count: Int, // State passed as a parameter
    onIncrement: () -> Unit, // Event callback for increment
    onDecrement: () -> Unit // Event callback for decrement
) {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(
            text = "Count: $count",
            style = MaterialTheme.typography.headlineMedium
        )
        Spacer(modifier = Modifier.height(16.dp))
        Row(
            horizontalArrangement = Arrangement.spacedBy(16.dp)
        ) {
            Button(onClick = onDecrement) {
                Text(stringResource(R.string.decrement))
            }
            Button(onClick = onIncrement) {
                Text(stringResource(R.string.increment))
            }
        }
    }
}
```

10. State management - State restoration

- To restore state across configuration changes (e.g., screen rotation), we use `rememberSaveable` instead of `remember`

```
@Composable
fun CounterApp() {
    // State is hoisted to the parent composable
    var count by rememberSaveable { mutableIntStateOf(0) }

    // Stateless Counter composable
    Counter(
        count = count,
        onIncrement = { count++ },
        onDecrement = { count-- }
    )
}
```



10. State management - View model

- For more complex state management, we use a `ViewModel` to separate UI logic from the composable

A `ViewModel` object holds state (e.g., using *`mutableIntStateOf`* or other) and composables observe this state and update the UI when the state changes

```
@Composable
fun CounterApp(viewModel: CounterViewModel = viewModel()) {
    // Observe the state from the ViewModel
    val count by viewModel.count.asIntState()

    // Stateless Counter composable
    Counter(
        count = count,
        onIncrement = { viewModel.increment() },
        onDecrement = { viewModel.decrement() }
    )
}

// ViewModel to manage the state
class CounterViewModel : ViewModel() {
    private val _count = mutableIntStateOf(0) // Mutable state
    val count: State<Int> get() = _count

    fun increment() {
        _count.intValue++
    }

    fun decrement() {
        _count.intValue--
    }
}
```

https://developer.android.com/topic/libraries/architecture/viewmodel

10. State management - View model

- To use view model, we need the following dependency:

build.gradle.kts

```
dependencies {
    implementation(libs.androidx.lifecycle.viewmodel.compose)
}
```

libs.version.toml

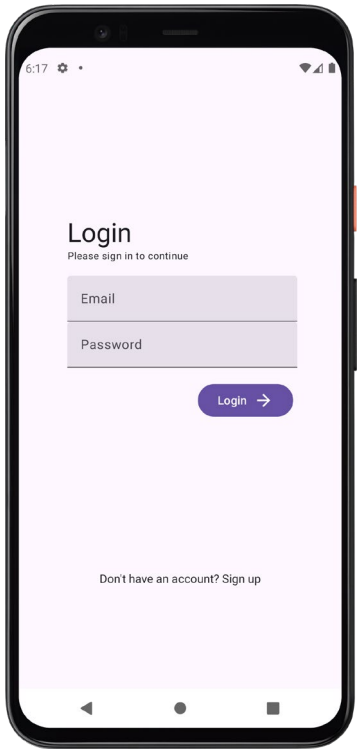
```
[versions]
lifecycleViewmodelComposeVersion = "2.8.7"

[libraries]
androidx-lifecycle-viewmodel-compose = { group = "androidx.lifecycle",
  name = "lifecycle-viewmodel-compose", version.ref = "lifecycleViewmodelComposeVersion" }
```

10. State management - Text fields

- We also use state variables to manipulate text fields:

```
var login by rememberSaveable { mutableStateOf("") }
var password by rememberSaveable { mutableStateOf("") }

TextField(
    value = login,
    onValueChange = { login = it },
    modifier = Modifier
        .constrainAs(loginField) {
            top.linkTo(text2.bottom, margin = 16.dp)
            start.linkTo(startGuideline)
        },
    placeholder = {
        Text(stringResource(R.string.email_edit_text))
    },
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email)
)
TextField(
    value = password,
    onValueChange = { password = it },
    modifier = Modifier
        .constrainAs(passwordField) {
            top.linkTo(loginField.bottom)
            start.linkTo(startGuideline)
        },
    placeholder = {
        Text(stringResource(R.string.password_edit_text))
    },
    visualTransformation = PasswordVisualTransformation(),
    keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password)
)
```

- `value = login`: Binds the text field's value to the login state variable
- `onValueChange = { login = it }`: Updates the login state whenever the user types in the field (**`it`** is the *implicit name* of a single parameter in lambda expressions)
- `placeholder`: Displays a hint when the field is empty
- `keyboardOptions`: Sets the keyboard type to `Email` (for easier email input) and `Password` for secure input
- `visualTransformation`: Masks the input with `PasswordVisualTransformation()` (to show dots instead of text)



https://developer.android.com/develop/ui/compose/text/user-input

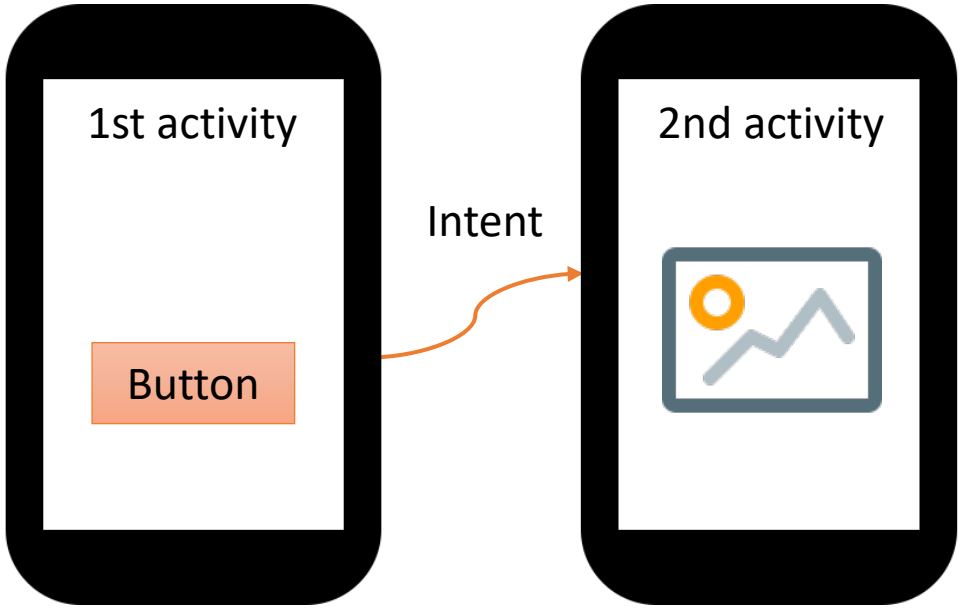
Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
    - Intents
    - Navigation Component
12. List and grids
13. Animations
14. Takeaways

11. Navigation

- Navigation refers to the interactions that let users navigate across, into, and back out from the different pieces of the UI
- The traditional Android View System (XML-based UI) was based on navigation between different activities using **intents**
  - An Intent is a messaging object you can use to request an action from another app component (e.g., from an activity to other)
- Jetpack Compose uses the **Navigation Component**, which is part of Android Jetpack
  - The Navigation Component is a library designed to work seamlessly with Compose's declarative UI model
  - With this component we implement single-activity apps with different screens implemented as composable functions

11. Navigation - Intents



Let's see an example of an app composed by two activities which uses an (explicit) intent to start the second activity

AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:theme="@style/Theme.MyApp">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".SecondActivity"
            android:exported="false" />
    </application>

</manifest>
```

https://developer.android.com/reference/android/content/Intent

11. Navigation - Intents

MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyAppTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    MyLayout(modifier = Modifier.padding(innerPadding))
                }
            }
        }
    }
}

@Composable
fun MyLayout(modifier: Modifier = Modifier) {
    var text by rememberSaveable { mutableStateOf("") }
    val context = LocalContext.current

    Column(modifier = modifier) {
        Text(text = stringResource(R.string.text_msg))
        TextField(
            value = text,
            onValueChange = { text = it },
            modifier = Modifier.fillMaxWidth()
        )
        Button(
            onClick = {
                val intent = Intent(context, SecondActivity::class.java).apply {
                    putExtra("name", text)
                }
                context.startActivity(intent)
            },
        ) {
            Text(stringResource(R.string.button_msg))
        }
    }
}
```

11. Navigation - Intents

SecondActivity.kt

```
class SecondActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyAppTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    val name = intent.getStringExtra("name")
                    val hello = String.format(stringResource(R.string.hello_msg), name)
                    MyLayout(modifier = Modifier.padding(innerPadding), hello)
                }
            }
        }
    }

}

@Composable
fun MyLayout(modifier: Modifier = Modifier, text: String) {
    Text(text = text, modifier = modifier)
}
```

Although this type of navigation is possible, in Jetpack Compose it is preferred single-activities apps using the Navigation library

11. Navigation - Navigation Component

- The Navigation component is a library that enables declarative navigation between screens in a Jetpack Compose apps
- The key concepts of the Navigation Component are the following:
  - `NavController`: Object that manages navigation between composables. It keeps track of the back stack and the current destination
  - `NavHost`: Composable that acts as container to hosts the navigation graph and displays the current destination
  - Navigation graph: A collection of *`composable`* destinations that maps out all the screens in your app and the paths (*routes*) that users can take to navigate between them

https://developer.android.com/develop/ui/compose/navigation

11. Navigation - Navigation Component

- To use Navigation Component, first we need to setup the following dependency in our Android project:

build.gradle.kts

```
dependencies {
    implementation(libs.androidx.navigation.compose)
}
```

libs.version.toml

```
[versions]
navigationCompose = "2.8.7"

[libraries]
androidx-navigation-compose = { module = "androidx.navigation:navigation-compose", version.ref = "navigationCompose" }
```

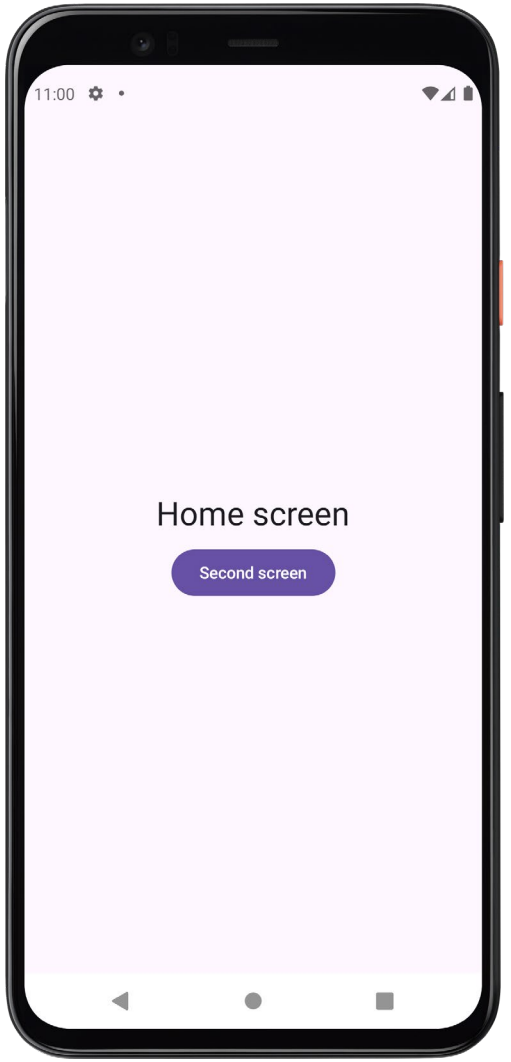
11. Navigation - Navigation Component


```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyAppTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    val navController = rememberNavController()
                    NavHost(
                        navController = navController,
                        startDestination = "home"
                    ) {
                        composable("home") {
                            HomeScreen(navController)
                        }
                        composable("second") {
                            SecondScreen(navController)
                        }
                    }
                }
            }
        }
    }
}
```

- `rememberNavController()` creates a `NavController` object to manage navigation between composables
- `NavHost` is a container for the navigation graph. It hosts the composable destinations (`HomeScreen` and `SecondScreen` in this example)

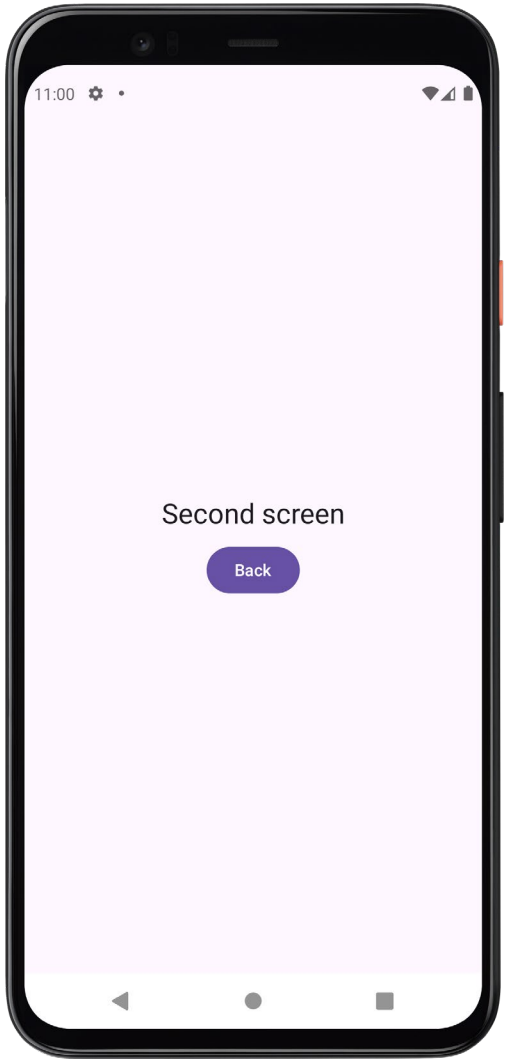
11. Navigation - Navigation Component


```
@Composable
fun HomeScreen(navController: NavController) {
    Column(
        Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = stringResource(R.string.home_msg),
            style = MaterialTheme.typography.headlineMedium
        )
        Spacer(Modifier.height(8.dp))
        Button(
            onClick = { navController.navigate("second") },
        ) {
            Text(stringResource(R.string.button_msg))
        }
    }
}
```



11. Navigation - Navigation Component


```
@Composable
fun SecondScreen(navController: NavController) {
    Column(
        Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = stringResource(R.string.second_msg),
            style = MaterialTheme.typography.headlineSmall
        )
        Spacer(Modifier.height(8.dp))
        Button(
            onClick = { navController.popBackStack() },
        ) {
            Text(stringResource(R.string.back_msg))
        }
    }
}
```



11. Navigation - Navigation Component

- We can pass arguments between screens using the navigation library. For example:

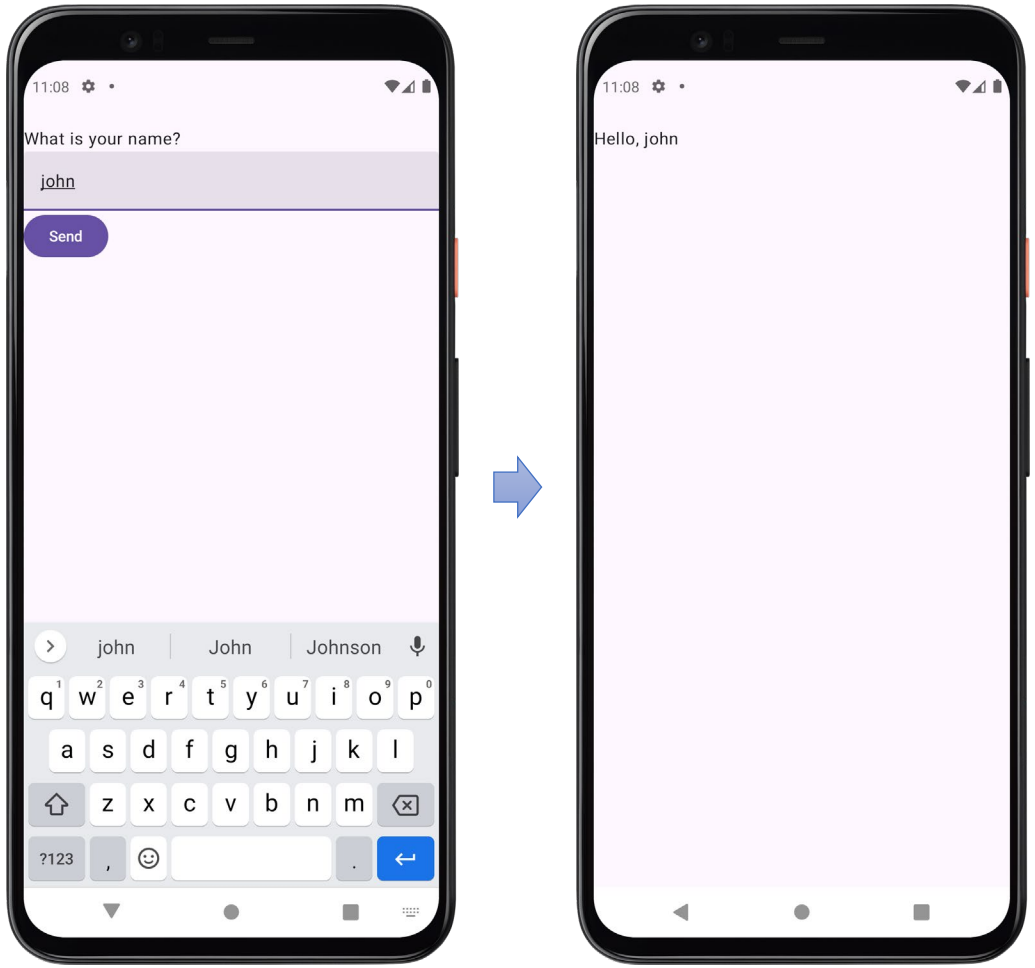
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyAppTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    val navController = rememberNavController()
                    val modifier = Modifier.padding(innerPadding)
                    NavHost(
                        navController = navController,
                        startDestination = "first"
                    ) {
                        composable("first") {
                            FirstScreen(navController, modifier)
                        }
                        composable(
                            "second/{name}",
                            arguments = listOf(navArgument("name") { type = NavType.StringType })
                        ) { backStackEntry ->
                            val name = backStackEntry.arguments?.getString("name")
                            name?.let { SecondScreen(modifier, it) }
                        }
                    }
                }
            }
        }
    }
}
```

- The second composable defines a route with a dynamic argument (`"second/{name}"`)
- The arguments list ensures that `name` is treated as a String type (`NavType.StringType`)
- `backStackEntry` (object that gives access to the navigation arguments) is used to retrieve the name argument from the navigation back stack. The operator ? is used to safely access `arguments` (if `arguments` is null, name will be also null)
- If name is not null, it is passes to `SecondScreen`

11. Navigation - Navigation Component

- We can pass arguments between screens using the navigation library. For example:

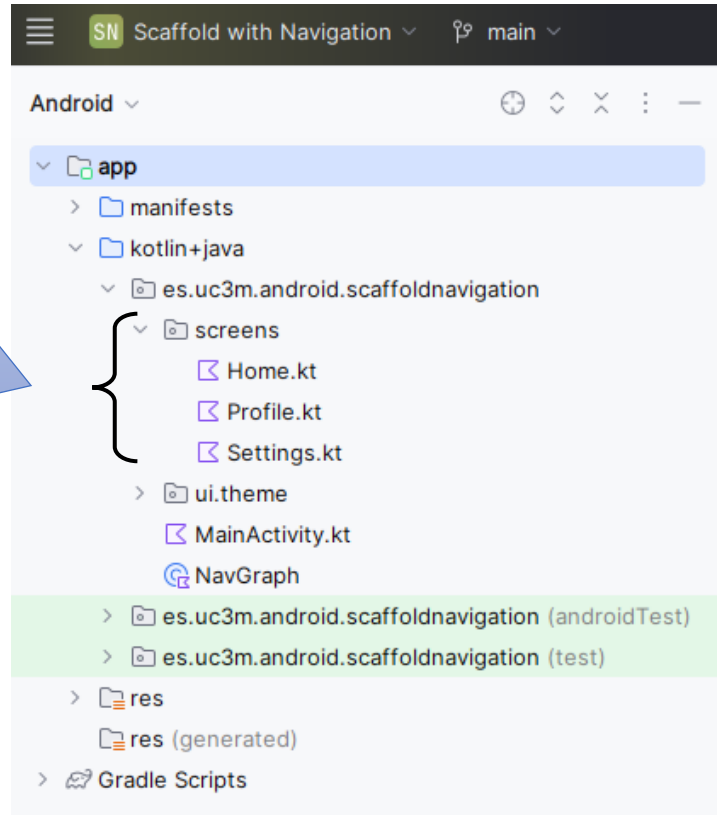
```
@Composable
fun FirstScreen(navController: NavController, modifier: Modifier = Modifier) {
    var text by rememberSaveable { mutableStateOf("") }

    Column(modifier = modifier) {
        Text(text = stringResource(R.string.text_msg))
        TextField(
            value = text,
            onValueChange = { text = it },
            modifier = Modifier.fillMaxWidth()
        )
        Button(
            onClick = { navController.navigate("second/$text") },
        ) {
            Text(stringResource(R.string.button_msg))
        }
    }
}

@Composable
fun SecondScreen(modifier: Modifier = Modifier, name: String = "") {
    val hello = String.format(stringResource(R.string.hello_msg), name)
    Text(text = hello, modifier = modifier)
}
```



11. Navigation - Navigation Component

- The following example combines an Scaffold structure with the use of the navigation component:

This folder contains the composables for the different screens



The NavGraph class defines the navigation graph using a *sealed* class (i.e., special type of class in Kotlin that restricts inheritance, used for representing a closed set of types, such as navigation routes in this case). This mechanism is a good practice to avoid hardcoding routes (string values)

```
const val HOME_ROUTE = "home"
const val PROFILE_ROUTE = "profile"
const val SETTING_ROUTE = "settings"

sealed class NavGraph(val route: String) {
    data object Home : NavGraph(HOME_ROUTE)
    data object Profile : NavGraph(PROFILE_ROUTE)
    data object Settings : NavGraph("$SETTING_ROUTE/{source}") {
        // Helper function to create the route with arguments
        fun createRoute(source: String) = "$SETTING_ROUTE/$source"
    }
}
```

11. Navigation - Navigation Component

The content composable defines the NavHost. The navigation controllers is used in all parts that requires to change navigation (e.g., navigation drawer, bottom bar, screen composables)

```
@Composable
fun MyContent(modifier: Modifier = Modifier, navController: NavHostController) {
    NavHost(
        modifier = modifier,
        navController = navController,
        startDestination = NavGraph.Home.route,
    ) {
        composable(NavGraph.Home.route) {
            HomeScreen(navController = navController)
        }
        composable(NavGraph.Profile.route) {
            ProfileScreen(navController = navController)
        }
        composable(
            NavGraph.Settings.route,
            arguments = listOf(navArgument("source") { type = NavType.StringType })
        ) { backStackEntry ->
            val source = backStackEntry.arguments?.getString("source")
            SettingsScreen(navController = navController, source)
        }
    }
}
```

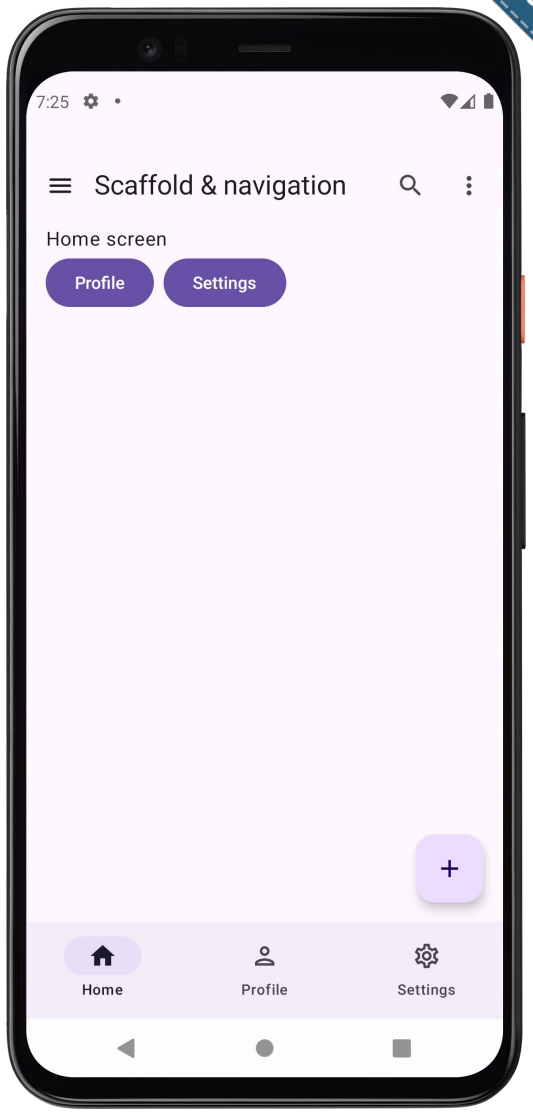


Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
    - Lists
    - Grids
13. Animations
14. Takeaways

12. Lists and grids

- In Jetpack Compose, **lists** and **grids** are essential components for displaying collections of items in a structured way
- Jetpack Composes provides different composables handle collections efficiently:
  - `LazyColumn`: For vertical lists
  - `LazyRow`: For horizontal lists
  - `LazyVerticalGrid`: For vertically grids
  - `LazyHorizontalGrid`: For horizontally grids

https://developer.android.com/develop/ui/compose/lists

12. Lists and grids - Lists

- **Lists** are used to display a vertically or horizontally scrollable collection of items
- Jetpack Compose provides two main composables for creating scrollable lists:
  - `LazyColumn`: For vertical lists
  - `LazyRow`: For horizontal lists
- These composables are **lazy**, meaning they only compose and render the items that are currently visible on the screen (*viewport*)
  - This makes them highly efficient for large datasets

https://developer.android.com/develop/ui/compose/lists

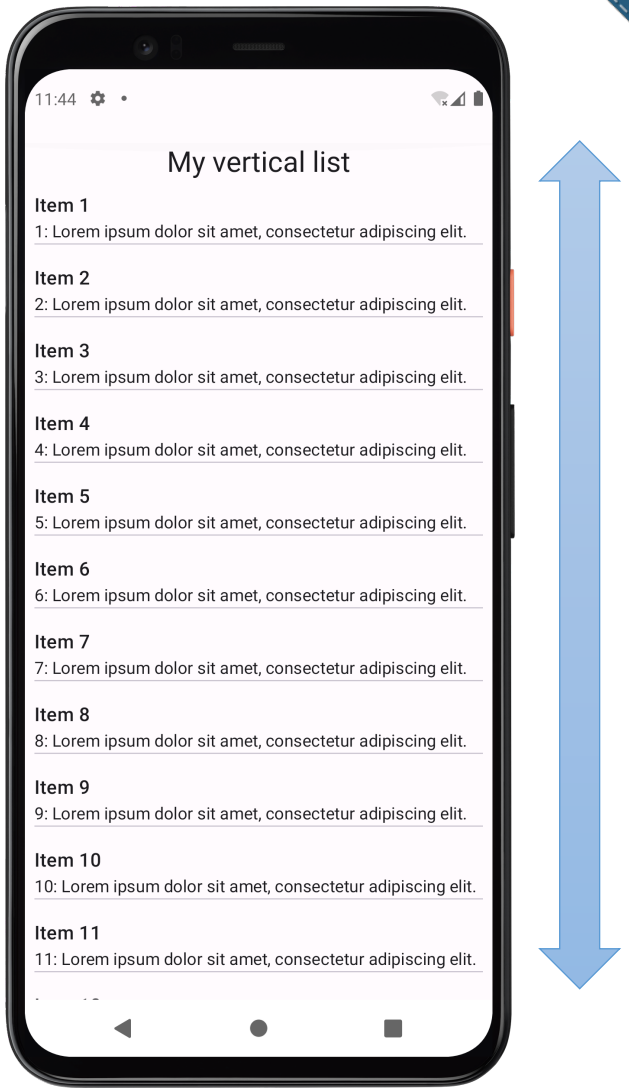
12. Lists and grids - Lists

This example defines a composable function that displays a scrollable vertical list of items

The `rememberLazyListState()` function ensures that the scroll position is not reset when the UI is recomposed

This part adds the `content` of `LazyColumn`, composed by a single item (`Text`) plus a list of items (iterating to create `myList` a `Column` composable for each item)

```
data class Item(val title: String, val description: String)

@Composable
fun MyLazyColumn(modifier: Modifier = Modifier) {
    val title = stringResource(R.string.item_title)
    val description = stringResource(R.string.item_description)
    val myList = (0..20).map {
        Item(
            title = String.format(title, it + 1),
            description = String.format(description, it + 1),
        )
    }

    LazyColumn(
        state = rememberLazyListState(),
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = modifier,
        content = {
            item {
                Text(
                    text = stringResource(R.string.my_list),
                    style = MaterialTheme.typography.headlineSmall
                )
            }
            items(myList) {
                Column(
                    modifier = Modifier.padding(8.dp)
                ) {
                    Text(
                        text = it.title,
                        style = MaterialTheme.typography.titleMedium
                    )
                    Text(
                        text = it.description,
                        style = MaterialTheme.typography.bodyMedium
                    )
                    HorizontalDivider()
                }
            }
        }
    )
}
```



12. Lists and grids - Lists

This example defines a scrollable horizontal list composed by 10 Text items

```
@Composable
fun MyLazyRow(modifier: Modifier = Modifier) {
    Column(modifier = modifier) {
        Text(
            modifier = Modifier.padding(16.dp),
            text = stringResource(R.string.my_list),
            style = MaterialTheme.typography.headlineSmall,
            textAlign = TextAlign.Start
        )
        LazyRow(
            state = rememberLazyListState(),
            contentPadding = PaddingValues(8.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            items(10) { index ->
                Text(
                    text = stringResource(R.string.item_text, index),
                    modifier = Modifier.padding(8.dp),
                    style = MaterialTheme.typography.bodyLarge
                )
            }
        }
    }
}
```

12. Lists and grids - Grids

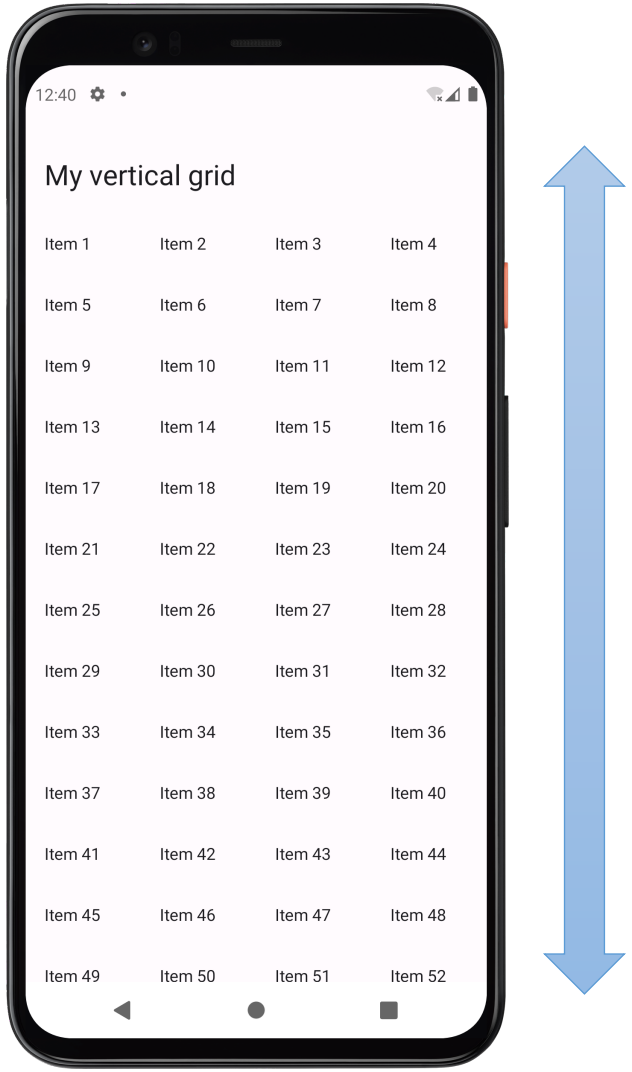
- **Grids** are used to display items in a grid layout (i.e., to organize content into rows and columns)
- Jetpack Compose provides two main composables for creating scrollable grids:
  - `LazyVerticalGrid`: For vertically grids
  - `LazyHorizontalGrid`: For horizontally grids
- Like `LazyColumn` and `LazyRow`, these composables are **lazy** and only render visible items

https://developer.android.com/develop/ui/compose/lists

12. Lists and grids - Grids

This example defines a composable function that displays a scrollable vertical grids of 100 items displayed in a grid of 4 columns

```
@Composable
fun MyVerticalGrid(modifier: Modifier = Modifier) {
    Column(modifier = modifier) {
        Text(
            modifier = Modifier.padding(16.dp),
            text = stringResource(R.string.my_grid),
            style = MaterialTheme.typography.headlineSmall,
            textAlign = TextAlign.Start
        )
        LazyVerticalGrid(
            columns = GridCells.Fixed(4), // 4 columns
            state = rememberLazyGridState()
        ) {
            items(100) { index ->
                Text(
                    text = stringResource(R.string.item_text, index + 1),
                    modifier = Modifier.padding(16.dp),
                    style = MaterialTheme.typography.bodyMedium
                )
            }
        }
    }
}
```



12. Lists and grids - Grids

This example defines a composable function that displays a scrollable horizontal grids of 42 items displayed in a grid of 6 rows

```
@Composable
fun MyHorizontalGrid(modifier: Modifier = Modifier) {
    Column(modifier = modifier) {
        Text(
            modifier = Modifier.padding(16.dp),
            text = stringResource(R.string.my_grid),
            style = MaterialTheme.typography.headlineSmall,
            textAlign = TextAlign.Start
        )
        LazyHorizontalGrid(
            rows = GridCells.Fixed(6), // 6 rows
            state = rememberLazyGridState()
        ) {
            items(42) { index ->
                Text(
                    text = stringResource(R.string.item_text, index + 1),
                    modifier = Modifier.padding(16.dp),
                    style = MaterialTheme.typography.bodyMedium
                )
            }
        }
    }
}
```

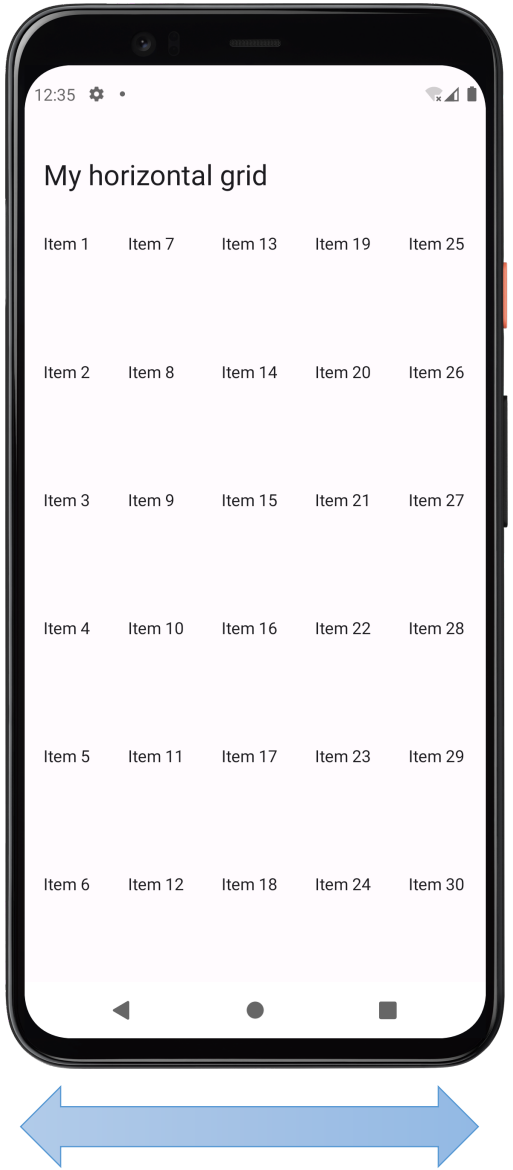


Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

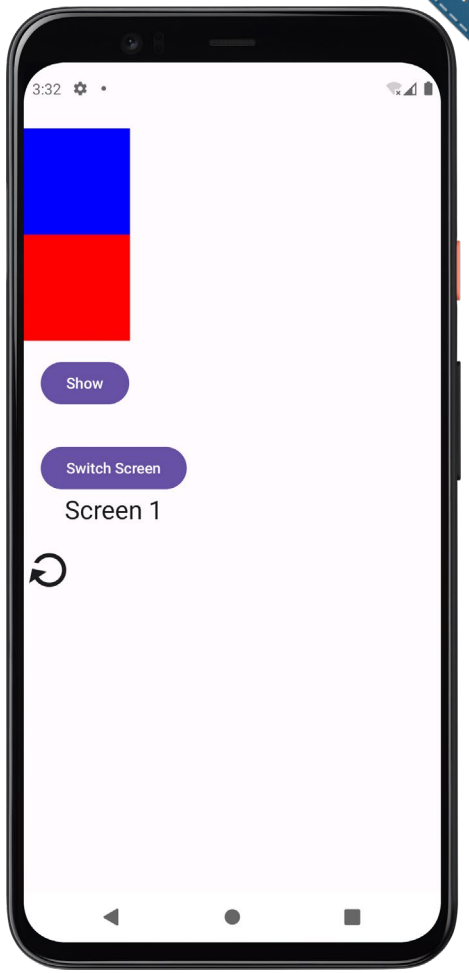
13. Animations

- Jetpack Compose provides built-in support for common animations, such as:
  - Visibility: appearance and disappearance
  - Content size changes
  - Transition between composables
- Some of these functions are:
  - `AnimatedVisibility`: to hide or show content
  - `AnimatedContent`: to animate between different contents
  - `animate*()`: to animate an individual property
  - `animate*AsState()`: to carry out state-drive animations
  - `Transition`: to animate multiple values at once
  - `InfiniteTransition`: to animate properties continuously

https://developer.android.com/develop/ui/compose/animation/introduction

13. Animations


```
@Composable
fun ChangeSize() {
    var expanded by remember { mutableStateOf(false) }
    val size by animateDpAsState(targetValue = if (expanded) 200.dp else 100.dp)

    Box(
        modifier = Modifier
            .size(size)
            .background(Color.Blue)
            .clickable { expanded = !expanded }
    )
}
```




```
@Composable
fun ChangeSizeAndColor() {
    var toggled by remember { mutableStateOf(false) }
    val transition = updateTransition(targetState = toggled)

    val color by transition.animateColor(label = "color") { state ->
        if (state) Color.Green else Color.Red
    }
    val size by transition.animateDp(label = "size") { state ->
        if (state) 150.dp else 100.dp
    }

    Box(
        modifier = Modifier
            .size(size)
            .background(color)
            .clickable { toggled = !toggled }
    )
}
```

```
@Composable
fun ToggleVisibility() {
    var visible by remember { mutableStateOf(false) }

    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier.padding(16.dp)
    ) {
        Button(onClick = { visible = !visible }) {
            Text(if (visible) "Hide" else "Show")
        }

        AnimatedVisibility(visible) {
            Box(
                modifier = Modifier
                    .size(100.dp)
                    .background(Color.Blue)
            )
        }
    }
}
```

Table of contents

1. Introduction
2. Activities
3. Jetpack Compose
4. Compose functions
5. Basic components
6. Resources
7. Layouts
8. Theme
9. Material components
10. State management
11. Navigation
12. List and grids
13. Animations
14. Takeaways

14. Takeaways

- Jetpack Compose is a modern UI toolkit for building native Android apps using Kotlin
  - It simplifies UI development by using a declarative approach, allowing us to define an app's UI as composable functions
  - This eliminates the need for XML layouts and reduces boilerplate code, making UI development faster and more intuitive
- Composables are the building block of Jetpack Compose
  - Composables are annotated with `@Composable` and describes how a portion of the UI should look based on the current state
  - Jetpack Compose offers an implementation of the Material components a collection of composable functions