
UNIT 7

ANDROID STUDIO

PMDM - 2DAM

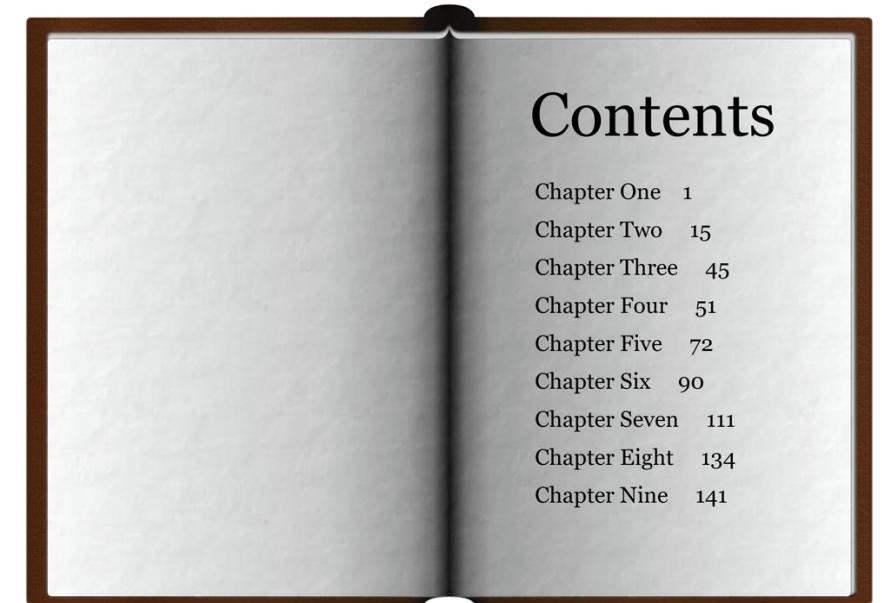
Àngel Olmos (a.olmosginer@edu.gva.es)

Jose Pascual Rocher (jp.rochercamps@edu.gva.es)



CONTENT

1. INTRODUCTION
2. ANDROID DESIGN WITH VIEWS
3. JETPACK COMPOSE INTRO
4. COMPONENTS



INTRODUCTION

Installation

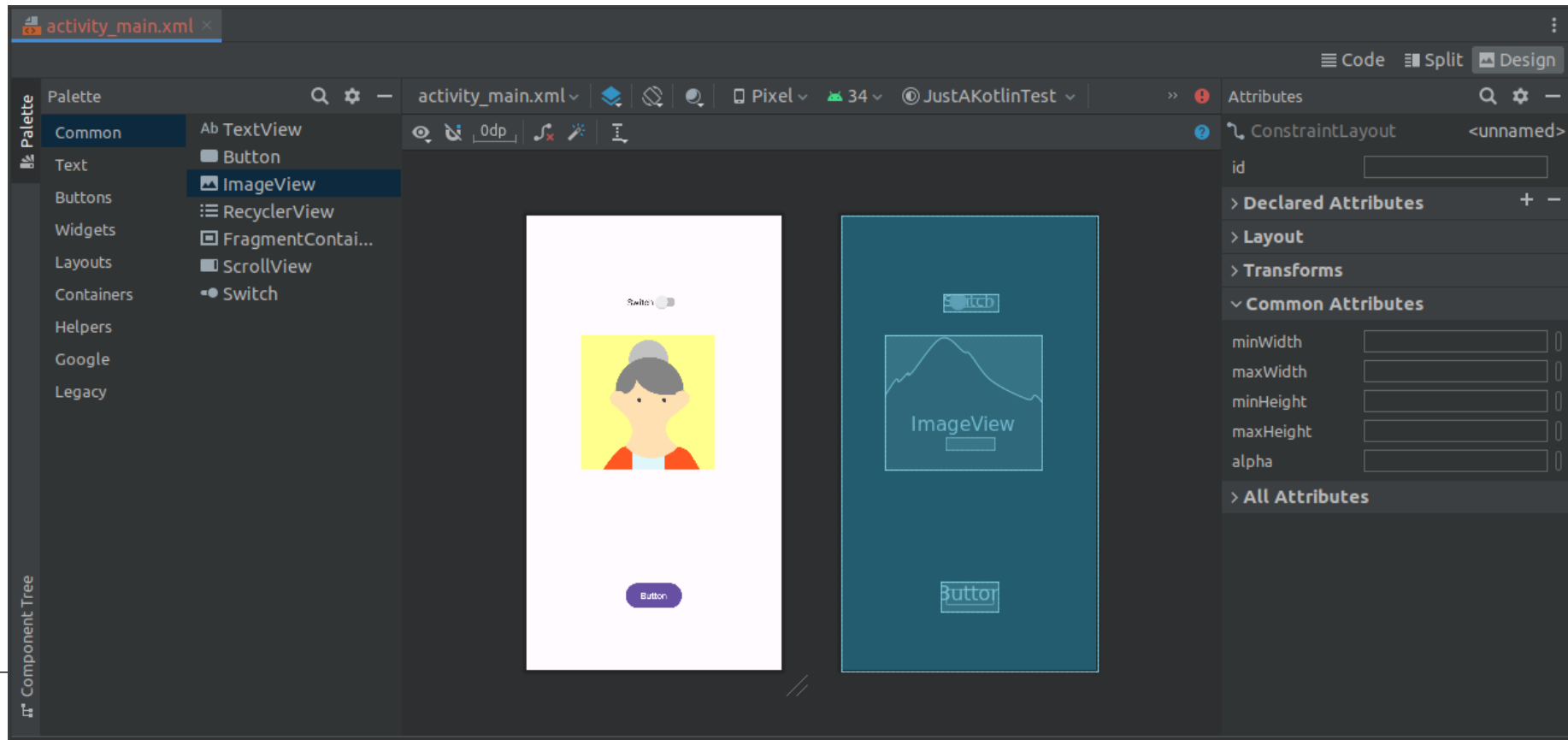
- Follow steps described in the AULES document
- Check / Install Android SDK
- Configure VM acceleration (Linux)
- Run apps on a hardware device (recommended)
- Create an Android Virtual Device (not recommended)



INTRODUCTION

APPs design (*Views*)

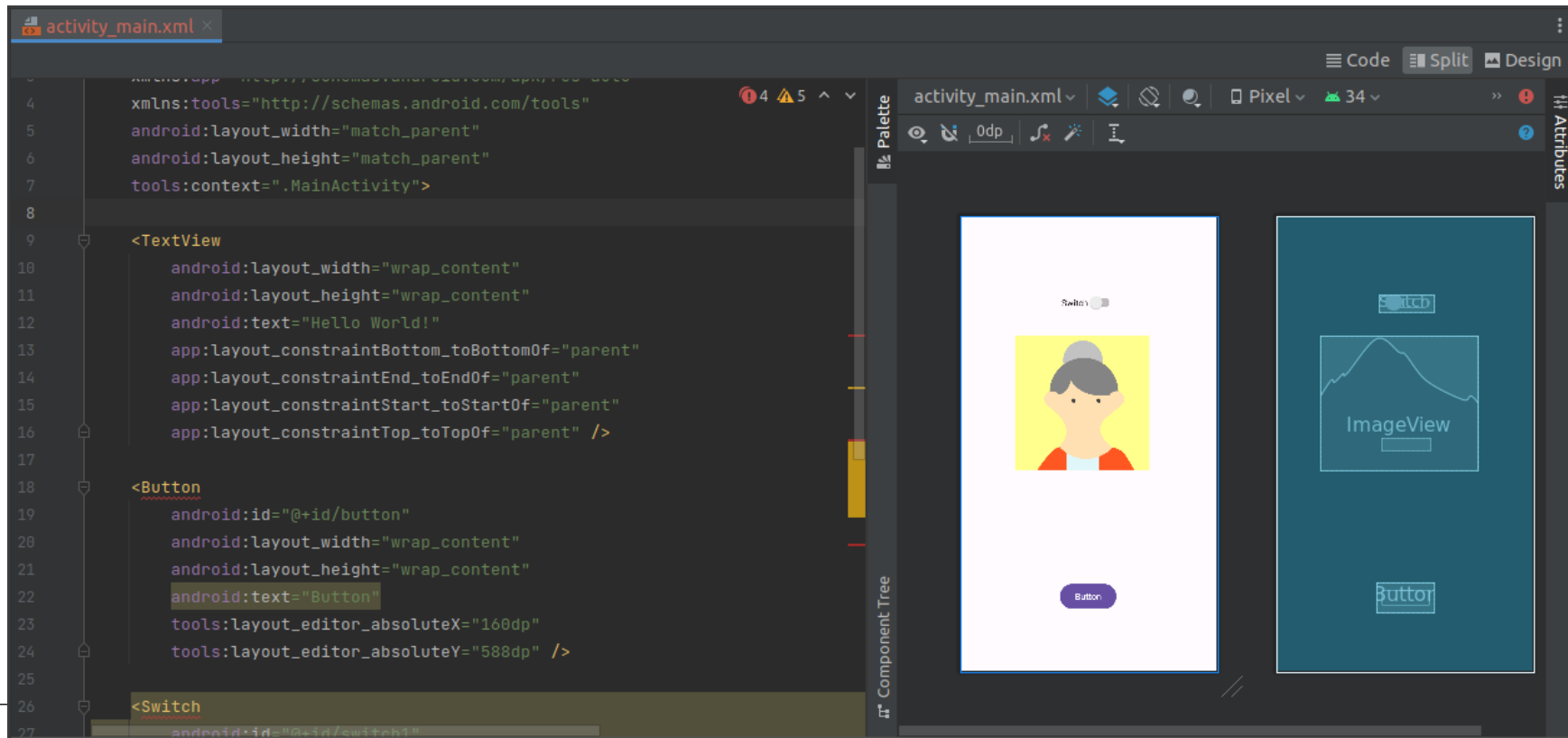
- Android Studio provides a visual layout editor for designing your APP's UI
- You can drag and drop UI components from the Palette to the layout editor



INTRODUCTION

APPs design (*Views*)

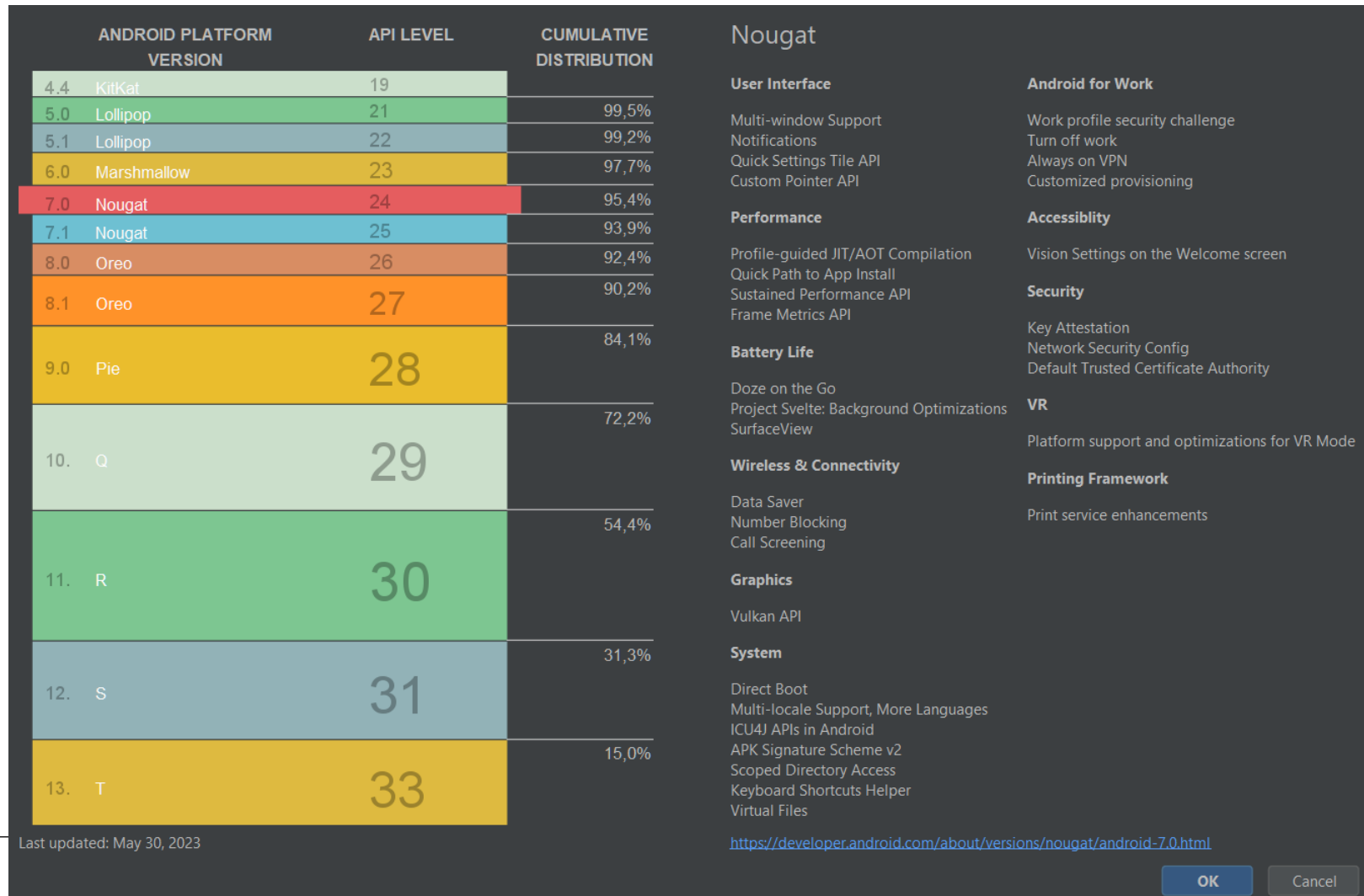
- You can also edit the XML layout file directly (app/src/main/res/layout/activity_main.xml)



INTRODUCTION

Which SDK to choose?

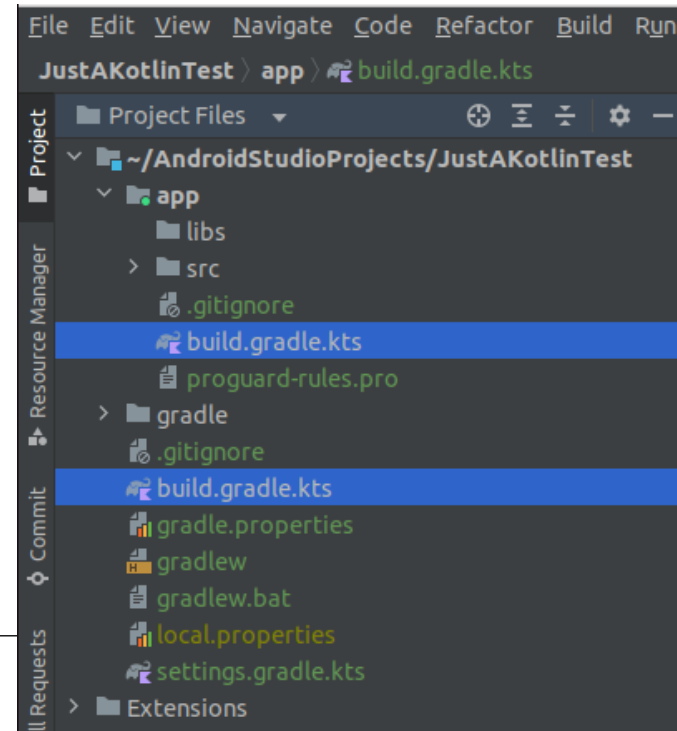
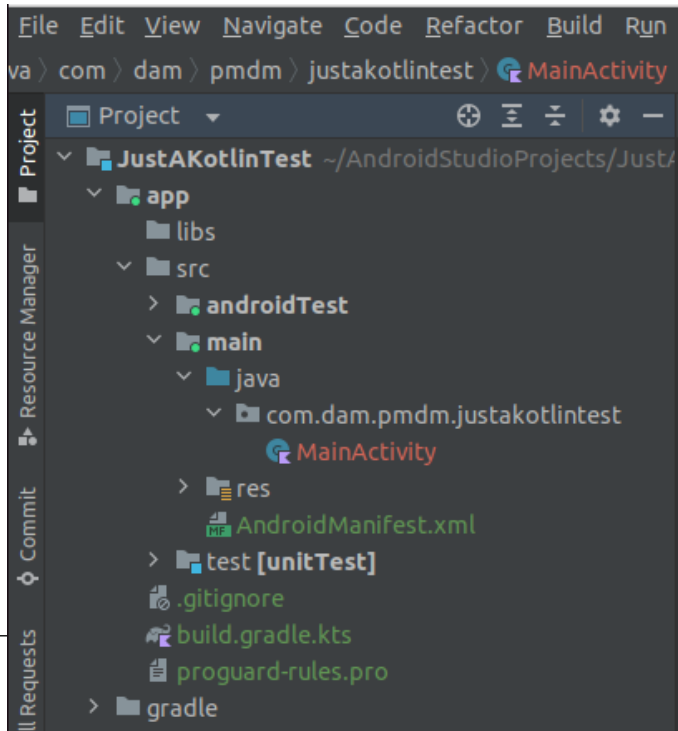
- Select the minimum SDK depending on the features needed for your APP
- The lower the SDK, the higher number of compatible devices



INTRODUCTION

Projects Structure

- **app/src/main/java** folder structure determined by the packages and the source files
- **Gradle files:** "build.gradle.kts" file in the project root folder and another one (same name) in the "app" folder



INTRODUCTION

Projects Structure

- The first Gradle file refers to configuration options common to all sub-projects/modules
- The second specifies the configuration of the APP (SDK version, JVM version, dependencies ...)

```
build.gradle.kts (JustAKotlinTest) × build.gradle.kts (:app) ×
1 // Top-level build file where you can add configuration options common to all sub-projects/modules
2 plugins { this: PluginDependenciesSpecScope
3     id("com.android.application") version "8.1.0" apply false
4     id("org.jetbrains.kotlin.android") version "1.8.0" apply false
5 }
```

```
build.gradle.kts (JustAKotlinTest) × build.gradle.kts (:app) ×
6 android { this: BaseAppModuleExtension
7     namespace = "com.dam.pmdm.justakotlintest"
8     compileSdk = 33
9
10    defaultConfig { this: ApplicationDefaultConfig
11        applicationId = "com.dam.pmdm.justakotlintest"
12        minSdk = 24
13        targetSdk = 33
14        versionCode = 1
15        versionName = "1.0"
16
17        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
18    }
19 }
```

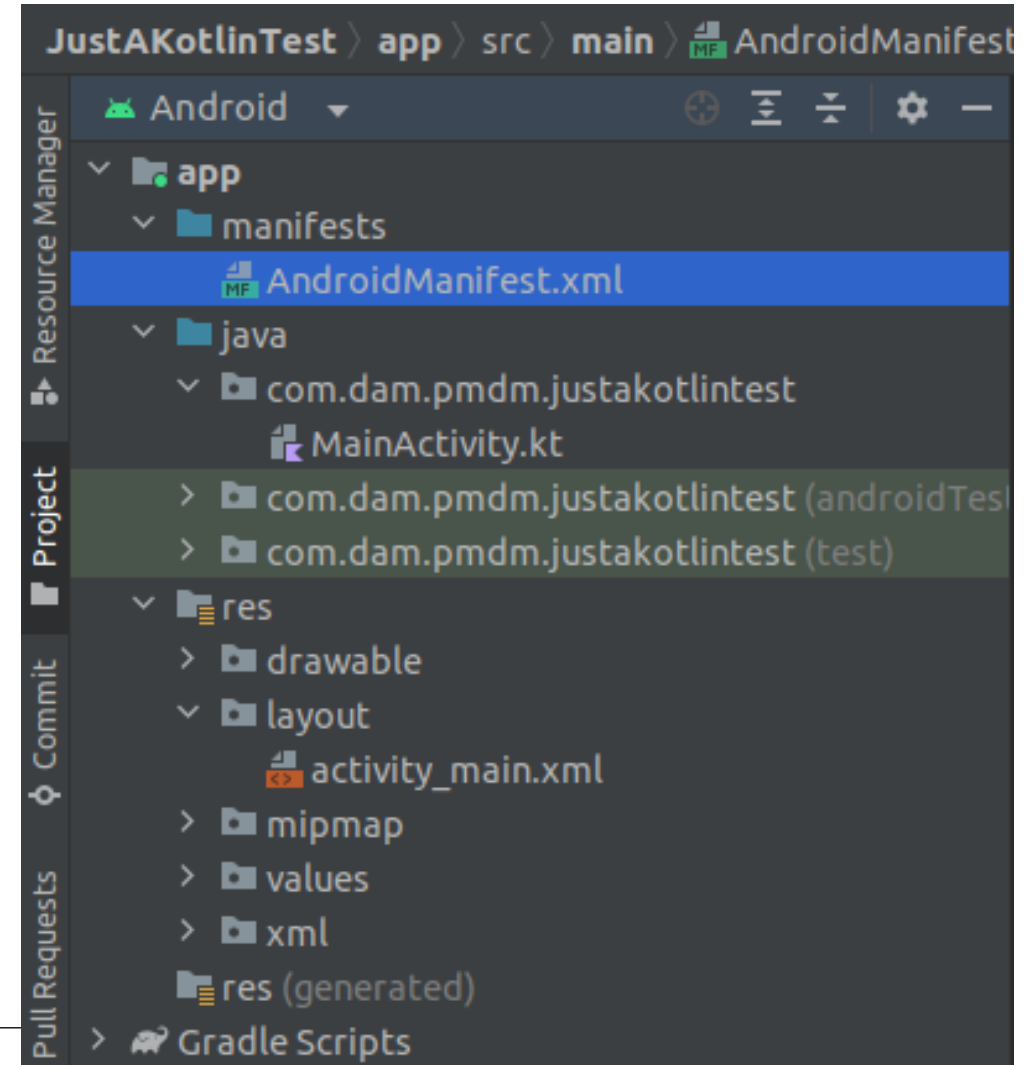

INTRODUCTION

Projects Structure

AndroidManifest.xml: important information to the Android build tools, the OS and Google Play

- Components of the application, launcher icon
- The permissions that the APP needs or that other applications must have to access to it
- The hardware and software features that the APP requires
- ...

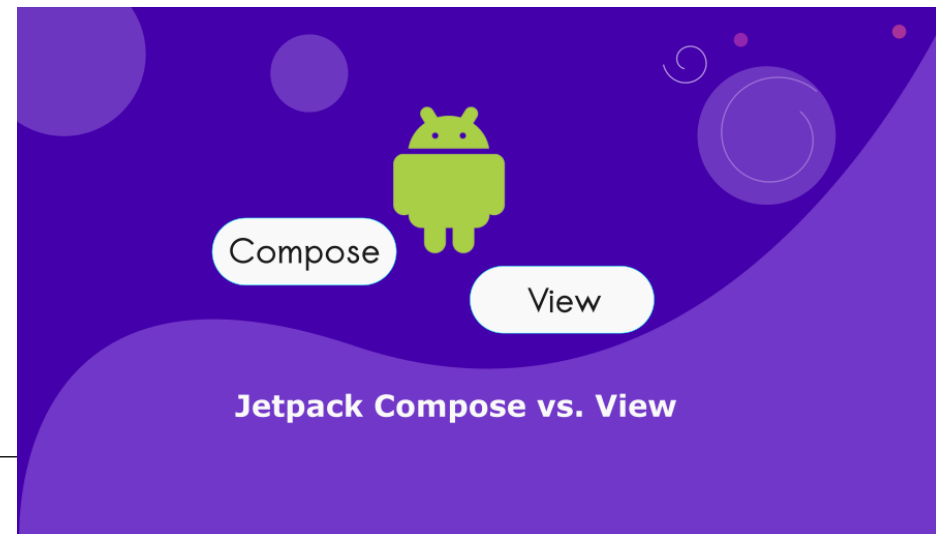
Declared **permissions** must be accepted by the **user** during installation



INTRODUCTION

Views Vs. Compose

- **Android Views**: long-standing method for creating Android UI
- Relies on XML-based layout files and *imperative coding* to define and update UI elements
- **Jetpack Compose**: modern evolution in Android UI development
- *Declarative approach* that simplifies UI creation through Kotlin code
- Encourages the creation of **reusable components**, offers **automatic UI updates** and enhances animation capabilities



INTRODUCTION

Views Vs. Compose



JETPACK COMPOSE

1. Less code
2. Android's modern toolkit for building native UI
3. It is Powerful
4. Uses declarative API (means you need to describe your UI)
5. Gives flexibility to implement the design you want.
6. Accelerate development with interoperability



XML

1. More code
2. It stands for Extensible Mark-up Language
3. It is a lightweight markup Language
4. Non-declarative API
5. Less flexible comparatively
6. Takes more time in development

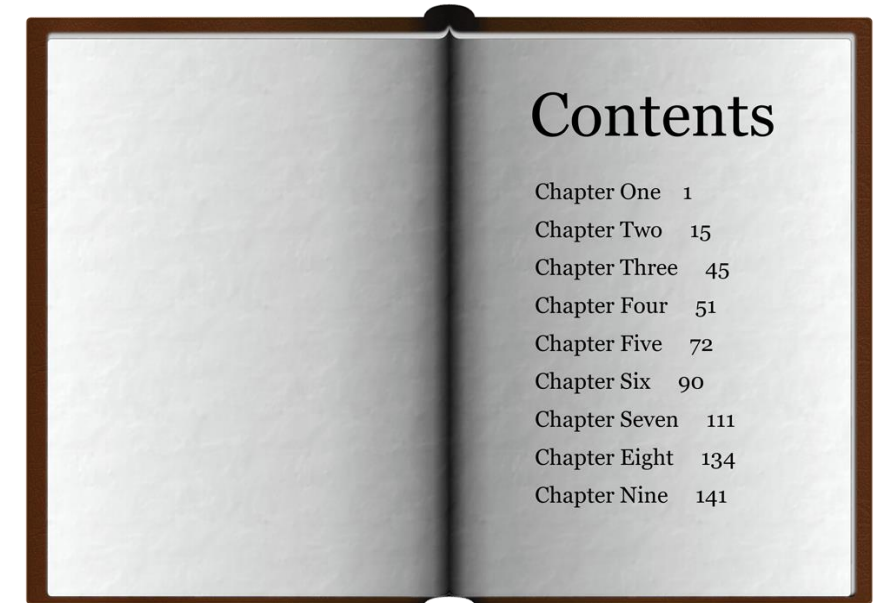
CONTENT

1. INTRODUCTION

2. ANDROID DESIGN WITH VIEWS

3. JETPACK COMPOSE INTRO

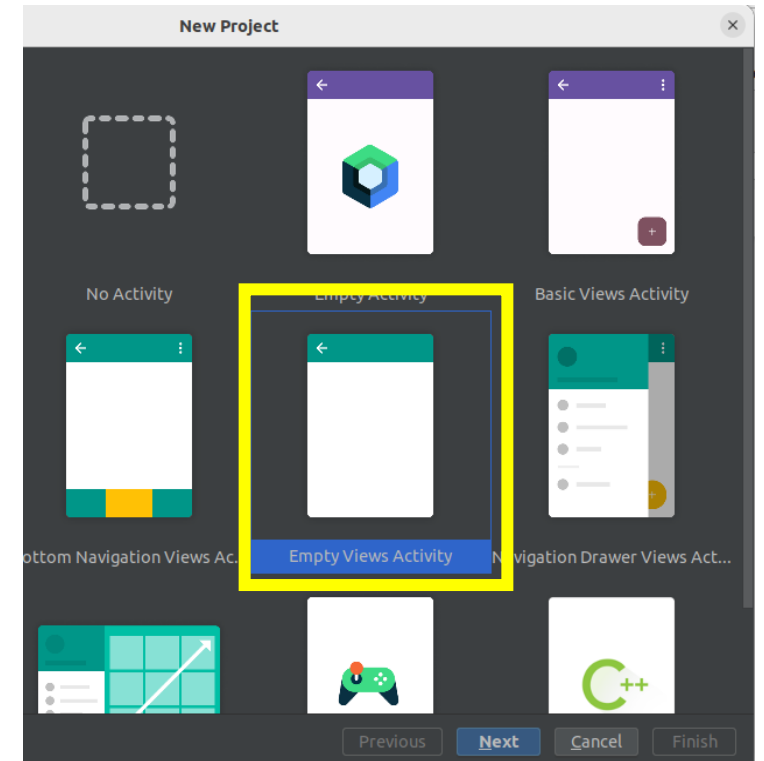
4. COMPONENTS



ANDROID DESIGN WITH *VIEWS*

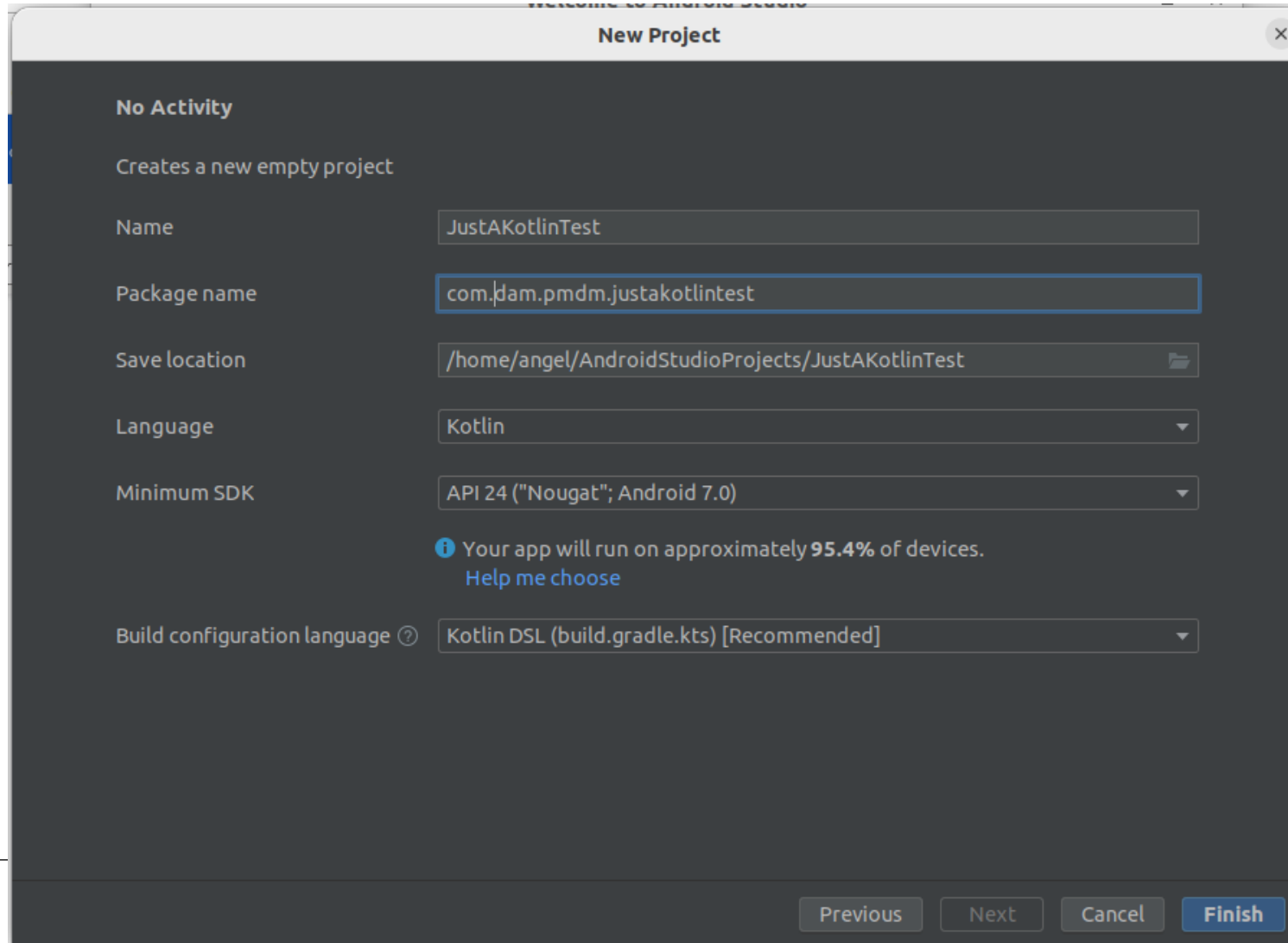
Create a new project

- Empty View Activity (do not select default "Empty Activity")
- Configure the project name, package name, and project location
- Select the language as "Kotlin"
- Choose the minimum API level (usually a lower API level ensures more devices compatibility)
- Select the *Gradle* automation system support (default)



ANDROID DESIGN WITH *VIEWS*

Create a new project



The screenshot shows the 'New Project' dialog box in Android Studio. The dialog has a title bar with 'New Project' and a close button. The main content area is dark gray and contains the following fields and options:

- No Activity**: A section header.
- Creates a new empty project**: A descriptive text.
- Name**: A text field containing 'JustAKotlinTest'.
- Package name**: A text field containing 'com.dam.pmdm.justakointest'.
- Save location**: A text field containing '/home/angel/AndroidStudioProjects/JustAKotlinTest'.
- Language**: A dropdown menu with 'Kotlin' selected.
- Minimum SDK**: A dropdown menu with 'API 24 ("Nougat"; Android 7.0)' selected.
- Information**: A blue information icon followed by the text 'Your app will run on approximately 95.4% of devices.' and a link 'Help me choose'.
- Build configuration language**: A dropdown menu with 'Kotlin DSL (build.gradle.kts) [Recommended]' selected.

At the bottom of the dialog, there are four buttons: 'Previous', 'Next', 'Cancel', and 'Finish'.

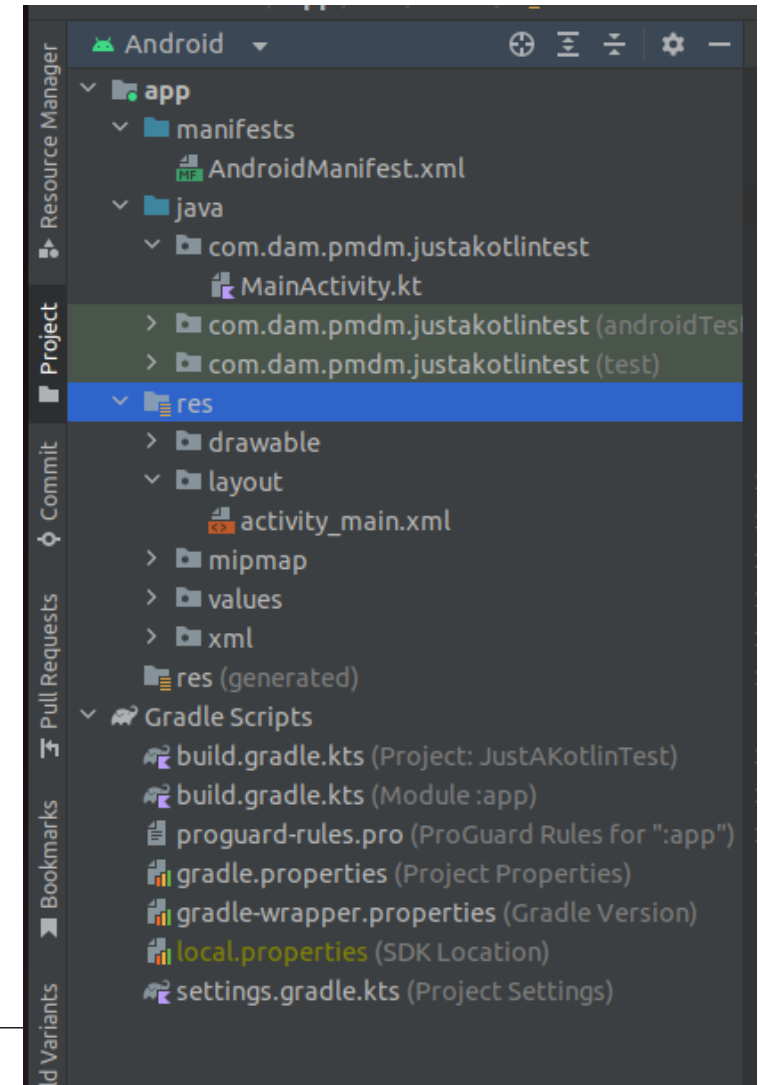
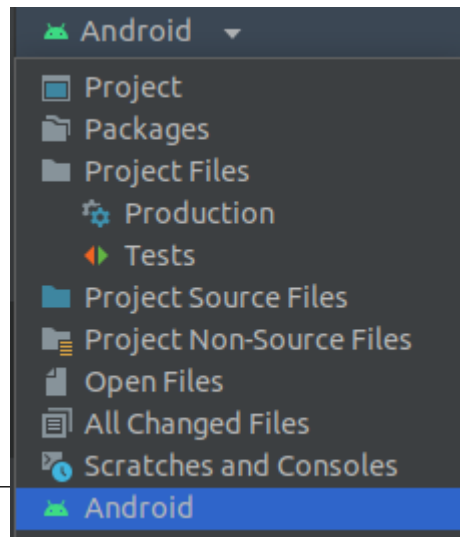
ANDROID DESIGN WITH *VIEWS*

Create a new project

Localize the explained configuration and design files:

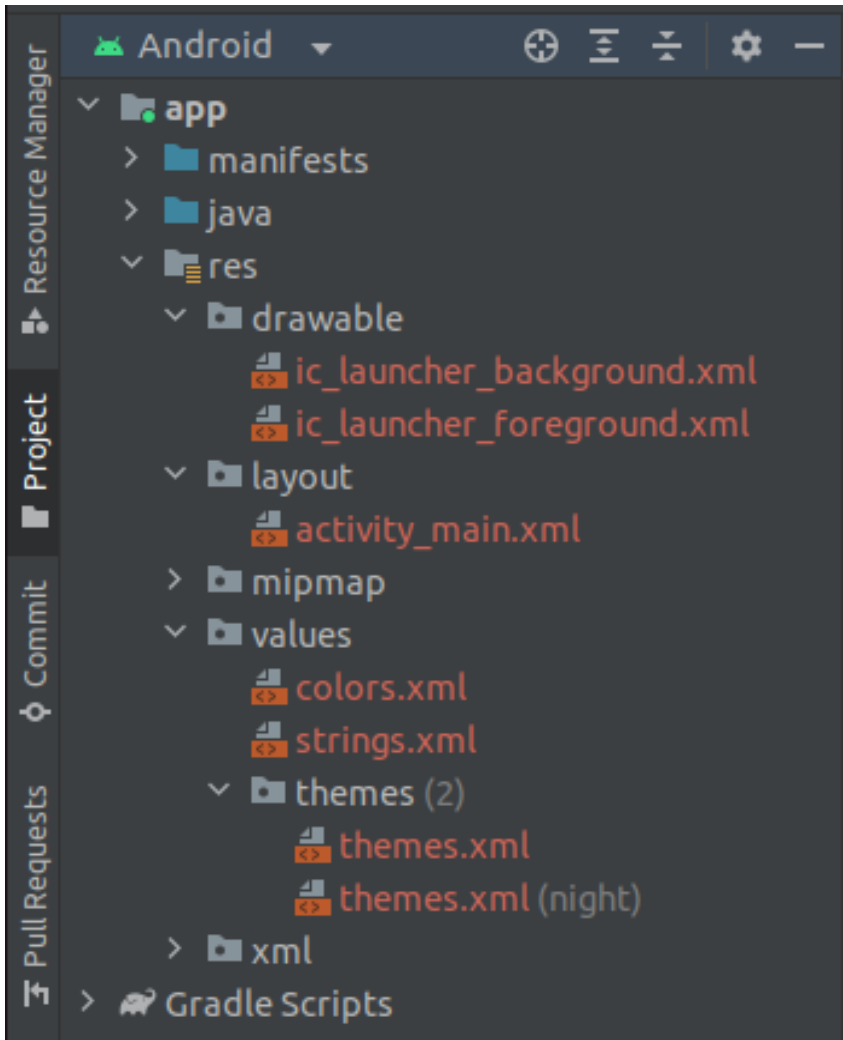
- activity_main.xml → Code / Split / Design
- Gradle files
- MainActivity.kt
- AndroidManifest.xml

Experience with the
different project views



ANDROID DESIGN WITH *VIEWS*

APP resources



All APP resources must be placed in its corresponding "res" subfolder:

- Drawable: images, icons ...
- Layout: UI layouts
- Values: colors, strings ...
- Themes: light and dark themes

ANDROID DESIGN WITH *VIEWS*

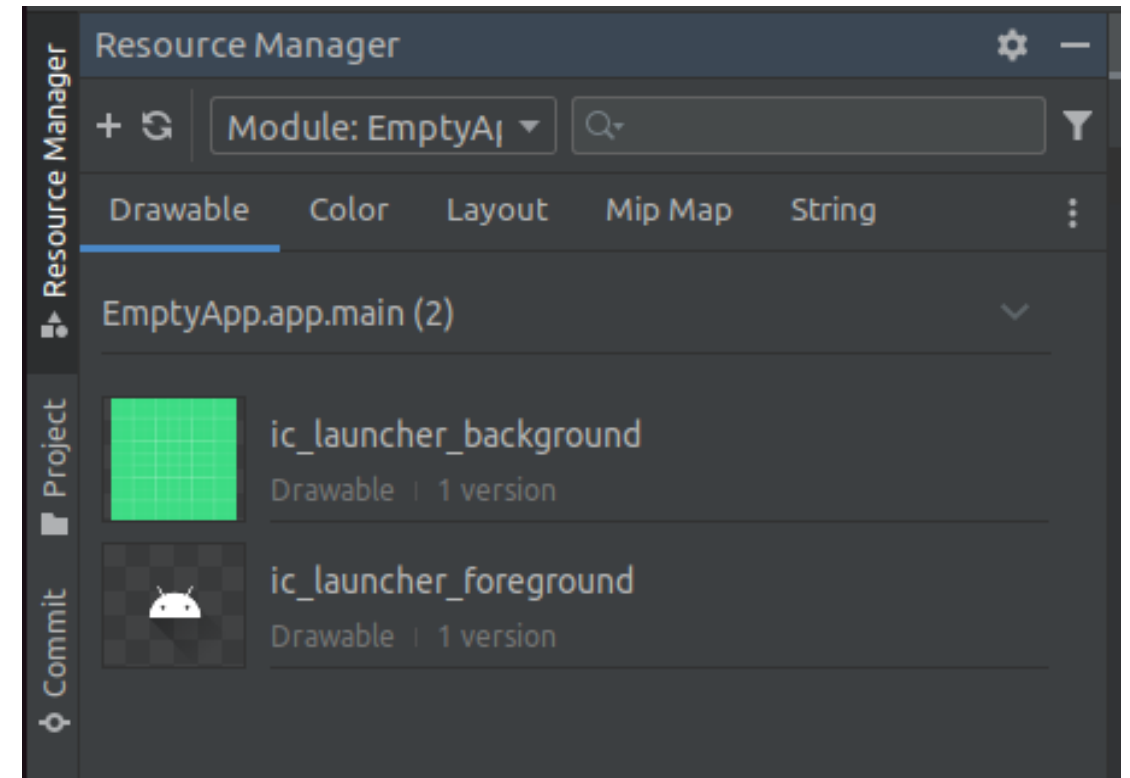
APP resources

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```

```
<resources>
    <string name="app_name">EmptyApp</string>
    <string name="author">Angel Olmos</string>
    <string name="iesme">IES Maria Enríquez</string>
</resources>
```

Different "strings.xml" files are used to have multi-language APPs

One **MUST** use the string.xml !!!



"Resource Manager" tab shows an interface to set all resources (not XML)

ANDROID DESIGN WITH *VIEWS*

activity_main.xml content

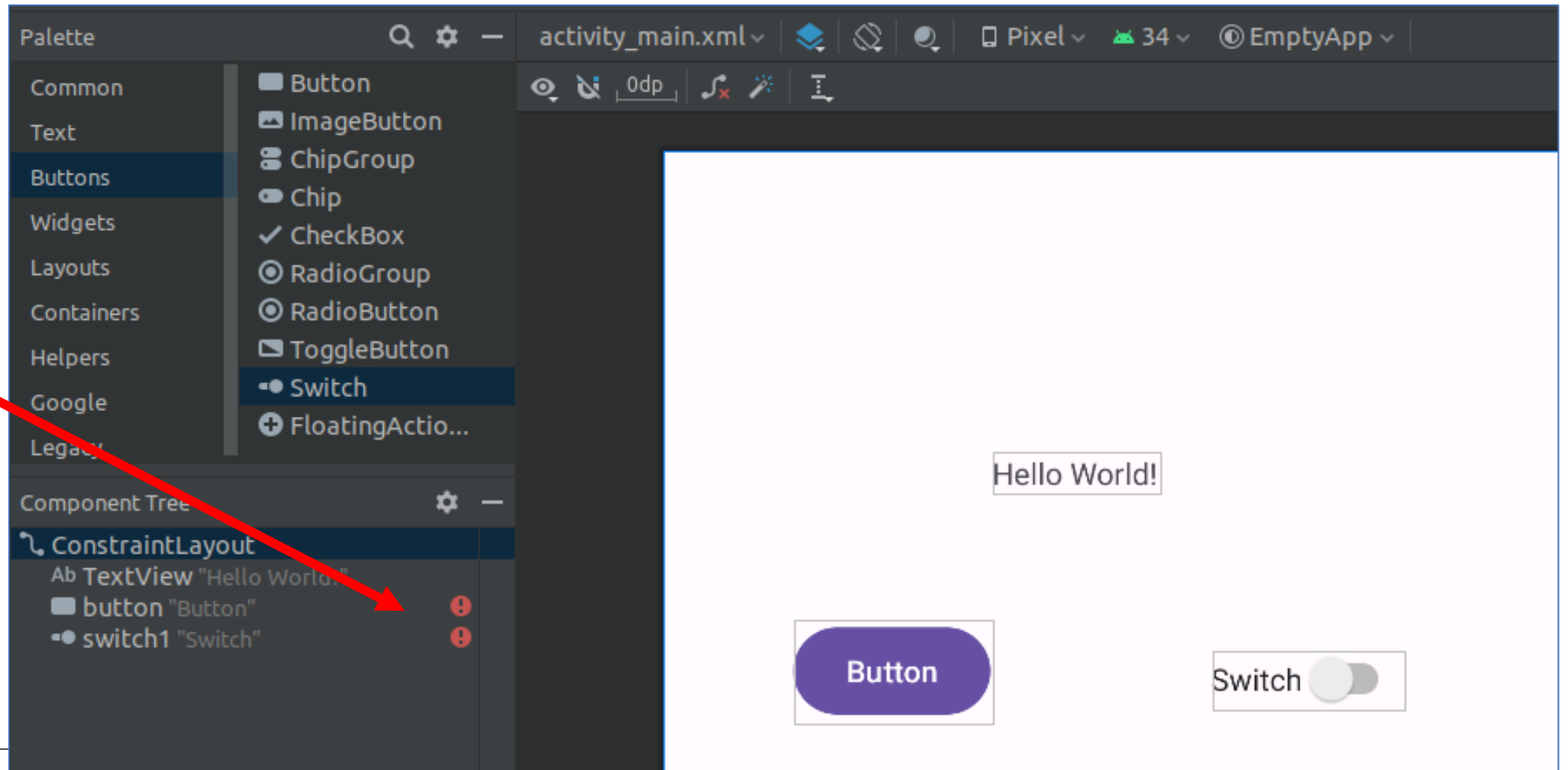
```
1  <?xml version="1.0" encoding="utf-8"?>
2  <androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      tools:context=".MainActivity">
8
9      <TextView
10          android:layout_width="wrap_content"
11          android:layout_height="wrap_content"
12          android:text="Hello World!"
13          app:layout_constraintBottom_toBottomOf="parent"
14          app:layout_constraintEnd_toEndOf="parent"
15          app:layout_constraintStart_toStartOf="parent"
16          app:layout_constraintTop_toTopOf="parent" />
17
18  </androidx.constraintlayout.widget.ConstraintLayout>
```

Text
component

ANDROID DESIGN WITH *VIEWS*

Add other components

Constraints
error



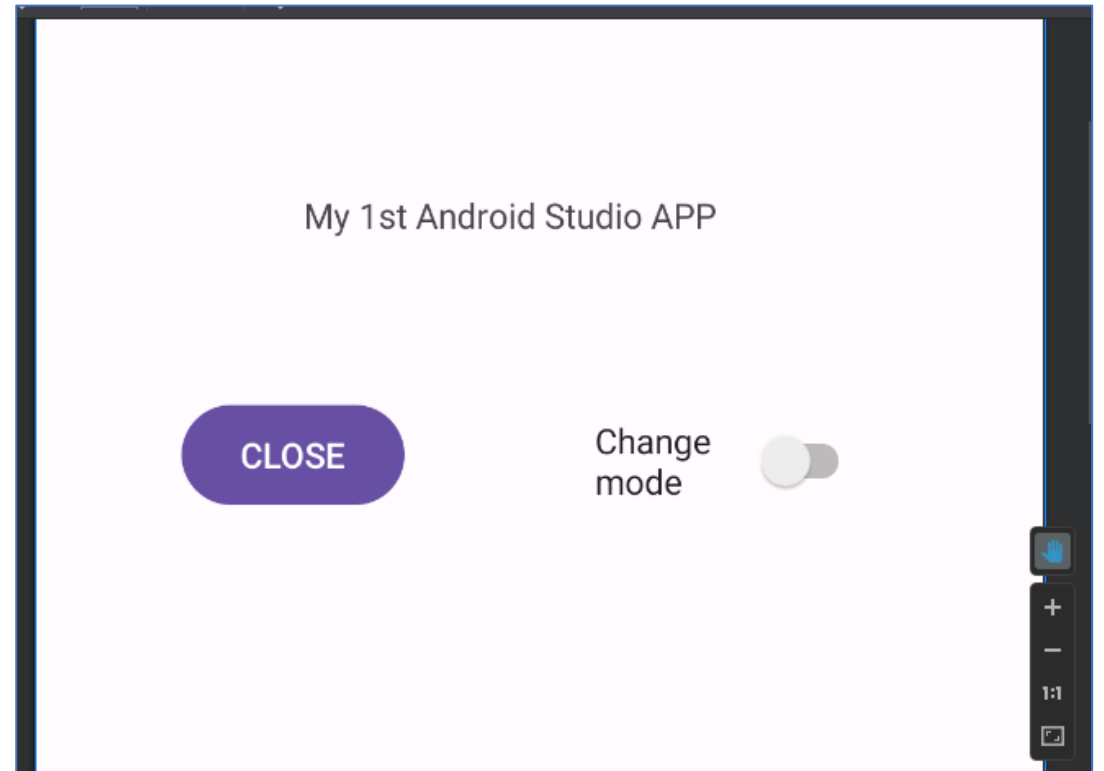
ANDROID DESIGN WITH *VIEWS*

Add other components

1- Correct errors and warnings

- Constraints errors
- Switch size errors
- Hardcoded string values ...

2- Change strings to meaningful values



ANDROID DESIGN WITH *VIEWS*

Add other components

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    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="@string/main_text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.452"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.196" />
```

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_close"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.187"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.325" />
```

```
<Switch
    android:id="@+id/switch1"
    android:layout_width="111dp"
    android:layout_height="60dp"
    android:text="@string/switch_mode"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.76"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.325" />
```

ANDROID DESIGN WITH *VIEWS*

MainActivity.kt

```
MainActivity.kt x
1  package com.dam.pmdm.emptyapp
2
3  import ...
4
5
6  class MainActivity : AppCompatActivity() {
7
8      override fun onCreate(savedInstanceState: Bundle?) {
9          super.onCreate(savedInstanceState)
10         setContentView(R.layout.activity_main)
11     }
12 }
```

What is what?

ANDROID DESIGN WITH *VIEWS*

MainActivity.kt

1- Define variables to access the different components

```
private lateinit var name : type
```

2- Assign components to variables

```
findViewById()
```

3- Modify UI and/or perform actions when interacting with components

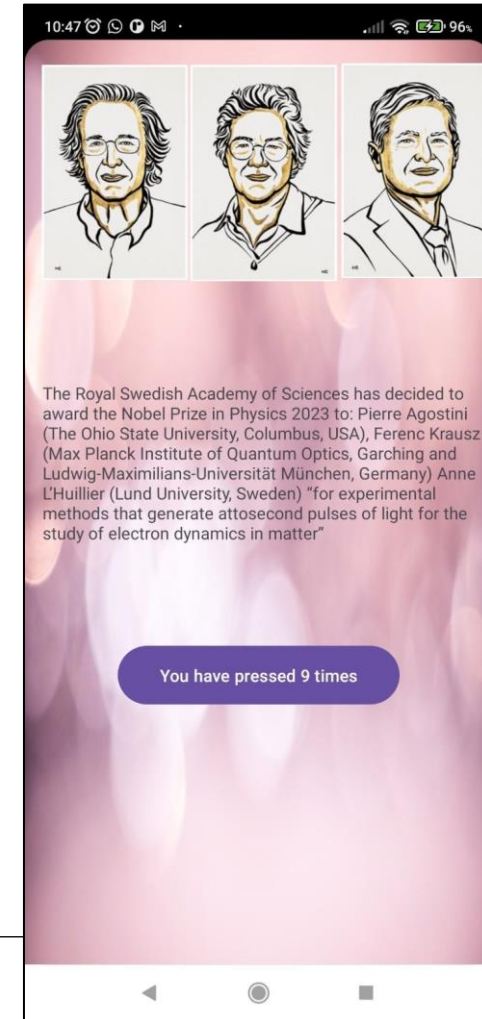
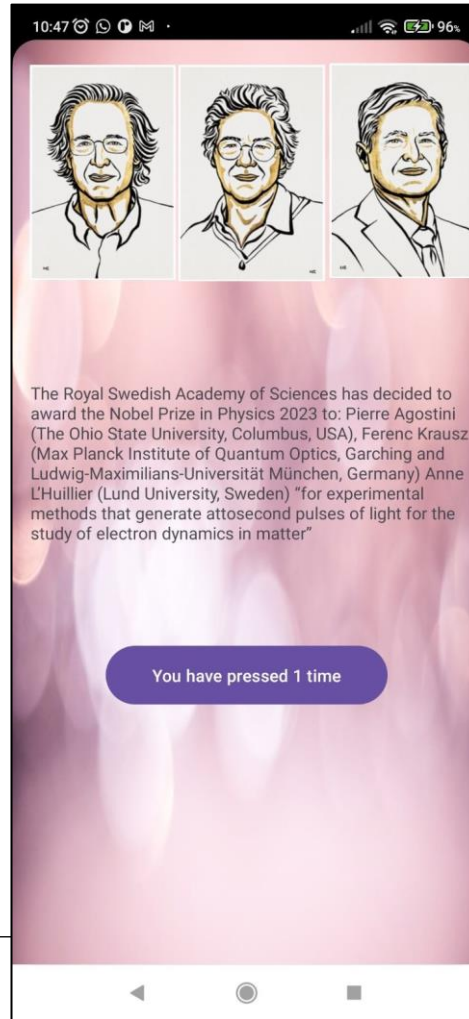
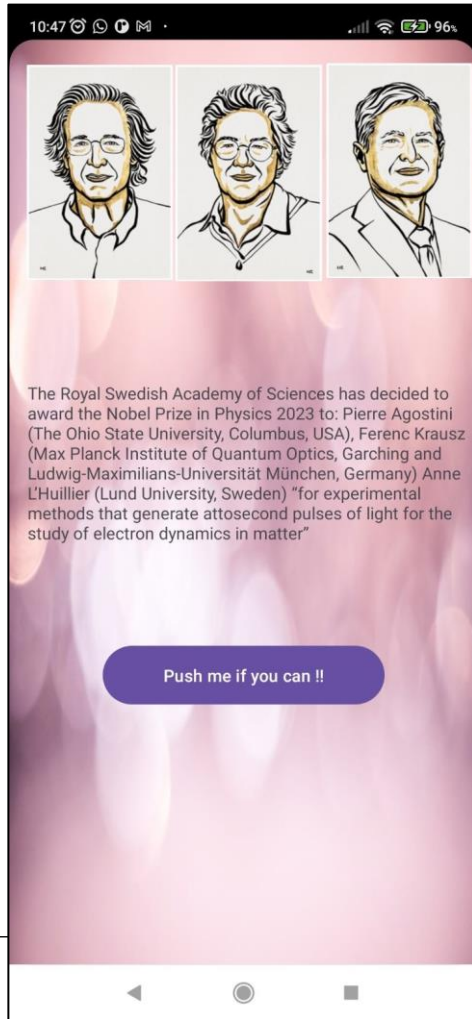
```
setOnCheckedChangeListener{ }
```

```
setOnClickListener{ }
```

```
...
```

ANDROID DESIGN WITH *VIEWS*

Android Views Activity



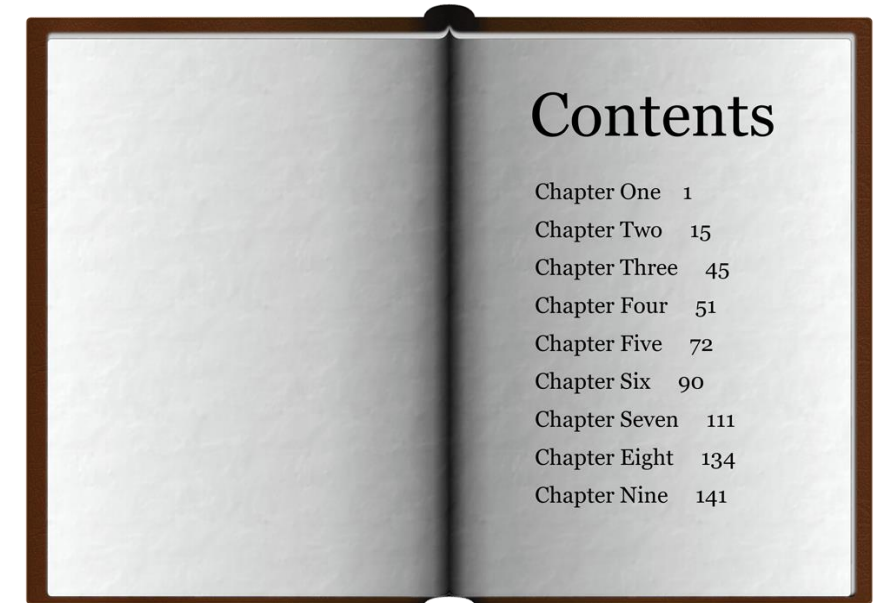
CONTENT

1. INTRODUCTION

2. ANDROID DESIGN WITH VIEWS

3. JETPACK COMPOSE INTRO

4. COMPONENTS



JETPACK COMPOSE INTRO

Characteristics

- **Declarative UI:** you describe how the UI should look like based on its current state
 - **Kotlin-Centric:** Developers can create UIs using Kotlin code
 - **Component-Based:** encourages the creation of reusable UI components
 - **Automatic UI updates:** when the underlying data changes, reducing the need for manual UI updates
 - **Modern Material Design:** Jetpack Compose seamlessly integrates with Material Design, making it straightforward to create apps that follow Google's design guidelines
 - **Gradual Adoption:** it can be gradually integrated into existing apps, allowing developers to migrate their UI components at their own pace
-

JETPACK COMPOSE INTRO

@notations

- Metadata that can be added to code elements, such as classes, functions, properties, variables, parameters ...
- Provides additional information or special configuration
- Do not have a direct impact on the behavior of the program **at run time**, but are used to provide additional information to the compiler, development tools, analysis processes ...
- *@Preview*: Preview the generated component (not on components that need an input variable)
- *@Composable*: Code that belongs to a composable declarative component

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
```

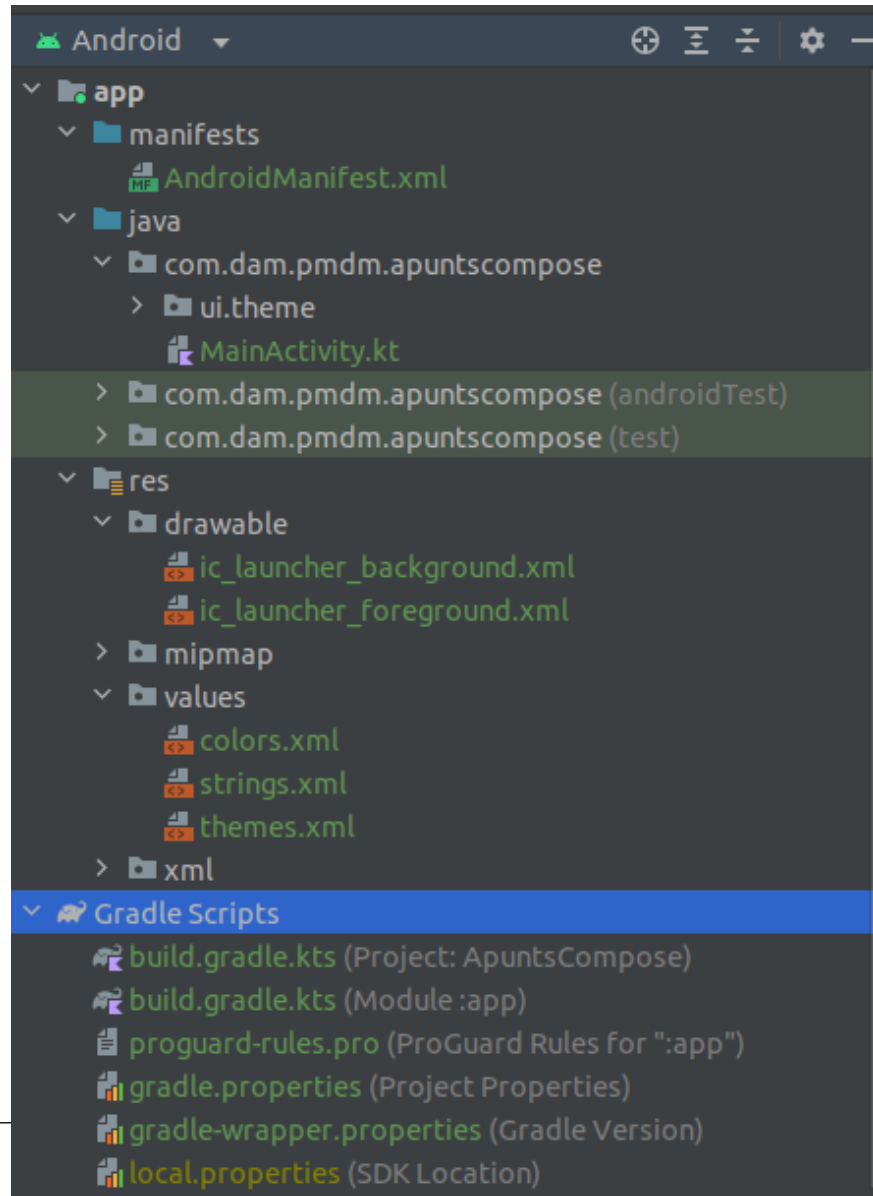
JETPACK COMPOSE INTRO

Units

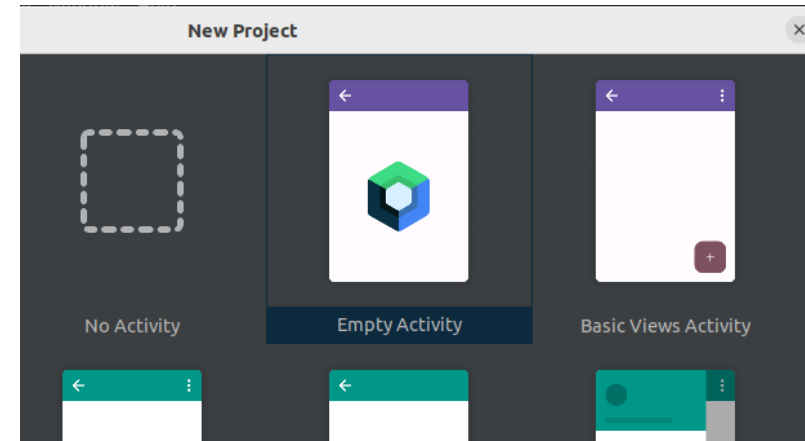
- **px**: "pixel", the smallest homogeneous unit in color that is part of a digital image
- **dp**: Density of Independent Pixels, based on the physical density of the screen. The dp-to-pixel ratio will change with screen density, not necessarily in direct proportion
- **dip**: it is the same as *dp*
- **sp**: Independent of Pixel scaling, *dp* unit scaled by the user's font size preference.

To be used in texts

JETPACK COMPOSE INTRO



Create a New compose project



Differences WRT Views?

JETPACK COMPOSE INTRO

```
class MainActivity : AppCompatActivity() {
    new *
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ApuntsComposeTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting(name: "Android")
                }
            }
        }
    }
}
```

What's this?
From where
does this come?

Surface layout
component
inside which the
UI is composed

A "composable" function that
defines what is inside the Surface

JETPACK COMPOSE INTRO

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

new *
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    ApuntsComposeTheme {
        Greeting(name: "Android")
    }
}
```

Only @Composable methods will show components in the UI

Text component and its parameters

@Preview methods are displayed in the preview tab

What's will appear in the preview?

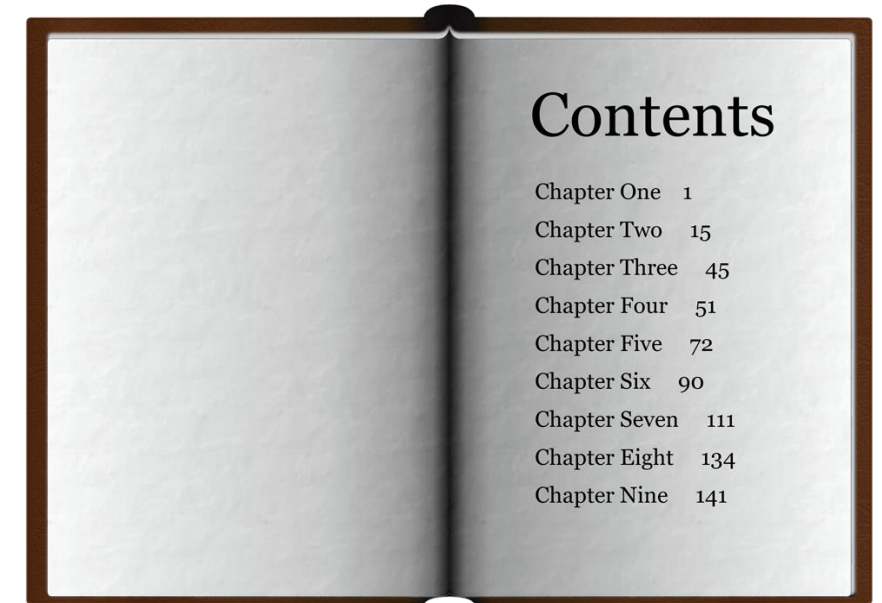
CONTENT

1. INTRODUCTION

2. ANDROID DESIGN WITH VIEWS

3. JETPACK COMPOSE INTRO

4. COMPONENTS



COMPONENTS

Libraries / Dependencies

- *Material 3* is the latest version of Google's open-source design system
- <https://developer.android.com/reference/kotlin/androidx/compose/material3/package-summary>

Be aware to have correct dependencies and imports

```
dependencies { this: DependencyHandlerScope

    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.2")
    implementation("androidx.activity:activity-compose:1.7.2")
    implementation(platform("androidx.compose:compose-bom:2023.03.00"))
    implementation("androidx.compose.ui:ui")
    implementation("androidx.compose.ui:ui-graphics")
    implementation("androidx.compose.ui:ui-tooling-preview")
    implementation("androidx.compose.material3:material3")
    implementation("androidx.navigation:navigation-compose:2.7.0") // a
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
```

```
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.material3.TextField
import androidx.compose.material3.TopAppBar
import androidx.compose.material3.TopAppBarDefaults
```

COMPONENTS

Text

- Displays highly customizable text in user interfaces
- One can specify a color using one of the values provided by the *Color* class, such as *Color.Red*, *Color.Blue*, *Color.Black* ...
- One of the most important properties in these elements is the use of the ***Modifier*** property, which allows us to set the components widths, paddings, backgrounds ...

... smells like??



Text

Explore the different *Text* parameters and modifiers to get the this



COMPONENTS

Text

Have a look at all the *Modifiers* to know what's available ... and try them

padding
background
size
border
fillMax...
height
width
...

```
Angel Olmos
@Composable
fun Greeting(name: String, modifier: Modifier) {
    Text(
        text = "Hello $name!",
        fontSize = 32.sp,
        color = Color.White,
        modifier = modifier
            .fillMaxWidth()
            .padding(15.dp)
            .background(color = Color.Red)
    )
}
```

background(color: Color, shape: Shape = ...) Modifier
background(brush: Brush, shape: Shape = ...) Modifier
padding(all: Dp) for Modifier in androidx.compose.foundation Modifier
padding(paddingValues: PaddingValues) for Modifier in androidx.compose.foundation Modifier
padding(horizontal: Dp = ..., vertical: Dp = ...) Modifier
absoluteOffset(x: Dp = ..., y: Dp = ...) for Modifier in androidx.compose.foundation Modifier
absoluteOffset {...} (offset: Density.() -> Int) Modifier
absolutePadding(left: Dp = ..., top: Dp = ..., right: Dp = ..., bottom: Dp = ...) Modifier
alpha(alpha: Float) for Modifier in androidx.compose.foundation Modifier
animateContentSize { initialValue, targetValue } Modifier
animateContentSize(animationSpec: FiniteAnimationSpec<Float>, initialValue: Float, targetValue: Float) Modifier
aspectRatio(ratio: Float, matchHeightConstraintsOnly: Boolean = false) Modifier

Press Intro to insert, Tabulator to replace Next Tip

COMPONENTS

Column / Row and LazyColumn / LazyRow

- Layout elements used to **organize UI components into columns or rows**
 - Difference between Lazy and not-Lazy:
 - the way they handle on-screen elements (recomposition)
 - Lazy components use *item* elements inside
 - LazyColumn is more efficient for displaying large lists or infinite scrolling. Loads and displays items as the user scrolls through the list, improving performance
-

COMPONENTS

Column / Row and LazyColumn / LazyRow

```
@Composable
fun ColumnRowExample(modifier: Modifier = Modifier) {
    Column { this: ColumnScope
        Text(text = "Item 1", fontSize = 20.sp, color = Color.Blue)
        Text(text = "Item 2", fontSize = 20.sp, color = Color.Green)
        Text(text = "Item 3", fontSize = 20.sp, color = Color.Red)
    }
    Column { this: ColumnScope
        Text(text = "Item 4", fontSize = 20.sp, color = Color.Yellow)
        Text(text = "Item 5", fontSize = 20.sp, color = Color.Gray)
        Text(text = "Item 6", fontSize = 20.sp, color = Color.Magenta)
    }
    Row { this: RowScope
        Text(text = "Item 7", fontSize = 20.sp, color = Color.Black)
        Text(text = "Item 8", fontSize = 20.sp, color = Color.Yellow)
        Text(text = "Item 9", fontSize = 20.sp, color = Color.Cyan)
    }
}
```

Copy+Paste and Try this
.... what's wrong?

```
@Preview(showBackground = true)
@Composable
fun PreviewColumnRow() {
    ApuntsComposeTheme {
        ColumnRowExample()
    }
}
```

COMPONENTS

Column / Row and LazyColumn / LazyRow

Copy + Paste + Try

```
@Composable
fun LazyColumnExample(){
    LazyColumn(
        modifier = Modifier
            .fillMaxWidth()
            .background(Color.LightGray)
    ) { this: LazyListScope
        item { this: LazyItemScope
            Text(
                text = "Module PMDM",
                fontSize = 32.sp,
                color = Color.Blue,
            )
        }
    }
}
```

```
item{ this: LazyItemScope
    Spacer(modifier = Modifier.height(20.dp))
}
```

```
item { this: LazyItemScope
    Text(
        text = "Maria Enriquez 2023",
        fontSize = 24.sp,
        color = Color.White,
    )
}
```

PreviewLazyColumnExample

Module PMDM

Maria Enriquez 2023

COMPONENTS

Column / Row and LazyColumn / LazyRow

These components have placement parameters like:

- *horizontalAlignment*
 - *verticalArrangement*
- } Column()
- *verticalAlignment*
 - *horizontalArrangement*
- } Row()

Try them at home, but try !!!!!

```
Column(  
    modifier = Modifier  
        .padding(40.dp)  
        .fillMaxSize(),  
    horizontalAlignment = Alignment.CenterHorizontally,  
    verticalArrangement = Arrangement.Center  
) { this: ColumnScope
```

... smells like??



```
.interno{  
    /* Centrado absoluto texto */  
    display: flex;  
    justify-content: center;  
    align-items: center;
```


COMPONENTS

Images

- *Image* component + *painterResource()* method --> used to load the image from your app's resources
 - *contentDescription* property is used to provide an optional description of the image for accessibility
 - You can further customize the appearance of the image using modifiers like *Modifier* to adjust its size, position, and other attributes
-

COMPONENTS

Images

```
item{ this: LazyItemScope
    Image(
        painter = painterResource(id = R.drawable.pmdm),
        contentDescription = "PMDM module image",
        modifier = modifier.padding(15.dp).height(150.dp)
    )
}
```

Add an image to
your *drawables*
and include it in
the LazyColumn

Most of components can behave as buttons
using the **.clickable** modifier ... **try it !!**

Module PMDM

Maria Enriquez 2023



COMPONENTS

States

- *remember*, *rememberSaveable* and *mutableStateOf* are used to manage state in a Compose application
 - *mutableStateOf*: used to create a mutable variable that cause a part of the UI to be recomposed when its value changes
 - *remember* & *rememberSaveable*: used to store and restore the state of a composable element across recompositions, lifecycles (as screen rotations) or device configuration changes
 - *remember* retains state only across recompositions (not lifecycles or configuration changes → *rememberSaveable*)
 - *rememberSaveable* automatically saves any value that can be saved in a *Bundle*
-

COMPONENTS

TextField

Add one to your
APP

- UI control to **get data** from the user (numbers or text)
- To update the *TextField* with the typed data from the user, one has to:
 1. Create a remembered variable that stores a mutable state of type *TextFieldValue*
 2. Reassign the *value* parameters every time it changes

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun SimplestTextField() {
    var text by remember { mutableStateOf(TextFieldValue(text = "")) }
    TextField(
        value = text,
        onValueChange = {newText -> text = newText},
    )
}
```

Change to: *var text = ""*
and see the difference

Shorter way of writing?

onValueChange = {text = it},

COMPONENTS

TextField

- **label:** text that will be floated on the top of the *TextField*
- **placeholder:** Displays descriptive text within the box when *TextField* is empty

Modify previous *TextField* composable to accept customizable label and placeholder ... and test it

```
fun SimpleTextField(label : String, placeholder : String) {  
    var text by remember { mutableStateOf(TextFieldValue(text = "")) }  
    TextField(  
        value = text,  
        label = { Text(text = label) },  
        placeholder = { Text(text = placeholder) },  
        onChange = { text = it },  
    )  
}
```

COMPONENTS

TextField

- **keyboardOptions**: parameter that defines the type of keyboard showed to the user and so the type of allowed data

```
keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
```

KeyboardType.Text

KeyboardType.Ascii

KeyboardType.Number

KeyboardType.Phone

KeyboardType.Uri

KeyboardType.Email

KeyboardType.Password

KeyboardType.NumberPassword

COMPONENTS

OutlinedTextField

Don't do it like this
How?

- Creates a *TextField* with an outline border
- One can use icons before and after the text:
 - *leadingIcon*: adds an icon in the starting area
 - *trailingIcon*: adds an icon in the ending area



Add one to your APP

```
fun emailTextField() {  
    var text by remember { mutableStateOf(TextFieldValue(text: "")) }  
    OutlinedTextField(  
        value = text,  
        label = { Text(text = "Email address") },  
        placeholder = { Text(text = "Enter your e-mail") },  
        leadingIcon = { Icon(  
            imageVector = Icons.Default.Email,  
            contentDescription = "emailIcon") },  
        trailingIcon = { Icon(  
            imageVector = Icons.Default.Add,  
            contentDescription = "trailingIcon") },  
        onValueChange = { text = it },  
    )  
}
```

COMPONENTS

Buttons

Buttons communicate actions that users can take

Important elements:

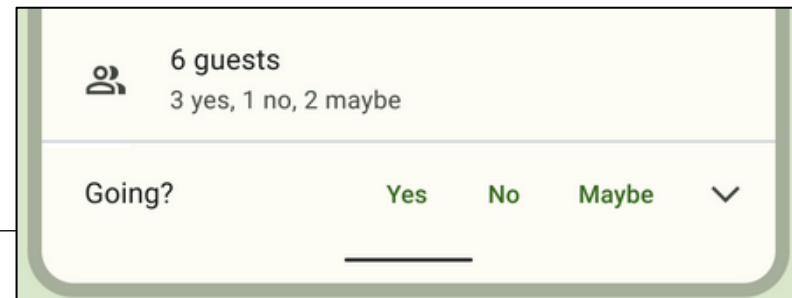
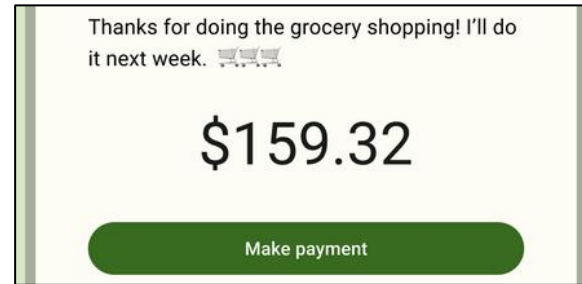
- Text: It describes the action that will occur if a user taps a button
 - Container: Button containers hold the label text and optional icon. Text buttons have a visible container only when hovered, focused, or pressed
 - Icon (optional): Icons visually communicate the button's action and help draw attention. They should be placed on the leading side of the button, before the label text
-

COMPONENTS

Buttons

There are 5 types of buttons:

1. Filled buttons
2. Outlined buttons
3. Filled tonal buttons
4. Elevated buttons
5. Text buttons



COMPONENTS

Buttons

Add a Subscribe button
to your APP

Module PMDM

Maria Enriquez 2023

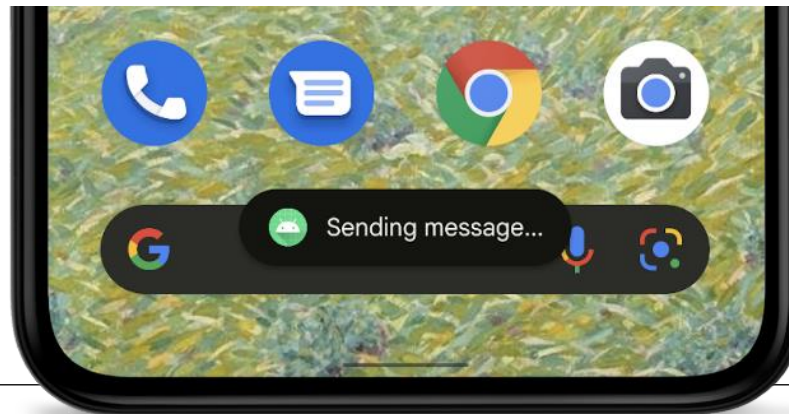


Subscribe

COMPONENTS

Toast

- A toast is a view containing a **quick little message** for the user
- When the view is shown to the user, appears **as a floating view over the application**
- It will never receive focus and **ongoing activity remains visible and supports interaction**
- Alerts **disappear automatically** after a timeout period



COMPONENTS

Toast

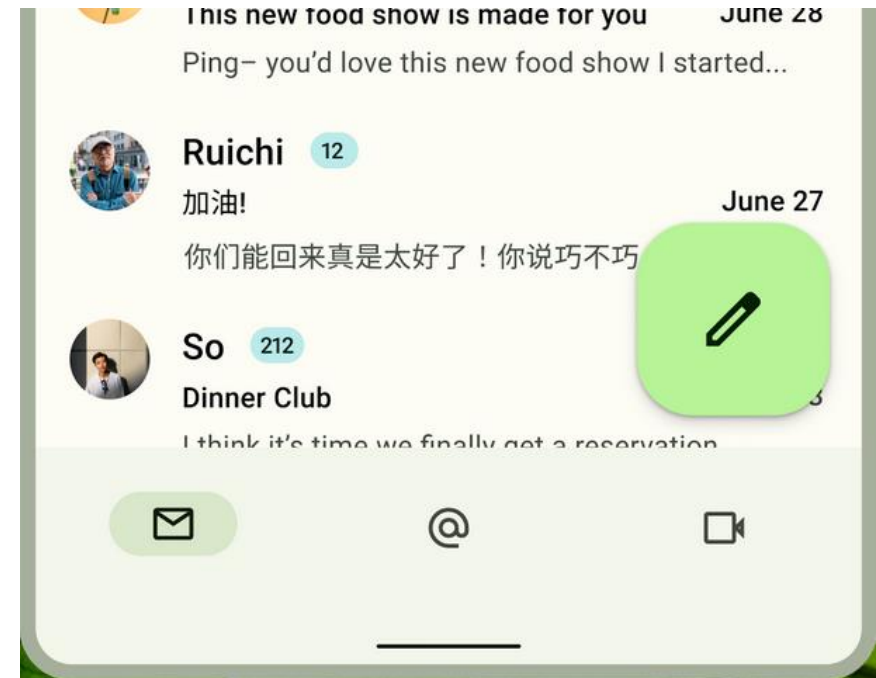
- The easiest way is to call the static method *makeText()* that constructs everything you need and returns a new Toast object
- **Instantiate a Toast object:** Use the *Toast.makeText()* method, which takes the following parameters:
 - The activity **Context** (*val context = LocalContext.current*)
 - The text that should appear to the user
 - The duration on the screen (*val duration = Toast.LENGTH_SHORT*)
- Call the *show()* method of the new Toast object to **display the toast**

Show a subscription Toast when a button is pressed

COMPONENTS

FAB Button

- Appears in front of all other content on screen
- **Persist** on the screen when content is **scrolling**
- Use a FAB for the most common or important action on a screen (primary action)
- Can be aligned left, center, or right. It can be positioned above the navigation bar, or nested within the bar



COMPONENTS

FAB Button

```
FloatingActionButton(  
    containerColor = colorResource(id = R.color.teal_700),  
    onClick = { }  
{  
    Text( fontSize = 24.sp, text = "+" )  
}
```

Add a "+" FAB to your
APP

Module PMDM

Maria Enriquez 2023



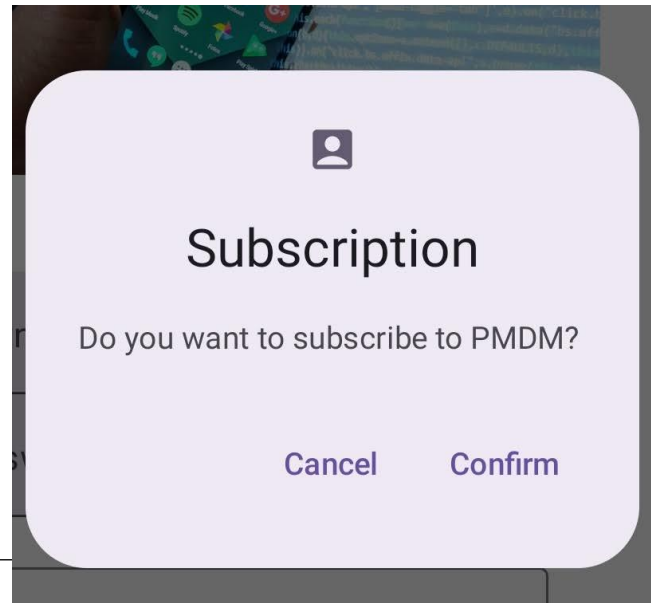
Subscribe



COMPONENTS

Dialog

- Dialogs provide important prompts in a user flow
- They can require an action, communicate information, or help users accomplish a task
- The dialog will position its buttons, typically *TextButtons*, based on the available space



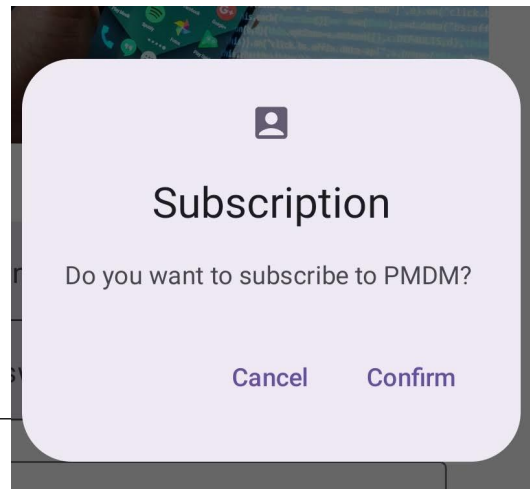
COMPONENTS

Dialog

- The *AlertDialog* is a subclass of *Dialog* that can display one, two or three buttons
- One of the main differences wrt the *Toast* is that the focus is fixed on the dialog and the **ongoing activity does not supports interaction** until the dialog is closed

Main parameters:

- *onDismissRequest*: called when the user tries to dismiss the *Dialog* by clicking outside or pressing the back button
- *confirmButton*
- *dismissButton*
- *icon* – optional
- *Title* and *text*



Create this dialog as a composable (**leave lambdas empty**) and launch it when pressing the FAB

COMPONENTS

Dialog

- Close dialog depending on the user action → Set input parameters

```
fun SubscribeDialog(show: Boolean, onDismiss: () -> Unit, onConfirm: () -> Unit) {  
    if (show) {  
        AlertDialog(  
            onDismissRequest = { onDismiss() },    // When pressed outside or going Back  
            confirmButton = {  
                TextButton(onClick = { onConfirm() }) { this: RowScope  
                    Text(text = "Confirm")  
                }  
            },  
            dismissButton = {  
                TextButton(onClick = { onDismiss() }) { this: RowScope  
                    Text(text = "Cancel")  
                }  
            },  
        )  
    }  
}
```

COMPONENTS

Dialog

- Close dialog depending on the user action → Show/Hide depending on ...

```
var show by rememberSaveable { mutableStateOf( value: false) }  
SubscribeDialog(  
    show = show,  
    onDismiss = { show = false },  
    onConfirm = {  
        Log.i( tag: "PMDM", msg: "Subscription accepted")  
        show = false  
    })
```

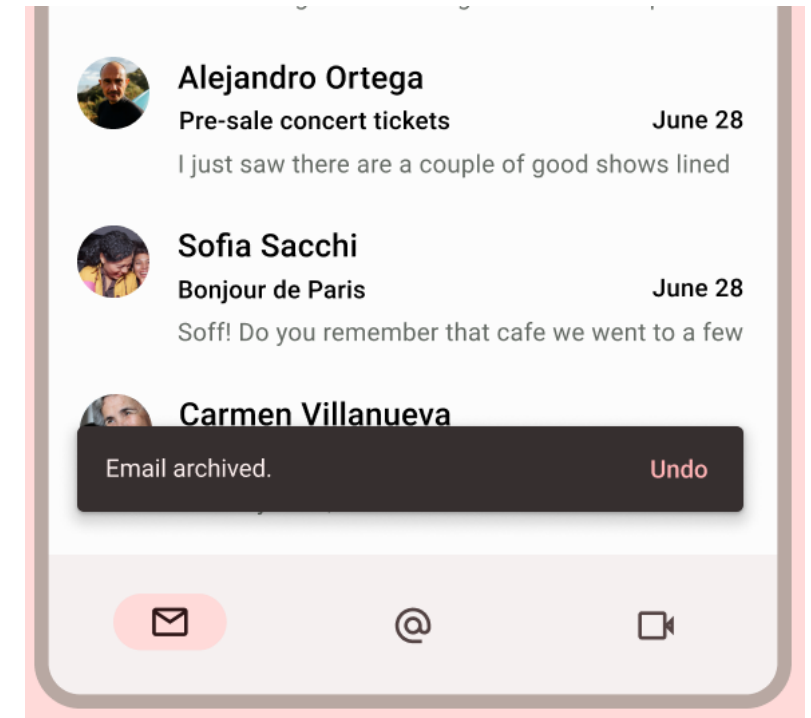
Who / When to
set it to *true*?

Check *Logcat* when confirming

COMPONENTS

Snackbar

- Serves as a brief notification that appears at the bottom of the screen
- It provides feedback about an operation or action **without interrupting the user experience**
- Snackbars **disappear after a few seconds**
- The user can also dismiss them with an action, such as tapping a button



COMPONENTS

Snackbar


- To implement a snackbar, you first create *SnackbarHost*, which includes a *SnackbarHostState* property
 - *SnackbarHostState* provides access to the *showSnackbar()* function which you can use to display your snackbar
 - This suspending function requires a *CoroutineScope* (as *rememberCoroutineScope*) in order to be called in response to UI events, such a buttons pressing
-

COMPONENTS

Snackbar

```
fun ScaffoldWithOnlySnackBar() {  
    val scope = rememberCoroutineScope()  
    val snackbarHostState by remember { mutableStateOf( SnackbarHostState() ) }  
  
    Scaffold(  
        snackbarHost = { SnackbarHost(hostState = snackbarHostState) },  
        content = { it: PaddingValues  
            Button(onClick = {  
                scope.launch {snackbarHostState.showSnackBar(message = "You've been subscribed to PMDM")}})  
            {Text(text = "Subscribe")}  
        },  
    )  
}
```

UI layout
We'll see
later

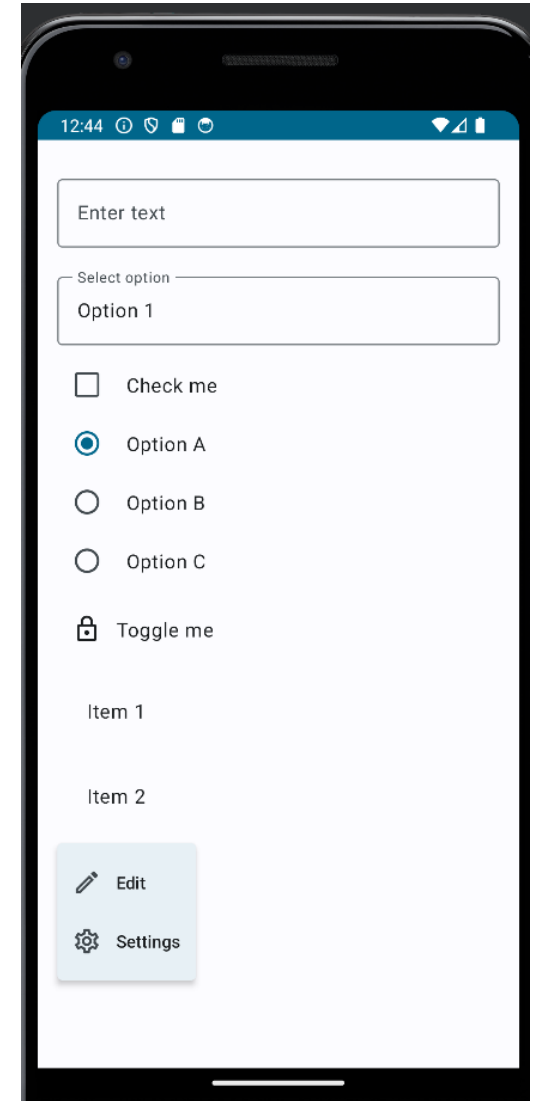


Try it

COMPONENTS

Form Components

- Forms are essential for data entry
- Among the most used components we find:
 - TextField
 - Dropdown
 - Checkboxes
 - Radiobuttons
 - ToggleButtons
 - Spinners



COMPONENTS

Form Components

```
// CheckBox
Row(
    verticalAlignment = Alignment.CenterVertically
) { this: RowScope
    Checkbox(
        checked = checkBoxChecked,
        onCheckedChange = { checkBoxChecked = it },
        modifier = Modifier.padding(end = 8.dp)
    )
    Text(text = "Check me")
}
```

☐ Check me

COMPONENTS

Form Components

```
val radioOptions = listOf("Option A", "Option B", "Option C")
var selectedRadioButton by remember { mutableStateOf(radioOptions[0]) }
```

```
// Radio Buttons
Column { this: ColumnScope
    radioOptions.forEach { option ->
        Row(
            verticalAlignment = Alignment.CenterVertically,
            modifier = Modifier
                .selectable(
                    selected = selectedRadioButton == option,
                    onClick = { selectedRadioButton = option }
                )
            .fillMaxWidth()
        ) { this: RowScope
```

- ☒ Option A
- ☐ Option B
- ☐ Option C



COMPONENTS

Form Components



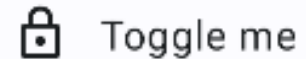
```
RadioButton(  
    selected = selectedRadioButton == option,  
    onClick = { selectedRadioButton = option },  
    colors = RadioButtonDefaults.colors(selectedColor = MaterialTheme.colors.  
    modifier = Modifier.padding(end = 8.dp)  
)  
Text(text = option)
```

- ☒ Option A
- ☐ Option B
- ☐ Option C

COMPONENTS

Form Components

```
// Toggle Button
Row(
    verticalAlignment = Alignment.CenterVertically,
    modifier = Modifier.padding(vertical = 8.dp)
) { this: RowScope
    var checked by remember { mutableStateOf( value: false) }
    IconToggleButton(checked = checked,
        onCheckedChange = { checked = it })
    {
        if (checked) {
            Icon(Icons.Filled.Lock, contentDescription = "Localized description")
        } else {
            Icon(Icons.Outlined.Lock, contentDescription = "Localized description")
        }
    }
    Text(text = "Toggle me")
}
```



LICENSE



Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0)

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



NonCommercial — You may not use the material for [commercial purposes](#).



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.

No additional restrictions — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.

<https://creativecommons.org/licenses/by-nc-sa/3.0/>
