

ANDROID COURSE

Android Development Fundamentals with Compose

9 Chapters

Philip Osztromok

Generated with Claude

Contents

Chapter 1	First Compose App	2
Chapter 2	Activities and Lifecycle	8
Chapter 3	Composables and Layouts	13
Chapter 4	State and Recomposition	19
Chapter 5	User Interaction	25
Chapter 6	Lists with LazyColumn	31
Chapter 7	Navigation Compose	38
Chapter 8	Resources and Themes	46
Chapter 9	Legacy Appendix	52

Android Development Fundamentals with Compose

Course 4 · Chapter 1 · Your First App, Built in Code

Older Android courses start by drawing a screen in an XML file. This one does not. With **Jetpack Compose** the screen is Kotlin code: you write **composable functions**, each one describing a piece of UI, and Android draws what they describe. By the end of this chapter you will have created a project, found your way around its files, read the generated `MainActivity.kt` line by line, changed what it shows and run it.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install (Android Studio Rabbit 2, from earlier work):** the new-project wizard's Empty Activity option, the default minimum SDK (API 24), and the list of files a generated Compose project contains (no `res/layout` folder). **Checked in the docs (one web search, summaries only):** that Empty Activity is the recommended starting point and that its `MainActivity` calls `setContent` with a `Scaffold` and a `Greeting` composable. **Not verified:** the exact code your Rabbit 2 template generates (the docs I found did not confirm it, so the listing below is a *shape*; compare it with your own file), and **all code in this chapter is unrun**, because there is no Kotlin compiler on the machine the course was written on.

Create the Project

The wizard is covered screen by screen in Android Studio 2: The New UI 3, so here is only the path. From the Welcome screen choose **New Project** (or **File > New > New Project**), pick **Empty Activity** (the Compose one: the Views templates have different names), then fill in the second screen:

FIELD	WHAT TO ENTER
Name	`ComposeHello` (any name; this course's examples use it).
Package name	Left as suggested, for example `com.example.composehello`.
Save location	A folder of your choice.
Minimum SDK	The default on the install was API 24 (Android 7.0). Accept it.

Press **Finish**. The first Gradle sync downloads what the project needs and can take a few minutes; wait for it to end before running anything (Android Studio 2: The New UI 4 explains what sync does).

What the Project Contains

In the Android view, with a few files omitted, the project looks like this (checked against a generated project):

```
ComposeHello/
├── app/
│   ├── build.gradle.kts
│   └── src/main/
│       ├── AndroidManifest.xml
│       ├── java/com/example/composehello/
│       │   ├── MainActivity.kt
│       │   └── ui/theme/
│       │       ├── Color.kt
│       │       ├── Theme.kt
│       │       └── Type.kt
│       └── res/            (no layout folder)
├── gradle/libs.versions.toml
├── build.gradle.kts
└── settings.gradle.kts
```

FILE	WHAT IT IS FOR
`AndroidManifest.xml`	Declares the app to Android: its name, icon and its activities. `MainActivity` is listed here as the screen that opens when you tap the app.
`MainActivity.kt`	The entry point: the one Activity, which puts a Compose UI on screen.
`ui/theme/`	`Color.kt`, `Type.kt` and `Theme.kt`: the app's colours, fonts and the theme function that wraps your UI (Chapter 8).
`res/`	Resources: icons, strings, colours. There is no `layout` folder, because there are no XML layouts.
`app/build.gradle.kts`	The module's build file: dependencies and settings, in Kotlin DSL.
`libs.versions.toml`	The version catalog: one place for library versions.

Reading MainActivity.kt

Compare your own file with this listing. It shows what the Empty Activity template is documented to generate; small differences (for example an extra `enableEdgeToEdge()` line, or different imports) are normal.

```
// MainActivity.kt (the shape of the generated file; yours may differ slightly)
package com.example.composehello

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.composehello.ui.theme.ComposeHelloTheme

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

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

```
            Greeting("Android")
        }
    }
```

PART	WHAT IT DOES
`ComponentActivity`	The base class for an Activity that can host Compose.
`onCreate`	Called when the screen is created. Chapter 2 covers the whole lifecycle.
`setContent { ... }`	Replaces `setContentView` and a layout file: the block inside *is* the screen.
`ComposeHelloTheme`	Your app's theme from `ui/theme/Theme.kt`; everything inside takes its colours and fonts.
`Scaffold`	A Material 3 layout skeleton (top bar, bottom bar and so on). It hands you `innerPadding` so your content does not slide under system bars.
`Greeting`	A composable function of your own: it takes data (`name`) and describes UI.
`Modifier`	A chain of instructions about size, padding, position and behaviour. Chapter 3 covers it.
`GreetingPreview`	Draws `Greeting` inside Android Studio without running the app (Android Studio 2: The New UI 5).

What Makes a Function Composable

- It is marked `@Composable`, which tells the Compose compiler that it describes UI.
- It is named in **PascalCase** (`Greeting`, not `greeting`), like a class: that is the Compose convention.
- It usually returns nothing. It does not hand back a view; calling it *emits* UI into the screen.
- It can only be called from another composable (or from `setContent`), not from ordinary code.
- It takes a `modifier: Modifier = Modifier` parameter as its first optional parameter, so callers can adjust its size and position.

Run It

Start an emulator or connect a phone (Android Studio 2: The New UI 6 shows how), choose it in the toolbar's device selector and press **Run**. You should see the greeting text in the top left. Press **Run** again after each change to see it. The preview shows the same thing without a device; it is faster for small visual changes.

Compared with the XML Way

	VIEWS AND XML (ANDROID DEVELOPMENT FUNDAMENTALS)	COMPOSE
Where the screen is defined	An XML layout file such as `activity_main.xml`	Kotlin functions
How the Activity shows it	`setContentView(R.layout.activity_main)`	`setContent { ... }`
Changing what is shown	Find a view by id and change it	Change the data; the UI is redrawn from it (Chapter 4)
Building blocks	View classes: `TextView`, `Button`	Composables: `Text`, `Button`
Seeing it without running	Layout Editor	`@Preview`

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 1 starts with an IDE tour and a first app built from XML. The IDE has changed (see Android Studio 2: The New UI 1, 3 and 4), and a new project is now a Compose project, so this chapter replaces both the tour and the first-app steps.

Try It Yourself

EXERCISE 1

Create a project called `ComposeHello` with the Empty Activity template, wait for the sync, and write down the files and folders you see in the Android view. Does your `MainActivity.kt` match the listing above? List any differences.

View solution

EXERCISE 2

Change `Greeting` so it shows two lines of text: `Hello, <name>!` and `Welcome to Compose`. Check the preview, then run the app on a device.

View solution

EXERCISE 3

Write your own composable `ProfileCard(name: String, role: String)` that shows the two values, give it a `@Preview`, then call it from `setContent` instead of `Greeting`.

View solution

NOT YET VERIFIED

Checked on the install: the Empty Activity option, API 24 as the default minimum SDK, and the generated project's file list.
Checked in the docs (one web search, summaries only): Empty Activity as the recommended Compose starting point; a template `MainActivity` with `setContent`, `Scaffold`, `Greeting` and a preview; `enableEdgeToEdge` being added by hand in the codelabs I found.
Not verified: the exact code and imports the Rabbit 2 template generates; the Compose conventions listed under "What Makes a Function Composable" (general knowledge); every code listing and solution in this chapter (unrun, not compiled); the exercises (not run by me). **If your generated file differs, trust your file and tell me what it contains.**
Sources: Create your first Android app (developer.android.com), Quick start, Jetpack Compose (developer.android.com), Create a project (developer.android.com).

COMING UP

Chapter 2 covers Activities and the lifecycle: what happens when the screen is created, paused or rotated, and how `rememberSaveable` keeps data across a rotation.

Android Development Fundamentals with Compose

Course 4 · Chapter 2 · When Android Creates and Destroys Your Screen

Your app does not decide when its screen starts, pauses or disappears: Android does, and tells your `MainActivity` through **lifecycle callbacks**. This matters in every app, because the classic surprise is that **rotating the phone destroys and rebuilds the Activity**, and anything you kept in an ordinary variable is gone. This chapter lists the callbacks, shows you how to watch them in Logcat, and shows the Compose tool for keeping a value across a rotation.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** the six core callbacks, the order they run in on a rotation, `isFinishing`, the `Log.d` pattern from a Compose codelab, and `rememberSaveable` as the way to survive a configuration change. **Checked on the install:** the Logcat filter box and its query form (Android Studio 2: The New UI 7). **General knowledge, not checked:** what each callback is *for* in practice, and exactly what `remember` loses. **All code is unrun** (no Kotlin compiler on the machine used to write this). Back-button and Home-button behaviour changes between Android versions, so the exercises ask you to record what *your* device does.

The Six Callbacks

(docs) An Activity has six core callbacks, called as it enters each new state:

CALLBACK	USUAL MEANING *(GENERAL KNOWLEDGE)*
`onCreate`	The Activity is being created. Set up the UI here: `setContent` goes in this one.
`onStart`	The screen is becoming visible.
`onResume`	The screen is in the foreground and the user can interact with it.
`onPause`	The user is leaving, or another window is covering part of the screen.
`onStop`	The screen is no longer visible.
`onDestroy`	The Activity is about to be destroyed.

Each override calls `super` first, so the base class still does its work. Compose code usually does not override these callbacks itself for every screen; a composable that needs to react to them

uses the Lifecycle API (the docs mention `LocalLifecycleOwner.current` for adding an observer), which a later chapter can show.

Watching Them in Logcat

(docs) A Compose codelab teaches this pattern: declare a `TAG` constant and call `Log.d(TAG, "...")` in each override.

```
// MainActivity.kt: log each lifecycle callback
import android.util.Log

private const val TAG = "MainActivity"

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(TAG, "onCreate")
        setContent { /* your UI */ }
    }

    override fun onStart() {
        super.onStart()
        Log.d(TAG, "onStart")
    }

    override fun onResume() {
        super.onResume()
        Log.d(TAG, "onResume")
    }

    override fun onPause() {
        super.onPause()
        Log.d(TAG, "onPause")
    }

    override fun onStop() {
        super.onStop()
        Log.d(TAG, "onStop")
    }

    override fun onDestroy() {
        super.onDestroy()
```

```
            Log.d(TAG, "onDestroy, finishing = $isFinishing")
        }
    }
```

Run the app, then in Logcat (Android Studio 2: The New UI 7) filter with `package:mine & tag:MainActivity`. You should see `onCreate`, `onStart` and `onResume` as the app opens.

Rotation Destroys the Activity

(docs) When the device rotates, the original Activity receives `onStop` and `onDestroy`, then a **new instance** is created and receives `onCreate`, `onStart` and `onResume`. The docs add that `onDestroy` runs either because the Activity is finishing (something called `finish()`) or because the system is destroying it temporarily; `isFinishing` tells you which, which is why the listing logs it. A new instance means new variables: anything held only in memory by the old one is gone.

Keeping a Value: remember and rememberSaveable

(docs) The codelab says to declare a value with `rememberSaveable` if it must survive a configuration change. *(general knowledge)* A plain `remember` keeps a value while the composable stays on screen, but not across the Activity being recreated. This small composable shows both. Chapter 4 explains `remember` and state properly; for now just compare what happens on rotation.

```
import androidx.compose.foundation.layout.Column
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.runtime.setValue
```

```
@Composable
fun Counter() {
```

```
    // remember: the value is lost when the Activity is recreated (rotation)
    var plain by remember { mutableIntStateOf(0) }
    // rememberSaveable: the value survives rotation
    var saved by rememberSaveable { mutableIntStateOf(0) }

    Column {
        Button(onClick = { plain++ }) { Text("remember: $plain") }
        Button(onClick = { saved++ }) { Text("rememberSaveable: $saved") }
    }
}
```

Tap each button a few times, rotate the device and watch which number resets.

Compared with the XML Way

	VIEWS (ANDROID DEVELOPMENT FUNDAMENTALS)	COMPOSE
Where the UI is built	`onCreate`, with `setContentView`	`onCreate`, with `setContent`
Saving a value across rotation	`onSaveInstanceState` and the `savedInstanceState` bundle	`rememberSaveable`
Reacting to lifecycle events	Override the callbacks	Override them, or observe them with the Lifecycle API
Saved data survives	Rotation and being killed in the background	The same *(general knowledge)*

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 2 teaches the same callbacks, Logcat and `savedInstanceState`. The callbacks are unchanged; what changes is how you keep state (`rememberSaveable`) and how you read the log (the New UI's query filter, Android Studio 2: The New UI 7).

Try It Yourself

EXERCISE 1

Add the logging overrides to `MainActivity`, run the app and write down the callbacks you see in Logcat as it opens. Then rotate the device and write down the new sequence.

View solution

EXERCISE 2

Add the `Counter` composable to `setContent`. Tap each button three times, rotate the device, and say which value was kept and which was lost.

View solution

EXERCISE 3

With logging in place, record the callbacks (and the `finishing` value) when you (a) press Back, (b) press Home and return, (c) rotate. Say which of them show `finishing = true`.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): the six callbacks and their order on rotation; `isFinishing`; the `TAG` and `Log.d` pattern; `rememberSaveable` for values that must survive a configuration change; `LocalLifecycleOwner.current` for observing the lifecycle in Compose.
Checked on the install: the Logcat query filter and the `package:mine` form.
Not verified: the "usual meaning" column and the exact limits of `remember` (general knowledge); what `rememberSaveable` can store beyond simple values; the callback order for Back and Home on your Android version; every listing and solution (unrun); the exercises (not run by me).
Sources: The activity lifecycle (developer.android.com), Stages of the Activity lifecycle (developer.android.com codelab), Activity state changes (developer.android.com).

COMING UP

Chapter 3 covers composables and layouts: `Row`, `Column`, `Box`, `Modifier` and the Material 3 components.

Android Development Fundamentals with Compose

Course 4 · Chapter 3 · Arranging Things on the Screen

A real screen is more than one line of text. In Compose you arrange composables by **nesting** them inside three layout composables: `Row` (side by side), `Column` (one above another) and `Box` (stacked on top of each other). You then adjust each piece with **modifiers**. This chapter covers those three, the modifiers you will use most, and a few Material 3 components.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** the roles of Row, Column and Box; that modifier order matters; `fillMaxWidth`, `size`, `padding`, `weight`, `Arrangement`, `Alignment` and `Spacer`. **Not found in the docs search:** Material 3 `Card` and `Scaffold` (described from general knowledge). **All code is unrun** (no Kotlin compiler on the machine used to write this): the imports are listed so you can check them, but expect to fix one or two with `Alt+Enter`. Nothing here was seen on your install.

Row, Column and Box

COMPOSABLE	ARRANGES ITS CHILDREN	LIKE IN XML *(GENERAL KNOWLEDGE)*
`Column`	One below another	A vertical `LinearLayout`
`Row`	Side by side	A horizontal `LinearLayout`
`Box`	Stacked, later children on top	A `FrameLayout`

(docs) By default a layout is only as big as its content. You nest them to build a screen: here a `Row` holds a coloured circle (a `Box`) and a `Column` of two lines of text.

```
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
```

```
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.shape.CircleShape
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Text
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
```

```
@Composable
fun ProfileRow(name: String, role: String) {
    Row(
        modifier = Modifier.fillMaxWidth().padding(16.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        Box(
            modifier = Modifier
                .size(48.dp)
                .background(MaterialTheme.colorScheme.primary, CircleShape)
        )
        Spacer(modifier = Modifier.width(12.dp))
        Column {
            Text(text = name, style = MaterialTheme.typography.titleMedium)
            Text(text = role, style = MaterialTheme.typography.bodyMedium)
        }
    }
}
```

Modifiers

A `Modifier` is a chain of instructions you attach to a composable: its size, spacing, background, what happens on a click. You write them one after another with dots. Common ones:

MODIFIER	EFFECT
`padding(16.dp)`	*(docs)* Adds space around the composable.
`fillMaxWidth()`	*(docs)* Takes the full width of the parent.
`size(48.dp)`	*(docs)* Gives an exact size.

MODIFIER	EFFECT
`background(color, shape)`	Paints behind the composable *(general knowledge)*.
`weight(1f)`	*(docs)* Inside a Row or Column, shares the free space by ratio.
`clickable { }`	Makes any composable respond to taps *(general knowledge; Chapter 5)*.

Order matters

(docs) A modifier chain is applied in sequence, so changing the order changes the result. The classic case is padding and background:

```
// Padding then background: the coloured area is INSIDE the padding (no coloured margin)
Text("A", Modifier.padding(16.dp).background(Color.Yellow))

// Background then padding: the colour fills the padded area too
Text("B", Modifier.background(Color.Yellow).padding(16.dp))
```

In `A`, padding is applied first, so the yellow covers only the text. In `B` the yellow is applied first and the padding then grows *inside* it, so the yellow includes the space. *(my reading of the docs' principle; check it in the preview)*

Weight

(docs) The official example gives one child twice the weight of another so it receives twice the width:

```
Row(Modifier.fillMaxWidth()) {
    Text("One third", Modifier.weight(1f))
    Text("Two thirds", Modifier.weight(2f))
}
```

Arrangement and Alignment

(docs) **Arrangement** controls how children are distributed along the layout's main axis (horizontal for a Row, vertical for a Column); **Alignment** controls the cross axis.

```
Row(
    modifier = Modifier.fillMaxWidth(),
    horizontalArrangement = Arrangement.SpaceBetween,
    verticalAlignment = Alignment.CenterVertically
) {
    Text("Left")
    Text("Right")
}

Column(
    modifier = Modifier.fillMaxWidth(),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.spacedBy(8.dp)
) {
    Text("One")
    Text("Two")
}
```

LAYOUT	MAIN AXIS	CROSS AXIS
`Row`	`horizontalArrangement`	`verticalAlignment`
`Column`	`verticalArrangement`	`horizontalAlignment`

`Arrangement.SpaceBetween`, `SpaceEvenly` and `spacedBy(8.dp)` are the ones you will use most *(general knowledge)*. Use `Spacer(Modifier.width(12.dp))` to put a fixed gap between two items, as in the profile row above *(docs: the tutorial does this)*.

Material 3 Components

(general knowledge, not found by my search) The generated project uses **Material 3**, whose components (`Text`, `Button`, `Card`, `Scaffold`) live in `androidx.compose.material3`. `Text` and `Button` you have already met. `Card` is a raised container with rounded corners; `Scaffold` (Chapter 1) is the screen skeleton. Text styles come from the theme: `MaterialTheme.typography.titleMedium` and the like (Chapter 8).

```
Card(modifier = Modifier.fillMaxWidth().padding(16.dp)) {
    Column(Modifier.padding(16.dp)) {
        Text("Card title", style = MaterialTheme.typography.titleMedium)
```

```
            Text("Some text inside a Material 3 card.")
        }
    }
```

Compared with the XML Way

	VIEWS AND XML (ANDROID DEVELOPMENT FUNDAMENTALS)	COMPOSE
Layout containers	`LinearLayout`, `FrameLayout`, `ConstraintLayout`	`Row`, `Column`, `Box` (and `ConstraintLayout` as a library for complex cases)
Sizes and spacing	Attributes such as `layout_width`, `padding`	Modifiers: `fillMaxWidth()`, `padding()`
Reusing a piece of UI	An `<include>` or a custom view	A function you call
Nesting depth	Deep nesting was a performance concern	Nesting is normal and cheap *(general knowledge)*

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 3 teaches `LinearLayout`, `ConstraintLayout`, common Views and view binding in XML. Here the same jobs are done with Row, Column, Box and modifiers; view binding is not needed, because there are no views to look up. The old Layout Editor's job is done by the preview (Android Studio 2: The New UI 5).

Try It Yourself

EXERCISE 1

Build a `ProfileCard(name, role)` that shows a coloured circle on the left and the two lines of text on the right, inside a Material 3 `Card`. Add a `@Preview` and check it.

View solution

EXERCISE 2

Make a row of three buttons spread evenly across the screen width. Then change it so the first button is pushed to the left and the other two to the right, and say which Arrangement you used for each.

View solution

EXERCISE 3

Use a `Box` to draw a grey square with a small red circular badge showing "3" in its top-right corner. Then swap the order of two modifiers on the badge and describe what changes in the preview.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): the roles of Row, Column and Box; that modifier order matters; `fillMaxWidth`, `size`, `padding`, `weight` (the 2:1 example), `Arrangement` and `Alignment` as main and cross axis; `Spacer` between items.
Not found in the docs search: Material 3 `Card` and `Scaffold`; the `background`, `clickable` modifiers and the XML comparison column (general knowledge).
Not verified: the padding-then-background explanation (my reading, check it in the preview); the imports, which may need adjusting; every listing and solution (unrun); the exercises (not run by me). Nothing in this chapter was checked on your install.
Sources: Compose layout basics (developer.android.com), Compose modifiers (developer.android.com), Constraints and modifier order (developer.android.com), Android Compose Tutorial (developer.android.com).

COMING UP

Chapter 4 covers state and recomposition: `remember`, `mutableStateOf`, state hoisting and why the UI redraws.

Android Development Fundamentals with Compose

Course 4 · Chapter 4 · Why the Screen Changes

This is the most important idea in Compose. In XML you change the screen by finding a view and telling it what to show. In Compose you never touch the UI directly: you change **state** (a value your composables read), and Compose **recomposes**, that is, calls the affected composables again so the screen matches the new value. Master this and everything else in Compose follows from it.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** `remember` and `mutableStateOf`, recomposition triggered by writing state, why `remember` is needed, `rememberSaveable` and configuration changes, and state hoisting with stateless and stateful composables. **Thin in the docs I found:** `mutableStateListOf` (only a performance note). **General knowledge, not checked:** the details of how the compiler tracks reads, and the list example. **All code is unrun** (no Kotlin compiler on the machine used to write this); the import lines are given so you can check them. Nothing was seen on your install.

State, and the Bug Without It

Here is the mistake almost every beginner makes. The variable changes when you tap, but the screen does not, because nothing told Compose to look:

```
@Composable
fun BrokenCounter() {
    var count = 0                       // plain variable: Compose is not watching it
    Button(onClick = { count++ }) {
        Text("Clicked $count times")    // never redrawn, and reset if redrawn for anoth
    }
}
```

(docs) A local `mutableStateOf` without `remember` is recreated on every pass, because recomposition can happen at any time and calls the composable again, resetting the value. The

fix has two parts: `mutableStateOf` makes a value Compose can watch, and `remember` keeps it between recompositions.

```
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableIntStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
```

```
@Composable
fun Counter() {
    var count by remember { mutableIntStateOf(0) }   // state Compose watches and keeps
    Button(onClick = { count++ }) {
        Text("Clicked $count times")
    }
}
```

PIECE	WHAT IT DOES
`mutableIntStateOf(0)`	Creates an observable `Int` holder starting at 0 *(general knowledge: the primitive version of `mutableStateOf`)*.
`remember { ... }`	*(docs)* Stores the object in memory in the composition and forgets it when the composable leaves the screen.
`by` (with `getValue`, `setValue`)	Lets you write `count` and `count++` instead of `count.value`.
`count++`	*(docs)* Writing to the state schedules recomposition of every composable that read it.

What Recomposition Is

When state changes, Compose calls again the composables that *read* it (*docs*), and only those *(general knowledge)*. Your composable is therefore a function from state to UI: same state, same screen. Two consequences:

- A composable may run many times, and at any time, so keep it free of slow work and surprises. Chapters 5 onward show where events and side effects belong.
- `remember` lasts while the composable stays on screen. To keep a value across rotation you use `rememberSaveable` (*docs*: it saves the state in a Bundle), as in Chapter 2.

State Hoisting: Stateless and Stateful

(docs) **State hoisting** means moving state up to the caller so the composable below becomes **stateless**: it receives a value and a callback instead of owning the state. The shape is always: value down, events up.

```
// Stateless: gets the value and an event callback; owns nothing
@Composable
fun CounterButton(count: Int, onIncrement: () -> Unit) {
    Button(onClick = onIncrement) {
        Text("Clicked $count times")
    }
}

// Stateful: owns the state and passes it down
@Composable
fun CounterScreen() {
    var count by remember { mutableIntStateOf(0) }
    CounterButton(count = count, onIncrement = { count++ })
}
```

	STATELESS	STATEFUL
Owns the state?	No: it receives the value and a callback	Yes: it holds `remember`ed state
Reuse	Easy: any caller can supply the state	Harder
Preview and test	Easy: pass any value	Awkward
Typical home	Most of your composables	A screen or a few top-level pieces

Prefer stateless composables and keep the state in as few places as possible *(general knowledge, and the usual Compose advice)*. The two counters in Exercise 2 show why: when one parent owns both values it can also show their total.

Lists in State

(general knowledge; the docs I found said little about it) To change a list on screen you need a list Compose can watch. `mutableStateListOf` gives one whose changes (add, remove) trigger recomposition. The one source on it that I found notes that it tracks each element's changes individually, so it has a cost; Chapter 6 returns to lists and `LazyColumn`.

```
@Composable
fun ShoppingList() {
    val items = remember { mutableStateListOf("Milk", "Eggs") }

    Column {
        Button(onClick = { items.add("Item ${items.size + 1}") }) {
            Text("Add item")
        }
        items.forEach { item ->
            Text(item)
        }
    }
}
```

Compared with the XML Way

	VIEWS (ANDROID DEVELOPMENT FUNDAMENTALS)	COMPOSE
Update the screen	Find the view and call `setText`, `setVisibility`	Change state; the UI is recomposed
Source of truth	Spread between views and variables	The state; the UI is derived from it
Forgetting to update a view	A common bug: the screen shows old data	Not possible for state-driven UI; the common bug is a plain variable that is not state
Keep across rotation	`onSaveInstanceState`	`rememberSaveable`

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals has no chapter like this one, because with Views you changed the screen by hand. State is the main new idea, and the next chapters (input, lists, navigation) all rely on it.

Try It Yourself

EXERCISE 1

Type in `BrokenCounter` and run it. Describe what happens when you tap. Then fix it with `remember` and `mutableIntStateOf`, and explain in a sentence why the first version failed.

View solution

EXERCISE 2

Using the stateless `CounterButton`, build `TwoCounters`: two independent counters in one parent that also shows their total. Say which composable owns the state.

View solution

EXERCISE 3

Make a `Switch` that shows the text "On" or "Off", keeping its value across rotation. Then switch `rememberSaveable` to `remember` and rotate the device to see the difference.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): `remember` and `mutableStateOf`; that writing state schedules recomposition; that `remember` forgets when the composable leaves the composition; `rememberSaveable` and Bundles; state hoisting with stateless and stateful composables; the note that `mutableStateListOf` tracks elements individually.
Not verified: that only affected composables rerun; how `mutableStateListOf` is meant to be used (not found); `mutableIntStateOf` as the primitive form (general knowledge); the "prefer stateless" advice (general knowledge); the XML comparison column; the imports; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: State and Jetpack Compose (developer.android.com), State in Jetpack Compose (developer.android.com codelab), Intro to state in Compose (developer.android.com codelab).

COMING UP

Chapter 5 covers user interaction: click handlers, `TextField` input, simple gestures, Intents and passing data to another screen. It builds directly on the state you met here.

Android Development Fundamentals with Compose

Course 4 · Chapter 5 · Taps, Typing and Other Screens

A screen the user cannot touch is a poster. This chapter adds the three kinds of interaction every app needs: reacting to **taps**, taking **typed input**, and moving to another screen or app with an **Intent**. All of it follows the pattern from Chapter 4: an event changes state, and the UI is recomposed from the state.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** the `TextField` pattern (a `value` plus `onValueChange` that updates a `remember`ed state), getting a context with `LocalContext.current` and calling `startActivity`, and `putExtra` for extra data. **Not found in the docs search:** `Button` and `onClick`, `clickable`, `Checkbox`, and the difference between explicit and implicit Intents (all described from general knowledge). **Newer text API:** the docs show a newer, state-based way to hold text-field state (`TextFieldState`) arriving and the `value` and `onValueChange` versions of the basic text field being deprecated in an alpha Compose UI release; I have not checked which API your project uses. **All code is unrun** (no Kotlin compiler on the machine used to write this). Nothing was seen on your install.

Reacting to Taps

Compose has no `setOnClickListener`. You pass a lambda when you create the composable. `Button` takes an `onClick`; any other composable can be made tappable with the `clickable` modifier *(general knowledge)*.

```
// A Button has onClick built in
Button(onClick = { /* do something */ }) {
    Text("Press me")
}

// Any composable can be made tappable with the clickable modifier
Text(
    text = "Tap this text",
    modifier = Modifier.clickable { /* do something */ }
)
```

The lambda usually changes state, as `count++` did in Chapter 4. The same pattern covers `Checkbox` and `Switch`: they receive a `checked` value and an `onCheckedChange` callback, exactly like a stateless composable (*general knowledge*).

```
var agreed by remember { mutableStateOf(false) }

Row(verticalAlignment = Alignment.CenterVertically) {
    Checkbox(checked = agreed, onCheckedChange = { agreed = it })
    Text("I agree")
}
```

Typed Input with TextField

(docs) A `TextField` shows the `value` you give it and reports a proposed new value through `onValueChange`. It does *not* change the text by itself: you store the new value in state, which recomposes the field with it. That is state hoisting (Chapter 4) in its most common form.

```
import android.content.Intent
import android.net.Uri
import androidx.compose.foundation.clickable
import androidx.compose.material3.Button
import androidx.compose.material3.Checkbox
import androidx.compose.material3.OutlinedTextField
import androidx.compose.material3.Text
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.runtime.setValue
import androidx.compose.ui.platform.LocalContext
```

```
@Composable
fun NameInput() {
    var name by rememberSaveable { mutableStateOf("") }

    Column(modifier = Modifier.padding(16.dp)) {
        OutlinedTextField(
            value = name,
            onValueChange = { name = it },
```

```
            label = { Text("Your name") }
        )
        Text(if (name.isBlank()) "Type your name" else "Hello, $name!")
    }
}
```

- `OutlinedTextField` is the Material 3 variant with an outline, and `TextField` the filled one *(general knowledge)*.
- If you leave out `name = it`, typing does nothing: the field redraws with the old value. This is the commonest beginner puzzle.
- `rememberSaveable` keeps the typed text across rotation (Chapter 2).
- The docs describe a newer `TextFieldState`-based text field replacing the `value` and `onValueChange` style over time. If Android Studio marks the code above as deprecated, look up the current version in the docs.

Other Gestures

Taps are the 95% case. Compose has further modifiers and APIs for dragging, scrolling, long presses and swiping, which are not covered in this course *(general knowledge; I did not research them)*.

Intents: Moving to Another Screen or App

An **Intent** is a message asking Android to start something. *(general knowledge, not found by my search)* Two kinds:

KIND	WHAT YOU NAME	EXAMPLE
Explicit	The exact Activity class to start	Open your own `DetailActivity`
Implicit	An *action* and let Android find an app that can do it	Open a web address, share text, dial a number

(docs) In Compose you get the context with `LocalContext.current` and call `startActivity()` on it, and you add extra data with `putExtra()`. An implicit example (the docs' sharing guide uses an action in the same way):

```
@Composable
fun OpenWebsiteButton() {
    val context = LocalContext.current
    Button(onClick = {
        val intent = Intent(Intent.ACTION_VIEW, Uri.parse("https://developer.android.com
        context.startActivity(intent)
    }) {
        Text("Open website")
    }
}
```

Passing Data to Another Activity

An explicit Intent names the class; `putExtra` attaches a value under a key, and the second Activity reads it from its own `intent`. The new Activity must also be declared in `AndroidManifest.xml` *(general knowledge; Android Studio's New Activity wizard normally adds it, which I have not checked)*.

```
// In the first screen
val context = LocalContext.current
Button(onClick = {
    val intent = Intent(context, DetailActivity::class.java).putExtra("name", name)
    context.startActivity(intent)
}) {
    Text("Show details")
}

// The second Activity
class DetailActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val name = intent.getStringExtra("name") ?: "nobody"
        setContent {
            ComposeHelloTheme {
                Text("Hello, $name!", modifier = Modifier.padding(16.dp))
            }
        }
    }
}
```

```
<!-- AndroidManifest.xml: inside <application> -->
<activity android:name=".DetailActivity" />
```

INTENTS OR NAVIGATION?

Opening a second *Activity* is one way to show another screen, but in a Compose app you normally keep one Activity and switch between composable screens with Navigation Compose (Chapter 7). Intents remain the right tool for reaching *outside* your app (a browser, a dialler, a share sheet) and for the system starting your app. Treat the second Activity here as an exercise in how Intents carry data.

Compared with the XML Way

	VIEWS (ANDROID DEVELOPMENT FUNDAMENTALS)	COMPOSE
React to a tap	`button.setOnClickListener { }` after `findViewById` or view binding	`onClick = { }` or `Modifier.clickable { }`
Read typed text	`editText.text.toString()`	The state that `onValueChange` updates
Start another Activity	`startActivity(Intent(...))`	The same, via `LocalContext.current`
Read extras	`intent.getStringExtra(...)`	The same

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 4 teaches `onClick`, event listeners and Intents. The Intent half is unchanged (same classes, same extras); the click and input half becomes lambdas and state.

Try It Yourself

EXERCISE 1

Build the `NameInput` screen: a text field and a line that greets the typed name. Then deliberately remove `name = it` and describe what happens when you type.

View solution

EXERCISE 2

Add a `Checkbox` labelled "I agree" and a `Button` that is enabled only when the box is ticked. (Hint: `Button` has an `enabled` parameter.)

View solution

EXERCISE 3

Add a button that opens a web page with an implicit Intent. Then create a `DetailActivity`, declare it in the manifest, and send it the typed name from Exercise 1 with an explicit Intent and `putExtra`.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): the `value` plus `onValueChange` pattern for `TextField`; `LocalContext.current` with `startActivity()`; `putExtra()`; that a state-based text field API (`TextFieldState`) is arriving and the older `value` and `onValueChange` basic text field overloads are deprecated in an alpha release.
Not found in the docs search: `Button` and `onClick`, `clickable`, `Checkbox` and `Switch` parameters, `OutlinedTextField`, the explicit and implicit Intent distinction, the manifest entry and what the New Activity wizard does (all general knowledge).
Not verified: whether `TextField` with `value` and `onValueChange` is flagged as deprecated in your Compose version; the imports; the `Uri.parse` call; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.
Sources: State and Jetpack Compose (developer.android.com), Send data to other apps (developer.android.com), Migrate to state-based text fields (developer.android.com), TextField API reference (developer.android.com).

COMING UP

Chapter 6 covers lists: `LazyColumn`, items and keys, item composables, click handling in a list and `LazyRow`, the Compose replacement for RecyclerView.

Android Development Fundamentals with Compose

Course 4 · Chapter 6 · Showing a List of Things

Almost every app shows a list: contacts, messages, products, items in a catalogue. In the Views world that meant a RecyclerView with an adapter, a ViewHolder and a DiffUtil. In Compose it is a `LazyColumn` and a lambda that says how to draw one row. This chapter shows that, then how to give rows stable identity with keys, how to react to a tap on a row, and how to scroll sideways with `LazyRow`.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** `contentPadding` (padding on the content, not the list), `Arrangement.spacedBy` for gaps, `key` in `items()` to keep per-item state when items move, and `LazyListState` (scroll position and scrolling methods). **Not found in the docs search:** click handling on rows (the `clickable` approach is general knowledge), `LazyRow`'s own parameters beyond the horizontal arrangement note, and the full `items` and `itemsIndexed` signatures. **All code is unrun** (no Kotlin compiler on the machine used to write this). Nothing was seen on your install.

A First LazyColumn

(docs) A lazy list only creates the rows that are on screen (or about to be), which is why a list of ten thousand items is not a problem. You describe the rows inside the `LazyColumn` block with `items`. A `Column` with a `forEach` would build every row at once and cannot scroll by itself *(general knowledge)*.

```
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Card
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
```

```
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
```

```
@Composable
fun NameList(names: List<String>) {
    LazyColumn(
        contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(names) { name ->
            Card(modifier = Modifier.fillMaxWidth()) {
                Text(text = name, modifier = Modifier.padding(16.dp))
            }
        }
    }
}

@Preview(showBackground = true)
@Composable
fun NameListPreview() {
    ComposeHelloTheme {
        NameList(listOf("Ada", "Grace", "Alan", "Katherine"))
    }
}
```

PART	WHAT IT DOES
`items(names) { name -> ... }`	Calls the lambda once per element, to describe that row.
`contentPadding`	*(docs)* Padding applied to the content rather than the list container, so rows scroll past the edges instead of being clipped.
`Arrangement.spacedBy(8.dp)`	*(docs)* A gap between rows. A `LazyRow` uses `horizontalArrangement` the same way.

Keys

(docs) Without a key, an item is identified by its position, so when rows move, any state remembered inside them goes to the wrong row. Pass a stable ID (not the position) through the

`key` lambda. The foundation release notes add that Compose can keep the scroll position tied to a key when items are added or removed above the visible one. In the example below the ID is the contact's `id`.

Rows, Selection and Taps

(general knowledge, not found by my search) Write one row as its own composable. Make it **stateless** (Chapter 4): it receives the data, whether it is selected and an `onClick`, and the screen owns the state. Make the row tappable with `Modifier.clickable`.

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.runtime.setValue
```

```
data class Contact(val id: Int, val name: String)

@Composable
fun ContactRow(contact: Contact, selected: Boolean, onClick: () -> Unit) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .clickable(onClick = onClick)
    ) {
        Text(
            text = if (selected) "${contact.name} (selected)" else contact.name,
            modifier = Modifier.padding(16.dp)
        )
    }
}

@Composable
fun ContactListScreen(contacts: List<Contact>) {
    var selectedId by rememberSaveable { mutableStateOf<Int?>(null) }

    LazyColumn(
        contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(contacts, key = { it.id }) { contact ->
```

```
            ContactRow(
                contact = contact,
                selected = contact.id == selectedId,
                onClick = { selectedId = contact.id }
            )
        }
    }
}
```

Tapping a row sets `selectedId`; the list recomposes and the chosen row shows "(selected)". The row never owns the selection, so it is easy to preview and reuse.

Changing the List

A list you add to or remove from must be state, or the UI will not follow. `mutableStateListOf` (Chapter 4) gives that. Keys matter here: with keys, removing the second row does not disturb the others.

```
@Composable
fun EditableList() {
    val people = remember {
        mutableStateListOf(Contact(1, "Ada"), Contact(2, "Grace"))
    }
    var nextId by remember { mutableIntStateOf(3) }

    Column {
        Button(onClick = { people.add(Contact(nextId, "Person $nextId")); nextId++ }) {
            Text("Add")
        }
        LazyColumn {
            items(people, key = { it.id }) { person ->
                Row(verticalAlignment = Alignment.CenterVertically) {
                    Text(person.name, modifier = Modifier.weight(1f).padding(8.dp))
                    Button(onClick = { people.remove(person) }) { Text("Remove") }
                }
            }
        }
    }
}
```

LazyRow and List State

```
@Composable
fun TagRow(tags: List<String>) {
    LazyRow(
        contentPadding = PaddingValues(horizontal = 16.dp),
        horizontalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(tags) { tag ->
            Card { Text(text = tag, modifier = Modifier.padding(12.dp)) }
        }
    }
}
```

(docs) `LazyListState` holds the scroll position, exposes `firstVisibleItemIndex` and `firstVisibleItemScrollOffset`, and has methods such as `animateScrollToItem` (run inside a coroutine) for scrolling by code. Create it with `rememberLazyListState()` *(general knowledge for that function name)*.

```
val listState = rememberLazyListState()
LazyColumn(state = listState) { /* items */ }

// Read where the user has scrolled to
val firstVisible = listState.firstVisibleItemIndex
```

A COMMON CRASH

(general knowledge, not checked) Do not put a `LazyColumn` inside a `Column` that scrolls vertically itself (`Modifier.verticalScroll`): a scrolling parent gives its child unlimited height, and Compose reports an error. Make the list the scrolling element, and put other content above it as a normal child or as extra items.

Compared with RecyclerView

RECYCLERVIEW (ANDROID DEVELOPMENT FUNDAMENTALS)	COMPOSE
An Adapter supplies the data	You pass a list to `items(...)`
A ViewHolder holds one row's views	A composable function draws one row; there are no views to hold
An item layout XML file	The lambda, or a row composable
DiffUtil finds what changed	Recomposition plus `key`
An item click listener set in the adapter	An `onClick` lambda passed to the row
LayoutManager (linear, grid)	`LazyColumn`, `LazyRow` and lazy grids *(grids not covered here)*

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 5 teaches the adapter pattern, ViewHolder, DiffUtil and click handling in RecyclerView. For a Compose app none of that machinery is needed: one composable per row, a key for identity, and state for changes. RecyclerView is still used in older apps (Chapter 9 covers working beside it).

Try It Yourself

EXERCISE 1

Show a list of 30 names in cards using `LazyColumn` (generate them with `List(30) { "Person $it" }`). Add content padding and spacing, and add a preview.

View solution

EXERCISE 2

Using the `Contact` data class, build `ContactListScreen` so tapping a row selects it. Then add a "Clear selection" button above the list. Which composable owns the selection?

View solution

EXERCISE 3

Build `EditableList`: a button that adds a person and a Remove button on each row. Run it with and without the `key` argument and describe any difference you can see. Then add a `LazyRow` of tags above the list.

View solution

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): `contentPadding` applying to content; `Arrangement.spacedBy` and `horizontalArrangement` for `LazyRow`; `key` in `items()`; scroll position kept by key; `LazyListState`, `firstVisibleItemIndex`, `firstVisibleItemScrollOffset`, `animateScrollToItem`.

Not found in the docs search: row click handling, `rememberLazyListState` by name, the scrolling-parent crash and the "Column builds everything" remark (general knowledge); the RecyclerView comparison column.

Not verified: the imports; whether keyed and unkeyed lists differ visibly in Exercise 3 (the docs say state follows the key; what you will actually see is for you to report); every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Lazy lists and lazy grids (developer.android.com), LazyListScope API reference (developer.android.com), Compose Foundation release notes (developer.android.com).

COMING UP

Chapter 7 covers Navigation Compose: `NavHost`, routes, arguments, the back stack, bottom navigation and deep links. It is how a tapped row opens a detail screen.

Android Development Fundamentals with Compose

Course 4 · Chapter 7 · Moving Between Screens

A real app has several screens. In a Compose app you usually keep **one Activity** and swap which composable fills it. The **Navigation Compose** library does the swapping: a `NavController` remembers where the user has been, and a `NavHost` says which composable belongs to which **route**. This chapter builds a two-screen app with an argument, then covers the back stack, bottom navigation and deep links.

WHAT WAS CHECKED, AND HOW: READ THIS FIRST

Checked in the docs (two web searches, summaries only): `rememberNavController()`, the `NavHost` and `composable` destinations, string routes with `navArgument`, screens taking `() -> Unit` callbacks that the `NavHost` wires to `navigate()`, `popBackStack()` (including its Boolean result), `popUpTo` with `inclusive`, `saveState` and `restoreState`, typed routes (`@Serializable`, `toRoute()`, version 2.8 and later), and **Navigation 3**. **Not found in the docs searches:** the exact dependency version, `NavigationBar` and `currentBackStackEntryAsState`, `launchSingleTop`, and deep-link setup (general knowledge). **All code is unrun** (no Kotlin compiler on the machine used to write this). Nothing was seen on your install.

THERE IS A NEWER NAVIGATION LIBRARY

(docs) Google now also documents **Navigation 3**, in which you own the back stack as a list of keys (adding to it to go forward, removing the last to go back) and a `NavDisplay` draws the matching screens. This chapter teaches **Navigation Compose** (`NavHost`), because it is what most existing Compose apps use and what the course outline set. For a brand-new app, check the current docs and whether Google recommends Navigation 3; I have not compared them in depth. If you want this chapter, or a later one, rewritten for Navigation 3, say so.

Add the Dependency

The Empty Activity template does not include Navigation Compose. Add it through the version catalog (Android Studio 2: The New UI 4 explains the catalog), then sync. Use the version Android Studio or the release notes suggest; I have not pinned one here.

```
# gradle/libs.versions.toml (add; use the version Android Studio suggests for navigation
[versions]
navigationCompose = "<current version>"

[libraries]
androidx-navigation-compose = { group = "androidx.navigation", name = "navigation-compos
```

```
// app/build.gradle.kts: inside dependencies { }
implementation(libs.androidx.navigation.compose)
```

Two Screens and an Argument

(docs) The recommended shape is that each screen is a composable taking plain callbacks, and the `NavHost` supplies lambdas that call `navigate()`. A screen therefore never touches the controller, which keeps it easy to preview and test (Chapter 4's stateless idea again).

```
@Composable
fun HomeScreen(onOpenDetail: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text("Home")
        Button(onClick = { onOpenDetail("Ada") }) { Text("Open Ada") }
        Button(onClick = { onOpenDetail("Grace") }) { Text("Open Grace") }
    }
}

@Composable
fun DetailScreen(name: String, onBack: () -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text("Details for $name")
        Button(onClick = onBack) { Text("Back") }
    }
}
```

```
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.Text
```

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.navigation.NavType
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
import androidx.navigation.navArgument
```

```
@Composable
fun AppNavigation() {
    val navController = rememberNavController()

    NavHost(navController = navController, startDestination = "home") {
        composable("home") {
            HomeScreen(onOpenDetail = { name -> navController.navigate("detail/$name") }
        }
        composable(
            route = "detail/{name}",
            arguments = listOf(navArgument("name") { type = NavType.StringType })
        ) { backStackEntry ->
            val name = backStackEntry.arguments?.getString("name") ?: "nobody"
            DetailScreen(name = name, onBack = { navController.popBackStack() })
        }
    }
}

// In MainActivity: setContent { ComposeHelloTheme { AppNavigation() } }
```

PART	WHAT IT DOES
`rememberNavController()`	*(docs)* Gives you the `NavHostController`, which moves between screens.
`NavHost(startDestination = ...)`	Holds the navigation graph and shows the current destination.
`composable("home") { }`	Registers a route and the composable to show for it.
`"detail/{name}"`	*(docs)* A route with an argument placeholder, declared with `navArgument`.
`navController.navigate("detail/$name")`	Goes to that route, pushing it on the back stack.

PART	WHAT IT DOES
`navController.popBackStack()`	*(docs)* Goes back; returns a Boolean, false when there was nothing to pop (for example the start destination was already removed).

The Back Stack

`navigate` pushes a screen; the system Back button, or `popBackStack()`, removes the top one. *(docs)* To clear part of the stack while navigating, use `popUpTo`; its `inclusive` flag decides whether the named destination is removed too.

```
// Go to home and remove everything above (and including) the old home, so Back does not
navController.navigate("home") {
    popUpTo("home") { inclusive = true }
}
```

Bottom Navigation

A bar of tabs along the bottom is a `NavigationBar` in a `Scaffold`. *(docs)* The documented pattern for keeping each tab's place uses `popUpTo` with `saveState`, and `restoreState` on the way back; the rest is general knowledge.

```
import androidx.compose.material3.NavigationBar
import androidx.compose.material3.NavigationBarItem
import androidx.compose.material3.Scaffold
import androidx.compose.runtime.getValue
import androidx.navigation.compose.currentBackStackEntryAsState
```

```
@Composable
fun MainScreen() {
    val navController = rememberNavController()
    val tabs = listOf("home", "search", "settings")
    val backStackEntry by navController.currentBackStackEntryAsState()
    val current = backStackEntry?.destination?.route

    Scaffold(
```

```
        bottomBar = {
            NavigationBar {
                tabs.forEach { route ->
                    NavigationBarItem(
                        selected = current == route,
                        onClick = {
                            navController.navigate(route) {
                                popUpTo(navController.graph.startDestinationId) { saveSt
                                launchSingleTop = true
                                restoreState = true
                            }
                        },
                        icon = { },
                        label = { Text(route) }
                    )
                }
            }
        }
    ) { innerPadding ->
        NavHost(navController, startDestination = "home", modifier = Modifier.padding(in
            composable("home") { Text("Home tab") }
            composable("search") { Text("Search tab") }
            composable("settings") { Text("Settings tab") }
        }
    }
}
```

Deep Links

(general knowledge; not found by my search) A deep link lets a web address or notification open one specific screen. You add a `navDeepLink` to the destination and a matching intent filter to the Activity in the manifest. This chapter does not build one; treat the snippet as a pointer, and read the official deep-link guide before using it.

```
composable(
    route = "detail/{name}",
    deepLinks = listOf(navDeepLink { uriPattern = "https://example.com/detail/{name}" })
) { /* ... */ }
```

```
<!-- AndroidManifest.xml: the Activity needs a matching intent filter for that web addres
```

Newer Ways to Write Routes

- **Typed routes** *(docs)*: instead of strings, routes are `@Serializable` classes or objects, you call `navigate` with an instance, and a destination reads its arguments with `toRoute()`. The docs say the typed overloads arrived in versions 2.8 and 2.9, so check your library version. They need the Kotlin serialization plugin, so they are not used in this chapter.
- **Navigation 3** *(docs)*: see the box above.

Compared with the XML Way

	NAVIGATION COMPONENT WITH VIEWS (ANDROID DEVELOPMENT FUNDAMENTALS)	NAVIGATION COMPOSE
The graph	A navigation XML file edited visually	Code inside `NavHost { }`
Destinations	Fragments	Composables
Passing data	SafeArgs generated classes	Route arguments (or typed routes)
Going back	`popBackStack()`	`popBackStack()` (the same name)
Deep links	In the graph XML	`navDeepLink` in code

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 7 teaches the Navigation Component with a nav graph, SafeArgs, deep links and back stack management, around Fragments. The ideas (a graph of destinations, a back stack, arguments) carry over; the graph moves into Kotlin code and Fragments disappear.

Try It Yourself

EXERCISE 1

Add the dependency, then build the two-screen app: Home with two buttons, and a Detail screen that shows the name passed in the route. Check that the Back button returns to Home.

View solution

EXERCISE 2

Add a third screen, "Done", reached from Detail. From Done, add a button that returns to Home and clears the stack so that pressing Back leaves the app instead of returning to Detail.

View solution

EXERCISE 3

Build a bottom navigation bar with three tabs. Open a tab, scroll or change something on it, switch tabs and come back. Say whether the tab remembers its state, and which flags you used.

View solution

NOT YET VERIFIED

Checked in the docs (two web searches, summaries only): `rememberNavController`; `NavHost` and `composable` destinations; string routes with `navArgument`; the callback pattern; `popBackStack` and its Boolean; `popUpTo` with `inclusive`; `saveState` and `restoreState`; typed routes from 2.8; Navigation 3's `NavDisplay` and key list.

Not found in the docs searches: the dependency's current version and artifact details beyond its name; `NavigationBar`, `NavigationBarItem`, `currentBackStackEntryAsState`, `launchSingleTop`; deep-link setup; the XML comparison column (general knowledge).

Not verified: whether Navigation 3 is now the recommended choice for new apps (not compared); the imports; whether the bottom-bar code and the pop-up flags behave as described; every listing and solution (unrun); the exercises (not run by me). Nothing was checked on your install.

Sources: Navigation and the back stack (developer.android.com), Navigate between screens with Compose (developer.android.com codelab), androidx.navigation.compose (developer.android.com), Navigation 3 release notes (developer.android.com), Navigation release notes (developer.android.com).

COMING UP

Chapter 8 covers resources, themes and dark mode: string and colour resources, the Material 3 theme and typography, dark mode and dynamic colour.

Android Development Fundamentals with Compose

Course 4 · Chapter 8 · Words, Colours and Dark Mode

So far the text in our composables has been written straight into the code, and every colour has been whatever the theme gave us. Real apps do two things better. They keep user-visible text in **resources**, so it can be translated, and they keep colours, fonts and shapes in a **theme**, so the whole app can change look (including dark mode) in one place. You met the theme files in Chapter 1 (`ui/theme/`); this chapter opens them.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked on the install (earlier work):** that a generated project has `ui/theme/Color.kt`, `Theme.kt` and `Type.kt`, and a `res/values` folder. **Checked in the docs (two web searches, summaries only):** `stringResource`, `dimensionResource` and `colorResource`; `isSystemInDarkTheme()` choosing between light and dark colour schemes; dynamic colour on Android 12 and above with `dynamicDarkColorScheme` and `dynamicLightColorScheme`; `MaterialTheme` taking `colorScheme`, `typography` and `shapes`. **General knowledge, not checked:** translation folders such as `values-es`, formatted strings, and the exact contents of your generated theme files (the listings are a *shape*). **All code is unrun** (no Kotlin compiler on the machine used to write this).

Resources: Keep Text Out of the Code

The `res/values` folder holds XML files of simple values. The most important is `strings.xml`. *(docs)* In Compose you read a string resource with `stringResource`:

```
<!-- app/src/main/res/values/strings.xml -->
<resources>
    <string name="app_name">ComposeHello</string>
    <string name="greeting">Hello, %1$s!</string>
    <string name="open_details">Open details</string>
</resources>
```

```
Text(text = stringResource(R.string.greeting, name))
Button(onClick = onOpen) { Text(stringResource(R.string.open_details)) }
```

- `%1$s` is a placeholder for the first argument *(general knowledge)*; `stringResource(R.string.greeting, name)` fills it in.
- *(docs, with the localization point from general knowledge)* `stringResource` retrieves localized text. To add a translation, add a folder such as `values-es` with the same names and Android picks the right file for the user's language:

```
<!-- app/src/main/res/values-es/strings.xml (a Spanish translation) -->
<resources>
    <string name="app_name">ComposeHello</string>
    <string name="greeting">¡Hola, %1$s!</string>
    <string name="open_details">Abrir detalles</string>
</resources>
```

Resource Manager (the left-hand tool window bar, Android Studio 2: The New UI 1) lists and previews your resources *(docs describe it; I have not used it here)*.

Dimensions and Colours

(docs) `dimensionResource` reads a value from a dimension XML file, and `colorResource` reads a colour; the docs describe `colorResource` as the tool for adopting Compose step by step, and note that it flattens colour state lists. *(general knowledge)* In a pure Compose app most people write sizes as `16.dp` in code and take colours from the theme, so these two are less common than `stringResource`.

```
<!-- res/values/dimens.xml -->
<resources>
    <dimen name="screen_padding">16dp</dimen>
</resources>

// In a composable
Column(modifier = Modifier.padding(dimensionResource(R.dimen.screen_padding))) { /* ...
```

The Theme Files

The three files in `ui/theme/` work together: `Color.kt` defines raw colours, `Type.kt` defines the `Typography` (text styles), and `Theme.kt` turns them into a `MaterialTheme` that you wrap around the app in `setContent` (Chapter 1). Compare your generated files with these shapes:

```kotlin
// ui/theme/Color.kt (the shape of the generated file; the values are examples)
val Purple80 = Color(0xFFD0BCFF)
val Purple40 = Color(0xFF6650A4)
```

```kotlin
// ui/theme/Theme.kt (the shape of the generated file; yours may differ)
private val DarkColorScheme = darkColorScheme(primary = Purple80)
private val LightColorScheme = lightColorScheme(primary = Purple40)

@Composable
fun ComposeHelloTheme(
    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(
        }
        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }

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

PART	WHAT IT DOES
`darkTheme = isSystemInDarkTheme()`	*(docs)* Follows the phone's light or dark setting by default.
`lightColorScheme` / `darkColorScheme`	Build a Material 3 colour scheme from your colours *(general knowledge)*.
`dynamicColor`	*(docs)* Dynamic colour builds the palette from the wallpaper on Android 12 and above; otherwise your own schemes are used.

PART	WHAT IT DOES
`MaterialTheme(colorScheme, typography, ...)`	*(docs)* Makes the scheme, text styles and shapes available to everything inside.

Using the Theme in Your Composables

(docs) Read the current values from `MaterialTheme.colorScheme`, `MaterialTheme.typography` and `MaterialTheme.shapes`, rather than hard-coding a colour or size. The text and the card then change automatically when the theme does.

```
Text(
    text = "Headline",
    style = MaterialTheme.typography.headlineMedium,
    color = MaterialTheme.colorScheme.primary
)
Card(shape = MaterialTheme.shapes.medium) { /* ... */ }
```

Seeing Dark Mode

In the preview: `@PreviewLightDark` shows light and dark at once (Android Studio 2: The New UI 5).

```
@PreviewLightDark
@Composable
fun GreetingPreview() {
    ComposeHelloTheme {
        Greeting("Android")
    }
}
```

- **On a device:** switch the emulator or phone to dark theme in its system settings, and the running app follows it because of `isSystemInDarkTheme()` *(general knowledge: where the setting sits varies by Android version).*

> **HARD-CODED COLOURS BREAK DARK MODE**

A composable that sets `Color.White` or `Color.Black` directly ignores the theme, so text can vanish against the opposite background. Use `MaterialTheme.colorScheme` entries instead *(general knowledge)*.

Compared with the XML Way

	VIEWS (ANDROID DEVELOPMENT FUNDAMENTALS)	COMPOSE
Text in resources	`@string/...` in layouts, `getString` in code	`stringResource(R.string...)`
Colours and styles	`colors.xml`, `themes.xml`, `style` attributes	`Color.kt` and `MaterialTheme` in Kotlin
Dark mode	A `values-night` folder and a DayNight theme	A dark colour scheme chosen with `isSystemInDarkTheme()`
Reading a theme colour	`?attr/colorPrimary`	`MaterialTheme.colorScheme.primary`
Translations	`values-xx` folders	The same `values-xx` folders

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 8 covers string, colour and dimension resources, themes and dark mode basics in XML. Resource folders and translations are unchanged; styling moves into Kotlin theme files and `MaterialTheme`.

Try It Yourself

EXERCISE 1

Move the text of your Home and Detail screens (Chapter 7) into `strings.xml` and read it with `stringResource`, including one string with a placeholder. Then add a `values-es` translation (or another language you know) and switch the device language to see it.

View solution

EXERCISE 2

Open `Color.kt` and `Theme.kt` in your project and write down what they contain. Change the primary colour in both the light and dark schemes, turn `dynamicColor` off, and compare the preview before and after.

View solution

EXERCISE 3

Preview a screen with `@PreviewLightDark`. Deliberately hard-code `Color.Black` on a `Text` and describe what happens in the dark preview. Fix it with a theme colour.

View solution

NOT YET VERIFIED

Checked on the install (earlier work): the `ui/theme` files and the `res/values` folder exist in a generated project.
Checked in the docs (two web searches, summaries only): `stringResource`, `dimensionResource`, `colorResource` (and its flattening of colour state lists); `isSystemInDarkTheme()`; dynamic colour (Android 12+); `MaterialTheme` parameters and the `MaterialTheme.colorScheme`, `typography`, `shapes` accessors; Resource Manager.
General knowledge, not checked: `values-es` translations and formatted strings (`%1$s`); the contents of your generated theme files; `lightColorScheme` and `darkColorScheme` details; the XML comparison column; where the device's dark-mode setting sits.
Not verified: the imports; every listing and solution (unrun); the exercises (not run by me). Nothing in the app was checked on your install beyond the file list.
Sources: Resources in Compose (developer.android.com), Material Design 3 in Compose (developer.android.com), Theming in Compose with Material 3 (developer.android.com codelab), Manage your app's UI resources with Resource Manager (developer.android.com).

COMING UP

Chapter 9, the last in the course, is the legacy appendix: why XML, Views and Fragments still exist, how to read an XML layout, and how `ComposeView` and `AndroidView` let the two worlds sit side by side.

Android Development Fundamentals with Compose

Course 4 · Chapter 9 · XML, Views and Fragments: the Old World

Everything so far used Compose. But most Android apps in the world were written before Compose existed, and many tutorials, libraries and job tasks still use **XML layouts, Views and Fragments**. You do not need to write them for a new app, but you do need to *read* them, and often to make them live next to Compose code. This last chapter shows how the old pieces map to the new ones, and how to mix the two.

> **WHAT WAS CHECKED, AND HOW: READ THIS FIRST**
>
> **Checked in the docs (one web search, summaries only):** hosting Compose with `ComposeView` and `setContent()`, the lifecycle owner requirement, the fragment `DisposeOnViewTreeLifecycleDestroyed` advice, unique IDs for several `ComposeView`s, `AndroidViewBinding` for small legacy XML layouts inside Compose, and that Compose and Views can coexist during a migration. **Not found in the docs search:** how View-based XML layouts and Fragments themselves work (general knowledge from the older course), and the `AndroidView` function itself. **All code and XML are unrun** (no Kotlin compiler on the machine used to write this). Nothing here was seen on your install.

Why the Old Pieces Still Matter

- **Existing apps.** *(docs)* The guidance for existing apps is that Compose and Views coexist in a codebase until the app is fully in Compose: you migrate a screen or a widget at a time.
- **Libraries and tutorials.** Many are written for Views; you need to recognise them *(general knowledge)*.
- **Android Development Fundamentals** (the XML course, `android1`) remains on this site for exactly that reason.

Reading an XML Layout

(general knowledge, from the older course) A layout file describes a tree of views with **attributes**. Each `android:id="@+id/..."` names a view so code can find it; `layout_width` and `layout_height` are required on every view; `@string/...` points at a string resource (Chapter 8).

```
<!-- res/layout/activity_main.xml (a typical old-style layout) -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/greeting_title" />

    <Button
        android:id="@+id/open_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/open_details" />
</LinearLayout>
```

```
// The Views way: find views by id, then act on them
setContentView(R.layout.activity_main)
findViewById<Button>(R.id.open_button).setOnClickListener { /* open details */ }
```

The same screen in Compose (Chapters 3 and 8):

```
// The same screen in Compose
@Composable
fun MainScreen(onOpen: () -> Unit) {
    Column(modifier = Modifier.fillMaxSize().padding(16.dp)) {
        Text(stringResource(R.string.greeting_title))
        Button(onClick = onOpen) { Text(stringResource(R.string.open_details)) }
    }
}
```

XML VIEWS	COMPOSE
`LinearLayout` (vertical, horizontal)	`Column`, `Row`
`FrameLayout`	`Box`

XML VIEWS	COMPOSE
`ConstraintLayout`	Nested layouts, or the `ConstraintLayout` library for Compose
`TextView`, `Button`, `EditText`	`Text`, `Button`, `TextField`
`android:padding`, `layout_margin`	`Modifier.padding(...)`
`android:id` plus `findViewById` or view binding	No ids: the data is the source of truth (Chapter 4)
`RecyclerView` with an Adapter and ViewHolder	`LazyColumn` (Chapter 6)
A Fragment	A composable screen, with a `NavHost` destination (Chapter 7)

What a Fragment Was

(general knowledge) A Fragment is a reusable piece of a screen with its own lifecycle and its own layout, hosted inside an Activity; the Navigation Component swapped Fragments in and out of one Activity. In a Compose app that job is done by composable screens and a `NavHost`, with no Fragments at all. You still meet Fragments in older code, in some libraries, and whenever you add Compose to an app that already uses them.

Compose Inside a View Layout

(docs) To add Compose to an existing layout or Fragment, use `ComposeView` and call its `setContent()`. Put the `ComposeView` in the XML like any other view, find it by ID, and call `setContent`:

```
<!-- res/layout/activity_main.xml: a ComposeView sits in the layout like any other view
<androidx.compose.ui.platform.ComposeView
    android:id="@+id/compose_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

```
// In the Activity (a ComponentActivity or AppCompatActivity)
setContentView(R.layout.activity_main)
findViewById<ComposeView>(R.id.compose_view).setContent {
    ComposeHelloTheme {
        Greeting("from Compose")
    }
}
```

- *(docs)* The `ComposeView` must be attached to a view-tree lifecycle owner; `ComponentActivity`, `FragmentActivity` and `AppCompatActivity` all provide one.
- *(docs)* If a layout has more than one `ComposeView`, give each a unique ID, or `savedInstanceState` will not work properly.
- *(docs)* In a Fragment, always set the `DisposeOnViewTreeLifecycleDestroyed` composition strategy:

```
// In a Fragment's onCreateView
return ComposeView(requireContext()).apply {
    setViewCompositionStrategy(ViewCompositionStrategy.DisposeOnViewTreeLifecycleDestroy
    setContent {
        ComposeHelloTheme { Greeting("from a Fragment") }
    }
}
```

Views Inside Compose

(docs) The other direction is for pieces that have no Compose version yet or are too big to rewrite. `AndroidViewBinding` (from the `ui-viewbinding` library) embeds an XML layout, and the docs say to use it only for smaller legacy layouts during an incremental migration, not for whole screens in a Compose-only app. *(general knowledge)* For a single classic view there is `AndroidView`:

```
@Composable
fun ClassicTextView() {
    AndroidView(
        factory = { context ->
            TextView(context).apply { text = "A classic TextView inside Compose" }
```

```
            }
        )
    }
```

Which Should I Use?

SITUATION	CHOICE
A new app	Compose only
An old app you are adding a screen to	Compose for the new screen, hosted with `ComposeView` if it must live inside a Fragment or XML layout
A Compose app that needs a widget with no Compose version	`AndroidView`
Reading or fixing an old XML screen	Leave it as XML unless you are migrating it on purpose

COMPARED WITH THE CLASSIC UI COURSE

Android Development Fundamentals 3 (XML layouts and Views), 5 (RecyclerView) and 6 (Fragments) are the chapters this appendix summarises. They are still the right place to learn the old pieces in depth, and they remain accurate for View-based projects.

Try It Yourself

EXERCISE 1

Using the XML layout in this chapter, write its Compose equivalent from memory first, then compare with the chapter's version. List the XML attributes that have no direct counterpart and say why.

View solution

EXERCISE 2

Create a project with an Empty Views Activity template, add a `ComposeView` to its layout, and show a Compose greeting inside it. Report what the template generated and anything that needed fixing.

View solution

EXERCISE 3

In your Compose project, embed a classic `TextView` with `AndroidView`. Then write two sentences on when you would choose that over rewriting the widget in Compose.

View solution

COURSE COMPLETE

You have covered a first Compose app (1), the Activity lifecycle (2), layouts (3), state (4), interaction and Intents (5), lists (6), navigation (7), resources and themes (8) and how the old XML world fits beside it (9). Next in the Android track: Android Development with Compose: Architecture & Data (`android5`, not yet written), and Personal Catalogue: Android (Kotlin & Compose), a project course that applies all of this.

NOT YET VERIFIED

Checked in the docs (one web search, summaries only): `ComposeView` with `setContent()`; the view-tree lifecycle owner requirement; unique IDs; `DisposeOnViewTreeLifecycleDestroyed` in Fragments; `AndroidViewBinding` for small legacy XML layouts; Compose and Views coexisting in a migration.
Not found in the docs search: View-based XML layouts and Fragments themselves; `AndroidView` (general knowledge); the Views-to-Compose mapping table.
Not verified: the imports; the Empty Views Activity template's current output (Exercise 2); every listing, layout and solution (unrun); the exercises (not run by me). Nothing was checked on your install.
Sources: Using Compose in Views (developer.android.com), Using Views in Compose (developer.android.com), Migrating to Jetpack Compose (developer.android.com codelab).