Kotlin

Computer: An electronic device that processes data and performs tasks based on instructions.
Programming language: A set of rules and syntax used to write instructions that a computer can understand and execute.
Syntax: The set of rules that define how instructions must be written in a programming language.
Expression: A combination of values, variables, and operators that produces a result (e.g., 2 + 3).
Statement: A complete instruction in a program that performs an action (e.g., print("Hello")).
Algorithm: A step-by-step procedure to solve a problem or perform a task.
Compiler: A tool that translates the program into machine code before it runs.
Interpreter: A tool that translates and runs the program line by line, while it runs.
Module: A file or unit(variables, function, class, or piece of code) that can be reused.
Library: A collection of modules grouped together to provide useful tools and functions.
Package: A structured group of modules or libraries.
Dependencies: The external code, tools, or resources that a program need to work properly.
Software: A set of programs and data used to operate a computer or perform tasks.
Importing: Usually means bringing external code, files, or data into a program or system
Template: A template is a model or guide that serves as a starting point for producing something.

1. Getting Started with Kotlin

Kotlin is a static programming language that allows you to write concise and typed code.

Technical requirements

IDE: IntelliJ IDEA Community Edition
OpenJDK 11 or higher

Introduction to Kotlin

Kotlin is a language that runs on the **Java Vritual Machine** developed by JetBrains. It was developed to overcome the following challenges that Java had:

- **Verbosity**: Java has a very verbose syntax and this leads to developers writing a lot of boilerplate code even for trivial tasks.
- **Null pointer exception**: By default, Java allows variables to have null values. This normally results in null pointer exceptions.

- **Concurrency**: Java has threads, but managing concurrency and thread safety can be such a hard task at times. This leads to a lot of performance and memory issues that seriously affect applications that need to do work off the main thread.
- **Slow adoption of features**: The Java release cycle is slow and it is difficult to use the latest Java version to develop Android apps as there's a lot to be done to ensure backward compatibility. This means it's hard for Android developers to easily adopt the new language features and improvements as they're stuck using older versions.
- **Lack of function support**: Java is not a functional language, which makes it hard for developers to write functional code in Java. It's hard to employ features such as high-order functions or treat functions as first-class citizens.

Some of the features where Kotlin has and edge over Java are:

- **Conciseness**: The syntax is concise, which in turn reduces the amount of boilerplate code that you write.
- **Null safety**: Kotlin was designed with null safety in mind. Variables that can have null values are indicated when declaring them, and before using these variables, the Kotlin compiler enforces checks for nullability, thereby reducing the number of exceptions and crashes.
- **Coroutines support**: Kotlin has built-in support for Kotlin coroutines. Coroutines are lightweight threads that you can use to perform asynchronous operations.
- **Data classes**: Kotlin has a built-in data class that automatically generate `equals()`, `hashCode()`, and `toString()` methods, reducing the amount of boilerplate code required.
- **Extension functions**: Kotlin allows developers to add functions to existing classes without inheriting from them, through extension functions.
- **Smart casting**: Kotlin's smart casting system makes it possible to cast variables without the need for an explicit cast. The compiler automatically detects when a variable can be safely cast and performs the cast automatically.

Kotlin has evolved over the years to support:

- **Kotlin Multiplatform**: This is used to develop applications that target different platforms such as Android, iOS, and web applications.
- **Kotlin for server side**: This is used to write backend applications and a number of frameworks to support server-side development.
- **Kotlin for Android**: Google has supported Kotlin as a first-class language for Android development since 2017.
- **Kotlin for JavaScript**: This provides support for writing Kotlin code that is transpiled to compatible JavaScript libraries.

- **Kotlin/Native**: This compiles Kotlin code to native binaries and run without **Java Virtual Machine (JVM)**.
- **Kotlin for data science**: You can use Kotlin to build and explore data pipelines.

Kotlin syntax, types, functions and classes

Kotlin is a strongly typed language. The type of a variable is determined at the time of compilation. Kotlin has a rich type system that has the following types:

- **Nullable types**: Every type in Kotlin can either be nullable or non-nullable. Nullable types are denoted with a question mark operator – for example, `String?`
- **Basic types**: Int, Long, Boolean, Double, and Char.
- **Class types**: As Kotlin is an object-oriented programming language, it provides support for classes, sealed classes, interfaces, and so on. You define a class using the `class` keyword.
- **Arrays**: There is support for both primitive and object arrays. To declare a primitive array, you specify the type and size, as follows:

```kotlin
val shortArray = ShortArray(10)
```

The following shows how you define object arrays:

```kotlin
val recipes = arrayOf("Chicken Soup", "Beef Stew", "Tuna Casserole")
```

- **Collections**: Kotlin has a rich collection of APIs providing types such as sets, maps, and lists. They're designed to be concise and expressive, and the language offers a wide range of operations for sorting, filtering, mapping, and many more.
- **Enum types**: These are used to define a fixed set of options. Kotlin has the Enum keyword for you to declare enumerations.
- **Functional types**: Kotlin is a functional language as well, meaning functions are treated as first-class citizens. You can be able to assign functions to variables, return functions as values from functions, and pass functions as arguments to other functions. To define a function as a type, you use the (Boolean) -> Unit shorthand notation. This example takes a Boolean argument and returns a Unit value.

Creating functions

Function is a block of code that does a specific task. Function name should be in camel case and descriptive to indicate what the function is doing.

```kotlin
fun main() {
    println("Hellow World!")
}
```

Creating classes

To declare a class in Kotlin, we have the `class` keyword.

```kotlin
class Recipe {
    private val ingredients = mutableListOf<String>()

    fun addIngredient(name: String) {
        ingredients.add(name)
    }

    fun getIngredients(): List<String> {
        return ingredients
    }
}

fun main() {
    val recipe = Recipe()
    recipe.addIngredient("Rice")
    recipe.addIngredient("Chicken")
    println(recipe.getIngredients())
}
```

Kotlin features for Android developers

Here are some of the features that developers can benefit from

- **Improved developer productivity**: Kotlin's concise and expressive syntax can help developers write code faster and with fewer errors.
- **Null safety**: Since Kotlin is written with nullability in mind, it helps us to avoid crashes related to the Null Pointer Exception.
- **IDE support**: Android Studio, which is built on top of IntelliJ IDEA, has been receiving tons of features to improve the Kotlin experience.
- **Jetpack libraries**: Jetpack libraries are available in Kotlin, and older ones are being rewritten with Kotlin. These are a set of libraries and tools to help Android developers write less code.
- **Jetpack Compose**: Jetpack Compose, a new UI framework, is completely written in Kotlin and takes advantage of features of the Kotlin language.

- **Kotlin Gradle DSL**: You are now able to write your Gradle files in Kotlin.
- **Coroutine support**: A lot of Jetpack Libraries support coroutines. For example, the `ViewModel` class has `viewModelScope` that you can use to scope coroutines in the lifecycle of the `ViewModel`. This aligns with the Structured Concurrency principles for coroutines. This helps cancel all coroutines when they're no longer needed. Some libraries including Room, Paging 3, and DataStore also support Kotlin coroutines.
- **Support from Google**: Google continues to invest in Kotlin. Currently there are resources ranging from articles to code labs, documentation, videos, and tutorials from the Android DevRel team at Google to assist you in learning new libraries and architecture for Android Development.
- **Active community and tooling**: Kotlin has a vibrant and active community of developers.

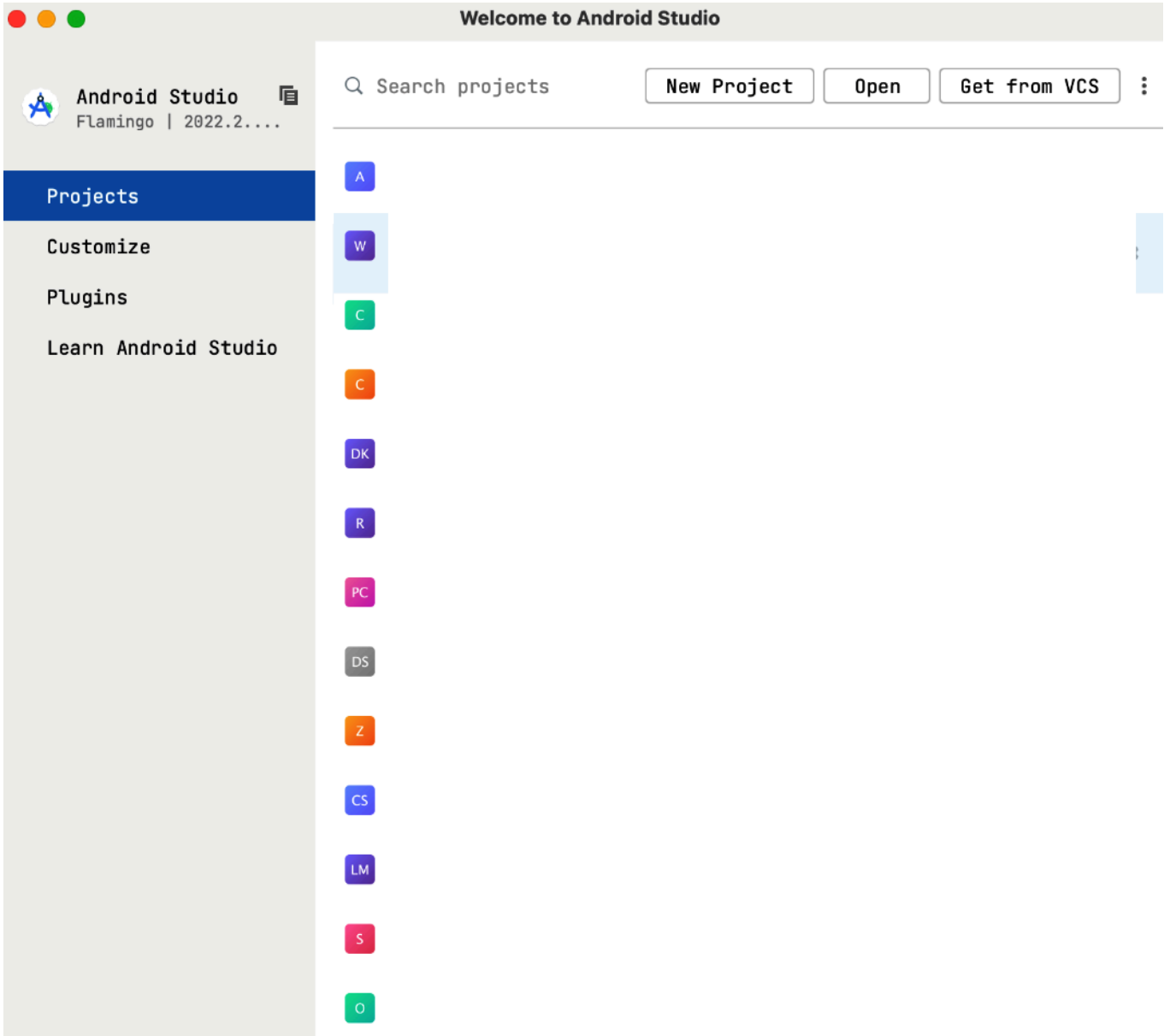# 2. Creating Your First Android App

Android, a mobile operating system developed by Google, runs on over two billion devices, such as smartphones, tablets, TVs, watches, and cars.
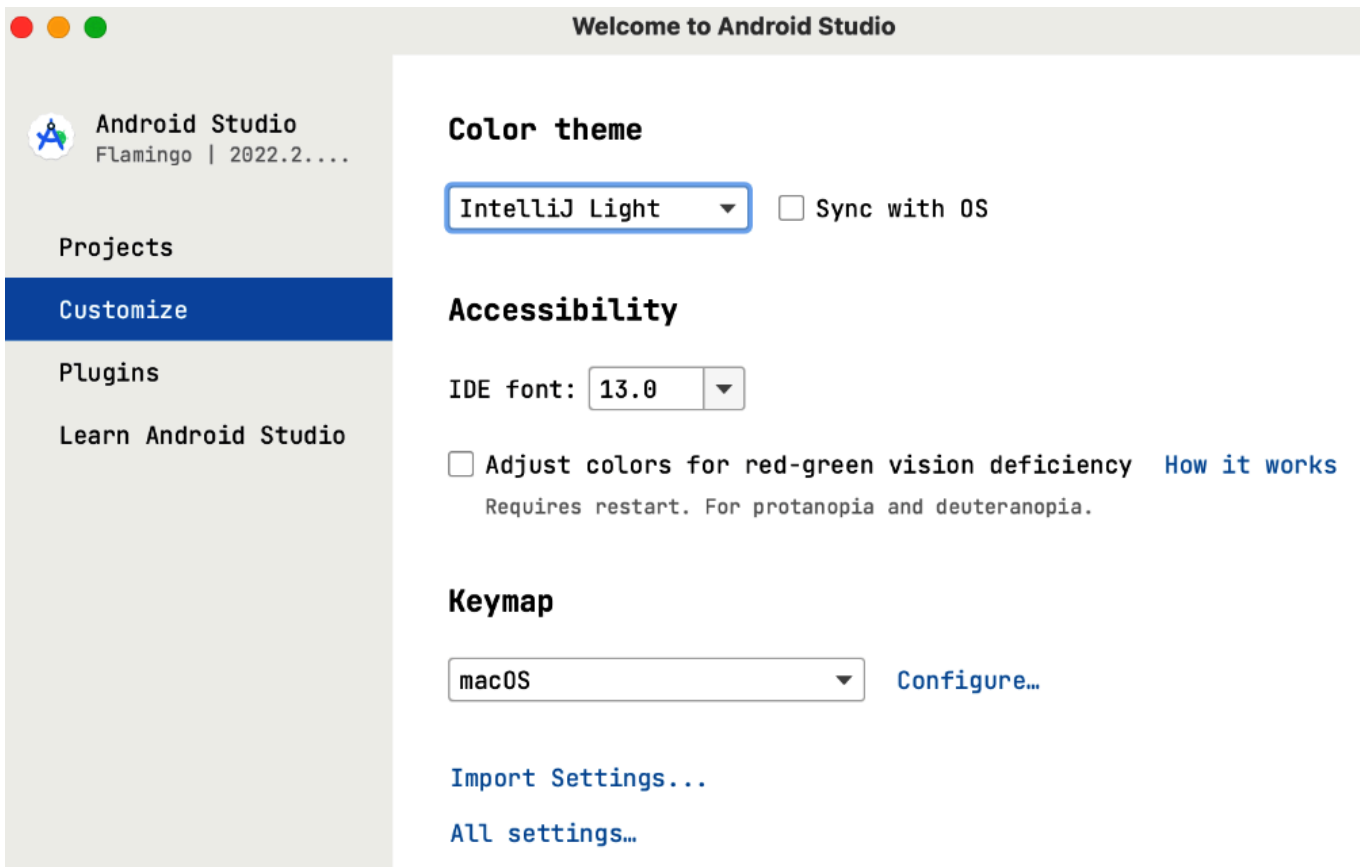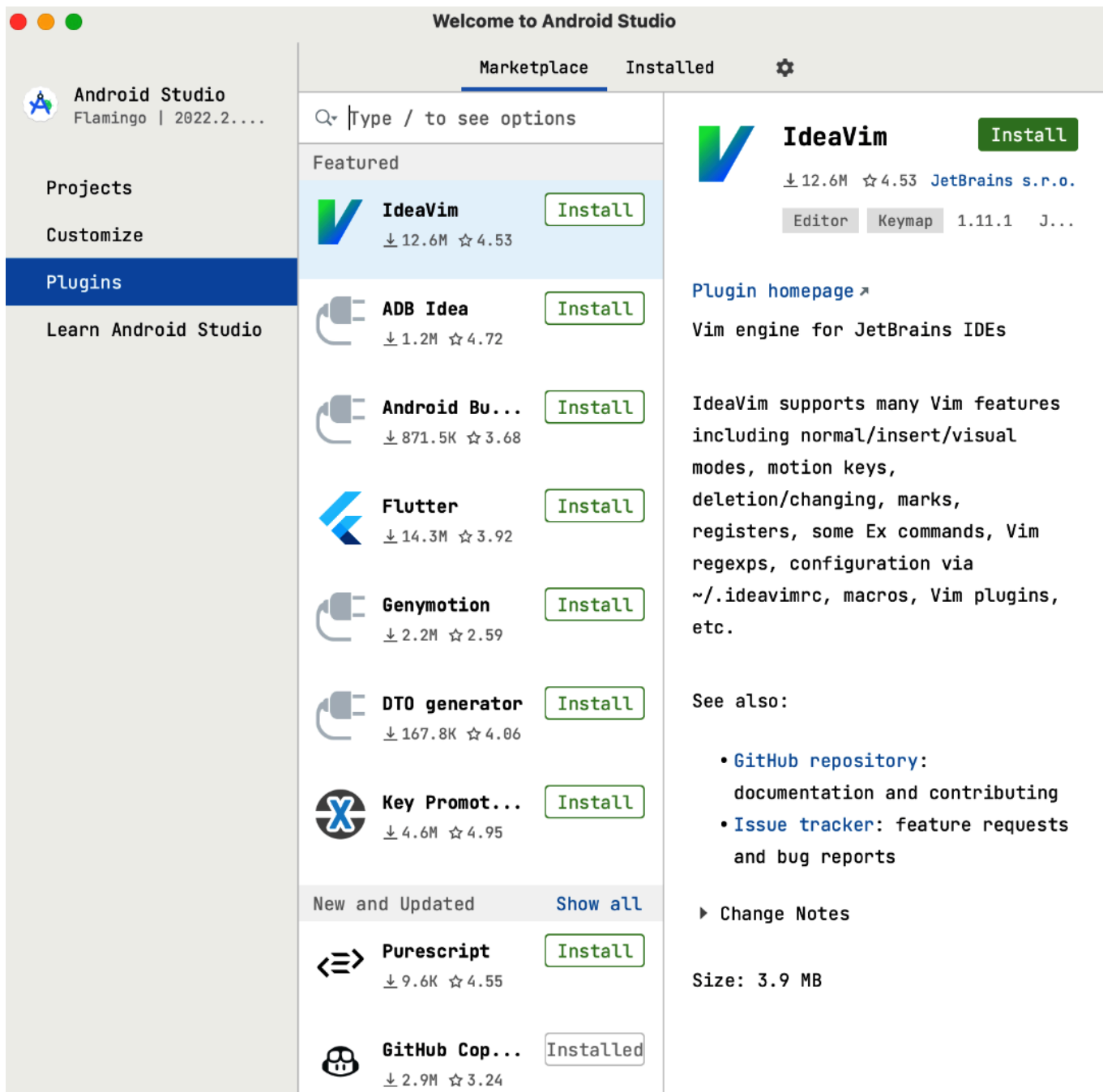
Technical requirements

IDE: Android Studio Hedgehog or later

Android Studio overview

Android Studio is the official IDE for creating Android applications. Built upon JetBrain's IntelliJ IDEA.

Android Studio
Flamingo | 2022.2....

Projects

Customize

Plugins

Learn Android Studio

Search projects

New Project

Open

Get from VCS

A

W

C

C

DK

R

PC

DS

Z

CS

LM

S

O

Android Studio
Flamingo | 2022.2....

Projects

Customize

Plugins

Learn Android Studio

Color theme

IntelliJ Light ▾   ☐ Sync with OS

Accessibility

IDE font: 13.0 ▾

☐ Adjust colors for red-green vision deficiency   How it works
    Requires restart. For protanopia and deuteranopia.

Keymap

macOS ▾   Configure…

Import Settings...

All settings…

Creating you Android app

1. Tap on the New Project button, which will take you to the Templates screen. To start with, on the right-hand side, we need to choose the specific form factor that we are targeting. By default, Phone and Tablet is selected. We are going to use the default option since we want to target Android and tablet devices. Next, we have to choose a template from the options

provided.
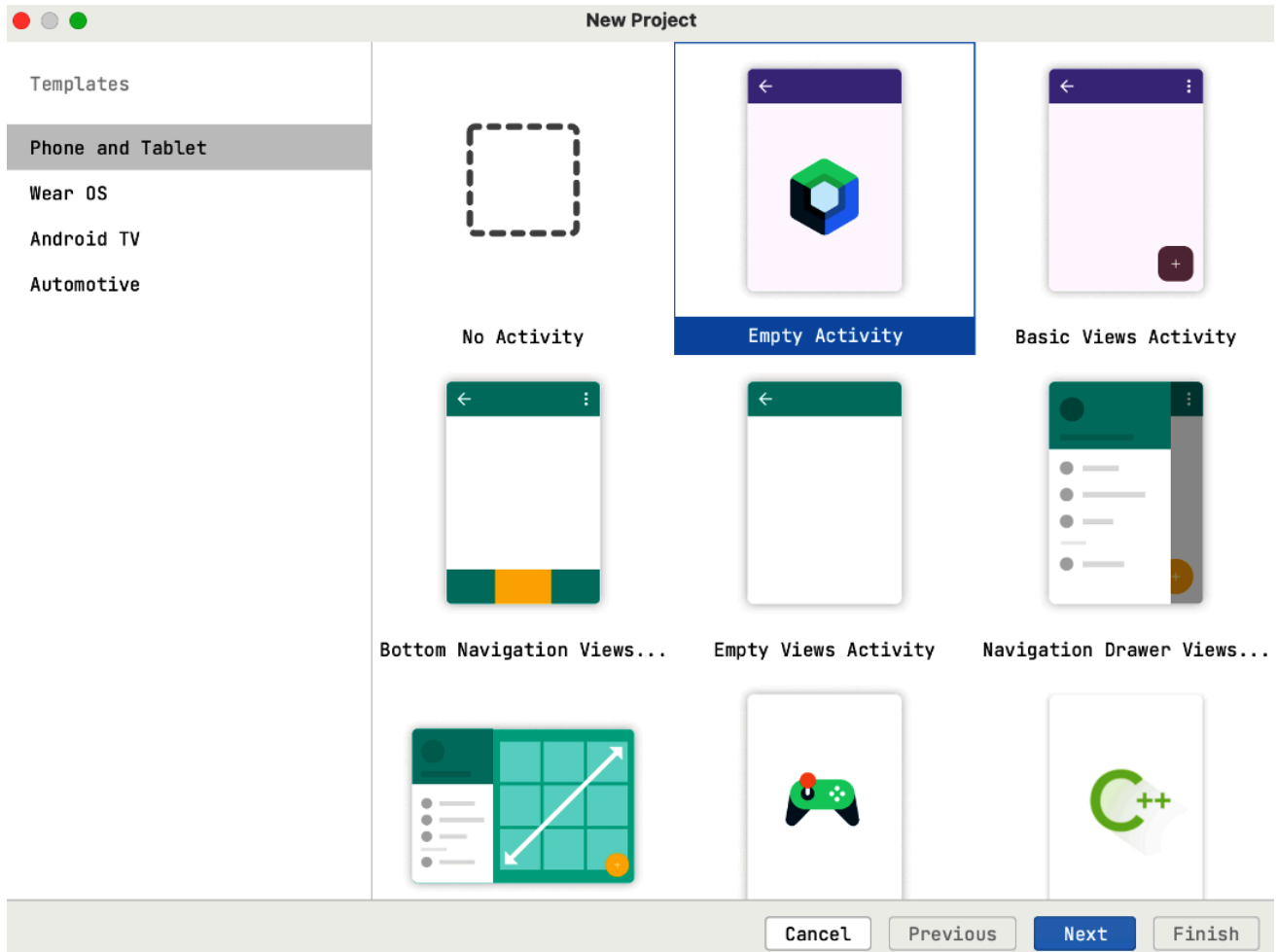
2. We will choose **Empty Activity** as we want to start from scratch. We use this instead of the **No Activity** option since this comes with some dependencies set up for us.

3. Tap Next, and we'll see the screen to configure the project details.
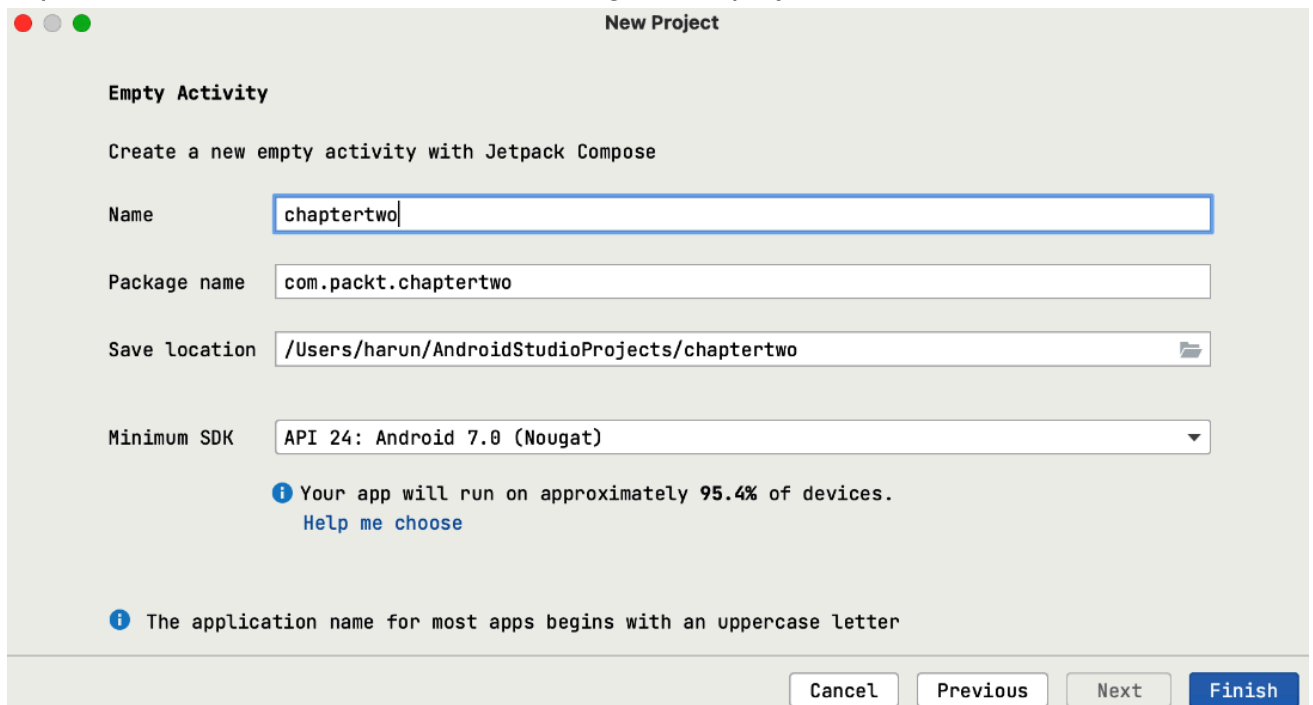
4. To finalize creating the project, we need to specify the following:
    1. **Name**: This is the unique name for our project.
    2. **Package name**: This is a unique identifier for our project. Normally it's a combination of the company website and app name.
    3. **Save location**: Here we specify the directory that our project will be in.
    4. Minimum SDK: This is the minimum Android version that our Android app will support. Android Studio gives us the percentage of devices using all the versions to help us decide the minimum Android version to support.
5. Lastly, tap **Finish** - this creates our project.
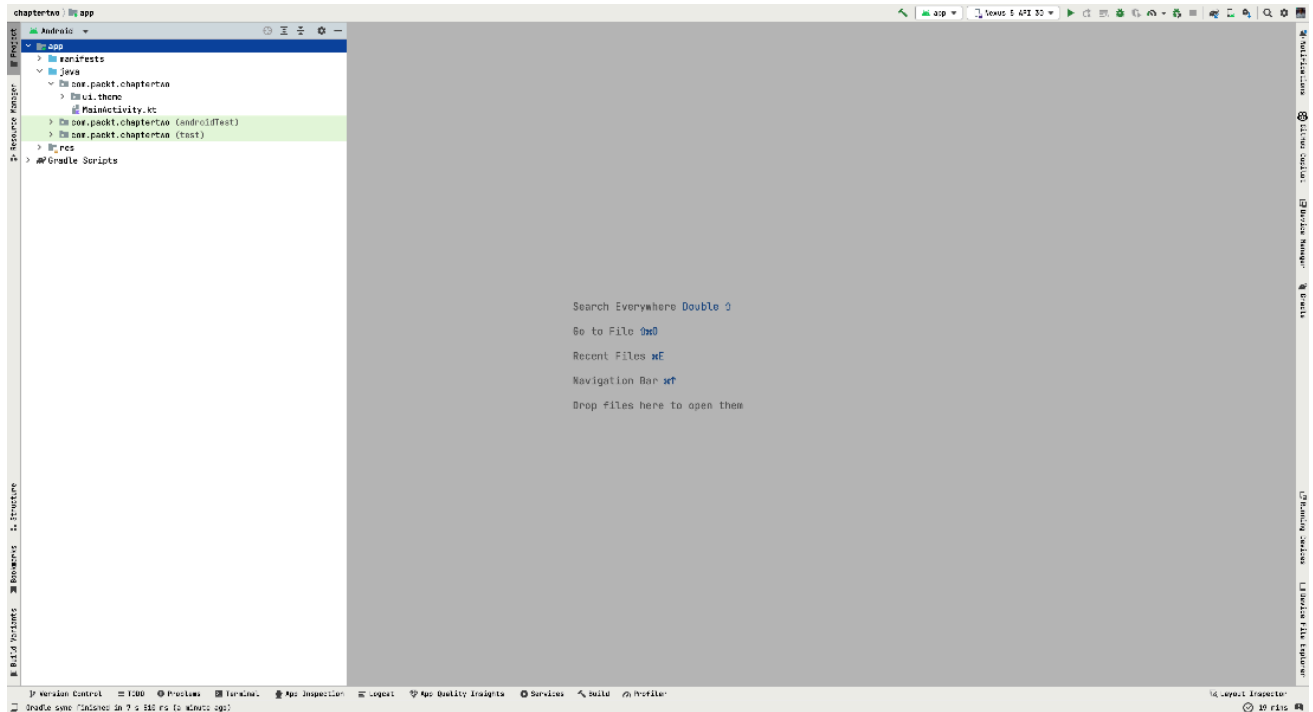
Exploring the new project
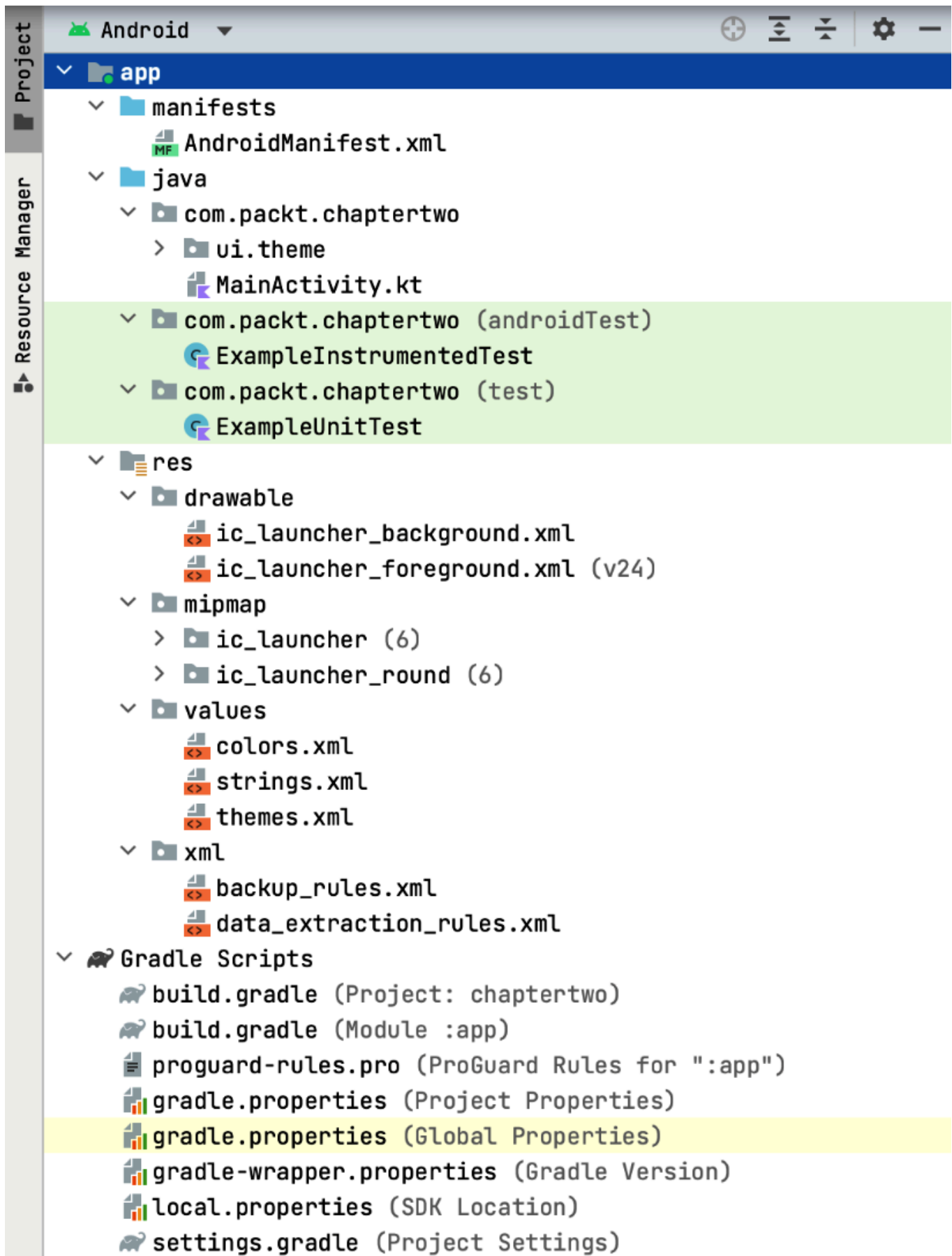
On the left, we have the project structure with different directories and packages. On the right is the editor section, which by default does not have anything. When you open any file inside Android Studio, this is where they appear.

- `manifests` : This has a single AndroidManifest.xml file, which is essential for our app configuration. A manifest file has a .xml extension and contains the information critical to your app. It communicates this information to the Android system. In this file, we define the permissions needed for our app, the app name, and icons. We also declare activities and services in this file. Without declaring them, it's hard for our app to use them.
- `java package` : This package, although named java, has all the Kotlin files for our project. If we need to add any files, this is where we add them. We can also create packages that help us group files with related functionality together.
- `com.packt.chaptertwo` : This is for the Kotlin files in our app
- `com.packt.chaptertwo (androidTest)` : Here, we add all the files for our instrumentation tests
- `com.packt.chaptertwo (test)` : Here, we add all the files for our unit tests
- `res` : This directory has all the resources needed for our app. These resources can include images, strings, and assets.
  - `drawable` : This folder contains custom drawables, vector drawables, or PNGs and JPEGs that are used in the app.
  - `mipmap` : This folder is where we place our launcher icons.
  - `values` : This folder is where we place our color, string, style, and theme files. In this folder, we define global values to be used all throughout the app.
  - `xml` : In this folder, we store XML files.
- Gradle Scripts: Here, we have all the Gradle scripts and Gradle property files needed for our project.
  - `build.gradle (Project: myapp)` : This is the top-level Gradle file where we add configurations that apply all over the project and submodules.
  - `build.gradle (Module: app)` : This is the app module Gradle file. Inside here, we configure the app module.

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}

android {
    namespace = "com.packt.chaptertwo"
    compileSdk = 35

    defaultConfig {
        applicationId = "com.packt.chaptertwo"
        minSdk = 34
        targetSdk = 35
        versionCode = 1
```

```kotlin
        versionName = "1.0"

        testInstrumentationRunner =
"androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_11
        targetCompatibility = JavaVersion.VERSION_11
    }
    kotlinOptions {
        jvmTarget = "11"
    }
    buildFeatures {
        compose = true
    }
    composeOptions {
        kotlinCompilerExtensionVersion = '1.3.2'
    }
    packagingOptions {
        resources {
            excludes += '/META-INF/{AL2.0, LGPL2.1}'
        }
    }
}

dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.activity.compose)
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.ui)
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)
    testImplementation(libs.junit)
```

```
        androidTestImplementation(libs.androidx.junit)
        androidTestImplementation(libs.androidx.espresso.core)
        androidTestImplementation(platform(libs.androidx.compose.bom))
        androidTestImplementation(libs.androidx.ui.test.junit4)
        debugImplementation(libs.androidx.ui.tooling)
        debugImplementation(libs.androidx.ui.test.manifest)
    }
```

- `namespace` : This is used as the Kotlin or Java package name for the generated `R` and `BuildConfig` classes.
- `compileSDK` : This defines the Android SDK version that will be used by Gradle to compile our app.
- `defaultConfig` : This is a block where we specify the default config for all flavors and build types. Inside this block, we specify properties such as `applicationId` , `minSDK` , `targetSDK` , `versionCode` , `versionName` , and `testInstrumentationRunner` .
- `buildTypes` : This configures different build types for our application, such as debug and release, or any custom build that we define. Within each build type block, we specify properties such as `minifyEnabled` , `proguardFiles` , or `debuggable` .
- `compileOptions` : We use this block to configure properties related to Java compilation. For example, we have defined `sourceCompatibility` and `targetCompatibility` , which specify the Java version compatibility for our project source code.
- `kotlinOptions` : We use this block to configure options related to Kotlin. A commonly used option is `jvmTarget` , which specifies which Java version to use for Kotlin compilation.
- `buildFeatures` : We use this block to enable and disable specific features in our project. For example, we've enabled compose in our project. We can enable or disable other additional features, such as `viewBinding` and `dataBinding` .
- `ComposeOptions` : This block is specific to projects that use Jetpack Compose. For example, inside this block, we can set `kotlinCompilerExtensionVersion` .
- `packagingOptions` : We use this block to customize the packaging options of our project, particularly regarding conflicts and merging.
- `dependencies` : Here we specify the dependencies in our project. We can add different libraries, modules, or external dependencies in this block.
- `proguard-rules.pro` : This is a file where you define rules for ProGuard to use when obfuscating your code.
- `gradle.properties (Project Properties)` : Here we define properties that apply to the whole project. Some of the properties include setting the Kotlin style and also specifying the memory to be used.
- `gradle.properties (Global Properties)` : This is a global file. We specify settings that we want to apply to all our Android Studio projects.

- `gradle-wrapper.properties (Gradle Version)` : In this file, we specify the Gradle wrapper properties, including the version and the URL from where to download the Gradle wrapper.
- `local.properties (Local Properties)` : In this file, we specify settings that need to apply to our local setup. Normally, this file is never committed to version control, so it means the configurations we add here only apply to our individual setup.
- `settings.gradle (Project Setting)` : We use this file to apply some settings to our project. For example, if we need more modules in our project, this is where they're specified.

When we build the project, Android Studio compiles all the resources and code using the configurations specified in our Gradle files and converts them into an Android Application Package (APK) or Android Application Bundle (AAB) that can run on our Android phones or emulators.

3. Jetpack Compose Layout Basics

Google introduced **Jetpack Compose**, a modern UI toolkit to help developers create UIs with less code.

Introduction to Jetpack Compose

Before Jetpack Compose, this is how we used to write UIs:

- Views were inflated from XML layout files. XML-based views are still supported alongside Jetpack Compose for backward compatibility.
- Themes, styles, and value resources were also defined in XML files.
- To access the views from XML files, we used view binding or data binding.
- This method of writing a UI required huge effort, requiring more boilerplate code and being error prone.

The **view system**, which was used to create UIs before Compose, was more procedural. Jetpack Compose is a whole other paradigm that uses declarative programming.

Declarative vs imperative UIs

In imperative UIs, we specify step by step the instructions describing how the UI should built and updated. We explicitly define the sequence of operations to create and modify UI elements.

In declarative UIs, we focus on describing the desired outcome rather than specifying the step-by-step instructions. We define what the UI should look like based on the current state, and the framework handles the rest.

In declarative approach UI state is represented by immutable data object. Instead of directly mutating the state, we create a new instance of the data objects to reflect the desired changes in the UI. We specify the relationship between the UI and the underlying state, and the framework automatically updates the UI to reflect those changes.

The Jetpack Compose version

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layoug.padding
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.runtime.remember
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp

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

@Composable
fun MyApp() {
    var count by remember { mutableStateOf(0) }

    Column(modifier = Modifier.padding(16.dp)) {
        Text(text="Counter: $count", style =
MaterialTheme.typography.bodyLarge)
        Spacer(modifier = Modifier.height(16.dp))
        Button(onClick = { count++ }) {
            Text("Increment")
        }
    }
}
```

The imperative approach, we must create the XML UI

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

```xml
    xmlns: tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
>

    <TextView
        android:id="@+id/counterTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:text="Counter: 0"
        android:textSize="20sp"
    />
    <Button
        android:id="@+id/incrementButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/counterTextView"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="16dp"
        android:text="Increment"
    />
</RelativeLayout>
```

With the layout file created, we can now create the activity class.

```kotlin
import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    private var count = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)

        val counterTextView: TextView = findViewById(R.id.counterTextView)
        val incrementButton: Button = findViewById(R.id.incrementButton)

        incrementButton.setOnClickListener {
            count++
```

```
            counterTextView.text = "Counter: $count"
        }
    }
}
```

Composable functions

Rendering: The process of converting code into viewable interactive content.

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

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

👤 Harun Wangereka
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    ChapterThreeTheme {
        Greeting( name: "Android")
    }
}
```

A composable function describes how to render a UI. This function must be annotated with the
`@Composable` function. Composable functions are meant to be **reusable**. Whenever the state
of the composable changes, it goes through a process of recomposition, which enables the UI
to display the latest state.

Composable functions are pure functions, meaning they don't have any side effects. They produce the same output when called several times with the same input. However launching a coroutine within a composable of calling external methods that do have side-effects, which should be avoided or handled carefully.

You can reuse and nest composables inside other composables.

```
@Composable
fun PacktPublishing(bookName: String) {
    Text(text="Title of the books is: $bookName")
}
```

Here Text is another Composed function from Material Design library.

Preview

In Jetpack Compose the `@Preview` annotation generates a preview of our composable function or a group of Compose components inside Android Studio.

```
// showBackground = true adds while background to our preview.
@Preview(showBackground = true)
@Composable
fun PacktPublishingPreview() {
    PacktPublishing("Android Development with Kotlin")
}
```

PacktPublishingPreview

Title of the book is: Android Development with Kotlin

Modifiers

Modifiers allow us to decorate our composable functions by enabling the following

- **Change** composables' size, behavior, and **appearance**
- **Add** more **information**
- **Process** user **input**
- **Add interactions** such as clicks and ripple effects

For example if we need to provide padding to our preview text

```
Text(
    modifier = Modifier.padding(16.dp),
    text = "Title of the book is: $bookName"
)
```

This will add 16.dp padding to the Text composable. 16.dp is a **density-independent** pixel unit in Jetpack Compose. This means it will remain consistent and adjust properly to different screen densities.

We can chain the different modifier functions in one composable. When chaining modifiers, the order of application is crucial.

```
Text(
    modifier = Modifier
        .fillMaxWidth()
        .padding(16.dp)
        .background(Color.Green),
    text = "Title of this book is: $bookName"
)
```

PacktPublishingPreview

Title of the book is: Android Development with Kotlin

Modifiers do not modify the original composable. They return a new, modified instance. This ensures our composable remains unchanged and immutable. In addition to using existing modifiers, we can also create our own modifiers when needed.

How Jetpack Compose transforms state into UI?

Tree of composables -> tree of layout -> render

State transforms into a UI in the following steps

1. **Composition** (initial phase)
   The **Compose compiler creates a tree of UI elements. Each element is a function that**

represents a UI element. Compose then calls the functions to create the UI tree. The composition step is responsible for determining which composables need updates and which ones can be reused. This happens by comparing a previous tree of composables with the new tree and only updating the ones that have changed. This makes this step very efficient as only elements with updates are updated.

2. **Layout**
   This step happens after the composition phase. Here, the Compose compiler takes the tree generated in the composition phase and **determines its size, position, and layout**. Each composable is measured and positioned within the layout based on its parent and any constraints set. It is also responsible for **creating the final layout tree** used in the drawing phase.

3. **Drawing**
   This is the last phase of transforming our UI to state. In this phase, the Compose compiler takes the **final layout tree** created in the layout phase and uses it **to draw the elements on the screen**. This is done by walking through the tree and issuing draw commands to the underlying graphics system. This phase is responsible for rendering the final UI on the screen.

Jetpack Compose layouts

Jetpack Compose offers the following layouts out of the box

- Column
- Row
- Box
- List

Column

Columns organize items vertically.

```
Column {
    Text(text = "Android")
    Text(text = "Kotlin")
    Text(text = "Compose")
}
```

Android

Kotlin

Compose

Lets polish it up a bit by using modifiers since Jetpack Compose also provides support for modifiers in these layouts.

```kotlin
Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp),
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Text(text = "Android")
    Text(text = "Kotlin")
```

```
        Text(text = "Compose")
}
```

Android
Kotlin
Compose

Row

Row organize items horizontally.

```
Row {
    Text(text = "Android")
    Text(text = "Kotlin")
    Text(text = "Compose")
}
```

PacktRowPreview

AndroidKotlinCompose

It supports addition of modifiers

```
Row(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp),
    verticalAlignment = Alignment.CenterVertically,
    horizontalArrangement = Arrangement.SpaceEvenly
) {
    Text(text = "Android")
    Text(text = "Kotlin")
    Text(text = "Compose")
}
```

Android   Kotlin   Compose
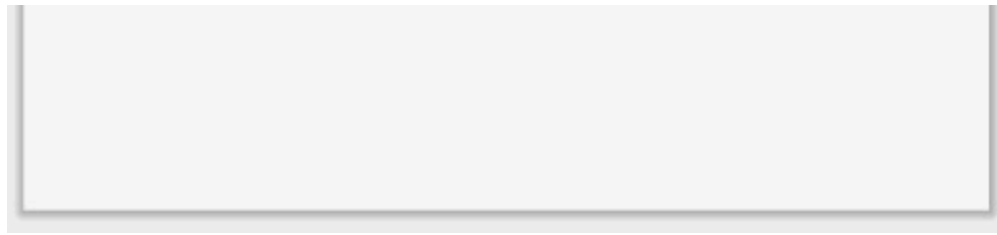
Box

The Box layout allows us to position child elements in a flexible way using the X and Y coordinates

```
Box(
    modifier = Modifier
        .size(100.dp),
    contentAlignment = Alignment.Center
) {
    Icon(
        modifier = Modifier
            .size(80.dp),
        imageVector = Icons.Outlined.Notifications,
        contentDescription = null,
        tint = Color.Green
    )
    Text(text = "9")
}
```

Lists

Compose provides the `LazyColumn` and `LazyRow` components, which can be used to display a list of items. They only render the items that are visible on the screen, rather than rendering all the items at once. `LazyColumn` and `LazyRow` are normally optimized for large datasets and at times are not suitable for all use cases.

Example

```
LazyColumn(
    modifier = Modifier
        .fillMaxSize()
        .background(Color.LightGray)
) {
    items(100) {
```

```kotlin
        Text(
            modifier = Modifier
                .padding(8.dp),
            text = "Item number $it"
        )
    }
}
```

```kotlin
        Text(
            modifier = Modifier
                .padding(8.dp),
            text = "Item number $it"
```

PackLazyColumnPreview

Item number 0

Item number 1

Item number 2

Item number 3

Item number 4

Item number 5

Item number 6

Item number 7

Item number 8

Item number 9

Item number 10

Item number 11

Item number 12

Item number 13

Item number 14

Item number 15

Item number 16

Item number 17

Item number 18

Item number 19

Item number 20

Item number 21

Item number 22

Item number 23

```kotlin
LazyRow(
    modifier = Modifier
        .fillMaxWidth()
        .background(Color.LightGray)
        .padding(8.dp)
) {
    items(100) {
        Text(
            modifier = Modifier
                .padding(8.dp),
            text = "Item number $it"
        )
    }
}
```

Item number 0   Item number 1   Item number 2   Item num

Jetpack Compose also have two more types of list layouts, `LazyVerticalGrid` and `LazyHorizontalGrid`.

`LazyVerticalGrid` creates a vertical list of items in a grid

```kotlin
LazyVerticalGrid(
    modifier = Modifier
        .fillMaxSize()
        .background(Color.LightGray)
        .padding(8.dp),
    columns = GridCells.Fixed(3)
) {
    items(100) {
        Text(
            modifier = Modifier
                .padding(8.dp),
            text = "Item number $it"
```

```
            )
        }
    }
}
```

PacktLazyVerticalGridPreview

Item number 0	Item number 1	Item number 2
Item number 3	Item number 4	Item number 5
Item number 6	Item number 7	Item number 8
Item number 9	Item number 10	Item number 11
Item number 12	Item number 13	Item number 14
Item number 15	Item number 16	Item number 17
Item number 18	Item number 19	Item number 20
Item number 21	Item number 22	Item number 23
Item number 24	Item number 25	Item number 26
Item number 27	Item number 28	Item number 29
Item number 30	Item number 31	Item number 32
Item number 33	Item number 34	Item number 35
Item number 36	Item number 37	Item number 38
Item number 39	Item number 40	Item number 41
Item number 42	Item number 43	Item number 44
Item number 45	Item number 46	Item number 47
Item number 48	Item number 49	Item number 50
Item number 51	Item number 52	Item number 53
Item number 54	Item number 55	Item number 56
Item number 57	Item number 58	Item number 59
Item number 60	Item number 61	Item number 62
Item number 63	Item number 64	Item number 65

| Item number 66 | Item number 67 | Item number 68 |
| Item number 69 | Item number 70 | Item number 71 |

`LazyHorizontalGrid` creates a horizontal list of items in a grid

```kotlin
LazyHorizontalGrid(
    modifier = Modifier
        .fillMaxSize()
        .background(Color.LightGray)
        .padding(8.dp),
    rows = GridCells.Fixed(3)
) {
    items(100) {
        Text(
            modifier = Modifier
                .padding(8.dp),
            text = "Item number $it"
        )
    }
}
```

Item number 0   Item number 3   Item number 6   Item num

Item number 1   Item number 4   Item number 7   Item num

Item number 2   Item number 5   Item number 8   Item num

Jetpack Compose also have `LazyVerticalStaggeredGrid` and `LazyHorizontalStaggeredGrid`, which are remarkably similar; the only difference is that they adapt to the children's height and width, respectively, meaning they all do not have uniform height or width.

ConstraintLayout

This layout enables to create responsive layouts. We can create complex layouts with relative positioning. `ConstraintLayout` uses chains, barriers, and guidelines to position child elements relative to each other.

It comes as a separate dependency, and we need to add it to our project. To add it, let us add this dependency to our app `build.gradle` file

Example for constraint layout

```
ConstraintLayout(
    modifier = Modifier
        .padding(16.dp)
) {
    val (icon, text) = createRefs()
    Icon(
        modifier = Modifier
            .size(80.dp)
            .constrainAs(icon) {
                top.linkTo(parent.top)
                bottom.linkTo(parent.bottom)
                start.linkTo(parent.start)
            },
        imageVector = Icons.Outlined.Notifications,
        contentDescription = null,
        tint = Color.Green
    )
    Text(
        modifier = Modifier
            .constrainAs(text) {
                top.linkTo(parent.top)
                bottom.linkTo(parent.bottom)
                start.linkTo(icon.end)
```

```
        },
        text = "9",
        style = MaterialTheme.typography.titleLarge
    )
}
```
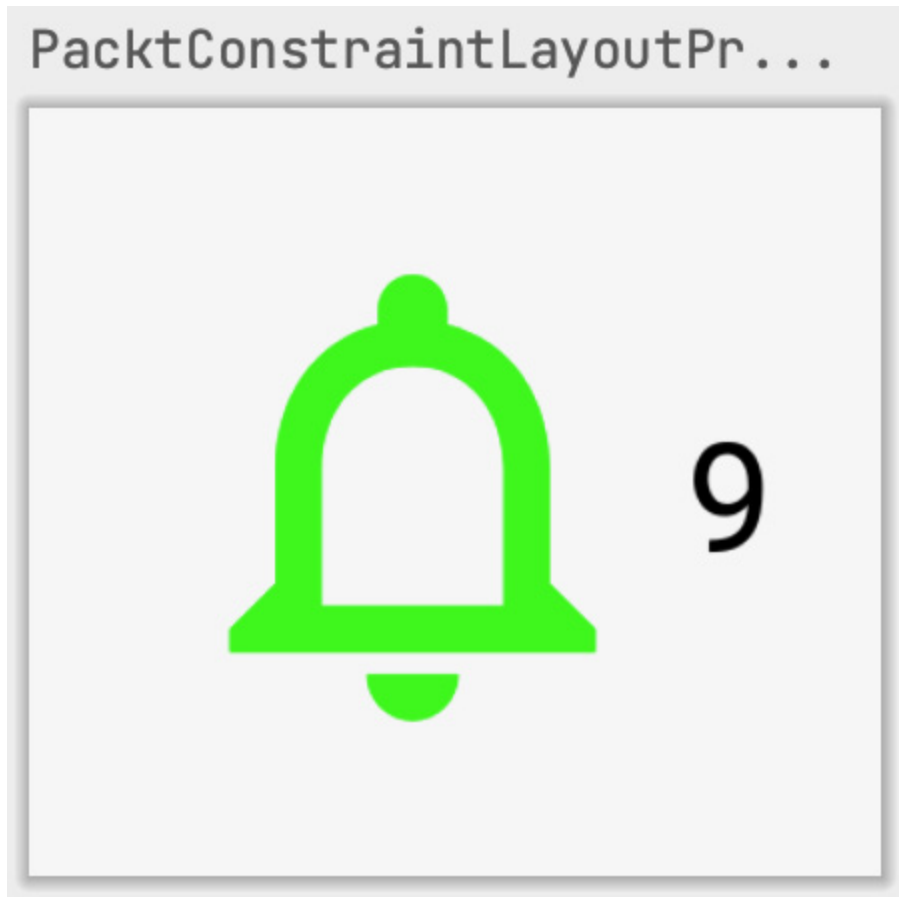

4. Design with Material Design 3

Material Design is a design system developed by Google. It helps us create beautiful UIs. It provides a set of guidelines and components for us to use as we're developing our Android apps.

Features of Material Design 3

- **Dynamic color**: This is a color system that sets the color of our apps to the color of the user's wallpaper. The System UI also adapts to this color.
- **More components**: Material 3 has a new set of improved components that are available for use. Some components have new UIs and others have been added to the APIs.
- **Simplified typography**: Material 3 has a much more simplified naming and grouping for typography. We have the following types: display, headline, title, body, and label, with each supporting small, medium, and large sizes.

- **Improved color scheme**: Material Design 3 support both dark and light color schemes in our apps. In addition to that, they created a new tool, Material Theme Builder (https://m3.material.io/theme-builder), which allows us to generate and export dark and light theme colors for our apps.
- **Simplified shapes**: Similar to typography, shapes have also been simplified to the following: Extra Small, Small, Medium, Large, and Extra Large. All these shapes come with default values, which we can always override to use our own.

Material components

The Material library comes with prebuilt components that we can use to build common UI components.

Top app bars

This is a component displayed at the top of the screen. It has a title and can also have some actions that are related to the screen the user is on. In Material 3, we have four types of top app bars: center-aligned, small, medium, and large
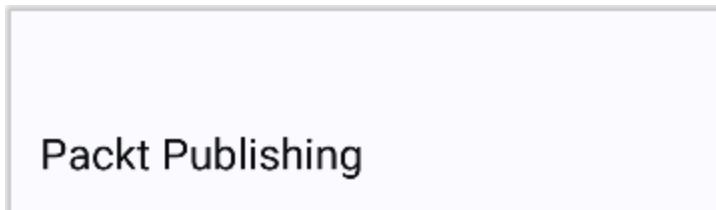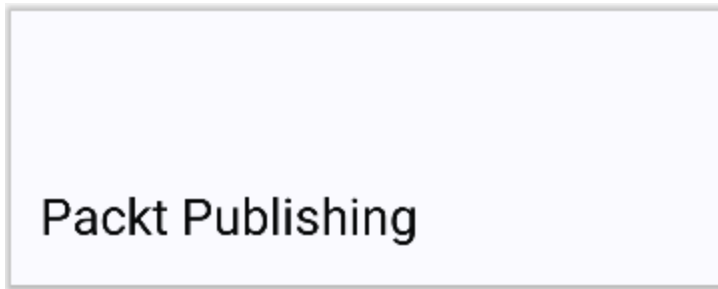
Small top app bar
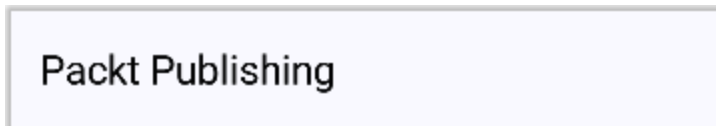
Center-aligned top app bar
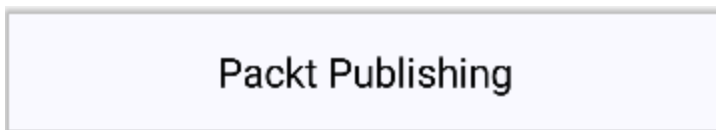
Medium top app bar

Large top app bar


```kotlin
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun packtCenterAlignedTopBar() {
    CenterAlignedTopAppBar(
        title = {
            Text(text = "Packt Publishing")
        }
    )
}
```

FloatingActionButton

`FloatingActionButton` is used to call a frequent action. ex: new chat

```kotlin
FloatingActionButton(
    onClick = { /*TODO*/ }
    content = {
        Icon(
            imageVector = Icons.Default.Add,
            contentDescription = "New Chat"
        )
    }
)
```


There is another type of `FloatingActionButton` known as `ExtendedFloatingActionButton` which look like floating action button with text. The only difference is that inside the content, we pass in a text since the `content` lambda exposes `RowScope`.

```
ExtendedFloatingActionButton(
    onClick = { /*TODO*/ }
    content = {
        Icon(
            imageVector = Icons.Default.Add,
            contentDescription = "New Chat"
        )
        Text(
            modifier = Modifier.padding(10.dp),
            text = "New Chat"
        )
    }
)
```
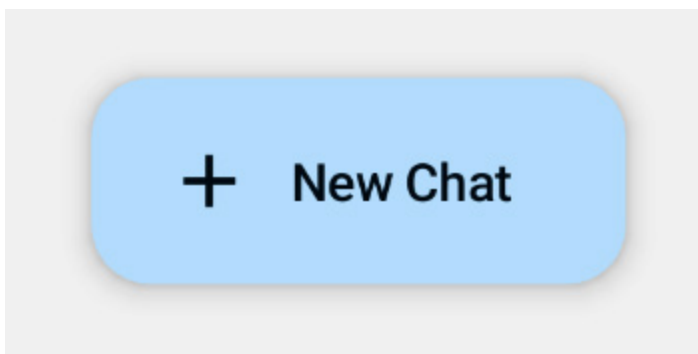

Bottom app bars

The bottom app bar components display navigation items at the bottom of the screen.

```
BottomAppBar(
    actions = {
        Icon(
            imageVector = Icons.Rounded.Home,
            contentDescription = "Home Screen"
        )
        Icon(
            imageVector = Icons.Rounded.ShoppingCart,
            contentDescription = "Cart Screen"
        )
        Icon(
            imaveVector = Icons.Rounded.AccountCircle,
            contentDescription = "Account Screen"
        )
    }
)
```
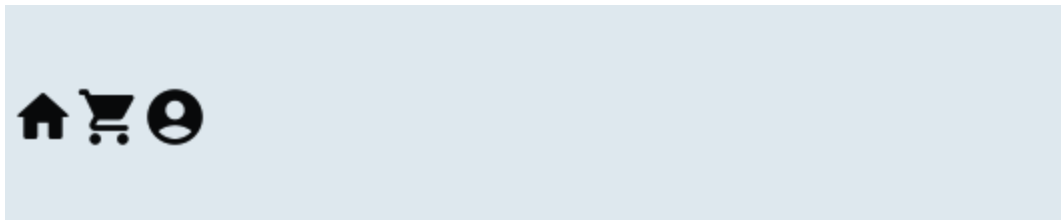
Additionally, in BottomAppBar, we can also provision a `FloatingActionButton` component.

```
BottomAppBar(
    actions = {
        Icon(
            imageVector = Icons.Rounded.Home,
            contentDescription = "Home Screen"
        )
        Icon(
            imageVector = Icons.Rounded.ShoppingCart,
            contentDescription = "Cart Screen"
        )
        Icon(
            imageVector = Icons.Rounded.AccountCircle,
            contentDescription = "Account Screen"
        )
    }
    floatingActionButton = {
        PacktFloatingActionButton()
    }
)
```
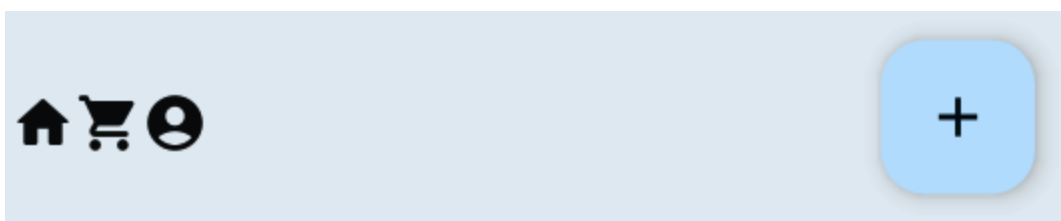

Scaffold

Scafflold layout helps place all components on your screen in their desired positions with ease.

```
Scaffold(
    topBar = {
        PacktSmallTopAppBar()
    },
    bottomBar = {
        PacktBottomNavigationBar()
```

```kotlin
        },
        floatingActionButton = {
            PacktFloatingActionButton()
        },
        content = { paddingValues -> Column(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(paddingValues)
                    .background(Color.Gray.copy(alpha = 0.1f)),
                verticalArrangement = Arrangement.Center,
                horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text(
                modifier = Modifier.padding(10.dp),
                text = "Kotlin for Android Development"
                textAlign = TextAlign.Center
            )
        }
    }
)
```

To view the full list of all the components, go to the Material 3 Components website (https://m3.material.io/components) to see them and their guidelines.

Using Material Design in our apps

`androidx.compose.material3:material3` is the dependency that contains the Material 3 components. We are using the Compose **Bill of Materials (BOM)** to manage our dependencies. This means that we do not have to specify the version of each dependency. Instead, we specify the version of the BOM, and it will manage the versions of the dependencies for us.

Adding Material Design 3 color schemes

```
// ui/theme/Color.kt
val Purple80 = Color(0xFFD0BCFF)
val PurpleGrey80 = Color(0xFFCCC2DC)
```

```
val Pink80 = Color(0xFFEFB8C8)

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

We will be using the Material Theme Builder tool to generate these colors. Let us open our browser and go to the Material Theme Builder tool (https://m3.material.io/theme-builder).
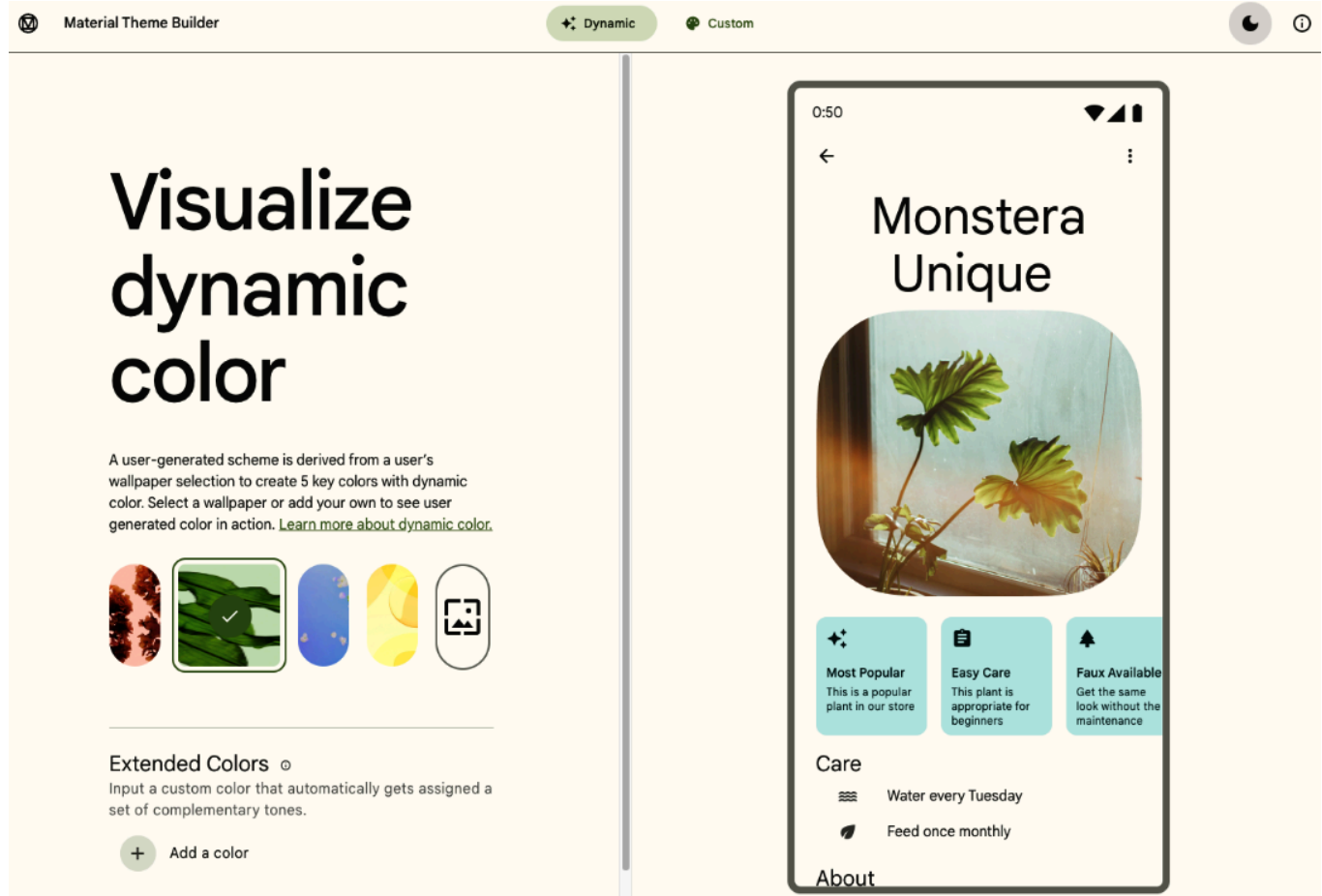
It makes it easier for us to customize and produce a consistent color scheme for our app. It has two tabs: Dynamic and Custom. In the Dynamic tab, we can select one of the preloaded colors or wallpapers to see how the color changes. One useful feature is that we can also add your

own wallpaper and generate the colors based on the wallpaper.
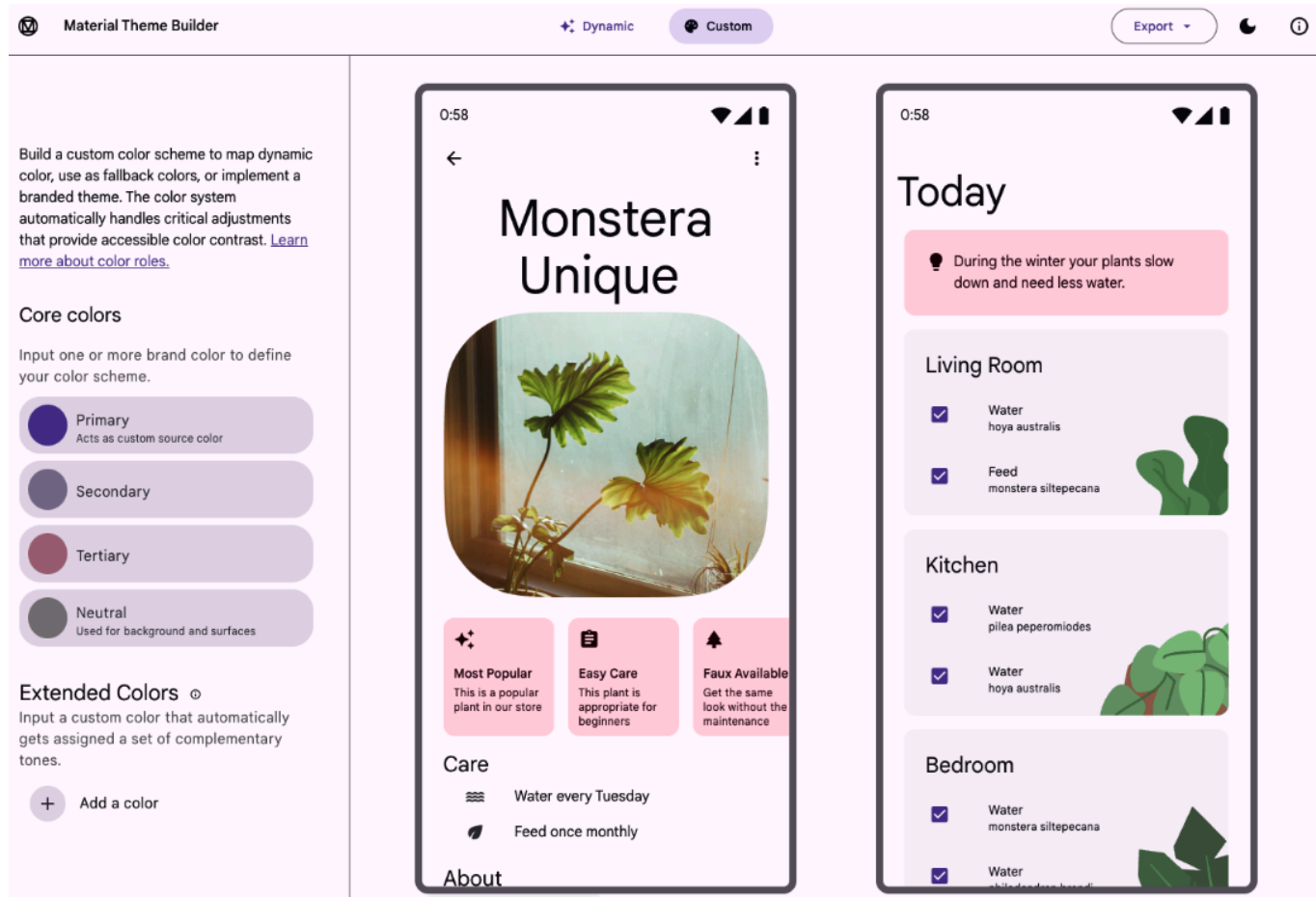
Designing UIs for large screens and foldables

Material 3 offers **canonical layouts** to serve as guidelines for creating UIs for large screens and foldables.

- **List-detail view**: Here, we place a list of items on the left and, on the right, we show the details of a single item.
- **Feed**: Here, we arrange content elements such as cards in a customizable grid, which provides a good view of a large amount of content.
- **Supporting pane**: Here, we organize app content into primary and secondary display areas. The primary area shows the main content while the secondary area shows the supporting content. The primary area occupies most of the screen while the secondary area occupies a smaller portion.

Jetpack Compose provides us with a way to get the screen size. We have the Material 3 `WindowSizeClass` to help us determine which layout to show in our app.

Using WindowSizeClass

Adding `implementation 'androidx.compose.material3:material3-window-sizeclass'` dependency to the app adds `WindowSizeClass` to the project.

WindowSizeClass classifies the available screen width into three categories:

- `Compact` : This is for devices whose width is less than 600 dp. Commonly, these are devices in portrait mode.
- `Medium` : This is for devices whose width is between 600 dp and 840 dp. Devices such as tablets and foldables in portrait mode fall into this category.
- `Expanded` : This is for devices whose width is greater than 840 dp. Devices such as tablets and foldables in landscape mode, phones in landscape mode, and desktops fall into this category.

`WindowSizeClass` uses `widthSizeClass` to get the width of the screen and `heightSizeClass` to determine height of the screen.

```
when(calculateWindowSizeClass(activity = this).widthSizeClass) {
    WindowWidthSizeClass.Compact -> {
        CharactersScreen(
            navigationOptions = NavigationOptions.BottomNavigation,
            showDetails = false
        )
    }
    WindowWidthSizeClass.Medium -> {
        CharactersScreen(
            navigationOptions = NavigationOptions.NavigationRail,
            showDetails = true
        )
    }
    WindowWidthSizeClass.Expanded -> {
        CharactersScreen(
            navigationOptions = NavigationOptions.NavigationDrawer,
            showDetails = true
        )
    }
    else -> {
        CharactersScreen(
            navigationOptions = NavigationOptions.BottomNavigation,
            showDetails = false
        )
    }
}
```

Making our app accessible

Jetpack Compose uses **semantics** to make our apps accessible. Semantics are used to describe the UI elements in our apps. Some of the best practices for making our apps accessible are:

- We should always ensure that all clickable or touchable elements or those that require user interaction are large enough to be easily tapped or clicked. Most Material components out of the box have a default size that is large enough to be easily tapped or clicked.
- We should add content descriptions to our composables. Components such as Icon and Image provide this argument to describe visual elements to accessibility services.

```
Icon(
    modifier = Modifier.size(48.dp),
    painter = painterResource(id = R.drawable.ic_launcher_foreground),
    contentDescription = "Icon"
)
```

- We should label our clickable elements. We can pass a clickable label to the clickable modifiers.

```
Text(
    modifier = Modifier
        .clickable(
            onClick = { /*TODO*/ },
            onClickable = "Click Me"
        )
        .padding(10.dp),
    text = "Click Me"
)
```

- By using semantics, we can also describe headers. Headers are used to describe the content that follows them.

```
Text( modifier = Modifier
    .semantics { heading() }
    .padding(10.dp),
    text = "Heading One"
)
```

- We can additionally provide information about the state of our composables. For example, we can provide information about the state of a button.

```
Button(
    modifier = Modifier
        .semantics { stateDescription = "Disabled" }
        .padding(10.dp),
    onClick = { /*TODO*/ },
    enabled = false
) {
    Text(text = "Disabled Button")
}
```

- For some groups of components, we can also use the mergeDescendants parameter to merge the semantics of the children composables.

```
Column(
    modifier = Modifier
        .padding(10.dp)
        .semantics(mergeDescendants = true) {
        }
) {
    Text(text = "Heading One")
    Text(text = "Heading Two")
    Text(text = "Heading Three")
}
```

5. Architect Your App

In **app development**, **architecture** refers to the **high-level structure of the app** — how the code, features, data, and components are organized and how they interact with each other.

Scaling: basically means growing or shrinking something in size, amount, or performance.

The process of developing apps needs to be scalable in such a way that you can maintain the app over a long time and easily hand over the development of it to other developers or teams.

Introduction to app architecture

Benefits of using an architecture:

- **Separation of concerns**: Using an architecture allows us to separate our code into different layers. Each layer only does one thing. This makes it easy to separate and group our code into different layers. Each layer has its responsibility. This prevents things from being mixed up and makes it easier to maintain our code.

- **Easy testing**: Using an architecture makes it easy to test our code. We can easily test each layer of our code in isolation since things are not tightly coupled.
- **Easy to maintain**: Using an architecture makes it easy to maintain our code. We can easily make changes to our code without affecting other parts of our code. This makes it easy to maintain our code over the long term. We can also swap in and out different parts of the code, and since we have tests, we can test the different implementations and ensure that nothing breaks.
- **Easy to scale**: Using an architecture makes it easy to scale our code. We can easily add new features to our code without affecting other parts of our code.
- **Easy to work in teams**: Using an architecture makes it easy to work in teams. Each team member or team can work on a different layer of the code. It makes it possible to work concurrently on different parts of the codebase.
- **Promotes reusability**: Over time, we can place some of the commonly used code across the projects in common packages or modules, which can then be re-used across the project without having to repeat the code.

We can architect our app either by feature or by layers. When we architect by feature, we have layers that represent a feature. When we architect by layers, we have layers that represent a layer of our app. An example of architecting by feature is shown in the following points:

- Home feature
- Profile feature
- Settings feature
  Each feature has its layers and all the code related to that feature.

Some of the architectures that we can use to build our apps:

- **Model, View, and ViewModel (MVVM)**: This is the most commonly used architecture. The app has the `Model View` and `ViewModel` layers. The Model layer is responsible for holding the data. The View layer is responsible for displaying the data. The `ViewModel` layer is responsible for holding the state of the data. It is also responsible for communicating with the Model and View layers. MMVM promotes the clear separation of concerns between the different layers. It also supports data binding, which makes it easy to update the UI when the data change.
- **Model View Intent (MVI)**: This has three key layers. The **Model** layer is responsible for holding the data. The **View** layer is responsible for displaying the data. The **Intent** layer represents user actions or events that are dispatched to the model to update a state. MVI promotes unidirectional data flow, where **data flow** in the same direction.
- **Model View Controller (MVC)**: This architecture has three layers. The **Model** layer represents the business logic and holds the data. The **View** layer is responsible for displaying the data. The **Controller** layer is responsible for and acts as an intermediary

between the model and the view. Its only problem is that it has a tight coupling between the layers, making it hard to test and scale.

- **Model View Presenter (MVP)**: This architecture has three layers. The **Model** layer represents the business logic and holds the data. The **View** layer displays the data and UI components and observes user interactions. The views delegate all UI-related logic to the presenters. The **Presenter** layer contains presentation logic and acts as an intermediary between the model and the view.