Mobile Applications

1. Introduction to Android

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

uc3m | Universidad Carlos III de Madrid

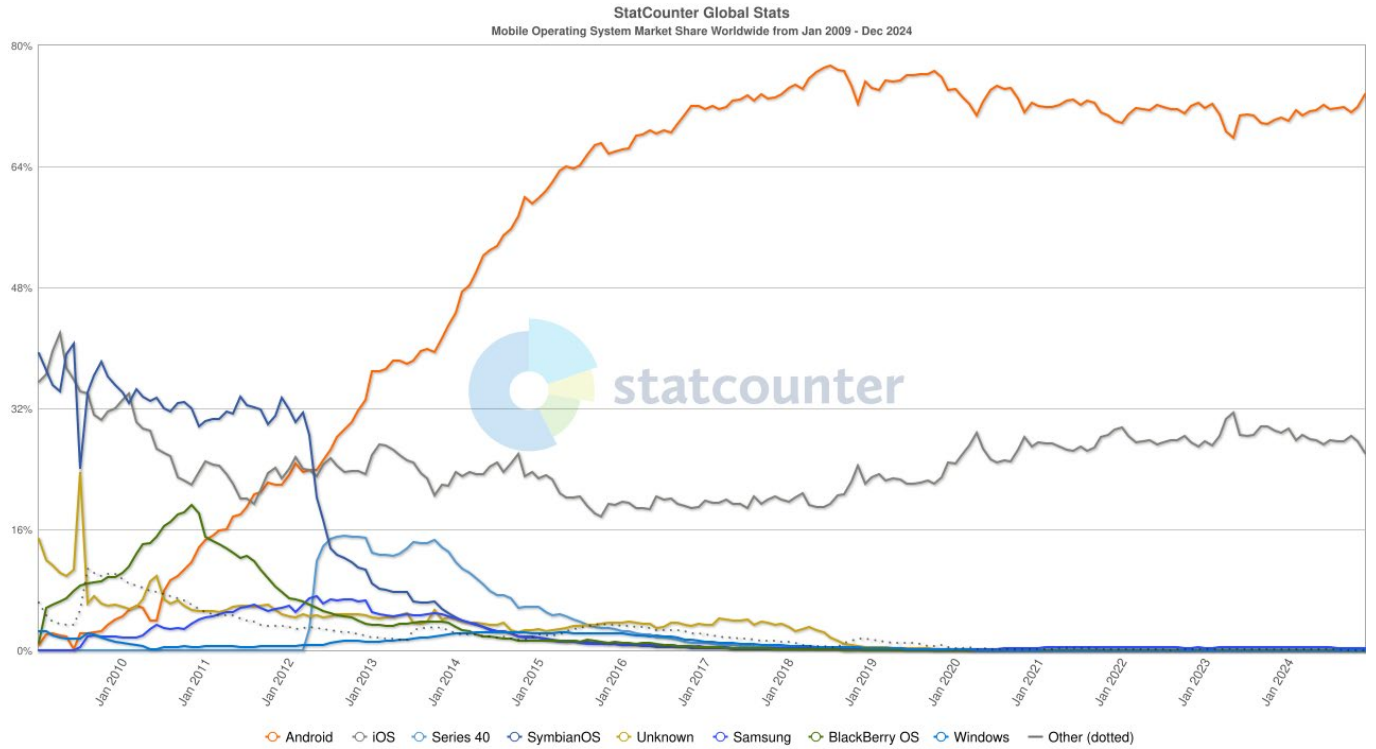
Table of contents

1. Introduction

- A **mobile application** (often called simply app) is a software application designed to run on a mobile device such as a phone, tablet, or watch
  - Mobile devices are portable electronic device designed to be carried and used on the go
  - Mobile apps stand in contrast to *desktop applications* (which are designed to run on desktop computers, such as Windows/macOS/Linux) or *web applications* which (run in web browsers, such as Chrome/Firefox/Edge/Safari)
- A **mobile operating system** (mobile OS) is a specialized software platform that manages the hardware and software resources of a mobile device, such as a smartphone, tablet, or wearable device
  - It serves as the interface between the user and the hardware, enabling apps to function and providing essential services to users

1. Introduction

- Nowadays, there are two main players in the mobile OS market:
  - **Android** by Google
  - **iOS** by Apple



https://gs.statcounter.com/os-market-share/mobile/worldwide/

1. Introduction

- There are different ways for developing mobile apps:

1. **Native** development involves creating apps specifically for a given platform (Android or iOS) using platform-specific programming languages and tools
2. **Hybrid** development combines web technologies (HTML, CSS, JavaScript) with a *WebView* native container to create apps that work across multiple platforms
3. **Cross-platform** development uses frameworks that allow developers to write code once and deploy it on multiple platforms
   - Unlike hybrid apps, cross-platform apps are compiled into native code
4. **Progressive Web Apps** (PWAs) are websites that behave like apps
   - They run in a browser but can be installed on a device and used offline
5. **Low-Code/No-Code Platforms** that allow non-developers to create apps using drag-and-drop interfaces and pre-built templates

1. Introduction

App type	Pros	Cons
Native	+ High performance and responsiveness + Best user experience and design consistency	- Requires separate codebases for each platform (more development time and cost)
Hybrid	+ Easy for developers familiar with web development + Single codebase for multiple platforms	- Slower performance compared to native apps - Limited access to advanced device features
Cross-platform	+ Saves development time and cost with a single codebase + Good performance for most apps	- Performance may not match fully native apps - Limited access to certain platform-specific features (depending on the framework)
PWAs	+ No app store submission required + Can work on any device with a browser + Cost-effective and fast to develop	- Limited access to device hardware and native features - Can't match the performance of native apps
Low-Code/ No-Code	+ Speeds up development for simple apps + Requires little to no programming knowledge	- Limited flexibility and scalability for complex apps



Table of contents

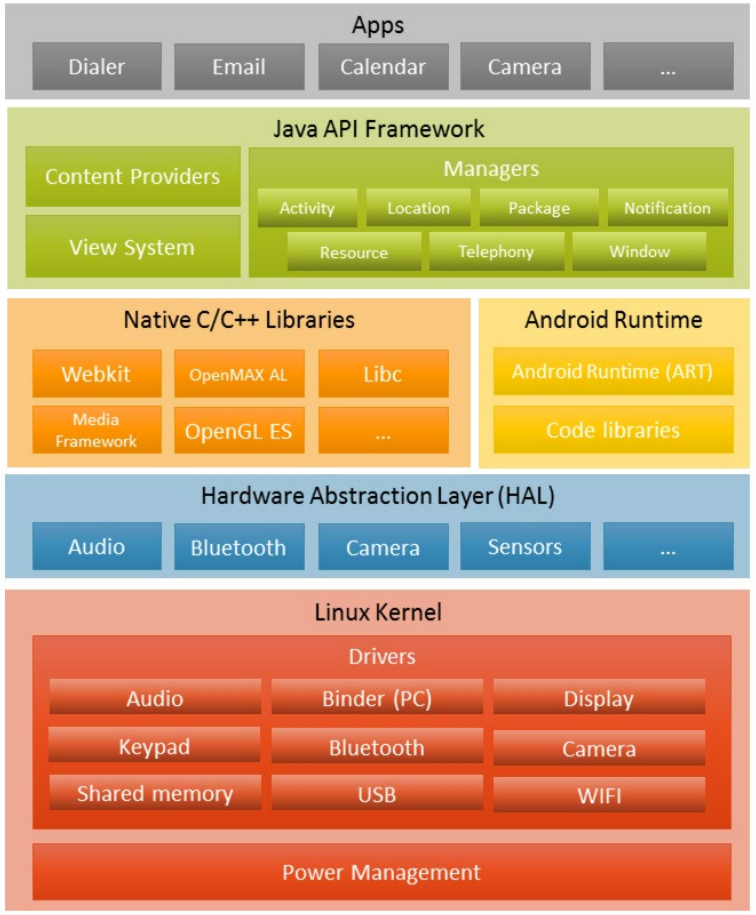
2. Android fundamentals

- **Android** is an open-source mobile operating system (OS) based on a modified version of Linux
  - A mobile OS is used in smartphones, tablets, smartwatches, or other mobile devices
- Android was originally developed by a startup named Android, acquired by Google in 2005
  - The Android source code is primarily licensed under the Apache 2.0 License
- Android is the most used operating system worldwide
  - Android controls around 74% of OS market share nowadays (the remaining is iOS)

https://www.android.com/

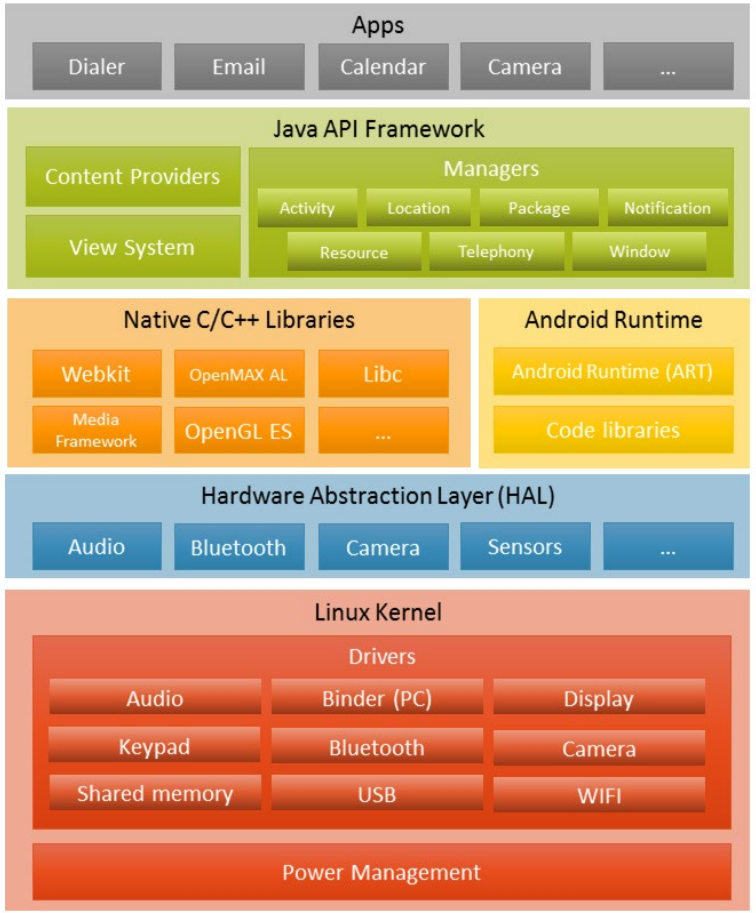
2. Android fundamentals - Architecture

- The architecture of Android (called Android platform) is a stack divided into several layers:
  - Linux kernel: This is the foundation of the Android platform. This layer contains all the low-level device drivers for the various hardware components of an Android device
  - Hardware Abstraction Layer (HAL): This layer provides standard interfaces that expose hardware capabilities to the higher-level Java API framework
  - Android Runtime (ART): It provides an application runtime environment for .dex files (a bytecode format designed for minimal memory footprint)



2. Android fundamentals - Architecture

- The architecture of Android (called Android platform) is a stack divided into several layers:
  - Native C/C++ libraries: Libraries such as OpenGL ES for high-performance 2D and 3D graphics processing
  - Java API framework: The entire feature-set of Android is available for developers through APIs written in Java
  - Apps: Android comes with a set of core apps, such as Phone, Contacts, Browser, and so on. In addition, many others apps can be downloaded and installed from Google Play (formerly Android Market)



2. Android fundamentals - Android API

- An API (**Application Programming Interface**) is a type of software interface that allows two applications to talk to each other
  - An API offers a service to other pieces of software
- All Android functionality is available to app developers through APIs written in the Java language
  - Android apps (written in Java or Kotlin) use that API
- The Android API contains the building block for creating Android apps, for instance:
  - Jetpack (suite of libraries to help developers follow best practices, reduce boilerplate)
  - View System (for apps UIs)
  - Resource Manager (for internationalization -I18N-, graphics, layouts)
  - Notification Manager (for custom alerts in the status bar)
  - Activity Manager (to manage the apps lifecycle)
  - Content Provider (to manage access data)

https://developer.android.com/reference

2. Android fundamentals - Android SDK

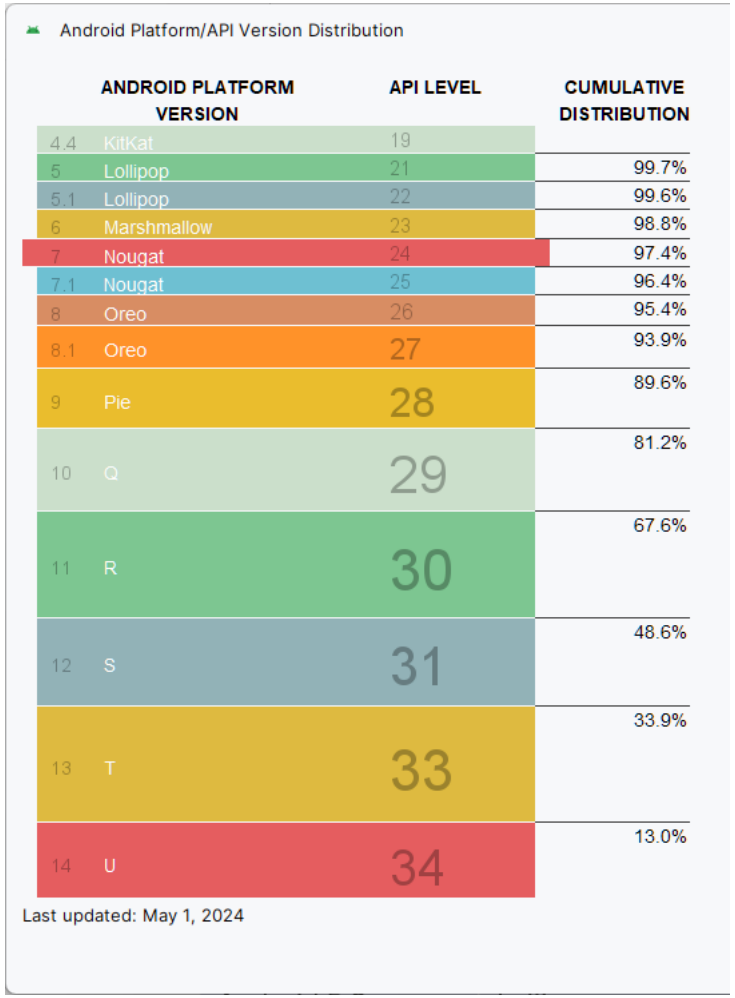
- A SDK (Software Development Kit) is a collection of software development tools in one installable package
- The **Android SDK** includes a comprehensive set of development tools, including a debugger, emulator, documentation, sample code, and tutorials
- There are different versions of the Android SDK
  - Each Android SDK is linked to a version of the Android platform
  - Each Android Platform uses a default version of the Android API (called API level)
  - In this course, it is recommended to use SDK 7.0 as the minimum

https://apilevels.com/

2. Android fundamentals - Android SDK

Platform version	Code name	API Level
Android 14	Upside Down Cake	34
Android 13	Tiramisu	33
Android 12	Snow Cone	31
Android 11	Red Velvet Cake	30
Android 10	Quince Tart	29
Android 9	Pie	28
Android 8.1	Oreo	27
Android 8.0	Oreo	26
Android 7.1	Nougat	25
Android 7.0	**Nougat**	**24**
Android 6.0	Marshmallow	23
Android 5.1	Lollipop	22
Android 5.0	Lollipop	21
Android 4.4	Kikkat	19

Recommended for this course (`minSdk`)



2. Android fundamentals - Programming languages

- Android apps can be developed with Java or Kotlin
  - **Java** is a popular high-level, object-oriented programming language
    - It was originally developed by Sun Microsystems in the mid-1990s
    - It is now owned and maintained by Oracle Corporation
  - **Kotlin** is modern high-level, object-oriented programming language
    - It was created by JetBrains in 2010, first released on 2016
    - It was designed to be concise, safer, and expressive
    - It is fully interoperable with Java, meaning that Kotlin code can coexist and interact seamlessly with existing Java code
    - It was officially endorsed by Google for Android development in 2017
- Google started to recommend using Kotlin in Android apps due to legal issues with Oracle
  - The use of Java by Google in Android was demanded by Oracle in 2010
  - In April 2021, the US Supreme Court declares Google's code copying fair

We use Kotlin for the code examples in the master lectures

Table of contents

3. Introduction to Kotlin

- Kotlin is a programming language officially supported by Google for Android development. The key features of Kotlin are:
  - Interoperability with Java: Kotlin is 100% interoperable with Java
  - Rich Android support: Kotlin integrates seamlessly with the Android APIs and tools
  - Concise syntax: Kotlin reduces boilerplate code, making it shorter and easier to read compared to Java
  - Null safety: Nullable and non-nullable types are clearly distinguished
  - Default and named parameters: Function parameters can have default values, and named parameters make method calls clearer
  - Extension functions: Kotlin allows developers to add new functionality to existing classes
  - Coroutines for asynchronous programming: Coroutines provide a clean, non-blocking way to handle tasks without the complexity of callbacks

https://kotlinlang.org/

3. Introduction to Kotlin

- Some code examples:

```
// Java
public class User {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

// Kotlin
data class User(var name: String)
```

Concise syntax

```
var name: String = "John" // Non-nullable
var nullableName: String? = null // Nullable
```

Null safety

```
fun String.isEmail(): Boolean {
    return this.contains("@")
}

val email = "example@gmail.com"
println(email.isEmail()) // true
```

Extension functions

```
fun greet(name: String = "Guest", age: Int = 18) {
    println("Hello, $name! You are $age years old.")
}

// Usage:
greet() // "Hello, Guest! You are 18 years old."
greet("Alice") // "Hello, Alice! You are 18 years old."
greet("Bob", 25) // "Hello, Bob! You are 25 years old."
```

Default parameters

```
fun greet(name: String = "Guest", age: Int = 18) {
    println("Hello, $name! You are $age years old.")
}

// Usage with named parameters:
greet(age = 25, name = "Charlie") // "Hello, Charlie! You are 25 years old."
greet(name = "Diana") // "Hello, Diana! You are 18 years old."
```

Named parameters

Table of contents

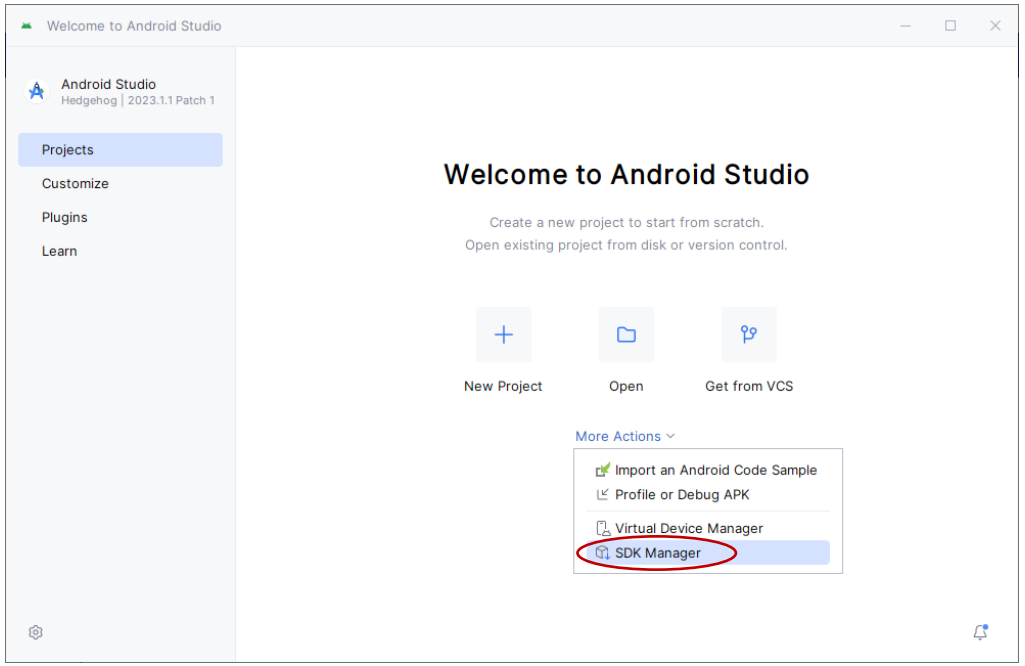
4. Android Studio

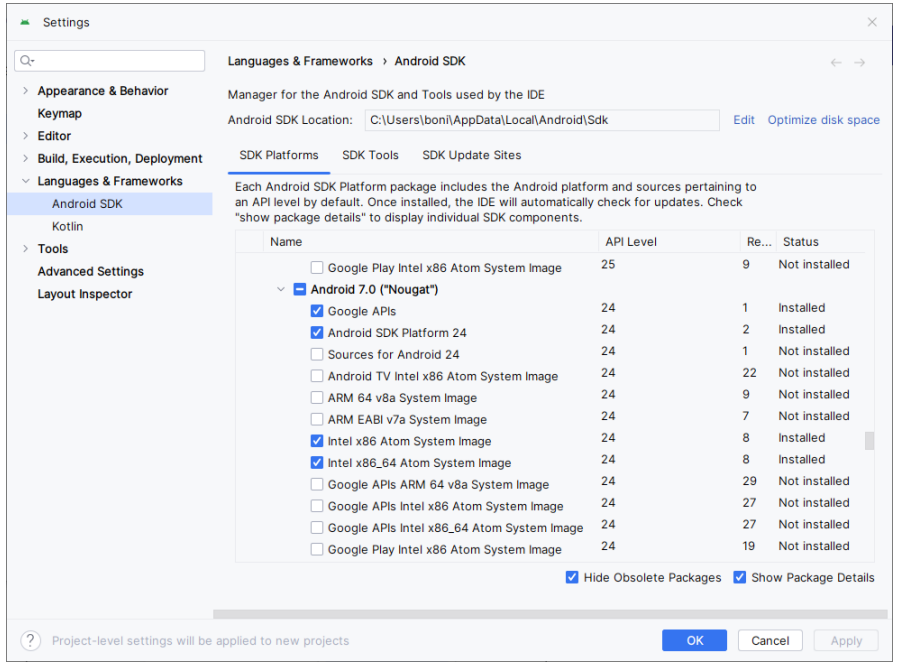
- **Android Studio** is the official development platform for Android
  - It is based on Intellij IDEA, which is a popular and comprehensive IDE (Integrated Development Environment)
    - An IDE is a software application that provides comprehensive facilities for software development (e.g., source code editor, build automation tools, debugger, etc.)
- Some relevant integrated tools in Android Studio are:
  - SDK Manager: tool that allows developers to download, update, and manage the Android SDK (Software Development Kit)
  - Virtual Device Manager (VDM): tool that allows to create and manage Android Virtual Devices (AVDs)

https://developer.android.com/studio

4. Android Studio - SDK manager

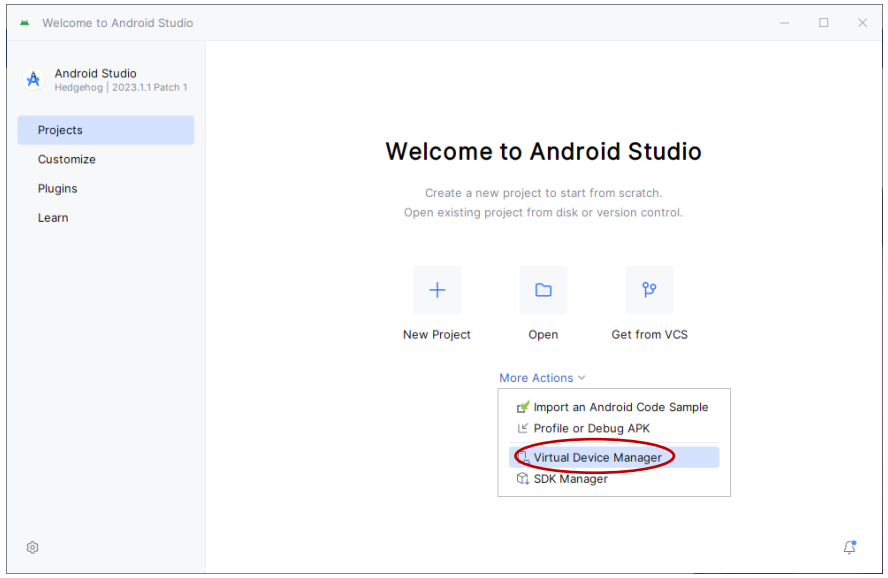
- The Android SDK is a set of development tools (debugger, emulator) and system images (Android versions) necessary for developing Android apps
- The SDK Manager is a tool that allows developers to download, update, and manage the Android SDK

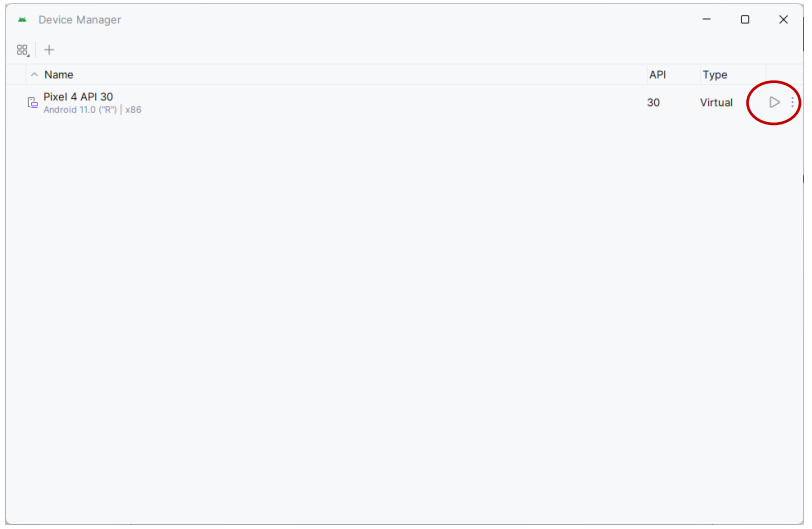




4. Android Studio - Virtual Device Manager

- An Android Virtual Device (AVD) is an emulator that allows us to model an actual mobile device
- The Virtual Device Manager is a tools that allows us to download and install different emulated Android virtual devices
  - AVDs can be phones, tables, TV, wearables, or automotive





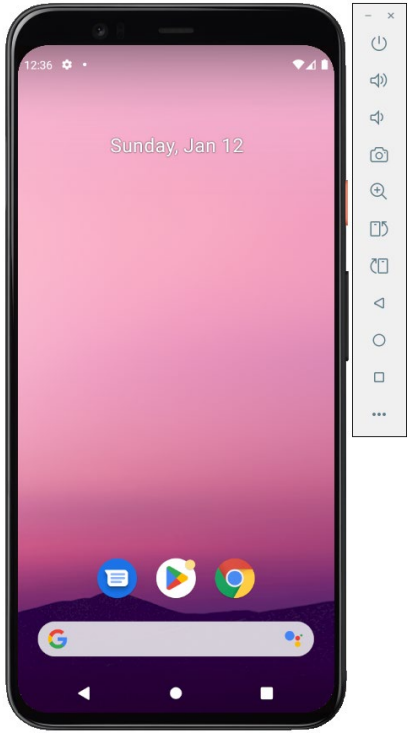


Table of contents

5. App components

- **App components** are the essential building blocks that define the structure and behavior of an Android app
  - We can see these app components as, modular pieces that work together to create a functional app
  - In other words, an Android app is the combination of one or more app components
  - App components are implemented using specific classes (Kotlin or Java) and configurations
  - Each component type has its own implementation pattern, and they must be declared in the `AndroidManifest.xml` file (except for dynamic broadcast receivers)

https://developer.android.com/guide/components/fundamentals

5. App components

- There are four different types of app components:

5. App components - Activities

1. **Activities**
   - Purpose: Represents a single screen with a user interface (UI). Activities are responsible for interacting with the user and handling UI-related tasks
   - Example: A login screen, a settings screen, or a details page
2. **Services**
   - Purpose: Runs in the background to perform long-running operations or tasks without a user interface. Services can continue running even if the user switches to another app
   - Example: Playing music in the background, syncing data with a server, or performing network operations

5. App components - Activities

3. **Broadcast receivers**
   - Purpose: Listens for system-wide or app-specific events (broadcasts) and responds to them. Broadcasts can be sent by the system (e.g., battery low, airplane mode) or by apps
   - Example: Responding to a low battery warning, detecting when the device is connected to wifi
4. **Content providers**
   - Purpose: Manages access to a structured set of data, enabling data sharing between apps. Content providers encapsulate data and provide mechanisms for defining data security
   - Example: Sharing contact information, accessing media files, or providing custom data storage for other apps

5. App components - Activities

We implement app components (e.g., activities) with Kotlin (or Java)

In this course will use **Jetpack Compose** to define the UI

The parent class in Jetpack Compose to implement activities is **`ComponentActivity`**

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

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

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

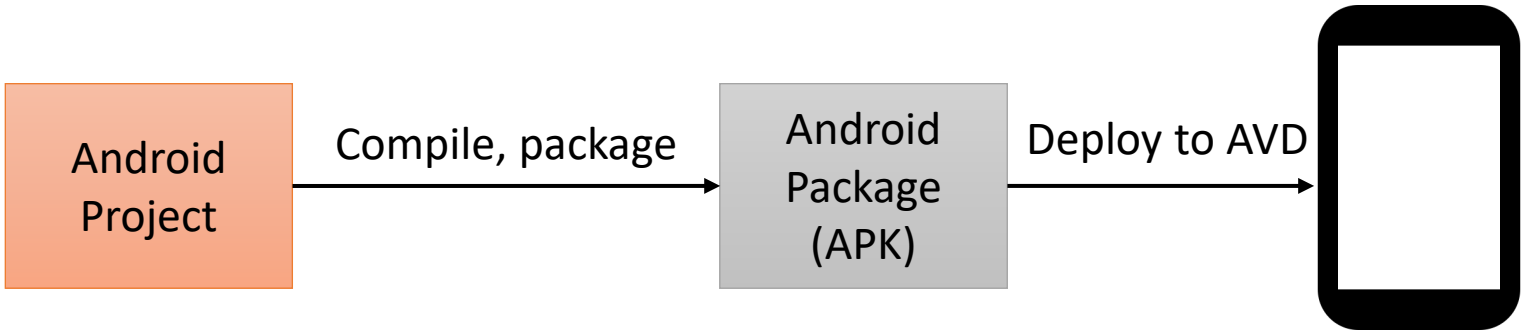
The activity lifecycle is Managed by the Android system through **lifecycle callbacks** (i.e., the methods `onCreate`, `onStart`, `onResume`, `onPause`, `onStop`, `onDestroy`)

Table of contents

6. Project structure - Gradle

https://gradle.org/

- Android Studio uses **Gradle** as build tool
  - Gradle is a popular build tool for Java/Kotlin
    - Another popular alternative of build tool for Java-based projects is Maven
  - Build tools are software utilities used to automate the creation of executable applications from source code
  - These tools ease the project management in terms of dependencies management, compilation, packaging, test execution, and deployment
- The simplified build lifecycle done in Android Studio with Gradle is:



An APK is an archive file (.apk suffix) which contains the contents of an Android app that are required at runtime

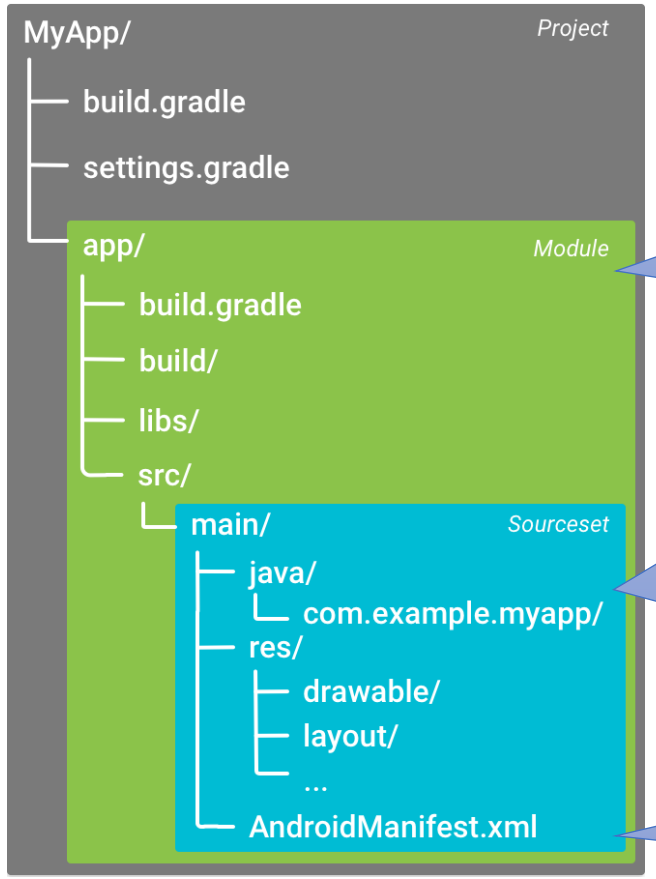
https://developer.android.com/build

6. Project structure - Gradle

- When starting a new project, Android Studio automatically creates the project scaffolding (i.e., the structure) automatically:

Each Gradle project contains a top-level setup files (`build.gradle` and `setting.gradle`). These files contain common configurations for all modules

Gradle modules can be seen as separate components within the same project. In an Android app, by default there will be a single module called app which contains the source code of the Android app



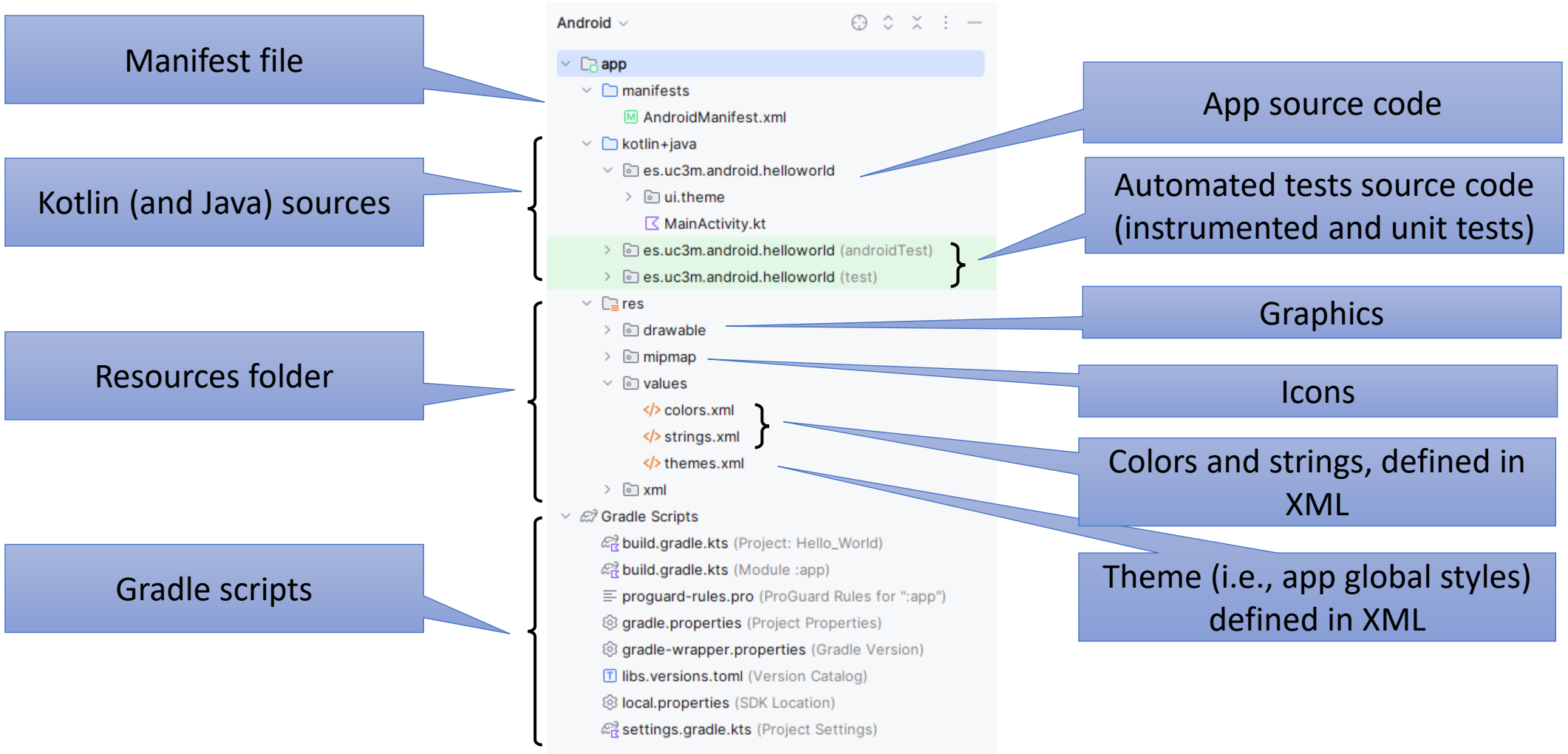
Each Gradle module has its own configuration file (`build.gradle` or `build.gradle.kts`)

The src folder contains the module source code. This folder contains the Java/Kotlin classes (*.java and *.kt files) and the resources (e.g., pictures and other Android setup files)

Manifest file (i.e., main configuration file for the app)

6. Project structure - Gradle

- A closer look to the structure of a Gradle project:



6. Project structure - The manifest file

- Every Android app project must have a file called **`AndroidManifest.xml`** file (with precisely that name)
  - XML (Extensible Markup Language) is a markup language for storing and transmitting arbitrary data
- This manifest file describes essential information about an app, such as the apps components, permissions, and application metadata
- This file (and the rest of the project scaffolding) is created by Android Studio when creating a new project

https://developer.android.com/guide/topics/manifest/manifest-intro

6. Project structure - The manifest file

- An example of manifest file create by Android Studio is as follows:

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

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.HelloWorld"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.HelloWorld">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

The tag `application` defines the app components (activities, services, broadcast receivers, and content providers)

The tag `activity` defines an activity

The tag `intent-filter` specifies the types of intents that an activity, service, or broadcast receiver can respond to. In this example:

- `android.intent.action.MAIN` : This activity will be the home page
- `android.intent.category.LAUNCHER` : This activity is listed in the app launcher

6. Project structure - The manifest file

- For basic apps, the previous manifest file (generated by Android Studio) can be simplified as follows:

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

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

</manifest>
```

6. Project structure - The build.gradle (app) file

- The file `build.gradle` is a configuration file used by Gradle to define how a project should be built
- In this file, we specify various aspects of your project, such as dependencies, tasks, plugins, and other settings
- In Android app, there are two syntaxes of defining this file:
  - Groovy (`build.gradle`). Traditional format
  - Kotlin DSL (`build.gradle.kts`). New format based on Kotlin aimed to improve the readability

Both formats achieve the same goal of defining and configuring the build process in Gradle. We will use Kotlin DSL in the examples

6. Project structure - The build.gradle (app) file

- Plugin section:
  - Gradle plugins are tools that extend the functionality of Gradle
  - For Android, we use the following plugins:
    - *android.application*: To build Android applications
    - *kotlin.android*: To use Kotlin for Android development
    - *kotlin.compose*: To enable Jetpack Compose (Android's modern UI toolkit)
  - The version of these plugins are located in a centralized location called the *version catalog* (`libs.versions.toml` file):

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.kotlin.compose)
}
```

build.gradle.kts

```
[versions]
agp = "8.8.0"
kotlin = "2.0.0"

[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
kotlin-compose = { id = "org.jetbrains.kotlin.plugin.compose", version.ref = "kotlin" }
```

libs.version.toml

6. Project structure - The build.gradle (app) file

- Android section. Relevant fields:
  - *namespace*: package name for the Java/Kotlin source code
  - compileSdk: Android SDK version used to compile the app
  - applicationId: unique ID of our app
  - minSdk: minimum Android API level required for running our app
  - targetSdk: highest API level that our app is aware

Common practice for SDK versions: decide which is minimum SDK we support (Android 7 in our case) and use the latest SDK for compilation and target

```
android {
    namespace = "es.uc3m.android.helloworld"
    compileSdk = 35

    defaultConfig {
        applicationId = "es.uc3m.android.helloworld"
        minSdk = 24
        targetSdk = 35
        versionCode = 1
        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
    }
}
```

build.gradle.kts

6. Project structure - The build.gradle (app) file

- Dependecies (external libraries that our project uses:):
  - *implementation*: main dependencies (i.e., required by the app)
  - *testImplementation*: unit test dependencies
  - *androidTestImplementation*: dependencies for instrumented tests (e.g., UI tests)
  - *debugImplementation*: debugging-only dependencies (e.g., UI tooling)

```
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)
}
```

build.gradle.kts

https://maven.google.com/
https://central.sonatype.com/

```
[versions]
coreKtx = "1.15.0"
junit = "4.13.2"
junitVersion = "1.2.1"
...

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx" }
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit", version.ref = "junitVersion" }
...
```

libs.version.toml

Table of contents

7. Takeaways

- Android is an open-source OS in which all its functionality is available to app developers through its Java Android API
- Android apps can be developed with Java or Kotlin
- Each Android Platform version (e.g. Android 7) uses a default version of the Android API (e.g. API 24)
- We use Android Studio as the development platform since it includes all the tools required for creating apps (IDE, SDK manager, AVD manager, Java compiler, debugger, …). Android Studio use Gradle for the project build
- An Android app is the combination of one or more app components: activities, services, broadcast receivers, content providers