Introduction to Android Mobile Application Development

Notes by José A. Espiño P. [1]

Spring Semester 2023–2024

[1] The content in these notes is sourced from what was covered in the course the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Introduction to Android

Mobile apps are installable software that run on a mobile device. As opposed to mobile websites, they use the device's hardware and software to provide a better user experience. The former are faster than mobile websites and can work without internet access, but are more expensive to build and maintain.
The most fundamental software for any mobile device is its operating system. It coordinates the communications between the hardware and software of a mobile device. The OS typically starts up when the device powers on and provides a user interface (UI) to interact with the device. Android OS is based on a modified version of the Linux kernel and is open–source. It gives users a multitouch experience, which means users interact with the device via touch gestures. The openness of Android has made it be popular not only with phones and tablets, but also with smartwatches, TVs, cars, and other devices. Some powerful features of the Android OS are

- NFC: Near Field Communication, which allows devices to communicate over short distances.
- Wi–Fi: Android provides in–built tech that supports Wi–Fi Direct, which allows devices to connect to each other without a Wi–Fi network. Other modes are hotspot and tethering.
- App Downloads
- Custom ROMs: Android devices can run customised versions of the OS called ROMs.

The Android OS architecture is made up of five sections:

- Linux Kernel: This layer provides a level of abstraction between the hardware and the rest of the software stack. It also provides the necessary hardware drivers to interact with the hardware.
- Platform Libraries: This layer provides the necessary libraries for the Android OS to run. It includes the Android Runtime (ART), which is the runtime environment for Android apps. It also includes the WebKit, which is the engine that powers the web browser.
- Android Runtime: This layer is where the apps run.

- Application Framework: This layer provides the necessary APIs for the Android OS to run. It includes the Activity Manager, which manages the lifecycle of the app, the Content Providers, which provide access to data, the View System, which provides the UI, and the Notification Manager, which provides notifications to the user.
- Applications

There are two main languages used to develop Android apps: Java and Kotlin. Java is a general–purpose programming language, used not only for Android development, but also for web development, embedded systems, etc. Kotlin is a modernised and more concise version of Java — and it is the preferred language for Android development. In comparison to Java, Kotlin is simpler, more expressive, and more powerful. It is also fully interoperable with Java, which means that you can use Kotlin and Java in the same project.
When developing an Android app, there are certain concepts you need to keep in mind:

- Android Software Development Kit (SDK): This is a collection of tools and libraries that you need to develop Android apps. It includes the Android Studio, which is the official IDE for Android development, and the Android Emulator, which is a virtual mobile device that runs on your computer.
- Top Level Component: This is the main building block of an Android app. It includes the Activity, which is the entry point of the app, the Service, which runs in the background, the Broadcast Receiver, which listens for system–wide events, and the Content Provider, which provides access to data.
- Activity Components: The content the users can interact with on the screen. These are the only components that deliver interactive content to the user. Although an app can provide multiple activities, many developers follow a *single–activity* architecture pattern when creating their apps. This means that the app has only one activity, and the rest of the content is delivered through fragments.
- Android Views: Only occupy a rectangular area on the screen and are responsible for drawing and event handle. A combination of these views form a design interface.
- Android Layout Files: These are XML files that define the layout of the app. They are used to define the UI of the app. Interfactes could be created with Android Views entirely in code using Kotlin or Java; Google has even created a new way of creating interfaces called Jetpack Compose, which is entirely in Kotlin code, without XML.
- Project Files: Files that make up the app. Configuration files define the project structure, code files provide the logic, and resource files provide everything else.

An Android app is made up of four main components:

1. Activities: These are the entry points of the app. They provide a window where the app draws its UI. An app can have multiple activities, but one of them must be marked as the main activity. This is the activity that is launched when the app is started. Each activity is independent of the other.

2. Services: These are background processes that run without a UI. They are used to perform long–running operations or to perform work for remote processes. They can be started and stopped, and can run in the background even if the app is not running.

3. Broadcast Receivers: Respond to messages from other applications or the system in real time.

4. Content Providers: Shares data between applications based on **requests**.

All of these components are bundled together through the Android Manifest file, an XML file that defines the major components of the app as well as the minimum requirements the app has to run.
XML, or Extensible Markup Language, is a way to created structured data and distribute it over the internet. It is a markup language, which means it is used to define the structure of a document. The structure of an XML document is built on **tags**, which are used to define the data. In order for an XML document to be valid, the document must be well formed and must comply with the rules of the XML syntax. The most basic rule of this syntax is that for every opening tag, there must be a closing tag.

2 Emulation and Development

An emulator is a computer program that is designed to replicate a different device. They are used to test software on different devices without needing different hardware. BlueStacks is a popular free emulator that allows you to use Android apps on Windows or Mac. Android Studio has a feature named Android Virtual Device Manager, or AVD that allows you to define features of a physical device and test your app on it. You can access it by opening the Android Studio interface, clicking on the upper right corner, click more actions, and then virtual device manager. You can then create a new virtual device. Then, you can download an Android Emulator by clicking on SDK manager from the Android Studio welcome page. Then, you choose a specific Android Emulator to download, install it, and click apply. An Android emulator needs a virtualization tool for hardware acceleration; Windows 10 comes preloaded with Hyper–V whereas macOS and Linux need to install KVM or HAXM.
In a phone, the CPU is like a translator between software and hardware. It translates high–level software instructions and translates them into hardware instructions. Most smartphones have one of three main architectures: ARM (ARMv7 or armeabi), ARM64 (AArch64 or arm64), and x86 (x86 or x86abi). Of these three, ARM is the most common and is optimised for battery usage; it also has simple instruction sets. x86 is more powerful, but it is quite punitive with the battery.
A very important tool you will need to use is an **OS image**. Android OS images adjust versions or copies of Android OS that you can run on your emulator, thus being able to develop apps for every version of Android. The way you can access different OS images is by clicking the three dotted menu icon on the upper right–hand side of the Android Studio welcome page, then click on Virtual Device Manager, click on Create Device, and then select the OS image you want to install onto the device.

In order to successfully use an emulator for testing and troubleshooting, you need to configure it adequately. You can achieve this by using the AVD manager; in Android Studio in particular you can specify the interaction between the development computer and the emulator. Here, you can specify some important AVD properties, such as the device type (AVD name), startup orientation (either landscape or portait mode), emulated performance (number of processes from your computer's CPU that the emulator can use), RAM in memory and storage. Although Android Studio provides predefined devices you can emulate, you can create your own by clicking on create device once you open the VDM.
There are several libraries and packages available for Android Studio developers to implement functions without having to code them from scratch. Some common ones are:

- Fresco: Image loading library that provides a smoooth scrolling experience when an image is loading. It uses smart caching to minimise storage overhead.
- ExoPlayer is a media player library. It is more customisable and robust than the Android local MediaPlayer APIs
- Retrofit is a networking library that allows you to make internet calls within your application

When you are developing an app, you need to make sure that all its project folders are well–structured, since each folder contains a separate part of the entire project. As soon as you start a project, some files and folders are generated by default:

- Sample App folder: This is the root folder of the project.
- .gradle folder: Gradle refers to the build toolkit that guides the project. It contains all the configurations and files used for project building.
- .idea folder: Used by Android Studio to store specific project metadata. In this folder, there is a set of configuration files containing configuration for specific functional areas. Some examples are *compiler.xml*, *encodings.xml*, and *modules.xml*
- app folder: Contains the source code for your app. This is where you write your code, create the UI, and store assets such as images and fonts. External Libraries are also displayed here.
- gradle folder: Gradle refers to Android's build system, a set of tools made available for developers to build, test, and run their applications. Whenever you build/compile a project, build related data is generated and stored in this folder. The gradle folder has some subfolders, such as `.gitignore`, which is where you specify sensitive files and folders that you may want to exclude from repository systems such as GitHub, `build.gradle` which is a build file that specifies and manages configuration options common to all sub–project folders, `gradlew`, which is only created once and is updateed whenever a new feature or plugin is required to build a project, `local.properties`, which is a file that contains information specific to your local configuration (do NOT push onto GitHub or similar), and `settings.gradle`, which handles various settings for projects and modules

The activity class is an essential part of any Android app — no matter how small or big the app, it will at least have one activity class. This class is automatically generated upon creating the app in Android Studio. An activity is an entry point for interacting with the user, a way for the user to interact with your app. An app normally is made up of several activities, referred to as "screens" that together form the overall app usage experience. Activities are represented by an activity class whose function is to respond to user input. Each activity class contains a layout that holds different pieces of a UI together so that the user can interact with the app. This means that the activity class is the gateway through which the user can interact dynamically with an app's UI. One activity in the app will be specified as the main activity, and will be the first screen that a user sees when they launch an app.
When you create a new Android project using the main activity template in Android Studio, a file named `MainActivity.kt` is generated. Within this file, different parts of Kotlin code are displayed. The main class, called MainActivity, displays different functions. A useful function is called `onCreate`, which defines the view the user will have upon opening the app and initialises the actions of the activity. This function is essential in order to run an Android app. Other functions, such as the actions the users will perform on the activity screen, follow this function.
Gradle is a build system that automates the building process through a set of build configuration files that define how a project is to be developed, what dependencies are needed, and what the result of the compilation should be. For every Android project, two *build.gradle* files are generated: one for the entire project and one for the app module. A module is a collection of source files and build settings that allow you to divide your project into different units of functionality. Each module hsa its own *build.gradle* file. A *build.gradle* file may contain:

- Android block: contains information about your project such as the minimum OS version
- Default config sub–block: contains information about the app such as the app's name, version, and the minimum SDK version
- Dependencies block: contains information about the libraries and packages that the app depends on. If a local copy is unavailable, it will automatically download the packages from the internet.

You can get full control over Gradle via the command line: the command `.gradlew/ build` will build the project and `.gradlew/ clean` will clean the project by deleting the contents of the build folder automatically generated. Lastly, `./gradlew wrapper` shows all the Gradle operations running in the background.
Let us now talk about the **Android Manifest**, a file automatically generated by Android Studio every time you create a new project. This is an .xml file that contains information about the app and that will allow you to define the permissions that your app needs in order to access protected parts of the operating system or other apps. This file contains sections such as the **permission tag**, which allows you to specify the requests of the application and the **application tag**, which specifies the app's name, icon, theme, and other UI elements. The **activity tag** specifies the main activity of the app, and the **intent filter tag**, found within the activity

tag, specifies the actions that the app can perform. The intent filler is the one that specifies the main activity as the entry point of the app.
Android development uses the term **resource** to refer to any non–Java file that your app needs. Resources are stored in the *Resource* folder and can be referenced in any part of the code. Some common resources are:

- String: A string resource is a simple string that is stored in the *res/values/strings.xml* file. It is used to store text that is displayed to the user. Each entry is a key (ID of the text) and a value (the text itself).
- Color: A color resource is a simple color that is stored in the *res/values/colors.xml* file. It is used to store colors that are used in the app's UI.
- Dimension: A dimension resource is a simple dimension that is stored in the *res/values/dimens.xml* file. It is used to store dimensions that are used in the app's UI.
- Font: A font resource is a simple font that is stored in the *res/font* folder. It is used to store fonts.

The concept of resource provides apps with portability, since you can update them without having to change the code itself. The **res** folder is generated by default upon creating a new project in Android Studio. Some subfolders contained in the res folder are **drawable** and **mipmap**. Although both folders are used to store images, the latter is an improvement of the former. There are several mipmap subfolders, each of which contains images of different resolutions; the application will display the most appropriate one based on the resolution of the phone being used, which guarantees the best possible image quality.
Another set of folders generated by Android Studio is the **layout** folder. This folder contains the XML files that define the user interface of your application. XML is preferable over code because it is separated from the code—which makes it easier to write apps that run on different devices. The layout folder contains the **activity_main.xml** file, which is the default layout file for the main activity. On it, you can check the layout of the app and dynamically change it, either via the code or the **design view**.

3 Building the App

Project planning is an essential step of the development process; good and clear planning can significantly decrease the planning time for every other step of the process. The first step of the planning process is to define the app's purpose and target audience. Then, you need to break the app down into principal components of functionality. This step of the process involves **requirements gathering**, which involves asking yourself questions about what *specifically* you want the app to do, thus narrowing down the platforms you want to use to develop the app, the libraries, the tools, and other aspects that you will need as you go. Once you know the requirements of the app, you could try to find a template that suits your needs. This can shorten the development time and give you a good headstart. Using the basic activity template for instance provides you with base UI components that you can build upon

with your specific actions.
UI is an important aspect of any app—it is a big factor that makes your app stand out from alternatives in the market. We can control the UI of an app in Android Studio by using **layouts**. An activity is a screen of the app that contains several UI objects (also known as **views**); a layout is an XML file that defines the structure of those UI components. The **view class** is the base class for all the UI components in Android. A special use of this class is the **ViewGroup** class, which can contain one or more other views.
The View object is created from the View class and occupies a rectangular screen area. It is responsible for drawing and event handling. Some of the most used Android Views are **TextView**, **EditText**, **Button**, **ImageView**, **ImmageButton**, **CheckBox**, **RadioButton**, **Recycler View**, **Date Picker**, and **Spinner**. The size of a view can be manually set or can be set to either occupy all the available space in its parent ViewGroup (match_parent) or to occupy as much space as it needs(wrap_content). Every view in XML has the following format:

```
<ViewName
    Attribute1=Value1
    Attribute2=Value2
    ...
    AttributeN=ValueN
/>
```

Two attributes that are required for every View are **android:layout_width** and **android:layout_height**. The former specifies the width of the view, and the latter specifies the height of the view. The value of these attributes can be either **match_parent**, **wrap_content**, or a specific size.
Some layouts available in Android Studio are:

- Linear Layout: This layout arranges its children in a single direction, either vertically or horizontally. It is the simplest layout available in Android Studio and is an extension of the **ViewGroup** class.
- Constraint Layout: ViewGroup subclass that allows you to position child views flexibly using constraints. This means that the position of child A can be defined in relation to child B, and so on.
- Android RecyclerView: Displays data that can be scrolled. The item views inside it are positioned using **layout managers**, which are responsible for measuring and positioning item views.
- Android WebView: It is a subclass of **ViewGroup** and is used to display web pages in the app.
- TableLayout: A layout that arranges its children into rows and columns.
- FrameLayout: A layout that displays a single view. If you want to display multiple views, you can use the **ViewGroup** class.

Even when using a template, you very often have to manually create the layout resource file for the activity. This is done by right–clicking on the **res** folder, clicking on **New**, and then clicking on **Layout Resource File**. You then specify the name of the file and the root element

of the layout. Once you have created the layout file, you can open it and start adding views to it. You can do this by dragging and dropping views from the **Palette** pane to the layout file. You can also add views by typing the XML code directly into the layout file.
Resources, which are images, videos, and other files, are used extensively in apps. That is why Android Studio has an app exclusive to these: the **Uniform Resource Identifier** (URI) class. It identifies a resource by either location, name, or both. After creating an **Android Resource Directory** and adding **raw** files to it, you can access the resources by referencing them within your code. An example of that would be:

```
    // Get a reference for video "capybara_shiba.mp4" from the raw folder
    val uri: Uri = parse(
        uriString = "android.resource://" + packageName + "/" + "
    capybara_shiba"
    )
```

The **parse** function is used to parse the URI string and return a URI object. The **uriString** parameter is the string to parse. The **packageName** parameter is the name of the package that the resource is in. The **capybara_shiba** parameter is the name of the resource.
Imports can make development a lot simpler by making it possible to reuse already existing code to perform specific tasks. The **import** statement is used to import a class or a package into your code. You can import a class or a package by using the **import** keyword followed by the name of the class or package you want to import. You can also use the **import** statement to import all the classes in a package by using the * wildcard. For example,

```
    import android.R
    // The R class is used to access resources in the app
    import android.net.Uri
    // The Uri class is used to identify a resource by location, name, or
     both
    import android.net.Uri.*
    // The parse function is used to parse the URI string and return a
    URI object
    import android.widget.MediaController
    // The MediaController class is used to control media playback
    import android.widget.VideoView
    // The VideoView class is used to display videos
    import android.os.Bundle
    // The Bundle class is used to pass data between activities
    import androidx.appcompat.app.AppCompatActivity
    // The AppCompatActivity class is used to create an activity
    import androidx.navigation.ui.AppBarConfiguration
    // The AppBarConfiguration class is used to configure the app bar
    import com.example.myapplication.databinding.ActivityMainBinding
    // The ActivityMainBinding class is used to bind the activity to the
    layout. myapplication is the name of the project itself.
```

Variables can be used to enable developers to capture user inputs, among other things. They can be declared in Kotlin using either the keyword **val** (for an immutable variable) or **var** (for a mutable variable). Common data types are **Int**, **String**, **Boolean**, **Float**, and **Double**. For example,

```
    val name: String = "Jose"
```

```
    // This is an example of a variable declaration. The variable name is "name" and its type is String. The value of the variable is "Jose"
    var age: Int = 21
    // This is an example of a variable declaration. The variable name is "age" and its type is Int. The value of the variable is 21
    println("My name is "+name+" and I am "+age+" years old")
```