Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE GRAU

TÍTOL DEL TFG: A study of Kotlin's: conciseness, safety and interoperability

TITULACIÓ: Grau en Enginyeria Telemàtica

AUTOR: Andreas Luca Crisan

DIRECTOR: Roc Meseguer Pallarès

DATA: 23 d' octubre del 2019

Títol: A study of Kotlin's: conciseness, safety and interoperability

Autor: Andreas Luca Crisan

Director: Roc Meseguer Pallarès

Data: 23 d' octubre del 2019

Resum

En l'últim esdeveniment del Google I/O, la conferencia més important del mon Android, que va tenir lloc al Maig del 2019, es van donar dos anuncis importants. El primer anunci va ser que hi ha més de 2.5 bilions de dispositius Android actius. I el segon indicava un canvi en quant al llenguatge preferit per desenvolupar aplicacions per als dispositius Android. Passa de ser Java, el llenguatge per defecte des del llançament del sistema operatiu Android al 2008, a ser Kotlin, un nou llenguatge de programació desenvolupat per JetBrains. Els dos anuncis criden molt l'atenció, perquè canviaria Google el llenguatge preferit per desenvolupar aplicacions Android quan tenen tants milions d'usuaris als quals els hi funciona perfectament?

L'objectiu principal d'aquest treball de fi de grau es donar una resposta a aquest pregunta avaluant Kotlin. Començarem examinant les característiques que ofereix, que van des de escriure codi concís a revolucionar el món de la programació asíncrona. També mirarem quin tipus de projectes es podem fer amb Kotlin, com per exemple aplicacions mòbils, servidores o fins i tot pagines web.

La primera avaluació serà teòrica, mirant que problemes pràctics resolen les característiques de Kotlin. Després escollirem una sèrie d'aquestes característiques i experimentarem amb elles, avaluant si aquestes compleixen la teoria i proposarem alguna millora a aquestes.

Una vegada familiaritzats amb Kotlin construirem una aplicació Android sencerament amb Kotlin. L'aplicació es connectarà amb un servidor per agafa dades de bitllets d'avió per a un determinat numero de persones amb diferents aeroports de sortida a un mateix aeroport d'arribada. D'aquesta manera podrem crear grups de viatge i buscar vols per a cadascun.

Per últim, tenint tot el que s'ha après durant el projecte, es validarà tant teòricament com pràcticament l'afirmació, trobada a la pàgina web oficial de Kotlin, que descriu Kotlin com a un llenguatge concís, segur i interoperable amb codi existent.

Title: A study of Kotlin's: conciseness, safety and interoperability

Author: Andreas Luca Crisan

Director: Roc Meseguer Pallarès

Date: October 23rd, 2019

Overview

In the latest Google I/O, Google's major conference on the Android world, that took place in May 2019, they stated two huge announcements. The first one is that there are already more than 2.5 billion active Android devices worldwide. And the second one is that since the launch of Android in late 2008 the preferred programming language for developing Android application has been Java, but this year, 2019, this changed. Kotlin, a new programming language developed by JetBrains, took its place. Both statements are huge, why would Google change its preferred programming language for Android development when they have that impressive number of active devices?

The goal of this project is to answer that question by evaluating Kotlin. We will first deep dive into its main features which go from writing concise code to revolutionizing asynchronous programming. We will also look at what can we build with Kotlin, which goes from mobile applications to servers or browser applications.

The first approach will be theoretical by researching what problem do Kotlin's features solve and how to use them. Then, we will move to select the most relevant features and we will experiment with them. In the experiment we will see if what the theory promises is true and at the end of evaluating these features, we will give some analysis or proposals for improving it.

Once we are more familiar with Kotlin we will build an actual Android application fully in Kotlin. The application will connect to a server and look for flights for a given group of people from different departure cities to a single destination. Therefore, we will have the possibility of creating travel groups of people and the possibility to look for flights for each of those.

Finally, we will take everything into consideration, and we will validate Kotlin's self-claim of being a concise, safe and interoperable language from both the theoretical and the practical points of view.

INDEX

INTRODUCTION

This paper aims to analyze Kotlin, a new general-purpose programming language, and evaluate its features and performance in Android application development. In Kotlin's official webpage, found in [1], Kotlin, is described as a programming language that is concise, safe, interoperable and tool-friendly, consequently in this paper we will focus on verifying these claims.

The Kotlin project started in 2011 but the first version wasn't release until 2016. Kotlin is open source and the Kotlin Foundation, composed by JetBrains and Google, supports its development. Kotlin is fully interoperable with Java, which means you can call Kotlin code from Java and vice versa. Kotlin has some great headlines. One year after its first major release, in 2017, it was added to Android Studio as a new language to build Android applications with. Two years later it was selected as the default language to build Android applications in Android Studio, the program in which developers use to write Android applications.

Kotlin, also claims to offer lots of new features and a modern language that made the Android team change Java which has been in the Android application world for 10 years. With Kotlin, Android application development is easier, faster and more robust.

The love that the development community has given to Kotlin and the amount of active Android devices, 2.5 billion, as stated in the last Google I/O, the major Android event that took place in May 2019 made me wonder why would they move from a language they are fluent as far as coding is concerned to a new language.

Motivation of the project

We are in the mobile-first era. When you launch your business you first think on attracting clients. This leads to thinking what the most likely device in which people will see their product is. According to the Guardian[1] we spend more than 3 hours and 15 minutes with our smartphones. For that reason, marketers centre their offered service in mobile application solutions. If a website is built, the first developed view frame is the one for smartphones and then the one for computer displays. Speaking about smartphones, Android is the operating system that owns most of the smartphone market. Therefor if you want your business, game, solution, whatever to be visible to most of the people, you need an Android application.

An Android application needs to be coded with a programming language. Almost every few months a new programming language appears, there are specific programming languages for a certain platform, or a certain task, but there are also general-purpose programming languages. Most of this programming

[1] The Guardian's article: https://www.theguardian.com/lifeandstyle/2019/aug/21/cellphone-screen-time-average-habits

languages vanish after some months or years. The reasons behind the failure of this programming languages normally are:

- The lack of documentation or sample code, the learning curve is bad.
- There already exists a better programming language for that purpose.
- The language is not maintained regularly, either because it is too complex, and it lacks people to work on and that leads in having a not robust programming language.

Therefore, how could Kotlin, a general-purpose programming language that is not mainly focused in just the mobile application world, officially take Java's place in Android development[2]?

Objectives

If we had to sum up the main goal of this project, it would be to evaluate in a theoretical and in a practical way the performance of Kotlin for application development. In other words, to validate if Kotlin is a safe, concise and interoperable language.

In order to achieve the global goal, we will set specific objectives that will guide the process of evaluating Kotlin for application development:

- Get a general overview on what is Kotlin, how is it build, how does it compile the code and its purposes.
- Research the features that Kotlin offers and analyse them in a theoretical way to see what problem they solve.
- Be fluent in writing and understanding Kotlin code. Also, being able to have the researched features in mind when starting a Kotlin project in order to make it with the best matching features available by doing proofs of concept with some of the researched features to see their actual benefit in a practical example.
- Learn about application architectures in general so that our code is more professional and for getting the feeling on how production applications are built.
- Build an entire Android application with Kotlin, so that we put everything learned together in one place, from the Kotlin features to application architecture. The Android application that we will build will consist in creating groups of people and finding flights for each member of the group for a given flight information.

Consequently, to achieving all the specific goals we will get the background of developing a full application in Kotlin so we will be able to extract a conclusion on why using Kotlin for Android development is a good idea.

[2] Google I/O 2019 it was announced Kotlin as the preferred Android application development language: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/

Project structure

This thesis will be divided into four main chapters. The first chapter will introduce Kotlin, we will look in a general scale what does it offer and what can we build with Kotlin. Then we will move to how does Kotlin code get compiled as there is not just one compilation target but more. This chapter is key for understanding the basics through a behind the scene of the compilation process for each compilation target. Then, we will end the chapter by giving several arguments on each compilation target and an explanation on why we will focus on Kotlin for Android development.

By narrowing the use case of Kotlin, in the second chapter we will research in a theoretical way the most remarkable features that Kotlin has to offer in Android development. For each feature an introduction will be given and then it will be followed by an explanation on what problem does this feature solve. The chapter will end by giving a series of key concepts that summarize the main benefits of using Kotlin for Android application development.

Once we have a theorical idea of what Kotlin is, what can we build with it and which are its main features, in chapter 3 we will get hands on the most relevant and attracting features for seeing their practical and real benefits. We will start with a small demonstration of Kotlin code that we will keep growing within each newly test feature. For each examined feature we will end up concluding some of their practical benefits and some proposal for enhancing these features, or a list of things to keep in mind when putting that certain feature into practice.

In the last, we will build a real-world Android application. We will study the most used Android application architectural patterns and choose the one that best fits our needs. We will also connect to a production server in order to retrieve real data and give the application a practical use. The chapter will end with a conclusion of the issues found during the development of the application and some proposals for improving the application.

After having gone through all the chapter we will close the thesis by giving an answer to the initial question, which can also be asked as: why choosing Kotlin for Android development? And we will also give some recommendations for starting Android applications with Kotlin.

CHAPTER 1. KOTLIN

1.1. What is Kotlin?

You may have or may have not heard about Kotlin, but according to StackOverflow's annual survey[3] it is the second most wanted and loved language in 2018, which means that if you sneak into a conversation between two android developers "Kotlin" is a word you will hear for sure.

Kotlin is a new programming language created by the JetBrains team, known by developing some of the most popular Integrated Development Environments (IDE), basically the computer programs used to write code, such as IntelliJ IDEA, Android Studio.... Everything started in 2011 but it was not very known before two events that made it really popular. The first one had place in 2016 when the first major version was released. And the second one was on May 2017 at Google's I/O, the biggest android event worldwide, where Kotlin was said to be the preferred Android development language.

Kotlin is popular, it is already being used by big tech companies such as Google, Netflix, Trello, Amazon or Uber. Kotlin's growth since the beginning is exponential, in the last Kotlin conference (KotlinConf) that took place in October 2018 they stated that Kotlin has more than 100 million lines of code in GitHub (multiplying by four lasts year's 25 million lines of code), which is a pretty outstanding stat taking in consideration that it was launched in late 2013, but as mentioned before only started to be popular in 2016. Even though, until Kotlin 1.3 was released in the middle of 2019, it was not that robust yet.

As you can deduct from the previous comment there are already a lot of Kotlin conferences and if you attend an Android Conference such as DroidCon most of the talks, not to say all, will be focused on Kotlin for Android development.

Let's define what Kotlin is more technically and describe some of its features. Kotlin is an open sourced, statically typed general-purpose programming language created not to be cross platform but multiplatform, which basically means that it's a programming language meant to be used in everything for everything. From developing a mobile app or a server to a script and even for creating domain-specific language (DSL) projects.

As all programming language, Kotlin needs to be compiled or interpreted. As mentioned before, it's cross platform so it compiles in different ways depending on what do we want to use it for. It can be compiled into Java bytecode which will run in the JVM or it either can be compiled it as a native binary, that can be executed in an iOS device for example, or it can even be transcompiled into JavaScript.

Each of the mentioned compilation ways has its purpose, as you can see in Figure 1.1, a different compilation environment and procedure which will be described in the next section in more detail.

[3] 2018 StackOverflow's annual survey: https://insights.stackoverflow.com/survey/2018

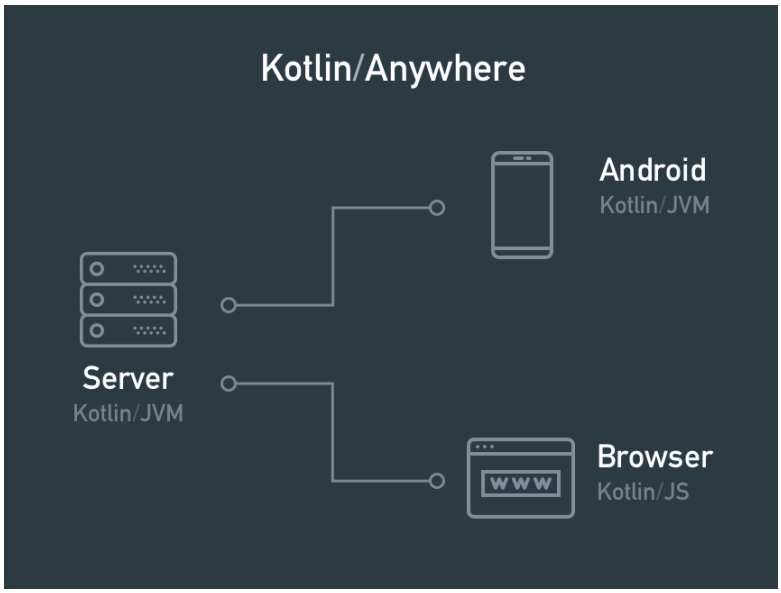

Fig. 1.1 Kotlin can be used anywhere

1.2. Compilation targets

Kotlin's long-term aim is not to be "the next Java" or what people thinks when they hear about it for the first time: "Yet another cross-platform framework" (such as: React Native, Flutter, Cordova...). It's much more than that, as said in the beginning its aim is to be on every platform, from servers, apps, desktop apps, executables to web browsers, that means everywhere. This means Kotlin is not only to be intercompatible with Java, but also with JavaScript and C/C++, see Figure 1.2.

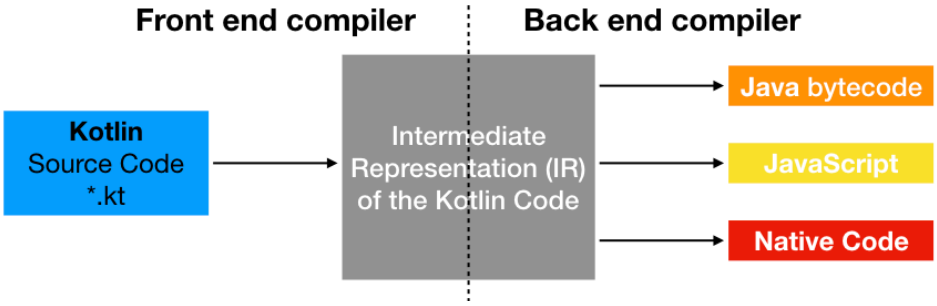

Fig. 1.2 Common compilation process

In order to go deeper in the compilation process of the different targets we will analyse the following quote from Kotlin's official FAQ when asked what does Kotlin compile down to:

"When targeting the JVM, Kotlin produces Java compatible bytecode. When targeting JavaScript, Kotlin transpiles to ES5.1 and generates code which is

compatible with module systems including AMD and CommonJS. When targeting native, Kotlin will produce platform-specific code (via LLVM).”

1.2.1. Java bytecode

Let’s start with the most popular compilation target: Java bytecode that runs on the Java Virtual Machine (JVM).

In order to understand how Kotlin code is compiled into Java bytecode we need to review the basis on how Java code is compiled into Java bytecode, see [3]. The process is straight forward, we have the Java classes of our application with their variables and methods. Then we call the Java compiler (*javac* that can be found in the JDK, Java Development Kit) which for each Java class will create a class file (*javaClassName.class*) that contains the Java bytecode, as you can see in Figure 1.3, usually this .class files are put together in a Java Archive known as a *jar* file, it also includes metadata files. Now that we have the jar that contains the Java bytecode of our application, we just have to execute it in the JVM.

Although the process is simple, we need to keep in mind that this jar that contains our code has dependencies on the core Java libraries. All Java applications have these dependencies, so they are put all together in the JVM, so we don’t have to worry about adding them. In our computer when we install Java, we concretely install the Java Runtime Environment (JRE), which includes the JVM, Java core libraries, etc...

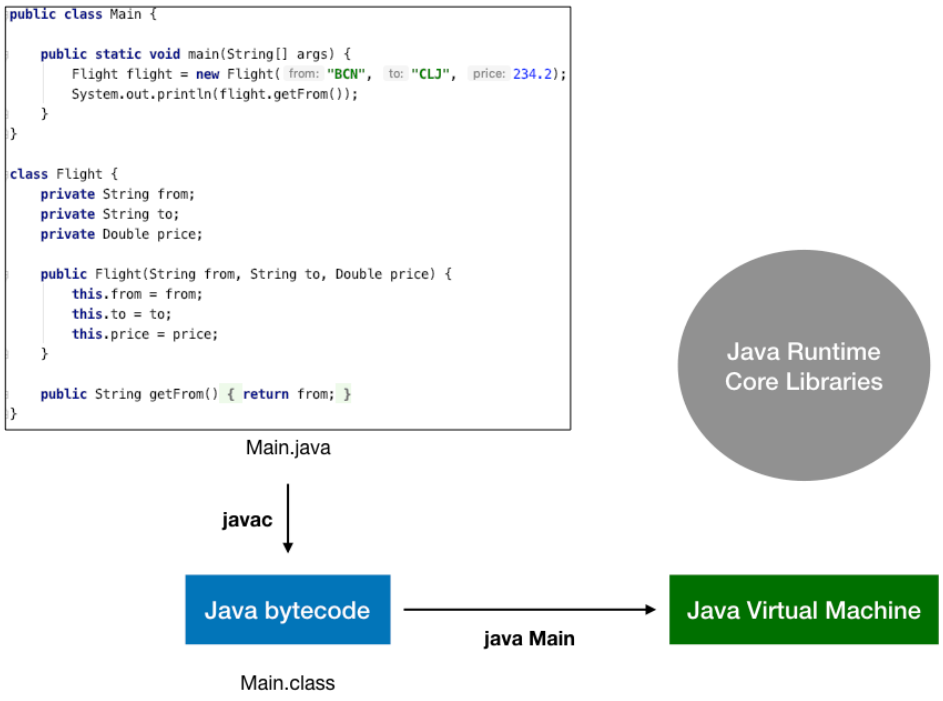

Fig. 1.3 Java code compilation process

When we compile out Kotlin application into the Java bytecode the process is very similar. We write our Kotlin classes with its respective variables and methods and then we call the Kotlin compiler (*kotlinc*) that produces the .class files containing Java bytecode that are put together in a jar file. Up until here it's the process is identical (except for the fact that now we are using the Kotlin compiler) as you can see in Figure 1.4. The problem is that unlike the Java runtime decencies are included in the JVM, currently the Kotlin standard libraries are not included, so whenever we call the JVM to execute Java bytecode (coming from the Kotlin compiler) we need to specify this Kotlin dependencies.

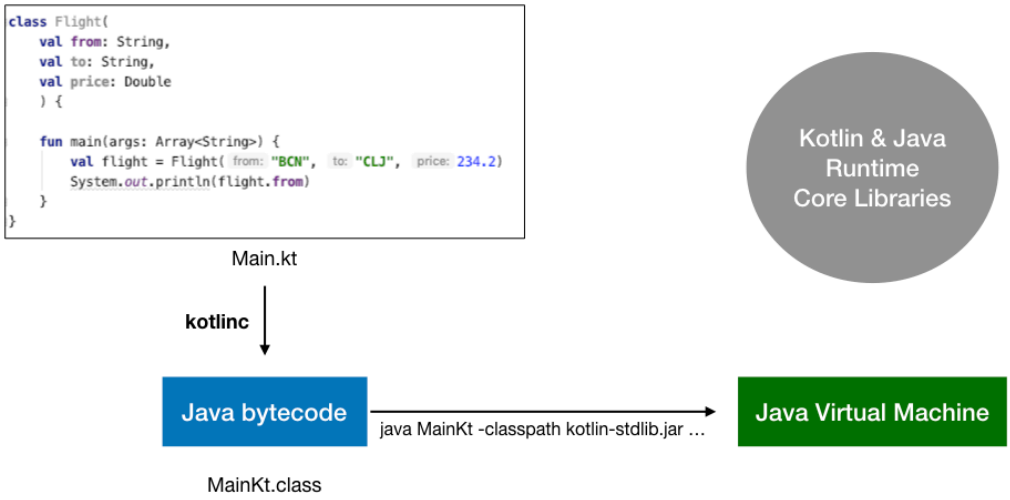

Fig. 1.4 Kotlin code compilation process

Therefore, as Kotlin is compiled into Java bytecode it can be used in a large variety of places where java is the main development language such as Android apps, Java servers using Spring Boot to build RESTful web services, scripts, or even for building Serverless Architectures in Amazon Web Services (AWS) lambdas.

1.2.2. JavaScript

When it comes to Kotlin JavaScript the word used instead of compilation is transpilation, which is a shortening of transcompilation. Transcompilation is the art of translating the source code written in a specific programming language into another programming language.

You can transpile your Kotlin code into JavaScript either by using a build system such as gradle or maven, command line or even from IntelliJ, which enables you to debug.

Whenever Kotlin is transpiled into JavaScript the outcome of the operation are three files. The first file is kotlin.js, this file will always be the same for a certain Kotlin version, it contains the standard library and the runtime library, the second file is basically your Kotlin source code transpiled into JavaScript as you can see

in Figure1.5. This transpilation of your main code is human-readable and you can understand it easily. And the last file is the one containing metadata.

When building code that interacts with the DOM, the JavaScript files need to be included in the html code. The first one to be included is the Kotlin standard library and then your module file.

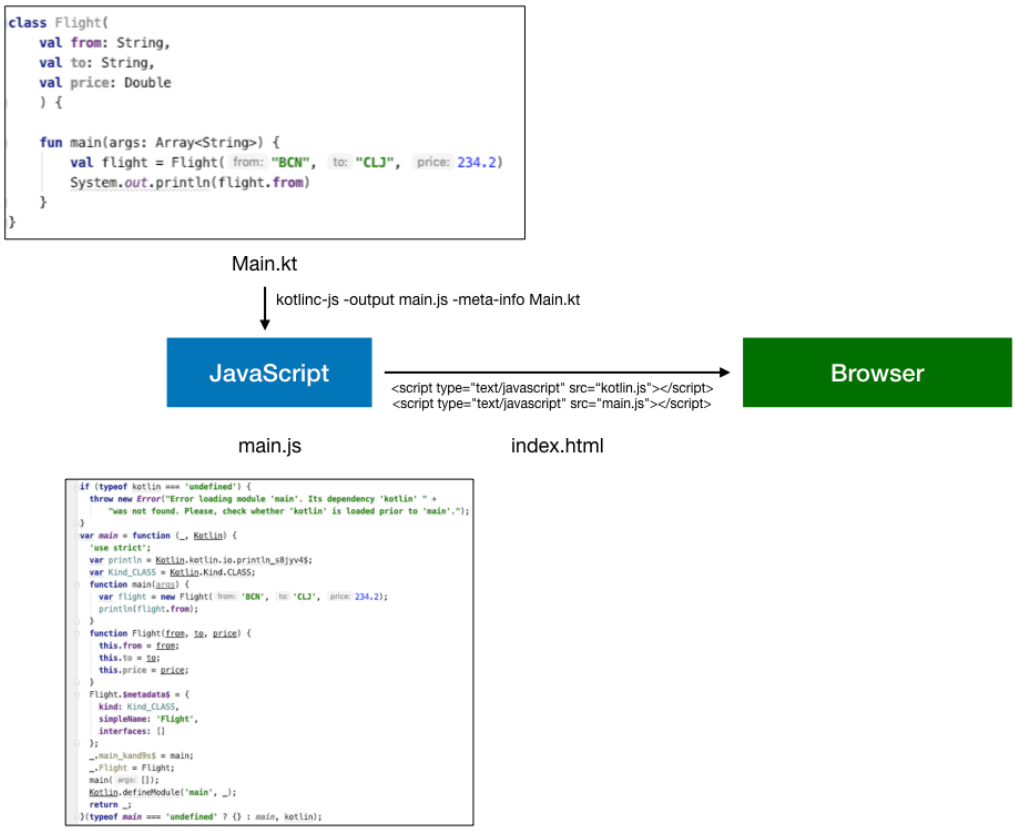

Fig. 1.5 KotlinJS transpilation process

JavaScript can be used in client-side application and server-side applications. Due to JavaScript's growth some modularization was needed to handle injection and encapsulation. Kotlin JavaScript is can be transpiled with Asynchronous Module Definition (AMD) for client-side development (which would be similar to how jQuery is used) and also with CommonJS for server-side development (which would be similar to how *node_modules* are used in node servers). But also, it can be transpiled with Universal Model Definition (UMD), which is the one by default with the command line compiler, allowing support for both client and server side. If you are transpiling your code with gradle or maven, you can set up what module to enable.

A cool feature that IntelliJ offers is the possibility to debug your Kotlin code in browsers. This is huge as normally when writing JavaScript for client applications you would be writing a lot of logs in order to debug it. The debugging itself is in the Kotlin code not in the transpiled JavaScript, so if you don't understand JavaScript you don't have to worry about it.

1.2.3. Native

The last but not least is the Kotlin/Native. It compiles Kotlin code into platform specific binaries that do not have the dependency of a virtual machine (as for the Java bytecode) or an interpreter (as for JavaScript).

Kotlin/Native is still in beta phase and it was announced in Kotlin 1.3 release. It currently supports quite some targets: iOS, Android (it obviously supports Java bytecode, but in this case we are referring to writing a native library that can be imported in an Android application), MacOS, Windows, Linux (there is an emphasis on RaspberryPis) and even WebAssembly.

One of the coolest features of Kotlin/Native is interoperability. There is quite some work here from the Kotlin team as for example in the Apple world, Kotlin/Native is interoperable with iOS and macOS main programming languages, Swift and Objective-C, but also with any native library you have (that can be written in C or C++), and it even gives you a wrapper for accessing platform POSIX for example. To simplify this concept, Kotlin/Native offers you the possibility to “talk” to your existing native libraries, and it also offers you the possibility to your actual native project to “talk” to your new Kotlin code. We’ll see examples about a practical approach to that in the next section.

Now let’s have a look at how the compilation of your Kotlin code to native works. Kotlin/Native is basically a project that holds two main items the Kotlin native standard library and a LLVM backend.

The native standard library would be the equivalent to kotlin.js in the Kotlin JavaScript world and the runtime standard library in the Java bytecode world. This is basically a native implementation of the main Kotlin core functionalities. Currently the native standard library is statically linked to the final compilation binary.

LLVM is a popular compiler toolchain, it is widely used. It started compiling C/C++, but nowadays it also compiles Objective-C, Java bytecode, Python, and a huge list.

The output of the compilation of the Kotlin code can be a Kotlin generic library *klib* that contains LLVM bitcode and Kotlin metadata, so that can be reused in any other Kotlin/Native project, or it can output binaries that are platform specific. The *klibs* are not platform specific, so it is a clean way to deliver a Kotlin/Native library that when integrated in a specific project will end up being compiled into the desired platform binary.

1.2.4. Multiplatform

As seen in the sections above, Kotlin offers a large broad of targets. Realising the size of targets, they supported and the possible solutions that Kotlin can offer to the development community, multiplatform projects appeared.

Multiplatform is not the same as cross-platform in the way that such frameworks as Ionic or React are known. Ionic is a popular JavaScript framework used to build Android and iOS applications, the problem is that when creating an Ionic application this idea of hybrid code comes into play. An Ionic application when deployed into a device it does not run the code natively, but in a container that interprets this JavaScript code. This makes the application slow and over-headed with unnecessary files, even though this kind of applications are used when you need a quick and static application. But when it comes to creating a secure, reliable, fast and responsive application (most cases scenario) you would build a native application.

Kotlin offers this possibility of writing code once for multiple platforms without losing this performance, reliability, security and responsiveness. This is possible due to the fact that Kotlin code it's either compiled into Java byte code (for Android and sever applications) and into Native code when needed (for iOS or Windows applications).

Most business that of Software as a Service (SaaS) that have an Android app, an iOS app and a website, have business logic repeated in the three platforms. Kotlin Multiplatform comes to the rescue. As Kotlin offers that large broad of supported platforms it ended up bringing a solution for the shared business logic.

Kotlin Multiplatform is still in an experimental phase, but it has evolved a lot since Kotlin 1.2 to Kotlin 1.3. The idea behind it is that the shared logic that will be in all your platforms can be written in a common place and then each platform includes that common logic. For example if you have an application that looks for flights, for sure in all your platforms Android, iOS and web you will have a data object containing the city from which the airplane will launch, the destination city, the departure time, the arrival time and the price. Why writing this code three times if you can write it just once and it will run as native code in each application? This will bring lots of benefits to your app. One of them is that you will have all your business logic in one place, which saves a lot of time and puts the accent on having a clean architecture, as you can see in Figure 1.6.

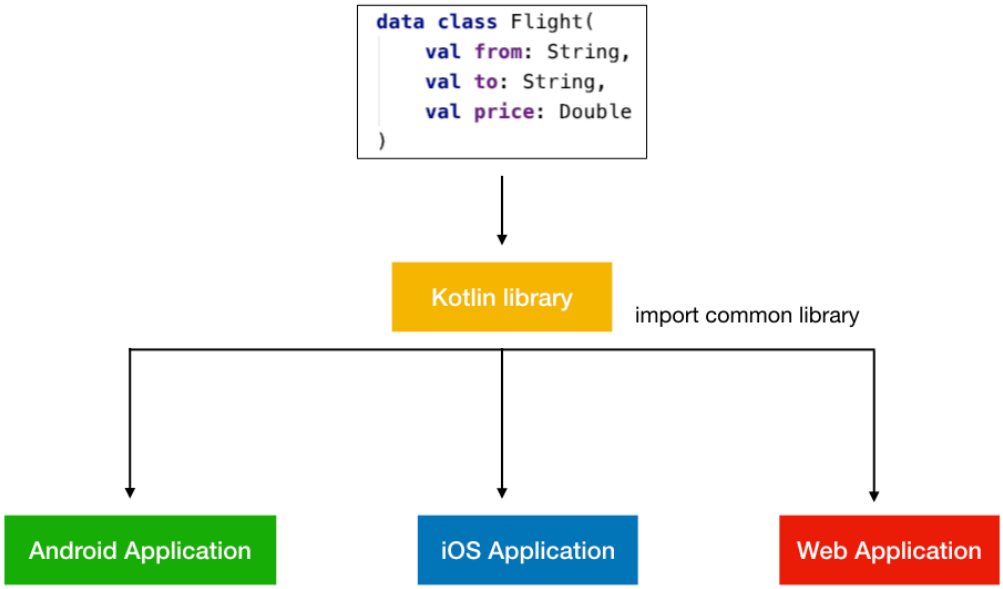

Fig. 1.6 Shared code between platforms

We will deep dive more into Kotlin Multiplatform in the next chapter.

1.3. Conclusion

After having a clearer view on what is Kotlin and seeing what Kotlin can be used for, in this paper will focus on Kotlin for Android development. The reason behind this decision is due to several factors. First of all, Kotlin is Android Studio's preferred language. Android development has a very big community and almost everybody is hyped with Kotlin. Building a curriculum on Android development using Kotlin is a safe bet. The selection of Kotlin as the preferred Android development language is very recent, is that recent that it was announced during the time that this paper was written, May 2019.

KotlinJS by its own I personally don't think it will have a lot of future, the state of Kotlin in 2019 already shows it is the least used compilation target. There are already very powerful JavaScript frameworks for web development, such as Angular, React and many more.

Kotlin/Native, in my honest opinion will have a good future too. After having investigated I came up with an interesting idea that I will try out in the development of this project. The idea is to try to get rid of the JNI layer in Android, which I will cover later in detail. Although the concept behind Kotlin/Native is super interesting it is still in a beta or experimental phase. This means it has an unstable API that may lead to huge refactors within the next Kotlin releases.

Kotlin Multiplatform also has a lot of use cases that can succeed, especially for developing Android and iOS apps. The idea is to have the business logic developed in just one place and then for each platform do the specifics. The

problem behind is that this is still in an experimental phase too, so future releases may lead to big code refactors.

So, during the next chapter we will try to have look at some of the features that Kotlin offers, but specially looking for the ones that have an effect in the Android development part.

CHAPTER 2. WHY KOTLIN? – KOTLIN FEATURES FOR ANDROID APP DEVELOPMENT

During this chapter the main goal is to see what features does Kotlin offer to Android development but also with the question: “Why Kotlin?” in mind.

Actually, the question “Why Kotlin?” comes from their official webpage[4] where the answer they give is because Kotlin is:

- Concise, it drastically reduces the amount of boilerplate code.
- Safe, it avoids entire classes of errors such as null pointer exceptions.
- Interoperability, it leverages existing libraries for the Java Virtual Machine (JVM), Android and the browser.
- Tool-friendly, you can choose any Java Integrated Development Environment (IDE) or build from command line.

Therefore, we will go through the most relevant features in a theoretical way, trying to understand what problem they solve and how they work.

At the end of the chapter we will conclude if the answers to why using Kotlin that their creators give in their webpage makes sense from the theoretical point of view.

2.1. Java Interoperability

We will start with the main feature, which is Java interoperability. Most Android apps are written in Java, which need to run in a JVM. The Android Operating System does not have a JVM, but it has something similar that makes possible Android apps to be written in Java. This is called Dalvik Virtual Machine (DVM) for Android OS lower than 4.4 and Android Runtime (ART) for higher.

Therefor a very smart move from JetBrain’s for Kotlin was to make it interoperable with Java. But, what does interoperability mean? Interoperability as far as programming languages are concerned is the possibility of using more than one programming language in the same project, in this case, Java in Kotlin without having any problem.

After having studied how Java and Kotlin get compiled and how it runs in the JVM in Chapter 1 this concept of interoperability between Java and Kotlin is easier to understand, both Java and Kotlin files are compiled into Java byte code and they can run together.

[4] Kotlin’s official webpage: https://kotlinlang.org/

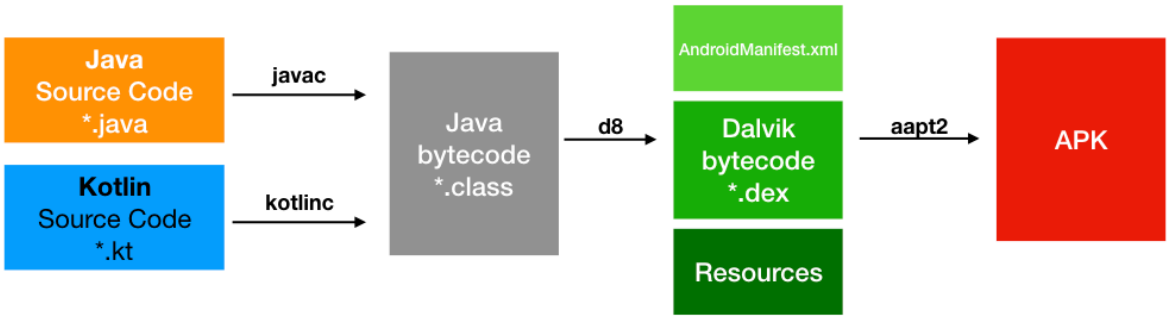

Fig. 2.1 Java and Kotlin interoperability in Android applications

But why would I want to start writing my Android apps in Kotlin if I am fluent in Java? This is a good question, and in order to answer it during this chapter we will look at some features that Kotlin offers that would make you want to use it instead of Java.

Even though Java and Kotlin can perfectly coexist together for Java developers there are some default issues that they need to keep in mind when starting to code in Kotlin for the first time (apart from the language itself).

Let's go through the most notorious differences between the defaults of both programming languages.

One of the most significant one is that you can forget about semicolons, if you are used to using semicolons, the first day will be a little annoying, but definitely after that day you won't want to go back using semicolons. Another significant detail is that objects can't be NULL, that means that for each existing object there will be an optional one. For example, *String* is different than *String?*, the first one can't be NULL but the second one yes (we will cover more about this in the next section). Also, in Kotlin there is no such thing as the *static* keyword for a method or property, which is broadly used in the Java world, you need to define it with a Java annotation. And the last main difference is that classes are *final* by default instead of being *open* as in Java, that means that if you want to extend a Kotlin class it will throw a compilation error if the class is not *open*.

When writing intercompatible code for Android applications, the Android documentation offers a best practice guide[5] that should be followed for writing better intercompatible code between Java and Kotlin for Android development. It even gives a tool that will run code checks for informing the user how intercompatible its code is.

2.1.1. Advantages

At the start of this chapter it was stated that making Kotlin interoperable with Java was a very smart move from the JetBrain's team, the practical meaning of this statement is that you don't have to recode your entire application in order to

[5] Android's Java-Kotlin interoperability guide: https://developer.android.com/kotlin/interop

migrate it to Kotlin. You could follow a smooth migration plan, for example, all new features can be written in Kotlin and then migrate little by little your Java classes into Kotlin. That's how popular applications started to introduce Kotlin into theirs. After having written your first Kotlin lines you can then decide to migrate all the application to Kotlin or not.

This interoperability also offers the possibility, of using existing Java libraries (JAR) or Android libraries (AAR), in the Android world, in Kotlin projects or vice versa. So, you don't have to worry of not having a library migrated into Kotlin in order to use it.

Another advantage is that as Kotlin and Java are compiled into Java bytecode the devices that are currently using a JVM or the Android OS, which in the newest devices uses ART to execute this Java bytecode, don't need any extra updates to run or execute the applications that use Kotlin, all the dependencies are included in the application.

The last advantage to be commented is that by having full Java interoperability, developers or application owners don't have to worry about thinking that if the application is built with a modern language such as Kotlin won't work on old devices. As Kotlin is a JVM language Kotlin application will work as always in old Android devices.

2.1.2. Disadvantages

Even though the advantages of Java interoperability are great there are few disadvantages depending on what you are building.

In case of building security frameworks or applications, Kotlin it's not the best language to use, as with Java, you wouldn't build a very secure application.
Android applications are very easy to reverse engineer with *apktool* (a very well-known tool in the Android reverse engineering community) for example. *Apktool*[6] takes the Android application (*APK*) and decompiles the archive that contains the Java bytecode (actually Dalvik bytecode in Android applications) into *smali* files which are easier to read and understand. And from *smali* files you can easily go to Java. Therefor all existing reverse engineering tools, at binary level, for Android applications will be able to decompile, inject, debug and hook Kotlin code.

Also, security by obscurity in Kotlin code is not the best option, even if some papers claim so, see [8], Kotlin produces a lot of metadata code where store its obfuscated strings, which are quite easy to deobfuscate[7].

The security issue can be addressed by either using a third-party security framework written in C/C++ or writing your own C code for sensitive operations.

[6] Apktool is a tool for reverse engineering 3rd party, closed, binary Android apps. You can find more information in their webpage: https://ibotpeaches.github.io/Apktool/.
[7] Security by obscurity is not the best for protecting an Android app. For more information read this article: https://www.nowsecure.com/blog/2019/07/11/think-twice-before-adopting-security-by-obscurity-in-kotlin-android-apps

Although, not all applications in the market need that level of security. For example, in the case of a bank it would need secure code and mechanisms for protecting very sensitive information and transactions, but an app that just sets an alarm does not have to worry about that. Most Android application attacks need physical access to the device in order to be compromised, so it's not a big issue. But for Android application developers they should not store sensitive keys in Kotlin code.

Another small disadvantage is that when you write an Android Archive (*AAR*), an android library, in Kotlin, the applications that integrates it apart from including this library it also has to include the Kotlin runtime libraries. Which may lead to add some extra size to the application.

2.2. Null Safety

Speaking about Java, NullPointerException is by far the most common thrown exception in all applications developed in Java. You know this is big when even the creator of the Null Reference concept even apologies for creating it[8]. The issue is caused when a member of a null reference is called, which happens more often than expected. This exception is produced at runtime so there is no way to catch it at compile time. In practical terminology the exception is thrown when we expect a variable to be initialized (and it is not) and we use one of its members.

JetBrain's decided to avoid the Null Reference issue in Kotlin. They did that by not allowing a variable to be nullable by default. As we described in the section before, for each object type in Kotlin there exists a second object type that can be null, this is also known as an optional type. So, nulls don't exist in pure Kotlin unless you want.

```
var airport: String = "BCN"
airport = null // This will throw a compilation error

var airport: String? = "BCN"
airport = null // This will work
```

Fig. 2.2 Kotlin object type declaration and optional declaration

Kotlin will check at compile time that a type is never initialized to null or ends up returning a null value. That sounds great, but then, if you can use optional types in Kotlin how is it still null safe? That is a good question, optional types will also be treated and examined by the compiler. If you declare an optional type value and later want to invoke one of its members you will either have to check if it is a none null value and then use it, or you could just use safe call operators, which will only be invoked when the object reference is not null or will return null if the

[8] In the QCon London software conference, Tony Hoare, the creator of the null reference apologizes for inventing it. Find more information here: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

object is null. The last option is to use the member with the assertion operator, which will escape the null check and will be executed even if the value is null. This last option may lead to NullPointerExceptions even in Kotlin, therefor you should avoid using it.

```
var flight: Flight? = Flight(“BCN”, “MAD”, “10-10-2019”, “20-10-2019”)

// Option number 1: check if is not null
if (flight != null) {
    flight.searchTicket()
}

// Option number 2: safe-call
flight?.searchTicket() // This will only be executed if flight is not null

// Option number 3: assertion operator
flight!!.searchTicket() // This may throw a NullPointerException
```

Fig 2.2 Different calls on a nullable object

In case you want to use the safe call operator but not return a null value if the object itself is null you can you the *elvis* operator, which is implemented in Kotlin, and return whatever you want.

```
val user: User? = null
val action = user?.login() ?: signup() // If user exist log in, sign up otherwise
```

Fig. 2.3 Elvis operator

Not only when speaking objects Kotlin is null safe but also when casting, if we want to cast an object into another but this is not the expect type of object with the smart cast operator, we can avoid the ClassNotFoundException. And also, when treating with arrays or collections Kotlin will always offer the option to filter all the null values of itself.

Kotlin also offers the possibility to handle nullables with Kotlin Extensions which we will cover in the next section.

Definitely Kotlin being null safe by default in every possible place is an awesome feature that can reduce around most of the 70% of all Android exceptions errors and it might be one of the most important factors on the reason why Kotlin has a great learning curve.

2.3. Kotlin extensions

Kotlin extensions is a great feature that, as its name says, lets you extend functions or properties for any class. Normally in Java when you want to extend an existing class with a function you would create a new Java class and then extend that class with the new function. One problem with this approach is that

you would need a new class for each extended class just for implementing one custom method, another problem is that instead of using the provided class you need to use the extended one, which may lead to errors and headaches. Since this was a problem Kotlin extensions provide the perfect solution: it lets you extend a method or parameter for any class without having to extend the class you want to add the extended function. This might sound overrated, but when programming this is a very powerful tool to avoid writing the same code snippet for a certain class method. It is also very useful to extend functionalities from a third-party library that might be missing some methods that we need, so we can modify it without touching it. If we extend an already existing method the original method will always be executed instead of the extended.

In order to extend a functionality to a class, we need to start the declaration with the type of the class we are extending (this is called the receiver type) followed by the extended method name with its arguments if needed. Basically, it means adding the class name you are extending before the method you just implemented separated by a dot. This is something easy to do but it can save a lot of repeated coding, or missing functionalities.

When developing Android applications, there is always a code snippet that is repeated when you want to get the view identifier of a certain element on the application view, and this is *findViewById,* as you can see in [4]. Using Kotlin extensions we can get rid of it by simply extending the View class. But you don't have to do this by your own, there is already an official Android Extensions gradle plugin[9] that will do the trick for you, it will let you bind views, set listeners to elements and work with fragments, custom views and view holders in an easier way. This ends up in a clearer and easier way of working with views in this case.

Kotlin's extensions is one of the most loved features by developers as it allows your interfaces to be as minimal as possible and that means having an understandable interface when implementing it. For example, a String is a collection of index characters, offering a method to return this same String in capital case would be an extension function as it is not an essential aspect of a String.

2.4. Kotlin standard library

The Java standard library is pretty powerful because it does a lot of stuff for you, but when the Kotlin standard library appeared it certainly surpassed Java's.

A standard library is a library for a programming language that offers you tools, APIs and components you can use all around your code that will cover low level details of the programming language so that you can just focus on writing what you want your application to do.

[9] More information about the Kotlin Android Extensions plugin can be found here: https://kotlinlang.org/docs/tutorials/android-plugin.html

Being honest, Kotlin's standard library is so powerful because it is a mix of Java's standard library and extension functions, as you can see in Figure 2.4. Java's standard library is very large and offers a lot of features, contrary to C's standard library, which is minimal, both have their benefits, one is hard to maintain and the other one not that much. One gives the developer a lot of features and the other requires writing their own code or using a third party for certain tasks. A good standard library can have a big influence on the success of a programming language, therefor Kotlin's currently in a good position as it's extending Java's.

The fact that Kotlin's standard library is an extension of Java's also helps a lot the Java interoperability and makes it super easy to call Java from Kotlin and vice versa.

Fig. 2.4 Kotlin Standard Library

There are many functionalities in the Kotlin standard library, we will look at some of them. One of the most notorious, or used, are the extensions on the collections. The collections offer a set of extended function in order to filter, map and reduce. It is also very easy to initialize collections; you don't have to do all the declarations in order to initialize it in your Java code. And other example, as mentioned before, the String class has the main functionalities and has extension functions that perform very common tasks, such as lower casing it, uppercasing it, split it… and many more.

Kotlin standard library also offers a set of standard functions which are meant to help the developer work easier with any object while solving common or repeated issues that they face. These functions let you change the scope of a given object and even apply multiple functions on the same object. Standard functions are also extension functions, as we mentioned before for the Kotlin standard library. An example of the usefulness of a standard function would be to do null-checks and executing a certain code only if a variable is not null with the *let* standard function. This is very attractive because we can get rid of a lot of if-else code in order to check if a variable is null. Another example is the *apply* standard function which is very useful when you want to call several methods for the same instance. More examples of standard functions can be seen in Figure 2.5.

Fig. 2.5 Kotlin standard function practical use cases by Jose Alcérreca

2.5. Coroutines

Coroutines is one of Kotlin's most envied features. Threading in Java and more specifically in Android app development has always been an issue.

In a first approach, we have learned to program synchronously, one line of code is executed when the above one has finished its execution. This normally worked fine for the majority of code we would write but it is also true that lately this approach has changed, especially in application development, now we tend to write code that waits for something. For example, we wait for a network call to be performed, we wait for a database query response, we wait for another player to move, basically we just wait for some action to be completed. We can't just execute all this code in the main thread and block the user until we received an event, this is completely inefficient and does not lead to a good user experience.

But then, how do we manage this problem? The solution was obviously using asynchronous programming. This meant using different threads for certain operations and then informing the main thread about the result of this certain operation. Writing this kind of code, is quite difficult and also too complex to maintain, which leads to a lot of overhead for doing a simple network call.

Then callbacks appeared which in Android development would work for making asynchronous programming a little easier. Callbacks are just functions passed as an argument to the asynchronous function that will be called when the asynchronous operation has finished. Coding wise, there is a problem that comes up when coding with callbacks known as the "callback hell" that can be visualized in the figure below.

```
function iIds(startAt, showSessionRoot, iNewNmVal, endActionsVal, iStringVal, seqProp, htmlEncodeRegEx) {
    if (SbUtil.dateDisplayType === 'relative') {
        iRange();
    } else {
        iSelActionType();
    }
    iStringVal = notifyWindowTab;
    startAt = addSessionConfigs.sbRange();
    showSessionRoot = addSessionConfigs.elHiddenVal();
    var headerDataPrevious = function(tabArray, iNm) {
        iPredicateVal.SBDB.deferCurrentSessionNotifyVal(function(evalOutMatchedTabUrlsVal) {
            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                iPredicateVal.SBDB.normalizeTabList(function(appMsg) {
                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                        iPredicateVal.SBDB.detailTxt(function(evalOrientationVal) {
                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                iPredicateVal.SBDB.neutralizeWindowFocus(function(iTokenAddedCallback) {
                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                        iPredicateVal.SBDB.evalSessionConfig2(function(sessionNm) {
                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                iPredicateVal.SBDB.iWindow2TabIdx(function(iURLsStringVal) {
                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                        iPredicateVal.SBDB.idx7Val(undefined, iStringVal, function(getWindowIndex) {
                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                addTabList(getWindowIndex.rows, iStringVal, showSessionRoot && showSessionRoot.length > 0 ? show
                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                        evalSAllowLogging(tabArray, iStringVal, showSessionRoot && showSessionRoot.length > 0 ?
                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                BrowserAPI.getAllWindowsAndTabs(function(iSession1Val) {
                                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                        SbUtil.currentSessionSrc(iSession1Val, undefined, function(initCurrentSe
                                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                                addSessionConfigs.render(matchText(iSession1Val, iStringVal, eva
                                                                                                    id: -13,
                                                                                                    unfilteredWindowCount: initCurrentSessionCache,
                                                                                                    filteredWindowCount: iCtrl,
                                                                                                    unfilteredTabCount: parseTabConfig,
                                                                                                    filteredTabCount: evalRegisterValue5Val
                                                                                                }] : [], cacheSessionWindow, evalRateActionQualifier, undefined,
                                                                                                    if (seqProp) {
                                                                                                        seqProp();
                                                                                                    }
                                                                                                });
                                                                                            }
                                                                                        });
                                                                                    }
                                                                                });
                                                                            }
                                                                        });
                                                                    }
                                                                });
                                                            }
                                                        });
                                                    }
                                                });
                                            }
                                        });
                                    }
                                }, showSessionRoot && showSessionRoot.length > 0 ? showSessionRoot : startAt ? [startAt] : []);
                            }
                        });
                    }
                });
            }
        });
    }
});
```

Fig. 2.6 Callback Hell

This is because normally in all programming languages the declaration and implementation of a function follows this schema: first we define the function name, then between brackets we specify the arguments taken by this function and finally, between curly brackets we implement the function. Therefore, as when writing asynchronous code with callbacks, a function is an argument of another function ends up with the above shown piece of art.

The proposed solution by JetBrains for all this complex asynchronous programming is called Kotlin Coroutines. Kotlin Coroutines is a library (that is not in the Kotlin Standard library) that offers a clean an easy API for doing all this asynchronous programming. Coroutines are not threads, and this is a major concept to keep in mind, they are like tasks or jobs that are executed in threads, the Kotlin team defines them as lightweight threads. Coroutines can be executed in any thread depending on the specified context for the coroutine but they can even change from on thread to another. This means that you forget about creating threads and maintaining them, which is complex and can lead to bugs and security flaws.

In Kotlin coroutines a new concept of functions appears; they are known as suspend functions. The main difference between a function and a suspendable function is that this first can be suspended, and it can only be executed in the context of a coroutine, as you can see in Figure 2.7. Suspended functions can be suspended and resumed in any thread.

Fig. 2.7 Coroutine use case and timeline

2.6. Anko

Anko is an official Kotlin library, which means it is maintained by JetBrains that claims to make Android app development easier, faster and smarter. Anko is built by defining Kotlin extensions that do the repetitive Android tasks for you. This way you just focus on building the essential parts of your app. Its name, Anko is the combination of Android and Kotlin first two letters.

Anko offers four main modules for creating Android applications:

- Anko commons is the module that offers a lot of helpers for basic Android functions such as: starting new activities, simplifying the common Android callbacks like *onClickListener*, any communication event like sending emails or making calls and many more.
- Anko layouts is the module that you create UIs in its Domain Specific Language (DSL). In order to create UI in Android you either write the XML code or drag and drop items from the menu bar. With Anko's layout DSL you can create they UI and set all the callbacks for a certain item in the same place, which results in easy understand and less code. It is also compatible with existing UIs.
- Anko SQLite simplifies a lot the communication with the database. It does all the boiler plate for accessing the database and it also manages the cursors, threads and models.
- Anko coroutines, which gives you a pool that manages and executes all your coroutines and prevents leaks.

It seems that JetBrains does not only want to take over Java as the preferred language to code Android applications but also to redefine Android applications development.

2.7. Multiplatform

In the first chapter, we have seen Kotlin Multiplatform and its capabilities. And even if Kotlin is a general-purpose programming tool a much more important premise for the Kotlin team is code sharing between these platforms[10]. Not all programming languages have the ability to be built for as many platforms, therefor they plan no only saving time by using Kotlin because it is a modern language and because it is way more intuitive, but also by avoiding the developers to duplicate the same code twice or even three times for each platform.

In order to build multiplatform projects Kotlin defines the concept of an *expected class or function* and an *actual class or function*. A simple comparison can be a header for a C file, normally when we create a C project, we have two files a header file that defines the public methods and a C file that implements those methods. The expected/actual concept works in a similar way, see Figure 2.8 and 2.9.

```
enum class LogLevel {
    DEBUG, WARN, ERROR
}

internal expect fun writeLogMessage(message: String, logLevel: LogLevel)

fun logDebug(message: String) = writeLogMessage(message, LogLevel.DEBUG)
fun logWarn(message: String) = writeLogMessage(message, LogLevel.WARN)
fun logError(message: String) = writeLogMessage(message, LogLevel.ERROR)
```

compiled for all platforms

expected platform-specific API

expected API can be used in the common code

Fig. 2.8 Common logic for all platforms

```
internal actual fun writeLogMessage(message: String, logLevel: LogLevel) {
    println("[$logLevel]: $message")
}
```

Fig. 2.9 Platform specific implementation of the expected method

If we create a multiplatform project, we will have the common logic, where we implement business logic and we also declare what does each platform need to implement. And we will also have the specific logic for each platform, that as said before will have to implement the requirements that the common logic sets.

The most practical use case is for sharing the business logic in mobile applications. As usually, these, have the same logic duplicated, because usually the application has to do the same in the iOS application as in the Android application.

[10] More information on the Kotlin Multiplatform feature can be found here: https://kotlinlang.org/docs/reference/multiplatform.html

2.8. Conciseness

After having seen Kotlin's Standard library you can guess that by having Kotlin wrapping and extending a lot of functionalities for us our code will reduce its size quite a lot. But you haven't seen it all yet. We are going to have look at other features that Kotlin offers that will drastically reduce the size of our app.

Kotlin is a type inferred language. This means that when declaring a variable, noted as *var* in Kotlin, you don't have to specify its type if you initialize it right away. If you do so the compiler will deduce the type of the variable by the initialization. But if you don't initialize it when you declare it you have to specify its type.

Kotlin also does smart casting, if it can determine the type of a variable from a previous verification it will automatically cast it for you.

When we learn about Object Oriented Programming (OOP) we introduce objects that have their constructors, their setters and their getters. And when more advanced we also override the *hash* and *equals* methods for our objects. This are known as Plain Java Objects or POJO, Figure 2.10. They are usually not complex but when you start a project and you have to create some of them it consumes some time and a lot of lines of code. Kotlin has introduced data classes, Figure 2.11, which in one-line declaration will create a POJO. So, you can forget about having to write all this repetitive but important lines of code.

```
package com.andreas.fly.demo;

import java.util.Date;
import java.util.Objects;

public class Flight {

    private String from;
    private String to;
    private Date departure;
    private Date arrival;

    public Flight(String from, String to, Date departure, Date arrival) {
        this.from = from;
        this.to = to;
        this.departure = departure;
        this.arrival = arrival;
    }

    public String getFrom() {
        return from;
    }

    public void setFrom(String from) {
        this.from = from;
    }

    public String getTo() {
        return to;
    }
```

```
    public void setTo(String to) {
        this.to = to;
    }

    public Date getDeparture() {
        return departure;
    }

    public void setDeparture(Date departure) {
        this.departure = departure;
    }

    public Date getArrival() {
        return arrival;
    }

    public void setArrival(Date arrival) {
        this.arrival = arrival;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Flight flight = (Flight) o;
        return from.equals(flight.from) &&
                to.equals(flight.to) &&
                departure.equals(flight.departure) &&
                arrival.equals(flight.arrival);
    }

    @Override
    public int hashCode() {
        return Objects.hash(from, to, departure, arrival);
    }
}
```

Fig. 2.10 Plain Java Object

```
package com.andreas.fly.demo

import java.util.*

data class Flight (
    var from: String,
    var to: String,
    var departure: Date,
    var arrival: Date
)
```

Fig. 2.11 Kotlin data class

A very broadly used pattern in Java is the Singleton pattern, which consists in having a shared instance for any declaration of this certain object. The Singleton pattern in Java needs to be implemented and managed by the developer but in Kotlin you can create a singleton by just declaring the *object* keyword before the body or functions of itself.

Kotlin classes can have a primary constructor and secondary constructors, but the primary has to be declared in the class declaration. The constructor's parameters or any function parameters can have a default assigned value that can be skipped in the invocation of the method if you don't want to override it.

Taking everything into consideration, if we take a Java application and migrate it to Kotlin for sure we can reduce the size of coded lines between a 20% and 50%. Writing less code means less code to test and less probability of introducing bugs, less code to read and understand. And furthermore, having less code leads to less code to be compromised, therefore we have more secure applications.

2.9. Modern language

All the mentioned features in the previous chapters in combination and some others not mentioned such as string interpolation, which allows you to concatenate strings very easily, the fact that there are not checked exceptions, which means that you are not enforced to catch all thrown exceptions, an easy object destruction into its constituent parts and many more features make Kotlin a modern language.

Nowadays we hear too often that a new programming language appeared, but there are several significant differences between Kotlin and all these new languages. One of them is the fact that the language comes from a very strong industry in the programming world, JetBrains. JetBrains is a software development company that has built some of the most used Integrated Development Environments (IDE) such as CLion, IntelliJ IDEA, AppCode, GoLang, PyCharm… These gives them a lot of feedback form software engineers so they could start the Kotlin project. Another key aspect is that even if Kotlin appeared few years ago they started working on it many years ago earlier 2011 so actually, even if the release is new the actual product has been in progress for a while.

On the other hand, Java makes you write a lot of boilerplate code, and all the new programming languages don't have that issue. An example of a modern programming language that does not have this problem is swift. Swift is developed by Apple and it's used mainly to develop iOS and macOS applications, but it also can be used for server-side applications. Swift is clean and concise and even though for iOS developer's it has been a pain to migrate every time Swift had a major release (as it was breaking blackguard's compatibility) it is still very popular. Kotlin's syntax is very similar to Swift's, if you see a code snippet in each of those languages you can confuse them, for example look at Figure 2.12. But this is good from the point of view of mobile app developers, as normally, if you develop apps for iOS you also develop apps for Android, therefor it's easier to change the context of developing for a specific platform.

Swift

```swift
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
let increment = makeIncrementer()
increment(7)
```

Kotlin

```kotlin
fun makeIncrementer(): (Int) -> Int {
    val addOne = fun(number: Int): Int {
        return 1 + number
    }
    return addOne
}
val increment = makeIncrementer()
increment(7)
```

Fig. 2.12 Swift and Kotlin comparison

So, basically, we can tell Kotlin is a modern language currently because of its conciseness, its clear syntax, all its built-in extensions that do a lot of work for you and its low risk adoption.

2.10. Tool Friendly

Apart from being a modern language, JetBrains has made a lot of effort on making Kotlin tool friendly, mainly because Kotlin support is integrated in all its in-house IDEs that can make us of Kotlin, but the main ones are Android Studio and IntelliJ IDEA, even though it also offers a plugin for using Kotlin in Eclipse.

It is very easy to start a Kotlin project in the mentioned IDEs, when creating a new IntelliJ or Android Studio project you can checkmark the support Kotlin box. But not only that, the IDEs provide you a lot of sample or starter projects for specific Kotlin project purposes such as a multiplatform project. In the case of a multiplatform project it will create you the project structure, divided by platforms: JVM, macOS and JavaScript and also the common part of the project, not to forget that it will also add the dependency for each platform and some sample code that for each platform interacts with the base common code. It makes it so easy to start so that you don’t have to lose a lot of time in searching what you need to set it up.

Not only that, but IntelliJ and Android Studio also detect special Kotlin notations, such as the suspend functions and that way it highlights you that you are treating with a coroutine which helps a lot when debugging.

Another cool feature of the IDEs that support Kotlin is that it offers you the possibility to translate Java code into Kotlin code automatically! This sounds very cool but after a conversion you still need to review the converted code in order to see if something is mess up, which probably is. Also, whenever you copy Java code from a StackOverflow answer or from wherever and you paste it in a Kotlin file it will automatically translate it into Kotlin, but you also have to review this conversion in order to check that nothing is messed.

For KotlinJS, IntelliJ offers the possibility for you to debug web applications in the browser with IntelliJ’s Chrome plugin.

All of these makes starting to work with Kotlin super easy and this is very important as normally if you start something new and it takes a lot of time to set up and configure properly it makes you not wanting to continue working with it.

2.13 Conclusion

Taking all the theoretical features into consideration we can conclude the following about Kotlin:

1. **It is the Java replacement.** The compatibility with Java is one of the things in which they have but most of the effort. It is very easy to call Java code from Kotlin and vice versa. This not only permits the developer to use existing Java code in Kotlin projects but also to do smooth migrations from Java code to Kotlin code. It also offers all the features offered by Java but in an enhanced manner.

2. **It is a modern easy to learn language.** Even if there are some other modern languages that run on the Java Virtual Machine such as Scala, Kotlin is very easy to learn as it is very intuitive as it removes all the boilerplate that other languages have. Antonio Leiva, one of the pioneers in Android development with Kotlin stated in his webinar[11] that a person that has the programming fundamentals can learn basic Kotlin in two hours. Kotlin's documentation, that you can find in [2], is also way easier to read than Scala's for example, and this gives extra confidence to the developer, that doesn't get as scary when starting coding Kotlin.

3. **It revolutionizes asynchronous programming.** Asynchronous programming in Java has been a pain either because it was too complex or because it was very difficult to maintain. Kotlin coroutines is one of the most loved features from the Android development community as it makes asynchronous very easy to code and also maintainable as it is written as if it was synchronous.

4. **You build more robust applications with less code.** Not only you write less code because Kotlin avoids you writing boilerplate code but if you choose to create multiplatform projects you only write your business logic once and then build it for all your platforms. Also, by removing the null reference concept your code is less likely to have runtime crashes. And writing less code means, less entry points for bugs, more maintainable code and easier to read code.

Answering the question stated at the beginning: "Why Kotlin?" we can agree in the theoretical point of view agree with the answers that Kotlin gives, Kotlin is more concise and safer than Java, but it also allows you to migrate slowly by making it intercompatible with Java and with the help of the Integrated Development Environment (IDE). Now that we have a theoretical starting point in the next chapter, we will code some Kotlin with the accent on some of the studied features.

[11] Antonio Leiva's webinar on Kotlin development for Android applications: https://devexperto.com/training-gratis/

CHAPTER 3. HANDS ON THE KOTLIN FEATURES

In this chapter we will take a closer look to the features offered by Kotlin. We will try to analyse in a practical way how they work and how easy is to use them.

The main goals of this chapter are to build a multiplatform project for an Android and iOS, to see how does Kotlin and Java interoperability work together, the migration process of a Java Android application to a Kotlin Android application and how coroutines help to program asynchronous code.

During the development of proposed exercises some of the generic features described in the above chapter such as how modern o how tool friendly Kotlin is will also be analysed as these features will be tested by the simple fact of creating Kotlin projects.

3.1 Kotlin Multiplatform in action

Now that we know what Kotlin multiplatform is and which are its possibilities we are going to have a practical look on how it works. To develop a multiplatform project the recommended IDE is IntelliJ IDEA.

The first thing that the IDE asks us when creating a multiplatform project, as you can see in Figure 3.1, is what type of project do we want to start.

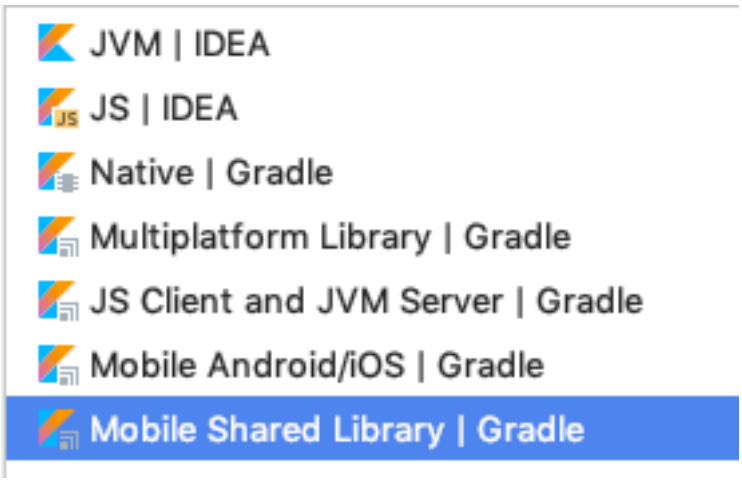

Fig. 3.1 IntelliJ options for a Kotlin project

As the focus of this paper is Android development the idea of this exercise is to create a Kotlin library that can be imported and used in either an Android application or in an iOS application[12]. When creating an application for your business, you duplicate your business logic in the Android application and in the

[12] You can find all the source code of the experiments in GitHub: https://github.com/andreasluca/TFG-KotlinSharedLibrary.

iOS application, with Kotlin multiplatform the possibility of writing this business logic and then using it in both platforms appears.

To create this shared library, you just need to create a new project and select the Mobile Shared Library option from Figure 3.1. The IDE will create for you lots of folders as you can see in Figure 3.2.

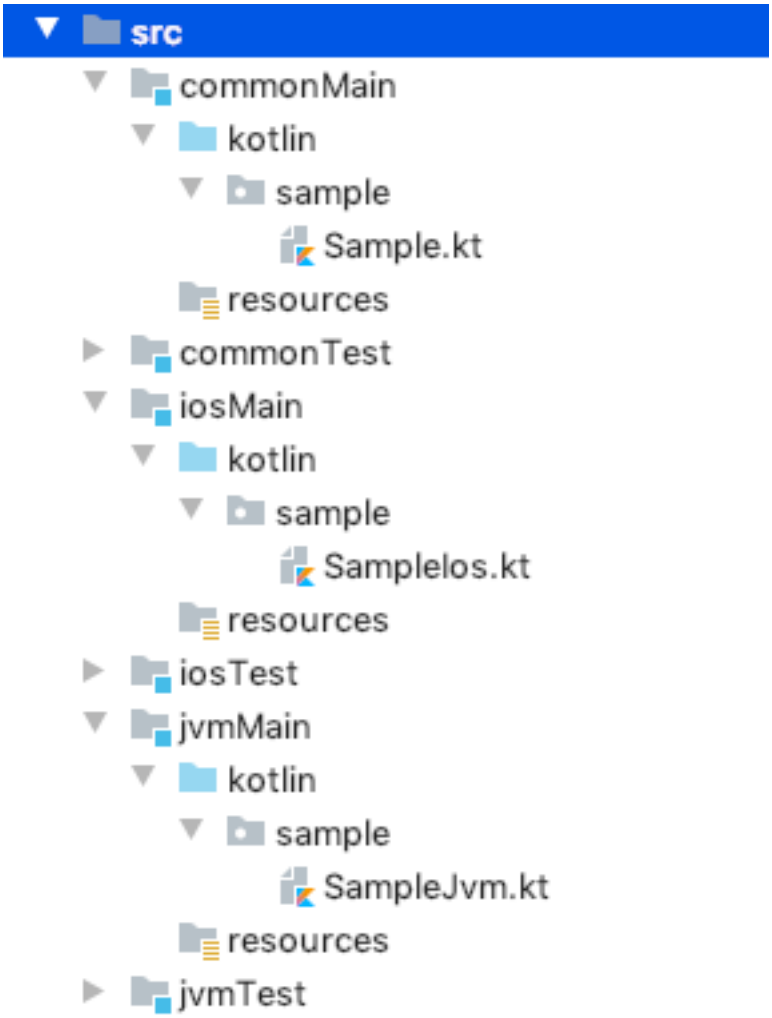

Fig. 3.2 Kotlin Mobile Shared Library project structure

The project structure contains some key folders:

- The *commonMain* folder is the folder that contains the declaration of the classes that both Android and iOS need to implement and the implementation of the shared logic.
- The *iosMain* folder contains the implementation of the declared classes for the iOS framework.
- The *jvmMain* folder contains the implementation of the declared classes for the java library.

- The tests folder: commonTest, iosTest and jvmTest. These folders give you the possibility of unit testing your common logic and then your platform specific logic.

The structure created it is very clear and it already adds all the dependencies required for each platform. There is a platform specific folder for iOS and for Android because there will be some components that are only available in the Android platform and some other that are only available in the iOS platform. For example, in the Android folder we are able to use the Java object types as it will be executed in a Java Virtual Machine. The project structure also includes some sample code that is easy to understand and gives you some headlines on how to proceed to implement your own code.

With the structure created, we will develop a library that for a given flight information it will calculate the price of the flight and the departure and arrival time. Therefor in the common logic we will declare that a Flight class is expected which has the parameters of a flight and we will also implement the function that returns an object of this Flight type with the price and the departure and arrival times. Then, in the platform specific folders we will implement the class that the common code expects and that's it. We hit the "*Build*" button and we have our Java library, a Java Archive file (JAR) and an iOS Framework that can be included in our application.

After having implemented the first Kotlin multiplatform library the main thing to highlight is that IntelliJ, the IDE used, helps a lot. JetBrains, the company under Kotlin and under IntelliJ, made a great job ensuring that creating projects with their new programming language using their IDE is very easy. Honesty, it saves a lot of set up time and having a big variety of multiplatform options makes you feel comfortable when building one. Also, the fact that it provides sample code for the common and each specific platform helps a lot to start, as you have something already that builds, runs and can be tested.

Something I didn't expect is the fact that the output of the build Android library is a Java Archive (JAR) instead of an Android Archive (AAR). Because normally when you build a library specifically for Android you create an Android Artifcat even if the Java Archive are also supported. But I guess as Kotlin multiplatform is divided in Java Virtual Machine, JavaScript and Native, it might build Java Archive by default. So, something to improve from their side is that when building a Mobile Shared Library to output an actual mobile library in the Android case, as for iOS frameworks are used and it is what the IDE builds.

3.2 Java interoperability in action

3.2.1 Calling Kotlin from Java

In order to see how Kotlin and Java are interoperable we will use the library created in the above subchapter. The idea is to create an Android application in

Java and import the library create before, which gives flight information for a given queried flight. Therefor we will have a Java application invoking Kotlin code[13].

The first thing is to create an Android application, so we will use Android Studio. When creating a new project in Android Studio we are asked what language we want build the project with either Kotlin or Java, Kotlin is the preselected one, we will select Java for the sake of seeing how they interact together.

After importing the flights library in the application, we try to invoke it, a curious thing is that the Flights class that we created in the library now has the name of FlightsKt and in order to invoke it form Java we just type the name of the class and the static method we created to retrieve a flight as you can see in the below code.

```
Flight flight = FlightsKt.getFlightFor("BCN", "MAD", "6-11-2019", "19-11-2019)
```

After running the application, we get an exception in the logcat, the tool we have to view the logs of an Android application in a device. As you can see in Figure 3.3, we get an exception that a class is not found.

```
2019-10-19 19:23:27.057 22961-22961/com.andreas.javaflights D/AndroidRuntime: Shutting down VM

    --------- beginning of crash
2019-10-19 19:23:27.060 22961-22961/com.andreas.javaflights E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.andreas.javaflights, PID: 22961
    java.lang.NoClassDefFoundError: Failed resolution of: Lkotlin/jvm/internal/Intrinsics;
        at com.andreas.flights.FlightsKt.getFlightFor(Unknown Source:8)
        at com.andreas.javaflights.MainActivity$1.onClick(MainActivity.java:33)
        at android.view.View.performClick(View.java:6597)
        at android.view.View.performClickInternal(View.java:6574)
        at android.view.View.access$3100(View.java:778)
        at android.view.View$PerformClick.run(View.java:25885)
        at android.os.Handler.handleCallback(Handler.java:873)
        at android.os.Handler.dispatchMessage(Handler.java:99)
        at android.os.Looper.loop(Looper.java:193)
        at android.app.ActivityThread.main(ActivityThread.java:6669) <1 internal call>
        at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:493)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:858)
```

Fig. 3.3 Kotlin class not found crash log

This was unexpected but after thinking what the problem could be, the fact that we are executing Kotlin code in a Java Android application that gets executed in the Dalvik Virtual Machine (*DVM*), which executes *dalvik* code (compiled Java code for the DVM) made sense, we are missing the Kotlin standard library for the Kotlin code. So, after adding the Kotlin standard library to the application dependencies the application worked as expected as you can see in Figure 3.4.

[13] You can find all the source code of the experiments in GitHub: https://github.com/andreasluca/TFG-JavaKotlinInteroperability.

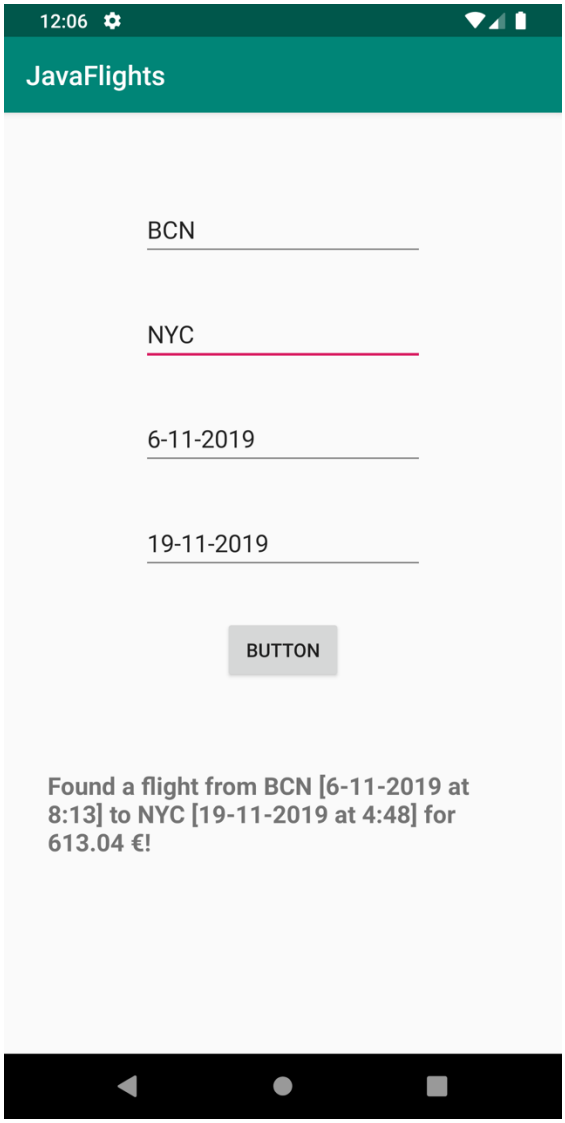

Fig. 3.4 Screenshot of the application that retrieves flight information

Using Kotlin code in a Java application is very easy but there are few things to keep in mind. The first is that if you don't specify the name of your Kotlin classes, when using them in a Java application the compiler will name them with the name of the file and it will add the "Kt" keyword at the end, so if your IDE does not help you when invoking the Kotlin class this is an important issue to keep in mind and not turn crazy because you can't find your file because the Kotlin class does not have the name you wrote. Nevertheless, you can add an annotation in the Kotlin code to tell the compiler what name to put to the class when compiled. Another issue to keep in mind is that if your application is in Java and you integrate Kotlin code you have to add the Kotlin standard library if you don't want it to crash, this is a problem because you get this error at runtime and not at compile, so you make think everything is working as expected until the application executes the Kotlin code and it does not find the standard library. Android Studio could help the developer by checking if there is any Kotlin library used in the project at compile time and throw a warning indicating that the Kotlin standard library is not included. Apart from the commented issues, the interaction between Java and

Kotlin is very fluid, so there isn't any problem in calling Kotlin code from Java and vice versa.

3.2.2 Calling Java from Kotlin

We have already called a Kotlin code from a Java application, now we are going to experiment, the Kotlin-Java interpolation the other way around, we are going to call Java code from Kotlin[14].

The experiment will consist in using a stable Android library such as GSON, a library developed by Google widely used to serialize JSON strings into Java objects, to serialize data.

We will first create an Android application project in Android Studio, selecting Kotlin as the main language. Then, we will add the GSON dependency to the gradle file as shown in Figure 3.5.

```
dependencies {
…
    implementation "com.google.code.gson:gson:2.8.5"
…
}
```

Fig. 3.5 Gradle file showing the dependencies section

Now we can parse JSON strings into Kotlin objects. We will create some data classes that will be parsed from the JSON. For the example we will create a "Group" object that has a name, which is a string, and an array of "Members", which is another object that is defined by a name, which is a string, a departure airport, which is also a string, and a "DNI" object, which contains the DNI number in a string format and a boolean saying if it is expired. You can see the created objects in Figure 3.6.

```
data class Group(
    val name: String,
    val members: Array<Member>
) {
    fun getDepartureAirports(): Array<String> {
        return members.map { member ->
            member.departureAirport
        }.toTypedArray()
    }
}

data class Member(
    val name: String,
```

[14] You can find all the source code of the experiments in GitHub: https://github.com/andreasluca/TFG-KotlinPoCs.

```
    val departureAirport: String,
    val dni: DNI
)

data class DNI(
    val number: String,
    val isExpired: Boolean = false
)
```

Fig. 3.6 Kotlin data classes

Now, we can parse a JSON string into these objects by writing the code shown in Figure 3.7.

```
val json = "{\n" +
        "  \"members\": [\n" +
        "    {\n" +
        "      \"name\": \"Marc\",\n" +
        "      \"departureAirport\": \"NYC\",\n" +
        "      \"dni\": {\n" +
        "        \"number\": \"AABBCC\",\n" +
        "        \"isExpired\": false\n" +
        "      }\n" +
        "    },\n" +
        "    {\n" +
        "      \"name\": \"Ruben\",\n" +
        "      \"departureAirport\": \"LDN\",\n" +
        "      \"dni\": {\n" +
        "        \"number\": \"BBCCDD\",\n" +
        "        \"isExpired\": true\n" +
        "      }\n" +
        "    },\n" +
        "    {\n" +
        "      \"name\": \"Benjamin\",\n" +
        "      \"departureAirport\": \"BCN\"\n" +
        "    },\n" +
        "    {\n" +
        "      \"name\": \"Invalid\"\n" +
        "    }\n" +
        "  ],\n" +
        "  \"name\": \"Brothers\"\n" +
        "}"
val group = Gson()
    .fromJson<Group>(
        Gson().fromJson<JsonElement?>(
            json,
            JsonElement::class.java
        )!!.asJsonObject,
        Group::class.java
    )

group.getDepartureAirports().forEach {
    Log.d("KotlinApp", "Departure airport: $it")
}
```

Fig. 3.7 Parse a JSON string into a Kotlin object

The parsing is correctly done, therefor calling Java code from Kotlin works perfectly.

One issue to be commented is that in the input JSON there is a member object, the “Invalid” one, that has not specified the “departureAriport” but regardless it is not specified even if we access it in the “getDepartureAirports” we don’t get a null pointer exception as we would get in Java, but it prints null as you can see in Figure 3.8.

```
com.andreas.kotlinapp D/KotlinApp: Departure airport: NYC
com.andreas.kotlinapp D/KotlinApp: Departure airport: LDN
com.andreas.kotlinapp D/KotlinApp: Departure airport: BCN
com.andreas.kotlinapp D/KotlinApp: Departure airport: null
```

Fig. 3.8 Logs printed by the application

After debugging the application, we could see that regardless the application didn’t throw a null pointer exception the value of “departureAirport” for the member “Invalid” was null! Therefor the null reference appeared even in Kotlin code.

Having realised that, I thought of a way of getting a null pointer exception in Kotlin by just adding a method to the “DNI” model and invoking it from a member that did not have the “DNI” value set as you can see in Figure 3.9.

```
data class DNI(
    val number: String,
    val isExpired: Boolean = false
) {
    fun createNullPointerException(): Int {
        // This has to throw a NPE if the DNI is not present in the Member
        return 123
    }
}


…

// This does not throw an exception
group.members[0].dni.createNullPointerException()
// This throws a Null Pointer Exception!!!
group.members[2].dni.createNullPointerException()
```

Fig. 3.9 Code modifications to get a Null Pointer exception in Kotlin

And after executing the code we get the null pointer exception as you can see in Figure 3.10.

```
.NullPointerException: Attempt to invoke virtual method 'int com.andreas.kotlinapp.DNI.createNullPointerException()'
```

Fig. 3.10 Null Pointer Exception in a Kotlin application

To sum up, Java code can be invoked from a Kotlin application perfectly. But during the experimentation of this feature we found two issues to comment. The first one is that the syntax of the Java code is modified when used into a Kotlin file. And lastly, the critical issue, we got a null reference in Kotlin code in a non-optional parameter. Therefore, combining Java with Kotlin needs to be done carefully as you cannot expect the full safety feature that full Kotlin code offers.

3.3 Asynchronous programming with Kotlin Coroutines

Kotlin coroutines is one of the most loved features. The possibility of writing asynchronous code as if it was synchronous when reading the code is awesome. Not only this, but the fact that asynchronous can be done in a very simple and clean way has given Kotlin Coroutines this popularity[15].

The first thing to do is to add the Kotlin Coroutines dependency to our project, as shown in Figure 3.11.

```
dependencies {
…
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.1.1"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.1.1"
…
}
```

Fig. 3.5 Gradle file showing the dependencies section

Now that coroutines are imported we will parse the JSON from the chapter above within the scope of an Android *AsyncTask* with callbacks, which is the way asynchronous programming was done in Android application, as you can see in Figure 3.6.

```
class JsonParser: AsyncTask<String, Void, Group?>() {

    private var onComplete: OnTaskCompleted? = null

    interface OnTaskCompleted {
        fun onTaskCompleted(group: Group)
```

[15] You can find all the source code of the experiments in GitHub: https://github.com/andreasluca/TFG-KotlinPoCs.

```
    }
    fun onFinish(onComplete: OnTaskCompleted) {
        this.onComplete = onComplete
    }

    override fun doInBackground(vararg p0: String?): Group? {
        val group = Gson()
            .fromJson<Group>(
                Gson().fromJson<JsonElement?>(
                    MainActivity.GROUP_JSON,
                    JsonElement::class.java
                )!!.asJsonObject,
                Group::class.java
            )
        onComplete!!.onTaskCompleted(group)

        return null
    }
}

// Then in the main thread
val parser = JsonParser()
parser.execute(GROUP_JSON)
parser.onFinish(object : JsonParser.OnTaskCompleted {
    override fun onTaskCompleted(group: Group) {
        group.members.forEach {
            Log.d("KotlinApp", "From: ${it.departureAirport}")
        }

        throwNullPointerException(group)
    }
})
```

Fig. 3.6 Implementation of an AsyncTask

To understand the code above you need to have some prior knowledge on the threading and asynchronous programming, also, in this case is not that difficult to follow but normally in large projects the asynchronous tasks are very difficult to understand due to the fact that you need to have many things in mind.

That why, the next step is converting the parsing of the JSON in a Kotlin Coroutine to see the differences, see Figure 3.7.

```
private suspend fun parseJsonAsyncTask(): Group {
    delay(1000) // We need to mock we are doing some slow task
    return Gson()
            .fromJson<Group>(
                Gson().fromJson<JsonElement?>(
                    GROUP_JSON,
                    JsonElement::class.java
                )!!.asJsonObject,
                Group::class.java
            )
}

// Then in the main thread
GlobalScope.launch {
    parseJsonAsyncTask().members.forEach {
```

```
        Log.d("KotlinApp", "${it.departureAirport}")
    }
}
```

Fig. 3.7 Kotlin coroutines in action

The Kotlin Coroutines approach is was easier to understand and shorter. You don't have to implement any interface, nor create boiler-plate code for doing asynchronous operations.

There for in any case an asynchronous operation is required, due to networks calls, storage operations, or any other kind of operation, using the coroutines approach will save a lot of time.

3.4 Conclusions

Now that we have done different proofs of concept of the different features, we know for fact that working with Kotlin is straight forward, the IDE helps a lot by providing examples and hints for enhancing our code. Also, the Kotlin developer community is starting to grow a lot, so, whenever we had a practical issue there was already some feedback given from the community, it is not the case for some theoretical aspects.

At the beginning, the concept of creating a multiplatform, sound a bit scary, because normally when you build something that has lots of dependencies you have to watch out a lot of different things and it is very difficult to find a solution, but with the support given from JetBrains creating a multiplatform project is not that scary. Also, we have built a shared mobile application but there are many other possibilities for multiplatform code, such as sharing client and server logic.

Also, the Java interoperability, is as good as expected. Following the theory, having Kotlin getting compiled in Java bytecode, we could proof that there is a lot of investment of that smooth interoperability that lets you, either migrate your project or use old libraries in Kotlin projects.

Finally, Kotlin coroutines revolutionize the way we do asynchronous programming, as they are very clean, very easy to use and very efficient. This allows a beginner to understand the flow of an application without losing its mind reading and remembering where there was a callback and what was that callback doing. And also helps to have a better code maintenance.

CHAPTER 4. BUILDING AN ANDROID APPLICATION WITH KOTLIN

So far, we know what Kotlin is, what can we build with Kotlin, what features does Kotlin have and we also have taken a practical look at some of its features. Now, let's go a step further and try to build an entire Android application with Kotlin.

Almost all applications we have in our smartphone developed by third party companies communicate with a Server that normally fetches data for them. Then, we want to communicate to a Server too in order to see how can to retrieve remote data.

After some brainstorming, some ideas came up on what application to build in order to test Kotlin as the development language for an Android application and the most interesting one was to build an application that looks for flights.

Not just a simple application that looks for a certain flight but an application that looks for flights from several airports and just one destination. Basically, an app for finding flights for you and your friends that live abroad, for example, to the same destination. This implies, creating groups of people for which to look for the flights, and retrieving a flight for each as shown in Figure 4.1.

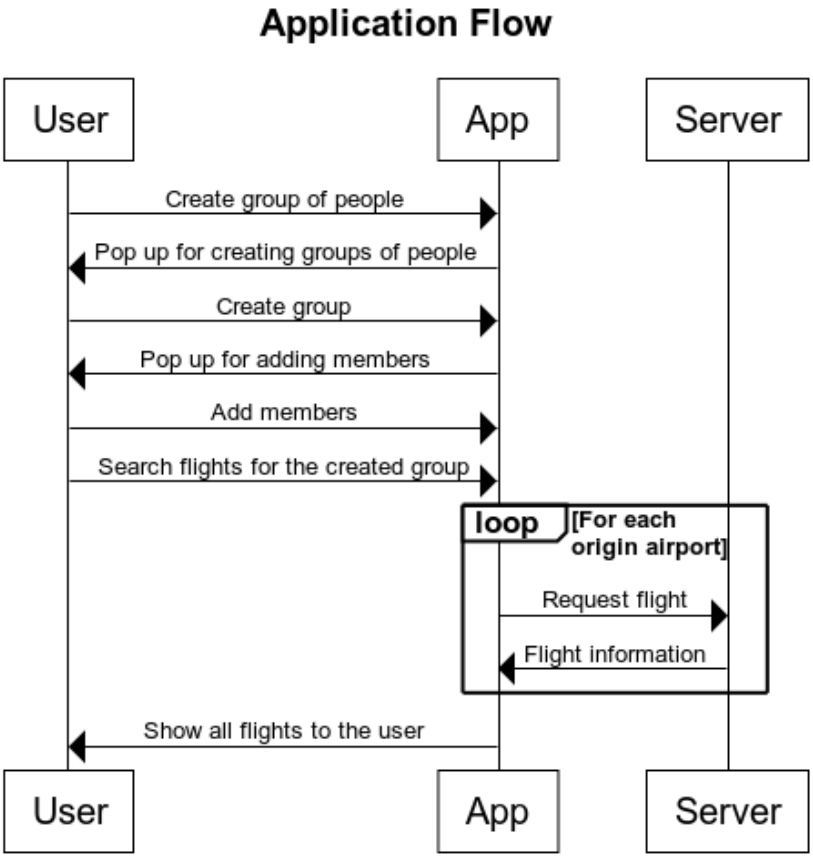

Fig. 4.1 High level detail of the application flow

The goal is to build an application[16] that uses and interacts with some of the features that Kotlin enhances for Android development. By the simple fact of using Kotlin for the whole development we are going to be able to see most of the features described in Chapter 2 such as the conciseness of Kotlin, the fact that it's a modern language and lets the developer write code in a more intuitive way, the benefits of using its standard library, the safeness of avoiding null references and so on.

For the development we are going to use Android Studio as the IDE, as it is based on the famous Java IDE IntelliJ IDEA, developed by the same company that developed Kotlin. Also, Android Studio is the IDE recommended by Android to develop Android applications and it already has Kotlin support.

4.1 Application flow

In order to have an easy to use application, we will try to have as less workflows as possible for the user. So, for an application that searches flights for a group of people we can define two concrete flows.

These two workflows consist in: creating groups of people for who to search the flights and the actual search for the flights.

In the first use-case, the user opens the application and clicks on the configuration menu to create a group by setting a name for the group. Then he just clicks on the created group and starts adding people as members of this group. The members added to the group need to specify an origin airport, from where to get the departure option for the flight request.

The second workflow consists in the actual search of the flights.

Once we have the groups of people created, in the landing view, we have a form to complete. We need to select the group of people, created in the previous workflow, for who we are searching the flights. The group of people parameter substitutes the typical "From:" input box that we have in all flight searches websites. What the group of people parameter actually does is giving an array of departure airports instead of just one. Then we have the destination airport input box and the option to select the one-way departure date and also the optional return date. Finally, for submitting the flight search request we have the search button. For a graphical insight of the view see Figure 4.2.

[16] You can find all the source code of the application in GitHub: https://github.com/andreasluca/coFly.

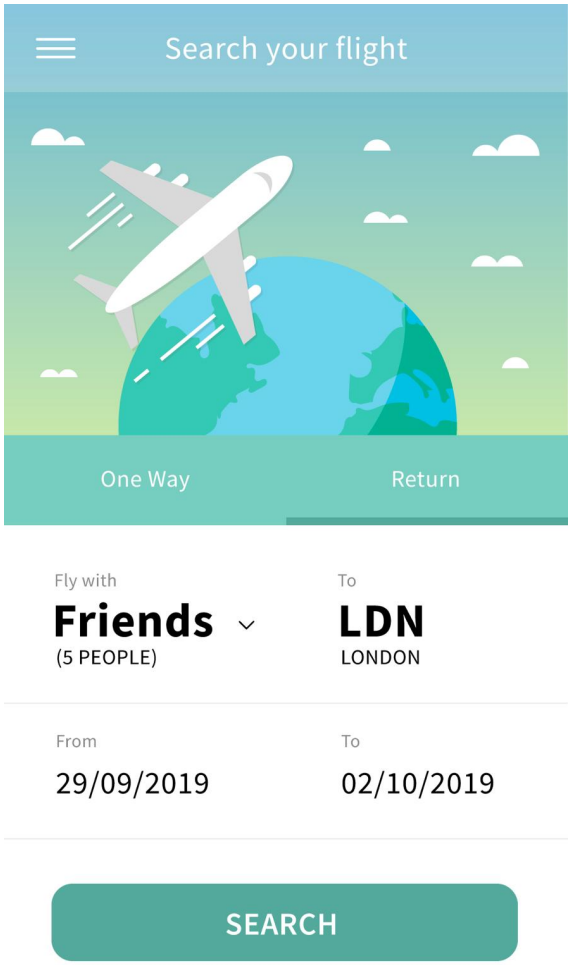

Fig. 4.2 Main user interface for looking for a flight

Therefore, after the user inputs the queried, they are sent to the business logic of the app that for each member of the group looks for a flight for the given destination airport and date.

After the network calls are performed for each member, the result is either a new view with the list of flights from each origin airport in the successful case. Or if the query fails, due to network issues or because there are no flights available for the requested parameters a pop-up message appears informing the user that there were no flights for the requested query.

For each found flight the information displayed is the departure and arrival airports and dates, the duration of the flights and the total price as you can see in Figure 4.3.

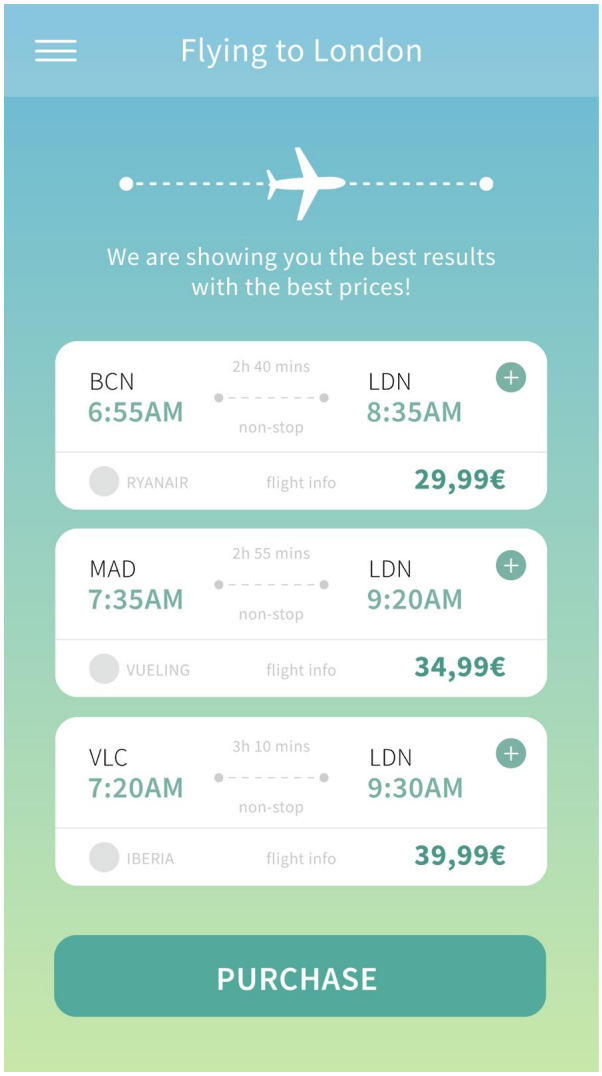

Fig. 4.3 Retrieved flights

4.2 Technical considerations

4.2.1 Retrieving flights information

Several options came to my mind when wondering where to retrieve the data for the flights search.

The first idea was to create a NodeJS server that would mock flights results and serve them via a REST interface. But there were several inconveniences, one of them was the fact that fake data is not real data (and mocking data is not that cool unless it is a test), another issue was the fact that normally in real world applications retrieving data from a server requires authentication, and normally not just a basic HTTP authentication but implementing secure protocols such as OAuth 2.0 and this would take a lot of time in the development of a mock server that goes out of the scope of this paper.

Secondly, I thought on using the most used search flight platform: SkyScanner. The only and important problem is that their API is closed to patterns. This didn't use to be the case but due to its popularity they decide it to close it.

So, I started researching for open flight search APIs and I found one that offered: OAuth 2.0 authentication on an open API for searching flights, Amadeus. Amadeus is an IT company that offers IT services for travel industry. Amadeus offers several open APIs for searching flights but with a 3000 calls per moth limit, which actually met my needs.

4.2.2 Android application architecture

In order to start coding the application there were several architectural decisions that needed to be taken. For example, coding an application from scratch or coding the application with the Anko library that we covered in Chapter 2.

In order to go code with as many Kotlin features as possible I decided to code the application without the Anko library, as it is not very commonly used, and I wanted the developed application to be coded with the most common techniques that our mobile applications have.

Having a clear application architecture form the beginning is key, as normally when an application is being developed without considering any architecture it ends up being unmaintainable and with a lot of headache whenever you want to change anything. Therefor I had to study the most common application architectures.

The Model-View-Controller (MVC) pattern is one of the most used for application architecture in general. The MVC pattern consists in three components, which are implicit in its name. The model is in charge of defining the data layer of the components of the application. The view is responsible of showing the models in the user interface, this is its only responsibility, showing the data to the user in a cool way. And the controller, which is the part that has all the logic. It's notified when the user interacted with the view and updates the model with the respective action.

After some years of Android development being active the expert developers propose that the Android classes: Activity, Fragment and View should take the role of the view in the MVC pattern, see [5]. Then the controller should be a separated class that does not extend any Android class and the same for the model.
The only problem with this approach is that the controller needs to tell the view to update when the model changed, therefor there should be an interface for the view, the controller should have a reference of the view and the view should implement the interface. This has multiple benefits, one of them is that whenever you want to refactor the whole view, which is something that usually happens due to new user interface updates, the new view just has to implement the view interface and the controller will be able to work with it. Another benefit it the application can be tested without problems, as all classes follow the single

responsibility principle. Also integrating new features should not be any problem and code readability is ensured.

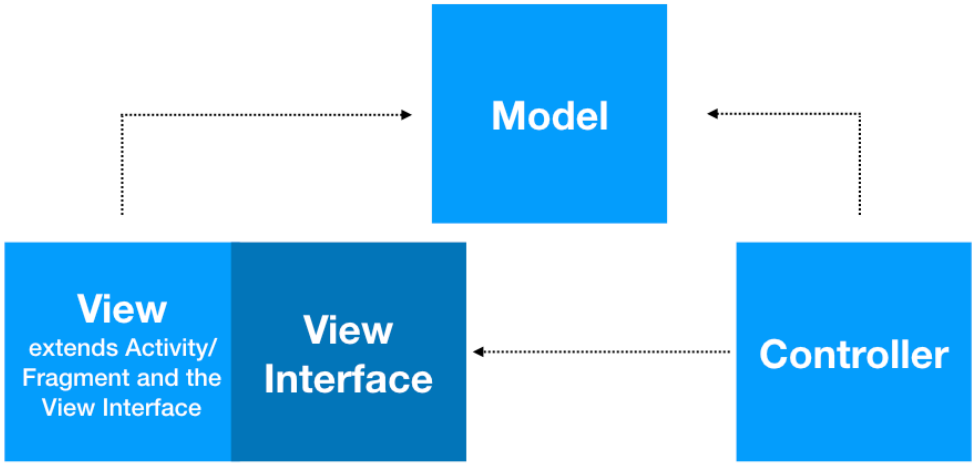

Fig. 4.4 Model-View-Controller structure

The Model-View-Presenter (MVP) pattern works very similar to the MVC pattern, see [6]. We again have three main components implied in the name of the pattern. The model, as in the MVC pattern, is in charge of defining the data layer of the components of the application. The view is responsible of showing the models in the user interface and notifying the presenter about any change or interaction with the user. The presenter reacts to the user interactions, updates the model and decides what to show in the view, basically it does all the trick, it's responsible of what to show, interact with the user actions and update the model.

Usually in order to make the whole application testable a contract is defined between the view and the presenter and that way they can be abstract to each other but on the other way they can still hold reference on each other. The contract defines an interface to be implemented by the presenter and referenced by the view and an interface to be implemented by the view and referenced from the presenter. This has its disadvantages such as the management of the user interface is done by both the view and the presenter, and the presenter has a lot of information but still this solution separates very well the three components and lets you have a clearer flow.
For Android applications this architecture makes more sense when the application is big, and for having a clearer flow to follow.

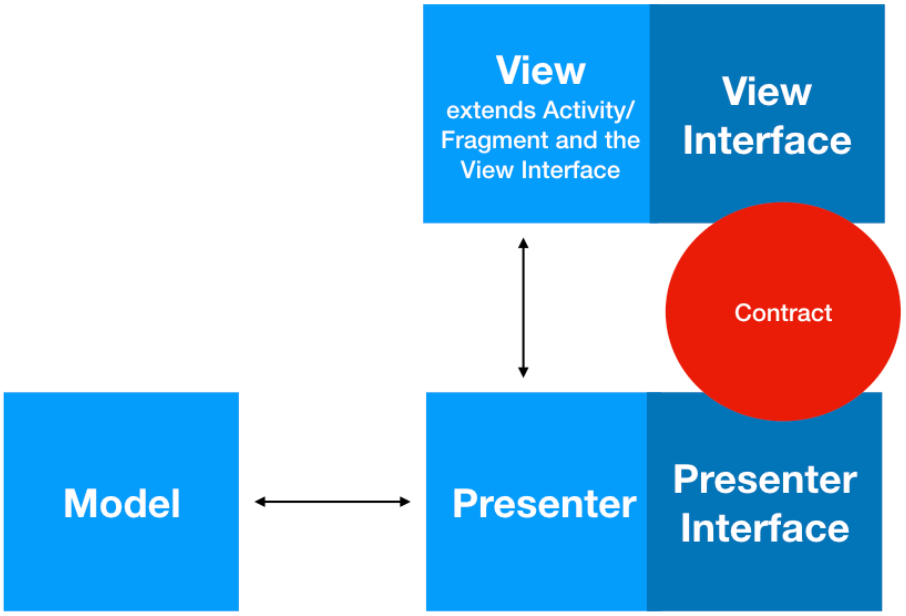

Fig. 4.5 Model-View-Presenter structure

The last architectural pattern studied is the Model-View-ViewModel (MVVM), see [8]. In this case we, again, have three major components. The model does the same as in the other two patterns, it is in charge of defining the data layer of the components of the application. The View notifies the ViewModel of user actions. And lastly, the ViewModel works with the model in order to tell the View what to show, therefor the ViewModel is a model of the view. This architectural pattern looks similar to the MVP pattern, in fact all of them have their similarities and their differences but choosing the correct one for the kind of application you do is key. In this case, the MVVM is more likely to be used in events-based applications, the view can easily bind the ViewModel.

Fig. 4.6 Model-View-ViewModel structure

The decision on what architecture to follow was difficult due to the similarity and the unknown factor of not having designed any system based on these architecture patterns before. But after some thought's the pattern that most convinced me for the kind of application built, the flight search app, was the MVP as it has a very clear separation of concerns that made me create a very straight forward flow.

Another aspect to keep in mind for the architecture of the application was the management of the data retrieved from the Amadeus REST API. Therefor researching how do real applications deal with the data source for their models I found the repository pattern. The repository pattern consists in a decoupling the data source of the application from the model, so that whenever changing the data source, the only change required would be on the repository part and we would have nothing to touch in the actual architecture of the application. Also, some other benefits that it offers are: the centralization of all the data sources of the application (in case there is more than one), the possibility to add very easily new data sources, the abstraction on where the data comes from, it can either come from a database or from a remote server.

While researching about application architecture there was an aspect that was not in my consideration, the dependency injection. Having a dependency injection strategy in your application is key for having a clean architecture overall the application. However, all the existing frameworks or dependency injection, such as Ktor or Dagger, add too much overhead in simple applications, they are very useful for large projects, therefor I came to the conclusion of having a self-implemented solution for dependency injection as you can see in Figure 4.7.

```
package com.andreas.fly

import AmadeusSecurityApi
import com.andreas.fly.repository.AmadeusRepository
import com.andreas.fly.repository.AmadeusShoppingApi
import com.andreas.fly.repository.Repository
import com.andreas.fly.repository.authentication.AccessTokenAuthenticator
import com.andreas.fly.repository.authentication.AccessTokenProvider
import okhttp3.OkHttpClient
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object Injector {

    private fun injectAmadeusShoppingRetrofit(): Retrofit {
        return Retrofit.Builder().apply {
            baseUrl(Constants.AMADEUS_SHOPPING_BASE_URL)
            addConverterFactory(GsonConverterFactory.create())
            client(injectAuthenticatorOkHttpClient())
        }.build()
    }

    private fun injectAmadeusSecurityRetrofit(): Retrofit {
        return Retrofit.Builder().apply {
            baseUrl(Constants.AMADEUS_SECURITY_BASE_URL)
            addConverterFactory(GsonConverterFactory.create())
        }.build()
    }

    private fun injectAmadeusSecurityApi(): AmadeusSecurityApi {
        return
injectAmadeusSecurityRetrofit().create(AmadeusSecurityApi::class.java)
    }

    private fun injectAuthenticatorOkHttpClient(): OkHttpClient {
        return OkHttpClient.Builder()
```

```
.authenticator(AccessTokenAuthenticator(AccessTokenProvider.getInstance(inje
ctAmadeusSecurityApi())))
            .build()
    }

    private fun injectAmadeusShoppingApi(): AmadeusShoppingApi {
        return
injectAmadeusShoppingRetrofit().create(AmadeusShoppingApi::class.java)
    }

    private fun injectRepository(): Repository {
        return AmadeusRepository(injectAmadeusShoppingApi())
    }

    fun injectMainPresenter(): MainPresenter {
        return MainPresenter(injectRepository())
    }
}
```

Fig. 4.7 Self implemented injector

So, the final product of the research for the architectural proposal for the application results in a combination of the MVP pattern plus the repository pattern with a self-implemented dependency injector as you can see in Figure 4.8 and Figure 4.9.

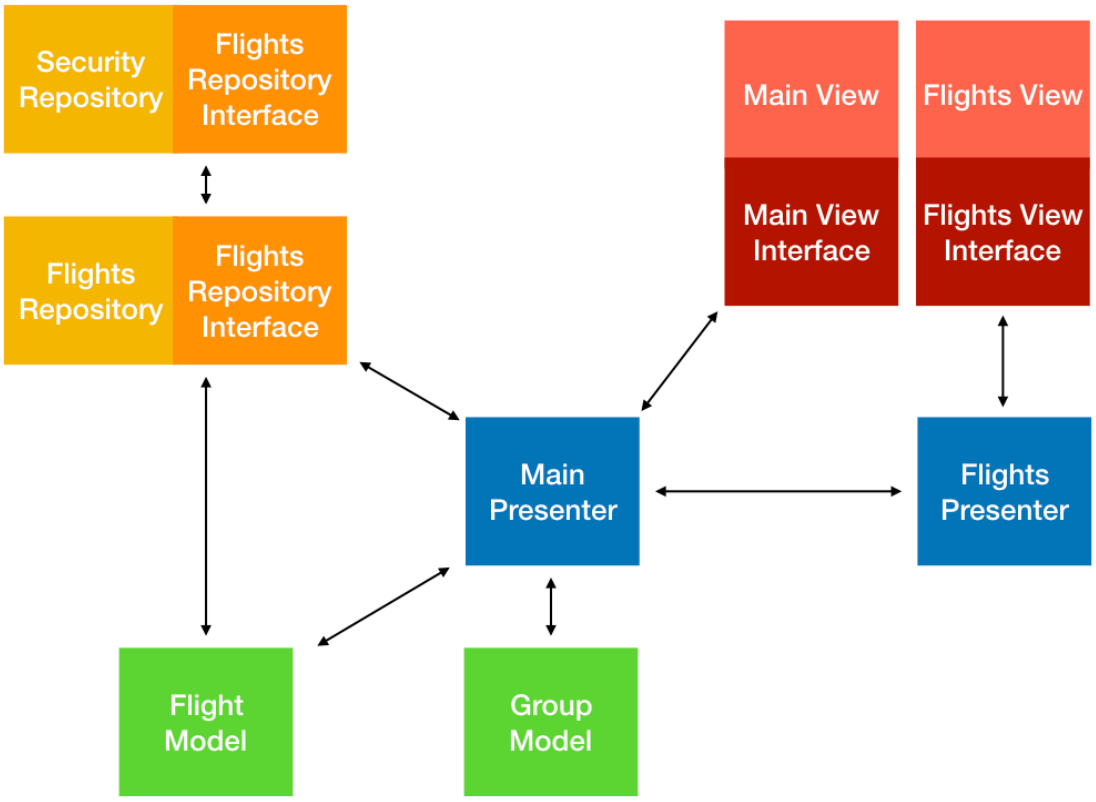

Fig. 4.8 High level architecture of the application

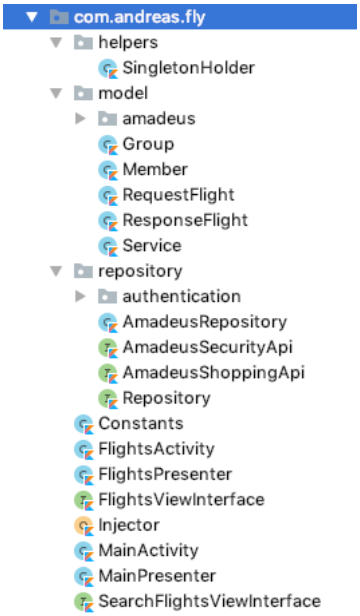

Fig 4.9 Project structure of the application

4.3 Conclusions

After having built an entire Android application with Kotlin that retrieves flights for groups of people taking full advantage of the features offered by Kotlin we can conclude that it makes you way more productive.

For example, by using the Kotlin Android Extensions to bind the data in the user interfaces, you forget of writing a lot of boilerplate code for a very common operation. Also, creating new data classes saves a lot of time not having to create all the getters, setters and override methods.

Also, using Kotlin Coroutines for the asynchronous operations makes the code very clear and easy to read, which end up saving a lot of time whenever you want to debug something.

Hence, by the mere fact of using Kotlin during the development made the amount of unexpected crashes decrease drastically. Actually, the only runtime crashes were due to the fact of a bad serialization of the JSON received from the server. Therefor we can conclude we have built safer code with less lines of code.

Lastly, it was a very good decision to study the common architectural patterns and applying them because it has organised the structure of the project in a very simple way, so that whenever a change is needed it will only affect the component that needs this change. Also, changing the flights provider would be very easy or any user interface change would not have any impact in the business logic of the applications.

CONCLUSIONS

Objectives

The result of the project is a positive validation of Kotlin's claim of being a concise, safe, interoperable and tool-friendly programming language.

First, we have gone through a theorical study of Kotlin's compilation process, mainly to understand how it builds the Kotlin code into platform specific code. In other words, from Kotlin code to Java bytecode, JavaScript or even native code. This proofed, in a theoretical way, that Kotlin is interoperable with existing code, as depending on the compilation target it is built with it has interoperability with Java libraries or JavaScript frameworks.

Secondly, we researched what features does Kotlin have and what problems do they solve. The result of such research happens to coincide with Kotlin's claim of being a concise programming language. In regard to conciseness what most of the features do is help the developer write less boilerplate code with a modern language. Some of the features that make Kotlin code concise are the possibility of adding extension functions to any class, the use of its standard library, a bunch of extension functions that do a lot of repetitive tasks for you, and the fact that it is a modern language with type inference.

After having a broader vision of Kotlin, we moved to do several proofs of concept in order to evaluate in a practical way some of the theoretical features. As a result, we found that creating multiplatform projects is very straight forward with IntelliJ IDEA, what else could we expect from the company under the programming language and the IDE. Also, working with Kotlin in Android Studio is straight forward as for when integrating the Kotlin library into a Java Android application it would automatically detect it an add its dependency. Therefor we could also validate that Kotlin is tool friendly. Not everything was perfect, we faced an issue when mixing Kotlin and Java code. Even if they were perfectly interoperable, we could see that Kotlin is not that null safe in combination with Java, as Java code can bring the null references into Kotlin code.

Lastly, we built an entire Android application to put everything together. During the development of the application the only runtime crashes found were because of a bad serialization of JSON data. All the other possible crashes were solved because a compilation error was thrown before launching the application. The issued runtime crash was due to a misspell of an expected object parameter when the JSON was being serialized, therefor the crash was because some parameters where not expected, and it was not handling properly. But if I didn't misspell the parameter, I would not have had any critical crash during the development. This makes Kotlin a very safe programming language. In the paragraph above we were saying that Kotlin in combination with Java is not that safe, but when we go for a full implementation in Kotlin it is.

The development of the application was fast as a previous analysis of the architecture was done, but also because a lot of boiler code was avoided with the Android Kotlin Extensions and with the conciseness of such modern language.

Consequently, using Kotlin in application development has also resulted in having better applications as for: there are null checks at compile time, avoidance of the callback hell and the reduction of code complexity.

Inconveniences

During the study of Kotlin, not many inconveniences appeared. One of them is that even if it has a great documentation, when you want to dig a little bit more into its internals it is difficult to find something useful, therefor the most difficult to write chapter was the first one, as it does an analysis of the compilation process for each platform.

Another inconvenience is that when building a mobile shared library multiplatform project the output is a Java Archive in order of being an Android Archive, this second one is more optimized and it already contains Dalvik bytecode, the code that Android understands, instead of Java bytecode, which needs to be compiled into Dalvik.

Also, the mix of Java and Kotlin can lead to bringing null reference back to Kotlin, as for so in mixed application it is recommend having a good isolation between the Kotlin code from the Java.

Finally, from the security point of view of an Android application built with Kotlin, even if Kotlin is a new programming language, Kotlin Android applications can be easily reverse engineered with current tools that reverse Java code. This means that people that were reversing Android applications did not have to make any effort for adapting any tool for reversing Kotlin built applications.

Personal conclusions

From the personal point of view, studying Kotlin has made me understand lots of concepts from the software engineering world. Some of those are:

- The Java Virtual Machine world, from how most JVM programs get compiled to understanding what JVM itself is.
- Architecture for mobile applications.
- Learning a top new programming language that is building a lot of professional careers.

Apart from that, I could also see what some foundations of Kotlin’s success as a programming language are:

- JetBrains has some of the most used IDE's, therefor it has a lot of feedback on what type of programming language do developers use so it started the Kotlin project based on this.
- Strong names, Google and JetBrains, are behind the Kotlin project maintaining it and investing in it.
- A good documentation and lots of sample code, that help the developer not being afraid of starting a new project.
- A very intuitive syntax that leads to a very good learning curve.
- The possibility of doing multiplatform projects where the business logic is shared.
- The fact that for Android development Kotlin can run in old Android devices as its first version is based in Java 6, hence there is not this fear or migrating to Kotlin for not breaking backwards compatibility.

In case I have to build an Android, I will for sure chose Kotlin as the feedback I can give from executing this project is very good.

Future work

This project can be extended in many ways. An approach would be to build the same final application but using some of the frameworks or features mentioned in the project. For example, using Anko, the Android Kotlin framework described in the second chapter, and analysing the differences between the application built with plain Kotlin and the one built with Anko. Also, another option would be to move all the logic of the application into a multiplatform project and use it in an iOS project. Not to go further, if the logic of the application is moved to a multiplatform project a website could also be built using that logic.

From another perspective, the project could also be extended by doing an analysis of several factors such as the performance of the same Kotlin application built in Java. Also, comparisons of the application size of a Java application and a Kotlin application and many more comparable parameters.

BIBLIOGRAPHY

[1] Kotlin’s official webpage, [Online] Available: https://kotlinlang.org/

[2] Kotlin’s official documentation, [Online] Available: https://kotlinlang.org/docs/reference/

[3] S. Wirtz, “Kotlin on the JVM - How can it provide so many features?”, Kotlin Experise Blog 2017 [Online] Available: https://kotlinexpertise.com/kotlin-byte-code-generation/

[4] A. Leiva, “Kotlin Android Extensions: Say goodbye to findViewById”, Anotonio Leiva’s Blog 2017 [Online] Available: https://antonioleiva.com/kotlin-android-extensions/

[5] F. Muntenescu, “Android Architecture Patterns Part 1: Model-View-Controller”, Medium 2016 [Online] Available: https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6

[6] F. Muntenescu, “Android Architecture Patterns Part 2: Model-View-Presenter”, Medium 2016 [Online] Available: https://medium.com/upday-devs/android-architecture-patterns-part-2-model-view-presenter-8a6faaae14a5

[7] F. Muntenescu, “Android Architecture Patterns Part 3: Model-View-ViewModel”, Medium 2016 [Online] Available: https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b

[8] Y. Shah, J. Shah and K. Kansara, "Code obfuscating a Kotlin-based App with Proguard", 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC) Advances in Electronics, Computers and Communications (ICAECC), 2018 Second International Conference on. :1-5 Feb, 2018, [Online] Available: https://ieeexplore-ieee-org.recursos.biblioteca.upc.edu/document/8479507/