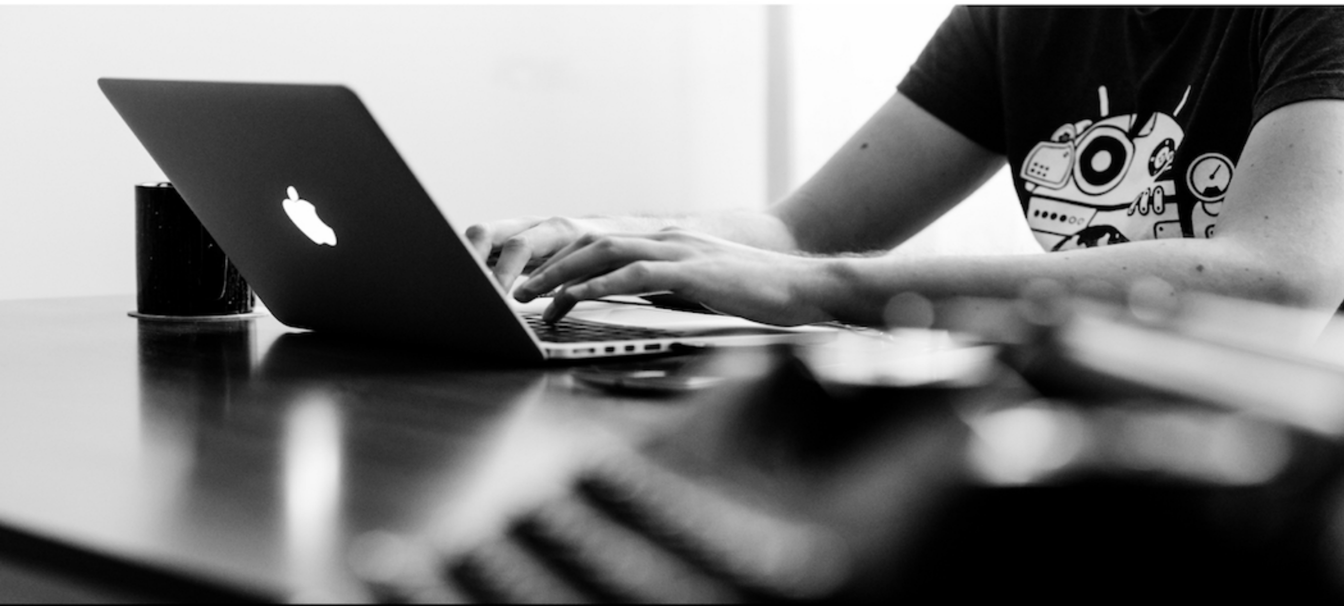
5 TOP GUIDES TO

MASTERING KOTLIN

Learn the tricks that will let you take the most out of the language

ANKO | KOTLIN DSL | COROUTINES
KOTLIN ANDROID EXTENSIONS |JAVA INTEROPERABILITY



antonioleiva.com

5 top guides to mastering Kotlin

Learn the tricks that will let you take the most out of the language

Antonio Leiva

This book is for sale at http://leanpub.com/mastering-kotlin

This version was published on 2018-09-25

Tweet This Book!

Please help Antonio Leiva by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just got ”5 top guides from mastering kotlin” by @lime_cl

The suggested hashtag for this book is #masteringkotlin.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#masteringkotlin

Contents

Introduction

Several years ago I discovered Android by chance. I was no longer enjoying the kind of software I developed at that time and was in search of new adventures.

Android fascinated me since the very beginning: in a few hours, you could have an App running on your mobile and, with little skills, create your first test screens. Everything could be done from your home and with free software.

But there was something in Android development that I didn't quite like much: the language. At work, I was using C#, a language that has always been at the forefront regarding features, and with Java, I was jumping several steps backward.

Concepts such as properties, null treatment or immutability were already present in most languages, but not in Java. Also, in Android we have the added problem that most devices need to compile using an old version of Java, so we couldn't even take advantage of the small language improvements.

That's why, for me, Kotlin was a turning point. The day I decided to try Kotlin, it was love at first sight. A world of possibilities appeared right in front of my eyes, giving me back things that I lost when I switched from C# to Java, but with other new features that made my code much cleaner and more flexible.

But when you learn a new language, not everything is a path of roses. Writing the same code with the new language is easy, but there are new concepts that you have to master and don't know when to use. You need a change of mentality to solve the same problems differently.

For me, that was an essential point in my transition, and that is why since then I dedicated myself to helping other Android developers to follow the same road.

That was 2015, and since then I've led thousands of developers and their companies to boost their Android productivity by switching to Kotlin, in many different ways: hundreds of articles, talks in conferences, the first Kotlin book ever[1], and extremely practical online course[2], live training, 1 to 1 mentoring[3], etc.

[1] https://antonioleiva.com/kotlin-android-developers-book/
[2] https://antonioleiva.com/online-course/
[3] https://antonioleiva.com/mentoring

So if you think I can help you or your company, please don't hesitate to write me to contact@antonioleiva.com[4].

In this ebook, I'll show you some other ways to take the most out of the language. Kotlin is super-powerful, and you will see here that there are little things in Android that you can't do with Kotlin.

So relax and enjoy what I'm bringing for you.

[4]mailto:contact@antonioleiva.com

1 Anko layouts on Android

If you're already developing Android Apps using Kotlin, you've probably heard about Anko layouts and been thinking about using them or at least considered taking a look at them.

The truth is that Anko[5] has been around for really long. In fact, when I started writing my book about Kotlin on Android[6] at the beginning of 2015, this library already existed.

But in case you hadn't heard about it, Anko is a library **developed by the Kotlin** team with the goal of **simplifying the interaction with the Android framework**.

Its most outstanding feature is Anko layouts, which is the main topic of this first guide. But it also has other features, such as small DSL to execute asynchronous tasks[7] in a very simple way, another one to build easy dialogs and alerts[8], a set of functions to deal with SQLite[9], and even an implementation of coroutines[10], among many other things.

I must admit that I'm a huge fan of this library because it has a lot of impressive features and it's been beneficial to me to understand Kotlin and how to apply it to Android development.

But there's something in this library that always pushed me back: **Anko layouts DSL**.

Anko layouts DSL

The idea of Anko layouts is to take the most out of Kotlin to provide a DSL that **makes declaring Android layouts easy and powerful, just by getting rid of the XMLs**.

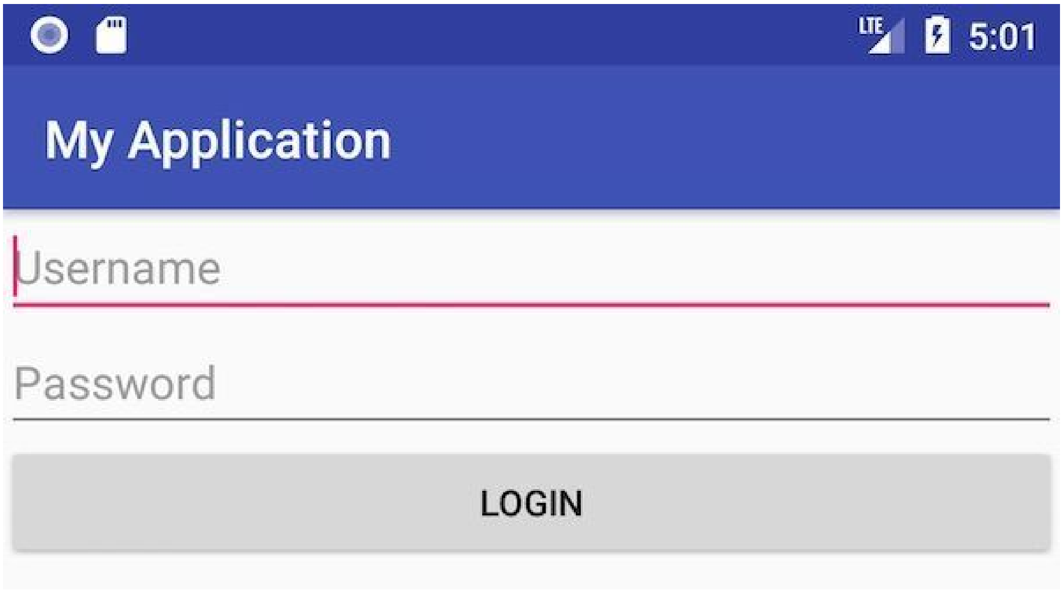
[5] https://github.com/Kotlin/anko/
[6] https://antonioleiva.com/book/
[7] https://antonioleiva.com/anko-background-kotlin-android/
[8] https://antonioleiva.com/dialogs-android-anko-kotlin/
[9] https://antonioleiva.com/databases-anko-kotlin/
[10] https://android.jlelse.eu/a-first-walk-into-kotlin-coroutines-on-android-fe4a6e25f46a

I've been developing Android Apps for like 6 years, and I'm really fluent with XML already. So when I tried to do something with Anko layouts, it was like starting from scratch to me.

But for simple examples, it looked really powerful! I could write a small login form in a breeze. With just this code:

```kotlin
UI(true) {
    verticalLayout {
        val user = editText {
            hint = "Username"
        }
        val pass = editText {
            hint = "Password"
        }
        button("Login") {
            setOnClickListener {
                longToast("User: ${user.text}, Pass: ${pass.text}")
            }
        }
    }
}
```

You'd get something like this:



Pretty awesome, right?

But we all know that small samples are never enough to get the whole picture, so I decided to convert my whole Bandhook-Kotlin repo[11] views to Anko layouts. And based on this experience, I decided to write this text.

Should I start using Anko layouts to build my projects?

The answer to this question is not black or white. Based on my experience, it depends a lot on your personal taste. These are the pros and cons I found:

Pros:

- It's ** straightforward to start with it**: the DSL is really intuitive for simple examples, and the code you need to write is clean and nice to read.

[11]https://github.com/antoniolg/Bandhook-Kotlin/

- It skips the inflation step, so the **views written with Anko are much faster to initialize** that those written in XML. There are some articles like this from Simon Vergauwen[12] that show it's even 4 times faster.
- Using Kotlin code means you can do more complex things: for instance, set a dimension that is the sum of two dimensions, ** more straightforward data binding, views are more natural to reuse and compose**…

But these are the cons I found:

- **You need to start learning from scratch**: new API, new rules. So, at the beginning, you'll find some things much more difficult to do that just by using XML. This shouldn't be a stopper though.
- Some things are inherently more complex, like styling your views, setting layout params.
- **You need to write your views blindly**. You don't have a powerful designer to implement the view, or at least see the view you're writing. That's half-true because there's a plugin that should show a preview, but I've never seen it working, it's broken quite often. You cannot use `tools:` attributes either, which are very useful when doing layouts.
- Some things may be even impossible. There are some XML properties that don't have their equivalent in Java, so you can't do much here.
- In general, **I'd encourage you to try it and take your own decisions**. I know people that love Anko layouts and can understand why. I can find an excellent use case for simple layouts like custom views for dialogs or for `RecyclerView` adapters.

How to start using Anko

For a full reference, I recommend you to take a look at the repository[13], but I'll tell you here some steps that will help you follow the code.

[12]https://android.jlelse.eu/400-faster-layouts-with-anko-da17f32c45dd
[13]https://github.com/antoniolg/Bandhook-Kotlin/

Add the corresponding dependencies

Anko is split into several libraries so that you only add what you need. For views, the base one is this:

```
implementation "org.jetbrains.anko:anko-sdk19:$anko_version"
```

The SDK you use should correspond to the minimum version you support (or previous). There's `sdk-15`, `sdk-19`, `sdk-21`, `sdk-23` and `sdk-25`. So let's say your minSDK is `17`, you should use `sdk-15`. Then, depending on what other support libraries you use, you will add the corresponding Anko one: implementation "org.jetbrains.anko:anko-appcompat-v7:$anko_version"

And there are others for design library, CardView, etc. The complete reference is at Anko repository[14].

The base activity

You can just do a `UI` block as I did in the example above, but this can make your activity grow fast. It's better to have the layout in another class.

So for this, I created a base activity, which will force to declare the property that holds the layout:

```
abstract class BaseActivity<out UI : ActivityAnkoComponent<out AppCompatActivity>>
    : AppCompatActivity() {
    abstract val ui: UI

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        (ui as ActivityAnkoComponent<AppCompatActivity>).setContentView(this)
        setSupportActionBar(ui.toolbar)
    }
}
```

In `onCreate`, it sets the view from the property as the view of the activity, and also uses the toolbar from this layout. The generic type of the activity is just an interface that forces implementers to hold a reference to the toolbar:

[14] https://github.com/Kotlin/anko/

```kotlin
interface ActivityAnkoComponent<T : AppCompatActivity> : AnkoComponent<T> {
    val toolbar: Toolbar
}
```

That way, the activity can use it in `onCreate`.

Building the layout

Each activity holds an `ActivityAnkoComponent`, that will be the one in charge to build the activity. Here's when we'll use Anko layouts:

```kotlin
override fun createView(ui: AnkoContext<MainActivity>) = with(ui) {

    coordinatorLayout {

        appBarLayout {
            toolbar = themedToolbar(R.style.ThemeOverlay_AppCompat_Dark_ActionBar) {
                backgroundResource = R.color.primary
            }.lparams(width = matchParent) {
                scrollFlags = SCROLL_FLAG_SNAP or SCROLL_FLAG_SCROLL or SCROLL_FLAG_ENTER\
_ALWAYS
            }
        }.lparams(width = matchParent)

        recycler = autoFitRecycler()
                .apply(AutofitRecyclerView::style)
                .lparams(matchParent, matchParent) {
                    behavior = AppBarLayout.ScrollingViewBehavior()
                }
    }
}
```

As you can see, the parent class is a `CoordinatorLayout`[15], which holds an `AppBar-Layout` with a `Toolbar`, and a `[RecyclerView]`.

An interesting thing is how the layout params are declared. It's a function where you set width and height by argument (default is `WRAP_CONTENT`), and then a block to add the rest of extra parameters:

[15]https://antonioleiva.com/coordinator-layout/

```
.lparams(matchParent, matchParent) {
    behavior = AppBarLayout.ScrollingViewBehavior()
}
```

Use custom views

By default, we only have functions for framework and support libraries views, but you can create you own. For instance, in the example above I created `autoFitRecycler`:

```
fun ViewManager.autoFitRecycler(theme: Int = 0) = autoFitRecycler(theme) {}
inline fun ViewManager.autoFitRecycler(theme: Int = 0, init: AutofitRecyclerView.() -> Un\
it)
    = ankoView(::AutofitRecyclerView, theme, init)
```

You can know more about this at the repository wiki[16]. ### Applying styles For each view, there exists a themed version that allows you to apply a theme, such as the toolbar here:

```
toolbar = themedToolbar(R.style.ThemeOverlay_AppCompat_Dark_ActionBar)
```

But, if what you want is to apply a style, you cannot use an XML one. Instead, you need to use the functions `apply` or `applyRecursively`. This second one, in case you want to apply the style also to the sub-views:

```
.applyRecursively { view ->
    when (view) {
        is EditText -> view.textSize = 18f
    }
}
```

As a way to extract styles, I implemented this extension function for the recycler:

[16]https://github.com/Kotlin/anko/wiki/Anko-Layouts#is-it-extensible

```
fun AutofitRecyclerView.style() {
    clipToPadding = false
    columnWidth = dimen(R.dimen.column_width)
    scrollBarStyle = View.SCROLLBARS_OUTSIDE_OVERLAY
    horizontalPadding = dimen(R.dimen.recycler_spacing)
    verticalPadding = dip(2)
    addItemDecoration(PaddingItemDecoration(dip(2)))
}
```

I can then apply the style this way:

```
recycler = autoFitRecycler()
        .apply(AutofitRecyclerView::style)
```

And these are basically the rough edges you need to know to understand the code.

There are some more complex views, which uses an `AppBarLayout` with a `CollapsingToolbarLayout`[17], which holds an `ImageView`, a `Toolbar` and a `TabLayout`. Then the main area is using a `ViewPager`. I'm leaving the code here for a reference on how it looks:

```
coordinatorLayout {

    themedAppBarLayout(R.style.ThemeOverlay_AppCompat_Dark_ActionBar) {
        fitsSystemWindows = true

        collapsingToolbarLayout = collapsingToolbarLayout {
            fitsSystemWindows = true
            collapsedTitleGravity = Gravity.TOP
            expandedTitleMarginBottom = dip(60)

            image = squareImageView {
                fitsSystemWindows = true
            }.lparamsC(matchParent) {
                collapseMode = COLLAPSE_MODE_PARALLAX
            }

            toolbar = toolbar {
                popupTheme = R.style.ThemeOverlay_AppCompat_Light
                titleMarginTop = dip(16)
```

[17]https://antonioleiva.com/collapsing-toolbar-layout/

```
            }.lparamsC(width = matchParent, height = dip(88)) {
                gravity = Gravity.TOP
                collapseMode = COLLAPSE_MODE_PIN
            }

            tabLayout = tabLayout {
                setSelectedTabIndicatorColor(Color.WHITE)
            }.lparamsC(width = matchParent) {
                gravity = Gravity.BOTTOM
            }

        }.lparams(width = matchParent) {
            scrollFlags = SCROLL_FLAG_SCROLL or SCROLL_FLAG_EXIT_UNTIL_COLLAPSED
        }

    }.lparams(width = matchParent)

    viewPager = viewPager {
        id = View.generateViewId()
    }.lparams {
        behavior = AppBarLayout.ScrollingViewBehavior()
    }
}
```

Kotlin and Anko Layouts, an exciting combination

It's true that working with this DSL is quite fun, and when you overcome the first part of the learning curve, you'll probably enjoy it.

I encourage you to find a view with some complexity and try to implement it using Anko. You will quickly find out whether this library is for you.

Remember there's a full example at Bandhook-Kotlin repository[18].

[18]https://github.com/antoniolg/Bandhook-Kotlin/

2 Kotlin DSL to write Gradle scripts on Android

There have been quite some months already since Gradle announced that they were working on supporting Kotlin[19] to write Gradle Scripts, by using a version of the language that has been recently revamped to Kotlin DSL.

At the beginning things were quite complicated, but nowadays, with latest versions of Kotlin DSL (at the time of writing this the version is 0.12[20]) the idea is more mature.

So, knowing how, it's not too difficult to start using Kotlin to build your Gradle files in Android.

One of the main issues of this is the lack of documentation, so I decided to write about my experience converting the Gradle files of Bandhook-Kotlin[21] so that you can replicate it in your project.

Using Kotlin DSL on your Gradle files. Is it worth it?

I guess this is the first question to solve. As of today, should I spend my time converting my files to Kotlin DSL?

My answer is probably a bit counterproductive to encourage you to continue reading this guide, but I don't want you to be hyped because of this: **you probably shouldn't**.

It has, of course, some pros:

- You can use a language you're more familiarized with, so **it's easier to start doing more complicated things**. I had never done anything on `buildSrc` folder,

[19]https://blog.gradle.org/kotlin-meets-gradle
[20]https://github.com/gradle/kotlin-dsl/releases/
[21]https://github.com/antoniolg/Bandhook-Kotlin/

and it was quite easy for me to create my own class and use it in the rest of the script files.
- The IDE helps you a lot more: **nice autocomplete, the errors are detected by the compiler, imports added automatically**… All you know and love from your regular Kotlin code is kind of extrapolated here.

But also has some cons:

- There's **not a straightforward way to convert from Groovy to Kotlin files**. I'll explain to you how to make it easier though
- **You need to know how the plugin is implemented to be able to use it**: while in Groovy you just use an equals to assign a value to every configuration, here you need to know whether it's a function[22] or a property[23] to know how to set it. Gladly the IDE can help. Gradle team says that this will only be solved when people write the plugins thinking also on Kotlin DSL. There are some rules to follow. An example (we'll see more later):

```
applicationId = Config.Android.applicationId
minSdkVersion(Config.Android.minSdkVersion)
targetSdkVersion(Config.Android.targetSdkVersion)
versionCode = Config.Android.versionCode
versionName = Config.Android.versionName
```

- **There's not much documentation** or examples: I think this is the main problem. If you get stuck, it's difficult to continue. It took me quite some time to know how to do some things.

That said, things are evolving really fast in this aspect: the final version of Kotlin Gradle Script is close and new Canary versions of Android Studio simplify things a lot. So I'm sure this alternative will be a real option in the close future, and lots of companies will start using it.

[22]https://antonioleiva.com/free-kotlin-android-course/
[23]https://antonioleiva.com/classes-kotlin/

How to convert your files

I want to give you here the fastest route, by skipping all the pain points I had to solve. So if I had to convert another project to use Kotlin on Gradle files, that's what I'd do.

Use the latest version of Gradle

The newer the Gradle version, the better, because it will include the latest Kotlin DSL version. When I converted this project, the latest one was 4.5.1. You can check the latest release here[24]. Modify your `gradle-wrapper.properties` file to use it:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-4.5.1-all.zip
```

Change the name of your files

I must admit that, since first time I tried[25], things have improved a lot. Before, you had to add many configurations that you wouldn't need when using Groovy.

Now, you just need to add an extension to your `build.gradle` files, and Gradle will be able to use Kotlin files. Rename them to `build.gradle.kts`.

Compile using the terminal

The IDE won't help you much here in understanding what's wrong, so I recommend you using *cmd* and the `--info` flag:

```
./gradlew assembleDebug --info
```

With this, you'll get a better idea of the things that are not working. You can do it now if you want, but it will obviously fail.

[24] https://github.com/gradle/gradle/releases
[25] https://github.com/antoniolg/android-kotlin-gradle-script

Configure your buildSrc folder

One of the pain points I found was a way to replicate the `ext` object in Groovy, that allows you to share variables between Groovy files:

```
ext {
    // Android config
    androidBuildToolsVersion = "27.0.3"
    ...
}
```

You can have that in your root Gradle file, and then use it in your modules:

```
buildToolsVersion parent.ext.androidBuildToolsVersion
```

That's pretty easy, and works fine. The alternative in Kotlin DSL is this: for each variable you need to create an extra like this:

```
var androidBuildToolsVersion: String by extra
androidBuildToolsVersion = "27.0.3"
```

And then you use it with this:

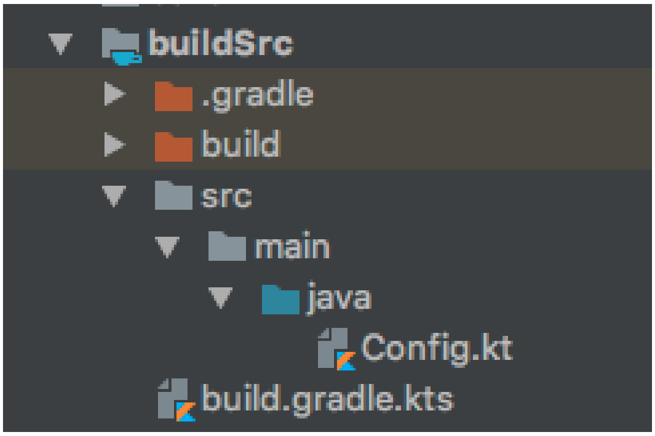
```
val androidBuildToolsVersion: String by extra
buildToolsVersion(androidBuildToolsVersion)
```

As you can imagine, this doesn't scale up very well. Having all this code for each variable is a pain.

So the alternative I found is to create a configuration file in the `buildSrc`, and add all you need in an object that you can then instantiate in any Gradle files. If you don't know about it, the `buildSrc` is basically a place where you put all the code that you want to use when building the project scripts. You can find more info in Gradle Docs[26].

To configure it, just create this folder structure under the `buildSrc` folder:

[26]https://docs.gradle.org/current/userguide/organizing_build_logic.html#sec:build_sources



Forget about `.gradle` and `build` folders and create the rest. Under that folder, also create a new `build.gradle.kts` with this content:

```
plugins {
    `kotlin-dsl`
}
```

The `Config` file will be the one to hold the variables you may use in your project. You can use whatever structure you want. While looking for info on how to do this, I found the repository from Arturo GutiÃ©rrez[27] that uses this structure, and I liked it. I've moved to use objects instead of classes though, which makes more sense here:

```
object Config {
    object BuildPlugins
    object Android
    object Libs
    object TestLibs
}
```

Then, each child object has its own values. For instance, the *Android* one:

[27]https://github.com/arturogutierrez/gradle-script-kotlin-example/blob/master/buildSrc/src/main/kotlin/ProjectConfiguration.kt

```
object Android {
    val buildToolsVersion = "27.0.3"
    val minSdkVersion = 19
    val targetSdkVersion = 27
    val compileSdkVersion = 27
    val applicationId = "com.antonioleiva.bandhookkotlin"
    val versionCode = 1
    val versionName = "0.1"
}
```

You can also use some top variables to make it easier to edit:

```
private const val supportVersion = "27.0.2"
...
object Libs {
    val appcompat = "com.android.support:appcompat-v7:$supportVersion"
    val recyclerview = "com.android.support:recyclerview-v7:$supportVersion"
    val cardview = "com.android.support:cardview-v7:$supportVersion"
    val palette = "com.android.support:palette-v7:$supportVersion"
    val design = "com.android.support:design:$supportVersion"
    ...
}
```

Then, using it in your `build.gradle` files is pretty straightforward:

```
defaultConfig {
    applicationId = Config.Android.applicationId
    minSdkVersion(Config.Android.minSdkVersion)
    targetSdkVersion(Config.Android.targetSdkVersion)
    versionCode = Config.Android.versionCode
    versionName = Config.Android.versionName

    testInstrumentationRunner = "android.support.test.runner.AndroidJUnitRunner"
}
```

As you see, it looks much cleaner.

Keep converting your Gradle files

Until the first time it completely compiles, you'll have to rely on what the terminal builds say, the IDE won't be handy here. So continue changing parts little by little, building the project, and interpreting the output. As a reference, I can leave you some parts of the Gradle files here (and you can, of course, check the complete project on Github[28]). For the root `build.gradle`:

```
buildscript {
    repositories {
        jcenter()
        google()
    }
    dependencies {
        classpath(Config.BuildPlugins.androidGradle)
        classpath(Config.BuildPlugins.kotlinGradlePlugin)
    }
}

allprojects {
    repositories {
        jcenter()
        google()
    }
}
```

This one becomes quite simple, as we've extracted all kinds of configuration to the `Config.kt` file. The module file is a bit more complicated. For the plugins, you do it like this:

```
plugins {
    id("com.android.application")
    kotlin("android")
    kotlin("kapt")
}
```

For regular plugins, you just use the function `id`, and Kotlin plugins use the function `kotlin`. Then the Android section is like you saw above. You need to try to discover whether it's a function or a property. Check the repository the most typical ones. Then, for the build types:

[28]https://github.com/antoniolg/Bandhook-Kotlin/

```
buildTypes {
    getByName("release") {
        isMinifyEnabled = false
        proguardFiles("proguard-rules.pro")
    }
}
```

You can't just create a `release` block, but instead, it's required to find it by name, and then you can configure it. The dependencies are easy, just functions where you set the name of the dependency:

```
dependencies {
    compile(Config.Libs.kotlin_std)
    ...
}
```

Keep building and polishing until the build succeeds.

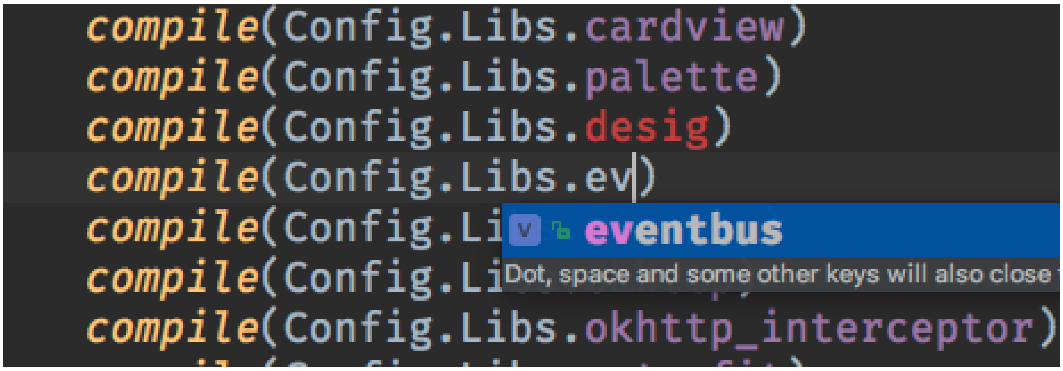
It's not easy, but it's cool!

It's really awesome to see how Kotlin is reaching to all development environments: JVM, JS, Gradle… and potentially everywhere thanks to Kotlin/Native[29].

This is just another example of the versatility of the language and gives an idea of how enthusiastic the different developers' communities are becoming about it. In the case of Gradle, maybe it's not yet production ready (though I know of people that are using it without any issues), but **it's worth giving it a try and check how beautiful it works** once everything is configured.

Having **compile time errors and autocomplete** is of great help when we're building our Gradle files, which otherwise requires just hard memory or searching.

[29]https://blog.jetbrains.com/kotlin/2017/04/kotlinnative-tech-preview-kotlin-without-a-vm/



```
    compile(Config.Libs.cardview)
    compile(Config.Libs.palette)
    compile(Config.Libs.desig)
    compile(Config.Libs.ev)
    compile(Config.Li  eventbus
    compile(Config.Li Dot, space and some other keys will also close
    compile(Config.Libs.okhttp_interceptor)
```

3 A first walk into Kotlin coroutines on Android

Coroutines were the most significant introduction to Kotlin 1.1. They are great because of how powerful they are, and the community is still finding out how to take the most out of them.

To put it simply, coroutines are a way to write asynchronous code sequentially. Instead of messing around with callbacks, you can write your lines of code one after the other. Some of them can suspend the execution and wait until the result is available.

If you were a former C# developer, async/await is the closest concept. However, coroutines in Kotlin are more powerful, because instead of being a specific implementation of the idea, they are a language feature that developers can implement in different ways to solve specific problems.

You can write your solution, or use one of the several options that the Kotlin team and other independent developers have built.

You need to understand that coroutines are an experimental feature in Kotlin 1.1. The Kotlin team uses this category for features that they want to release early but might change in the future. In the case of coroutines, they plan to support the current API, but you might want to migrate to the new definition. As we will see later, you need to opt in for this feature. Otherwise, the IDE shows a warning when you use it.

But this also means that you should take this chapter as an example of what you can do, not a rule of thumb. Things may change in the next few months.

Understanding how coroutines work

My goal here is that you can get some basic concepts and use one of the existing libraries, not to build your implementations. But I think it is important to understand some of the internals so that you do not blindly use what you are given.

Coroutines are based on the idea of *suspending functions*: functions that can stop the execution when they are called and make it continue once it has finished running their background task.

Suspending functions are marked with the reserved word `suspend`, and can only be called inside other suspending functions or a coroutine.

Thus, you cannot call a suspending function everywhere. There needs to be a surrounding function that builds the coroutine and provides the required context. Something like this:

```
fun <T> async(block: suspend () -> T)
```

I am not explaining how to implement the above function. It is a complicated process that is out of the scope of this book, and for most cases, there are solutions already implemented for you. If you are interested in building your own, you can read the specification written in coroutines Github[30]. What you need to know is that the function can have whatever name you want to give it and that it has at least a suspending block as a parameter.

Then you could implement a suspending function and call it inside that block:

```
suspend fun mySuspendingFun(x: Int) : Result {
    ...
}

async {
    val res = mySuspendingFun(20)
    print(res)
}
```

Are coroutines threads then? Not exactly. They work similarly, but are much more lightweight and efficient. You can have millions of coroutines running on a few threads, which opens a world of possibilities.

There are three ways you can make use of the coroutines feature:

- Raw implementation: it means building your way to use coroutines. This is quite complex and usually not required at all.

[30]https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md

- Low-level implementations: Kotlin provides a set of libraries that you can find in kotlinx.coroutines[31] repository, which solves some of the hardest parts and provides a specific implementation for different scenarios. There is one for Android[32], for instance.
- Higher-level implementations: if you want to have a solution that provides everything you need to start using coroutines right away, there are several libraries out there that do all the hard work for you, and the list keeps growing. I am going to stick to Anko, which provides a solution that works well on Android, and you are already familiar with the library.

Using Anko for coroutines

Since 0.10 version, Anko provides a couple of ways to use coroutines in Android.

The first one is very similar to what we saw in the example above, and also similar to what other libraries do.

First, you need to create an async block where suspension functions can run:

```
async(UI) {
    ...
}
```

The UI argument is the execution context for the `async` block.

Then you can create blocks that are executed in a background thread and return the result to the UI thread. Those blocks are defined using the `bg` function:

```
async(UI) {
    val r1: Deferred<Result> = bg { fetchResult1() }
    val r2: Deferred<Result> = bg { fetchResult2() }
    updateUI(r1.await(), r2.await())
}
```

`bg` returns a `Deferred` object, which suspends the coroutine when the function `await()` is called, just until it returns the result. We will see this option in the example below.

[31]https://github.com/Kotlin/kotlinx.coroutines
[32]https://github.com/Kotlin/kotlinx.coroutines/tree/master/ui/kotlinx-coroutines-android

As you know, as Kotlin compiler can infer the type of the variables, this could be simpler:

```
async(UI) {
    val r1 = bg { fetchResult1() }
    val r2 = bg { fetchResult2() }
    updateUI(r1.await(), r2.await())
}
```

The second alternative is to make use of the integration with listeners that is provided on specific sub-libraries, depending on which listener you are going to use. For instance, on `anko-sdk15-coroutines`, there exists an `onClick` listener whose lambda is indeed a coroutine. So you can start using suspending functions right away inside the listener block:

```
textView.onClick {
    val r1 = bg { fetchResult1() }
    val r2 = bg { fetchResult2() }
    updateUI(r1.await(), r2.await())
}
```

As you can see, the result is very similar to the previous one. You are just saving some code.

To use it, you will need to add some of these dependencies, depending on the listeners you want to use:

```
compile "org.jetbrains.anko:anko-sdk15-coroutines:$anko_version"
compile "org.jetbrains.anko:anko-appcompat-v7-coroutines:$anko_version"
compile "org.jetbrains.anko:anko-design-coroutines:$anko_version"
```

Using coroutines in our example

In the example that is explained in the book[33] (which you can find here on Github[34]), weâ€™re creating a simple weather App.

So, to use Anko coroutines, we first need to include the new dependency:

[33] https://antonioleiva.com/book
[34] https://github.com/antoniolg/Kotlin-for-Android-Developers

app/build.gradle

```
compile "org.jetbrains.anko:anko-coroutines:$anko_version"
```

Next, if you remember, I told you that you need to opt in for the feature. Otherwise, it will show a warning. To do that, simply add to the module`build.gradle`:

app/build.gradle

```
kotlin {
    experimental {
        coroutines "enable"
    }
}
```

Now you are ready to start using coroutines. Let's go first to the detail activity.

You can change this code:

```
doAsync {
    val result = RequestDayForecastCommand(intent.getLongExtra(ID, -1)).execute()
    uiThread { bindForecast(result) }
}
```

with this one:

ui/activities/DetailActivity.kt

```
async(UI) {
    val result = bg { RequestDayForecastCommand(intent.getLongExtra(ID, -1))
        .execute() }
    bindForecast(result.await())
}
```

The forecast is requested in a background thread thanks to the `bg` function, which returns a deferred result. That result is awaited in the `bindForecast` call until it is ready to be returned.

Though there is a problem here. Coroutines have a problem: they are keeping a reference to `DetailActivity`, leaking it if the request never finishes for instance.

No worries, because Anko has a solution. You can create a weak reference to your activity, and use that one instead:

```
val ref = asReference()
val id = intent.getLongExtra(ID, -1)

async(UI) {
    val result = bg { RequestDayForecastCommand(id).execute() }
    ref().bindForecast(result.await())
}
```

This reference allows calling the activity when it is available and cancels the coroutine in case the activity has been killed. Be careful to ensure that all calls to activity methods or properties are done via this `ref` object.

But this can get a little complicated if the coroutine interacts several times with the activity. In `MainActivity`, for instance, this solution becomes a little more convoluted:

```
private fun loadForecast() {

    val ref = asReference()
    val localZipCode = zipCode

    async(UI) {
        val result = bg { RequestForecastCommand(localZipCode).execute() }
        val weekForecast = result.await()
        ref().updateUI(weekForecast)
    }
}
```

You cannot use `ref()` inside the `bg` block because the code inside that block is not a suspension context, so you need to save the `zipCode` into another local variable.

I honestly think that leaking the activity for 1-2 seconds is not that bad, and probably will not be worth the boilerplate. In fact, we were already leaking it with the previous solution. So if you can ensure that your background process is not taking forever (for instance, by setting a timeout to your server requests), you are safe by not using `asReference()`.

This way, the changes to `MainActivity` would be simpler:

ui/activities/MainActivity.kt

```kotlin
private fun loadForecast() = async(UI) {
    val result = bg { RequestForecastCommand(zipCode).execute() }
    updateUI(result.await())
}
```

So with all this, you now have your asynchronous code written synchronously very easily. As I said at the beginning, this code is quite simple, but imagine convoluted cases where the result of one background operation is used by the next one, or when you need to iterate over a list and execute a request per item. All this can be written as regular synchronous code, which is much easier to read and maintain.

Bye bye, callback hell. You can check the exact changes in this branch from the book sample App[35].

[35] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-27

4 Kotlin Android Extensions

Kotlin Android Extensions is another plugin that the Kotlin team has developed to make Android development simpler. The plugin automatically creates a set of properties that give direct access to all the views in the XML. This way we do not need to explicitly find all the views in the layout before starting using them.

The names of the properties are taken from the ids of the views, so we must be careful when choosing those names because they now become a relevant part of our base code. The plugin also infers the type of these properties from the XML, so there is no need to do any extra castings.

Kotlin Android Extensions is that it does not require adding libraries to our project. The plugin generates the code it needs to work only when it is required.

How does it work under the hood? These properties delegate to functions that request the view, and a caching function that prevents from doing a `findViewById` every time a property is used. Be aware that this caching mechanism only works if the receiver is an Activity or a Fragment. It skips the cache if it is inside an extension function because the plugin is not able to add the necessary code.

How to use Kotlin Android Extensions

Let's see how easy it is. Though the plugin is part of the regular one (it does not require to install a new one), if you want to use it you have to add an extra *apply* in the Android module:

app/build.gradle

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
```

If you remember, we already did that at the beginning of the book.

Recovering views from the XML

From this moment, recovering a view is as easy as **using the view id you defined in the XML directly into your activity.** Imagine you have an XML like this one:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/welcomeMessage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Hello World!"/>

</FrameLayout>
```

As you can see, the `TextView` has `welcomeMessage` id. In the `MainActivity` you now could write:

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

    welcomeMessage.text = "Hello Kotlin!"
}
```

To use it, you require a special *import* (the one I write below), but the IDE can write the import for you:

```
import kotlinx.android.synthetic.main.activity_main.*
```

The new Android Studio activity templates now include nested layouts, by using the `include` tag. It is important to know that you must add a synthetic import for each XML you use:

```
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.android.synthetic.main.content_main.*
```

As I mentioned above, the generated code includes a view cache. So if you ask for the view again, this does not require another `findViewById`. Let's see what it is doing behind the scenes.

The magic behind Kotlin Android Extensions

When you start working with Kotlin, it is helpful to understand the bytecode generated when you use a new feature. This practice helps you understand the hidden costs of your decisions.

There is an action below *Tools –> Kotlin*, called *Show Kotlin Bytecode* . If you click here, you can see the bytecode generated when the class file you opened is compiled.

The bytecode is not helpful for most humans, but there is another option here: *Decompile*.

This section shows a Java representation of the bytecode generated by Kotlin. That way, you can compare the Java equivalent to the Kotlin code you wrote.

I am going to use this on the previous sample activity, and see the code generated by Kotlin Android Extensions.

The interesting part is this one:

```
...
public View _$_findCachedViewById(int var1) {
   if(this._$_findViewCache == null) {
      this._$_findViewCache = new HashMap();
   }

   View var2 = (View)this._$_findViewCache.get(Integer.valueOf(var1));
   if(var2 == null) {
      var2 = this.findViewById(var1);
      this._$_findViewCache.put(Integer.valueOf(var1), var2);
   }

   return var2;
}

public void _$_clearFindViewByIdCache() {
   if(this._$_findViewCache != null) {
      this._$_findViewCache.clear();
   }

}
```

Here it is the view cache we were talking about. When a view is requested, it tries to find it in the cache. If it is not there, it uses `findViewById` and adds it to the cache. Pretty simple indeed.

Besides, it adds a function to clear the cache: `clearFindViewByIdCache()`. You can use it for instance if you rebuild the view, as the old views are not valid anymore. Then this line:

```
welcomeMessage.text = "Hello Kotlin!"
```

is converted into this:

```
((TextView)this._$_findCachedViewById(id.welcomeMessage))
    .setText((CharSequence)"Hello Kotlin!");
```

So the properties are not real, the plugin is not generating a property per view. It replaces the code during compilation to access the view cache, cast it to the proper type and call the method.

Kotlin Android Extensions on fragments

Fragments can also use this plugin. The problem with fragments is that the view can be recreated while the fragment instance keeps alive. What happens then? This means that the views inside the cache would no longer be valid.

Let’s see the code it generates if we use a fragment. I am creating this simple fragment, that uses the same XML I wrote above:

```
class Fragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
            savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.fragment, container, false)
    }

    override fun onViewCreated(view: View?, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        welcomeMessage.text = "Hello Kotlin!"
    }
}
```

In `onViewCreated`, I change the text of the `TextView`. What about the generated bytecode? Everything is the same as in the activity, with this slight difference:

```
// $FF: synthetic method
public void onDestroyView() {
   super.onDestroyView();
   this._$_clearFindViewByIdCache();
}
```

When the view is destroyed, this method calls `clearFindViewByIdCache`, so we are safe here.

Kotlin Android extensions on a Custom View

It will work very similarly on a custom view. Imagine we have a view like this:

```xml
<merge xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"
              android:layout_width="match_parent"
              android:layout_height="match_parent">

    <ImageView
        android:id="@+id/itemImage"
        android:layout_width="match_parent"
        android:layout_height="200dp"/>

    <TextView
        android:id="@+id/itemTitle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

</merge>
```

I am creating a straightforward custom view and generating the constructors with the new intent that uses `@JvmOverloads` annotation. When a class extends a `View` or a subclass of it, you can press *Alt + Enter*, where you find the option to create a constructor like this:

```kotlin
class CustomView @JvmOverloads constructor(
        context: Context, attrs: AttributeSet? = null, defStyleAttr: Int = 0
) : LinearLayout(context, attrs, defStyleAttr) {

    init {
        LayoutInflater.from(context).inflate(R.layout.view_custom, this, true)
        itemTitle.text = "Hello Kotlin!"
    }
}
```

At the end of the book, I will tell you more about these annotations and how to use them to improve the interoperability with Java code.

In the example above, I am modifying the text of `itemTitle`. The generated code should be trying to find the view from the cache. It does not make sense to copy all the same decompiled code again, but you can see this in the line that modifies the text:

```
((TextView)this._$_findCachedViewById(id.itemTitle))
    .setText((CharSequence)"Hello Kotlin!");
```

We are only calling `findViewById` first time in custom views too.

Recovering views from another view

The last alternative Kotlin Android Extensions provide is to use the properties directly from another view. I am using a layout very similar to the one in the previous section. Imagine that this is inflated in an adapter for instance. You can also access the subviews directly, just by using this plugin:

```
val itemView = ...
itemView.itemImage.setImageResource(R.drawable.image)
itemView.itemTitle.text = "My Text"
```

The plugin also helps you fill the *import*. Check that this one is different:

```
import kotlinx.android.synthetic.main.view_item.view.*
```

There are a couple of things you need to know about this:

- In compilation time, you can reference any views from any other views. This means you could be referencing a view that is not a direct child of that one. Of course, this crashes in execution time when it tries to recover a view that does not exist.
- In this case, views are not cached, as opposed to *Activities* and *Fragments*.

Why is this? Here the plugin does not have a place to generate the required code for the cache. If you again review the code that is generated by the plugin when calling a property from a view, you will see this:

```
((TextView)itemView.findViewById(id.itemTitle)).setText((CharSequence)"My Text");
```

As you can see, there is no call to a cache. Be careful if your view is complex and you are using this in an adapter. It might impact the performance.

Alternatively, if you are using Kotlin 1.1.4 or newer, you have another option.

Kotlin Android Extensions in 1.1.4

From this version of Kotlin, the Android Extensions have incorporated some new exciting features: caches in any classes, and a new annotation called `@Parcelize`. There is also a way to customize the generated cache.

At the moment of writing these lines, these features are still experimental, so you need to enable them by adding this to the `build.gradle`:

```
androidExtensions {
    experimental = true
}
```

This *experimental* flag means that the API is not final, so it can change in the future.

View cache on a ViewHolder (or any custom classes)

You can now build a cache on any classes in a simple way. The only requisite is that your class implements the `LayoutContainer` interface. This interface provides the view that the plugin uses to find the subviews. Imagine we have a `ViewHolder` that is holding a view with the layout described in the previous examples. The required code is:

```
class ViewHolder(override val containerView: View)
        : RecyclerView.ViewHolder(containerView), LayoutContainer {

    fun bind(title: String) {
        itemTitle.text = "Hello Kotlin!"
    }
}
```

The `containerView` is the one that we are overriding from the `LayoutContainer` interface, and that is all you need. From now on, you can access the views directly, no need of prepending `itemView` to get access to the subviews.

Again, if you check the code generation, you will see that it is taking the view from the cache:

```
((TextView)this._$_findCachedViewById(id.itemTitle))
    .setText((CharSequence)"Hello Kotlin!");
```

I have used it here on a `ViewHolder`, but you can see this is generic enough to be used in any classes.

Kotlin Android Extensions to implement Parcelable

With the new `@Parcelize` annotation, you can effortlessly implement `Parcelable`. Simply write the annotation, and the plugin does all the hard work:

```
@Parcelize
class Model(val title: String, val amount: Int) : Parcelable
```

Then, as you may know, you can add the object to an intent:

```
val intent = Intent(this, DetailActivity::class.java)
intent.putExtra(DetailActivity.EXTRA, model)
startActivity(intent)
```

And recover the object from the intent where you need it (in this case, in the target activity):

```
val model: Model = intent.getParcelableExtra(EXTRA)
```

Customize the cache build

The last new feature included in this experimental set is a new annotation called `@ContainerOptions`. This one allows you to customize the way the cache is built, or even prevent a class from creating it.

By default, it uses a `Hashmap`, as we saw before. This behavior can be changed to use a `SparseArray` from the Android framework, which may be more efficient under certain situations. Finally, if for some reason you want to disable the cache for a specific class, you also have that option.

Here it is an example of how to change the cache implementation:

```
@ContainerOptions(CacheImplementation.SPARSE_ARRAY)
class MainActivity : AppCompatActivity() {
    ...
}
```

Currently, the existing options are these:

```
public enum class CacheImplementation {
    SPARSE_ARRAY,
    HASH_MAP,
    NO_CACHE;

    ...
}
```

Conclusion

You've seen how easy is to deal with Android views in Kotlin. With a simple plugin, we can forget about all that awful code related to view recovery after inflation. This plugin will create the required properties for us casted to the right type without any issues.

Besides, Kotlin 1.1.4 has added some interesting features that will be really helpful in some cases that were not previously covered by the plugin.

5 Java interoperability

So far, we have been talking about creating an app from scratch. However, you probably find yourself in a situation where you already have an App written in Java with thousands of lines of code, and you cannot convert all your code. I will cover this topic here.

One of the great wonders of Kotlin is that it is entirely interoperable with Java. Therefore, although all your application code is written Java, you can create a class in Kotlin and use it from Java without any issues. Calling Kotlin from Java code cannot be easier. This potentially gives you two advantages:

- You can use Kotlin in a Java project: In any project, you have already started, you can decide to start writing new code in Kotlin. You can then call it from Java code.
- If you do not know how to do something in Kotlin, you can write that part in Java. You may be wondering if there is a case where Kotlin is not enough to do something on Android. In theory, everything can be done, but the fact is that it does not matter. If you cannot do it in Kotlin, then implement that part in Java.

Let's see how this compatibility works, and how Kotlin code looks when used from Java.

Package-level functions

In Kotlin, functions do not need to be inside a class, but this is not the case in Java. How can we call a function then? Imagine that we have a file called `utils.kt` that looks like this:

```
fun logD(message: String) {
    Log.d("", message)
}

fun logE(message: String) {
    Log.e("", message)
}
```

In Java we can access them through a class that will be called `UtilsKt`, with some static methods:

```
UtilsKt.logD("Debug");
UtilsKt.logE("Error");
```

Extension functions

We have been using extension functions a lot throughout this book. However, how do they look in Java? If you have the following function:

```
fun ViewGroup.inflate(resId: Int, attachToRoot: Boolean = false): View {
    return LayoutInflater.from(context).inflate(resId, this, attachToRoot)
}
```

This is applied to a `ViewGroup`. It receives a layout and inflates it using the parent view. What would we get if we want to use it in Java? This is the result:

```
View v = UtilsKt.inflate(parent, R.layout.view_item, false);
```

As you can see, the object that applies this function (the receiver) is added as an argument to the function. Besides, the optional argument becomes mandatory, because in Java we cannot use default values.

Function overloads

If you want to generate the corresponding overloads in Java, you can use `@JvmOverloads` annotation for that function. In the previous example, you would not need to specify `false` for the second argument in Java:

```
@JvmOverloads
fun ViewGroup.inflate(resId: Int, attachToRoot: Boolean = false): View {
    return LayoutInflater.from(context).inflate(resId, this, attachToRoot)
}

View v = UtilsKt.inflate(parent, R.layout.view_item);
```

If you prefer to specify the name of the class when calling Kotlin from Java, you can use an annotation to modify it. In the `utils.kt` file, add this above the `package` sentence:

```
@file:JvmName("AndroidUtils")
```

And now the class in Java will be named:

```
AndroidUtils.logD("Debug");
AndroidUtils.logE("Error");
View v = AndroidUtils.inflate(parent, R.layout.view_item, false);
```

Instance and static fields

In Java, we use fields to store the state. They can be instance fields, which means that each object can store a different value, or static (all instances of a class share them). If we try to find an equivalent of this in Kotlin, it would be properties and companion objects. If we have a class like this:

```
class App : Application() {

    val appHelper = AppHelper()

    companion object {
        lateinit var instance: App
    }

    override fun onCreate() {
        super.onCreate()
        instance = this
    }

}
```

How does this work in Java? You can simply access the companion object properties as static fields, by using `getters` and `setters`:

```
AppHelper helper = App.instance.getAppHelper();
```

As a `val`, it only generates the `getter` in Java. If it were `var`, we would also have a `setter`. The access to `instance` has worked automatically because it uses the `lateinit` annotation, which also exposes the field that Kotlin uses to store the state. But imagine we create a constant:

```
companion object {
    lateinit var instance: App
    val CONSTANT = 27
}
```

You find that you cannot use it directly. You are forced to access through a `Companion` internal class:

```
KotlinClass.Companion.getCONSTANT()
```

This previous snippet does not look particularly readable. To expose the field in Java the same way a static field would look, you need a new annotation:

```
@JvmField val CONSTANT = 27
```

And now you can use it from Java code:

```
int c = App.CONSTANT;
```

If you have functions in a companion object, they are converted to static methods using the `@JvmStatic` annotation. There are several ways to define constants that, when we use Kotlin from Java, generate different bytecode. If you remember, we used `const val` before, and those properties are accessible without using a `getter`.

Data classes

Some features are clear, but some others are more difficult to know how they may behave in Java. So let's take a look at those features that Kotlin has, but Java does not. One example is *data classes*. Let's say we have a data class like this:

```
data class MediaItem(val id: Int, val title: String, val url: String)
```

We can create instances of this class:

```
MediaItem mediaItem = new MediaItem(1, "Title", "https://antonioleiva.com");
```

But are we missing something? First, let's check if `equals` works as expected:

```
MediaItem mediaItem = new MediaItem(1, "Title", "https://antonioleiva.com");
MediaItem mediaItem2 = new MediaItem(1, "Title", "https://antonioleiva.com");

if (mediaItem.equals(mediaItem2)) {
    Toast.makeText(this, "Items are equals", Toast.LENGTH_SHORT).show();
}
```

Of course, it shows the `Toast`. The bytecode the class generates has everything it needs to compare two items and, if the state is the same, then the items are also the same. However, other things are more difficult to replicate. Remember the `copy` feature data classes have? The method is there, but you can only use it passing all arguments:

```
mediaItem.copy(1, "Title2", "http://google.com");
```

So it is not better than just using the constructor. Also, we lose destructuring, as the Java language does not have a way to express that.

Sealed classes

Another feature that may come to your mind is *sealed classes*. How do they work when used from Java? Let's try it:

```kotlin
sealed class Filter {
    object None : Filter()
    data class ByType(val type: Type) : Filter()
    data class ByFormat(val format: Format) : Filter()
}
```

We have a `Filter` class that represents a filter that can be applied to items. Of course, in Java we cannot do:

```java
public void filter(Filter filter) {
    switch (filter) {
        ...
    }
}
```

`switch` in Java only accepts a small number of types, and for Java, sealed classes are regular classes. So the best you can do is:

```java
if (filter instanceof Filter.None) {
    Log.d(TAG, "Nothing to filter");
} else if (filter instanceof Filter.ByType) {
    Filter.ByType type = (Filter.ByType) filter;
    Log.d(TAG, "Type is: " + type.getType().toString());
} else if (filter instanceof Filter.ByFormat) {
    Filter.ByFormat format = ((Filter.ByFormat) filter);
    Log.d(TAG, "Format is: " + format.getFormat());
}
```

We cannot make use of any of the extra features from Java.

Inline functions and reified types

As you may remember, in Kotlin you can make generic functions use reified types. That way, you can use the generic type inside the function.

When we saw them, I mentioned that they need to use the reserved word `inline`, which is used to substitute the calls to the function by the body of the function when compiling. Can we use that from Java?

Let's start with the `inline` functions, which are easier to test. If we have a `toast` function that receives the message as a lambda:

```
inline fun Context.toast(message: () -> CharSequence) {
    Toast.makeText(this, message(), Toast.LENGTH_SHORT).show()
}
```

We can use it without issues like this from Java:

```
ExtensionsKt.toast(this, () -> "Hello World");
```

So inline functions work, but there is an interesting thing here. When used from Kotlin, the decompiled code looks like this:

```
Toast.makeText(this, "Hello", 0).show();
```

The function is being inlined as expected. What happens when used from Java?

```
ExtensionsKt.toast(this, DetailActivity$$Lambda$0.$instance);
```

So, though you can use `inline` functions from Java, they are actually not inlined. It is calling the function and creating an object for the lambda. That is something to take into account.

Now, what happens to reified types? This is a function that navigates to the activity specified in the generic type:

```
inline fun <reified T : Activity> Context.startActivity() {
    startActivity(Intent(this, T::class.java))
}
```

Then, if you try to call this function from Java, you will see that this method appears to be private, so we cannot use it. Reified functions are not are not available from Java code.

So now you understand better how all new Kotlin features behave when used from Java. You see that using the code we write in Kotlin from Java is effortless. Most of them can still be used, though certainly, we cannot take advantage of some Kotlin features from Java.

Conclusion

I hope you enjoyed these 5 guides that cover some not so known uses of the language.

As you can see, Kotlin is very flexible, which allows solving the same problems in different ways.

From now on, I encourage you to try all these guides with practical examples (most of them include code samples for you to follow) and to keep learning about this fascinating language.

If you want to do it faster and efficiently, I can help you in many different ways.

The simplest one is the Kotlin book[36], where you can learn the language on your own little by little.

But if you're really committed to learning the language, I recommend you to take a look at the online course[37], 1 to 1 mentoring[38], or I even do live training at companies. Ask your company, I'm sure they have some budget saved for training their employees and they can help you get one of these options.

Thanks for reading this guide! If you need something, as usual, you can find me at contact@antonioleiva.com[39].

If you find typos or wrong content in this ebook, please write me and let me know so that I can fix it.

Best,

Antonio

[36]https://antonioleiva.com/kotlin-android-developers-book/
[37]https://antonioleiva.com/online-course/
[38]https://antonioleiva.com/mentoring
[39]mailto:contact@antonioleiva.com