Kotlin
for Android Developers

Learn Kotlin in an easy way while developing an Android App.

Antonio Leiva

Kotlin for Android Developers

Learn Kotlin in an easy way while developing an Android App

Antonio Leiva

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

This version was published on 2015-09-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

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

The suggested hashtag for this book is #kotlinandroiddev.

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

https://twitter.com/search?q=#kotlinandroiddev

This book is dedicated to all the loyal readers of antonioleiva.com, who made me believe that writing about Android development was a powerful tool to help others learn about it. I felt this book as a necessary step forward.

And, of course, this is specially dedicated to you. With your support and your help this book is growing, and I hope it will become a reference. So any claim or suggestion you think it will improve the quality of this book will be welcomed. Feel free to write anytime to contact@antonioleiva.com.

Contents

I. About this book **1**

II. Is this book for you? **2**

III. About the author **3**

1 Introduction **4**
1.1 What is Kotlin? 4
1.2 What do we get with Kotlin? 5

2 Getting ready **9**
2.1 Android Studio 9
2.2 Install Kotlin plugins 10

3 Creating a new project **11**
3.1 Create the project in Android Studio 11
3.2 Configure Gradle 11
3.3 Convert MainActivity to Kotlin code 13
3.4 Test that everything works 13

4 Classes and functions **15**
4.1 How to declare a class 15
4.2 Class inheritance 15
4.3 Functions 16
4.4 Constructor and functions parameters 16

5 Writing your first class **19**
5.1 Creating the layout 19
5.2 The Recycler Adapter 20

6 Variables and properties **22**
6.1 Basic types 22
6.2 Variables 23
6.3 Properties 24

7 Anko and Extension Functions **26**

7.1 What is Anko? . . . 26
7.2 Start using Anko . . . 26
7.3 Extension functions . . . 27

8 Retrieving data from API . . . 28
8.1 Performing the request . . . 28
8.2 Moving request out of the main thread . . . 29

9 Data Classes . . . 30
9.1 Extra functions . . . 30
9.2 Copying a data class . . . 30
9.3 Mapping an object into variables . . . 31

10 Parsing data . . . 32
10.1 Converting json to data classes . . . 32
10.2 Shaping the domain layer . . . 33
10.3 Drawing the data in the UI . . . 35

11 Operator overloading . . . 37
11.1 Operators tables . . . 37
11.2 The example . . . 38
11.3 Operators in extension functions . . . 39

12 Making the forecast list clickable . . . 40

13 Lambdas . . . 45
13.1 Simplifying setOnClickListener() . . . 45
13.2 Click listener for ForecastListAdapter . . . 46
13.3 Extending the language . . . 47

14 Visibility Modifiers . . . 49
14.1 Modifiers . . . 49
14.2 Constructors . . . 50
14.3 Refactoring the code . . . 50

15 Kotlin Android Extensions . . . 52
15.1 How to use Kotlin Android Extensions . . . 52
15.2 Refactoring our code . . . 53

16 Application Singleton and Delegated Properties . . . 56
16.1 Application Singleton . . . 56
16.2 Delegated Properties . . . 57
16.3 Standard Delegates . . . 58
16.4 How to create a custom delegate . . . 61
16.5 Reimplementing App Singleton . . . 62

17 Creating an SQLiteOpenHelper . . . 63
17.1 ManagedSqliteOpenHelper . . . 63
17.2 Tables definition . . . 64
17.3 Implementing SqliteOpenHelper . . . 65
17.4 Dependency injection . . . 67

18 Collections and functional operations . . . 69
18.1 Aggregate operations . . . 70
18.2 Filtering operations . . . 72
18.3 Mapping operations . . . 74
18.4 Elements operations . . . 75
18.5 Generation operations . . . 77
18.6 Ordering operations . . . 78

19 Saving and requesting data from database . . . 80
19.1 Creating database model classes . . . 80
19.2 Writing and requesting data . . . 81

20 Null safety in Kotlin . . . 88
20.1 How Null types work . . . 88
20.2 Nullity and Java libraries . . . 90

21 Creating the business logic to data access . . . 92

22. Control flow and ranges . . . 97
22.1 If Expression . . . 97
22.2 When expression . . . 98
22.3 For loops . . . 99
22.4 While and do/while loops . . . 100
22.5 Ranges . . . 100

23 Creating a Detail Activity . . . 102
23.1 Preparing the request . . . 102
23.2 Providing a new activity . . . 104
23.3 Start an activity: reified functions . . . 108

I. About this book

In this book, I'll be creating an Android app from ground using Kotlin as the main language. The idea is to learn the language by example, instead of following a typical structure. I'll be stopping to explain the most interesting concepts and ideas about Kotlin, comparing it with Java 7. This way, you can see what the differences are and which parts of the language will help you speed up your work.

This book is not meant to be a language reference, but a tool for Android developers to learn Kotlin and be able to continue with their own projects by themselves. I'll be solving many of the typical problems we have to face in our daily lives by making use of the language expressiveness and some other really interesting tools and libraries.

The book is very practical, so it is recommended to follow the examples and the code in front of a computer and try everything it's suggested. You could, however, take a first read to get a broad idea and then dive into practice.

As you could read in previous pages (and probably the site were you downloaded), this is a lean publication. This means that the book is written and progresses with you. I'll continually write new content and review the existing based on your comments and your suggestions. In the end, it will also be your book. I want this book to be the perfect tool for Android developers, and as such, all the help and ideas will be welcomed.

Thanks for becoming part of this exciting project.

II. Is this book for you?

This book is written to be useful for Android developers who are interested in learning Kotlin language.

This book is for you if you are in some of the following situations:

- You have a basic knowledge about Android Development and the Android SDK.
- You want to learn how to develop Android apps using Kotlin by following an example.
- You need a guide on how to solve many of the typical situations an Android developer finds everyday, using a cleaner and more expressive language.

On the other hand, this book may not be for you. This is what you won't find in it:

- This is not a Kotlin Bible. I'll explain all the basics of the language, and even more complex ideas when they come out during the process just when we need them. So you will learn by example and not the other way round.
- I will not explain how to develop an Android app. You won't need a deep knowledge of the platform, but you will need some basics, such as some knowledge of Android Studio, Gradle Java programming and Android SDK. You may even learn some new Android things in the process!
- This is not a guide to learn functional programming. Of course, I'll explain what you need, as Java 7 is not functional at all, but I won't dive deep in functional topics.

III. About the author

Antonio Leiva is an Android Engineer who spends time learning about new ways to explode Android potential and then writes about it. He writes a blog at antonioleiva.com[1] about many different topics related to Android development.

Antonio started as a consultant in CRM technologies, but after some time, looking for his real passion, he discovered the Android world. After getting some experience on such an awesome platform, he started a new adventure at a mobile company, where he led several projects for important Spanish companies.

He now works as an Android Engineer at Plex[2], where he also plays an important role in the design and UX of the Android applications.

You can find Antonio on Twitter as @lime_cl[3]

[1] http://antonioleiva.com
[2] http://plex.tv
[3] https://twitter.com/lime_cl

1 Introduction

You've decided that Java 7 is obsolete and you deserve a more modern language. Nice choice! As you may know, even with Java 8 out there, which includes many of the improvements we would expect from a modern language, we Android developers are still obliged to use Java 7. This is part because of legal issues. But even without this limitation, if new Android devices today started shipping a virtual machine able to understand Java 8, we could't start using it until current Android devices are so obsolete that almost nobody uses them. So I'm afraid we won't see this moment soon.

But not everything is lost. Thanks to the use of the Java Virtual Machine (JVM), we could write Android apps using any language that can be compiled to generate a bytecode the JVM is able to understand.

As you can imagine, there are a lot of options out there, such as Groovy, Scala, Clojure and, of course, Kotlin. In practice, only some of them can be considered real alternatives.

There are pros and cons on any of these languages, and I suggest you to take a look to some of them if you are not really sure which language you should use.

1.1 What is Kotlin?

Kotlin, as described before, is a JVM based language developed by JetBrains[4], known for the creation of IntelliJ IDEA, a powerful IDE for Java development. Android Studio, the official Android IDE, is based on IntelliJ, as a plugin of the platform.

Kotlin was created with Java developers in mind, and with IntelliJ as its main development IDE. And these are two very interesting characteristics for Android developers:

- **Kotlin is very intuitive and easy to learn for Java developers**. Most parts of the language are very similar to what we already know, and the differences in basic concepts can be learnt in no time.
- **We have total integration with our daily IDE for free**. Android Studio is perfectly capable to understand, compile and run Kotlin code. And the support for this language comes from the company who develops the IDE, so we Android developers are first-class citizens.

But this is only related to how the language integrates with our tools. What are the advantages of the language when compared to Java 7?

[4] https://www.jetbrains.com/

- **It's more expressive**: this is one of its most important qualities. You can write more with much less code.
- **It's safer**: Kotlin is null safe, which means that we deal with possible null situations in compile time, to prevent execution time exceptions. We need to explicitly specify if an object can be null, and then check its nullity before using it. You will save a lot of time debugging null pointer exception and fixing nullity bugs.
- **It's functional**: Kotlin is basically an object oriented language. However, as many other modern languages, it uses many concepts from functional programming, such as lambda expressions, to resolve some problems in a much easier way. One of its most awesome features is the way it deals with collections.
- **It makes use of extension functions**: This means we can extend any class with new features even if we don't have access to the code of the class.
- **It's highly interoperable**: You can continue using all the libraries and code written in Java, because the interoperability between both languages is excellent. It's even possible that both languages coexist in the same project.

1.2 What do we get with Kotlin?

Without diving too deep in Kotlin language (we'll learn about it in next chapters), these are some interesting features we miss in Java:

Expresiveness

With Kotlin, it's much easier to avoid boilerplate because most typical situations are covered by default in the language. For instance, in Java, if we want to create a typical data class, we'll need to write (or at least generate) this code:

```
1  public class Artist {
2      private long id;
3      private String name;
4      private String url;
5      private String mbid;
6
7      public long getId() {
8          return id;
9      }
10
11     public void setId(long id) {
12         this.id = id;
13     }
14
```

```
15      public String getName() {
16          return name;
17      }
18
19      public void setName(String name) {
20          this.name = name;
21      }
22
23      public String getUrl() {
24          return url;
25      }
26
27      public void setUrl(String url) {
28          this.url = url;
29      }
30
31      public String getMbid() {
32          return mbid;
33      }
34
35      public void setMbid(String mbid) {
36          this.mbid = mbid;
37      }
38
39      @Override public String toString() {
40          return "Artist{" +
41                  "id=" + id +
42                  ", name='" + name + '\'' +
43                  ", url='" + url + '\'' +
44                  ", mbid='" + mbid + '\'' +
45                  '}';
46      }
47  }
```

With Kotlin, you just need to make use of a data class:

```
1   data class Artist(
2       var id: Long,
3       var name: String,
4       var url: String,
5       var mbid: String)
```

This data class auto-generates all the fields and property accessors, as well as some useful methods such as `toString()`.

Null Safety

When we develop using Java, most of our code is defensive. We need to check continuously if something is null before using it if we don't want to find unexpected *NullPointerException*. Kotlin, as many other modern languages, is null safe because we need to explicitly specify if an object can be null by using the safe call operator (written `?`).

We can do things like this:

```
// This won't compile. Artist can't be null
var notNullArtist: Artist = null

// Artist can be null
var artist: Artist? = null

// Won't compile, artist could be null and we need to deal with that
artist.print()

// Will print only if artist != null
artist?.print()

// Smart cast. We don't need to use safe call operator if we previously
// checked nullity
if (artist != null) {
    artist.print()
}

// Only use it when we are sure it's not null. Will throw an exception otherwise.
artist!!.print()

// Use Elvis operator to give an alternative in case the object is null
val name = artist?.name ?: "empty"
```

Extension functions

We can add new functions to any class. It's a much more readable substitute to the typical utility classes we all have in our projects. We could, for instance, add a new method to fragments to show a toast:

```
fun Fragment.toast(message: CharSequence, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(getActivity(), message, duration).show()
}
```

We can now do:

```
fragment.toast("Hello world!")
```

Functional support (Lambdas)

What if instead of having to write the creation of a new listener every time we need to declare what a click should do, we could just define what we want to do? We can indeed. This (and many more interesting things) is what we get thanks to lambda usage:

```
view.setOnClickListener { toast("Hello world!") }
```

This is only a small selection of what Kotlin can do to simplify your code. Now that you know some of the many interesting features of the language, you may decide this is not for you. If you continue, we'll start with the practice right away in the next chapter.

2 Getting ready

Now that you know some little examples of what you may do with Kotlin, I'm sure you want to start to put it into practice as soon as possible. Don't worry, these first chapters will help you configure your development environment so that you can start writing some code immediately.

2.1 Android Studio

First thing you need is to have Android Studio installed. As you may know, Android Studio is the official Android IDE, which was publicly presented in 2013 as a preview and finally released in 2014.

Android Studio is implemented as a plugin over IntelliJ IDEA[5], a Java IDE created by Jetbrains[6], the company which is also behind Kotlin. So, as you can see, everything is tightly connected.

The adoption of Android Studio was an important change for Android developers. First, because we left behind the buggy Eclipse and moved to a software specifically designed for Java developers, which gives us a perfect interaction with the language. We enjoy awesome features such as a fast and impressively smart code completion, or really powerful analyzing and refactor tools among others.

And second, Gradle[7] became the official build system for Android, which meant a whole bunch of new possibilities related to version building and deploy. Two of the most interesting functions are build systems and flavours, which let you create infinite versions of the app (or even different apps) really easily while using the same code base.

If you are still using Eclipse, I'm afraid you need to switch to Android Studio if you want to practice many of the parts included in this book. The Kotlin team is creating a plugin for Eclipse, but it will be always far behind from the one in Android Studio, and the integration won't be so perfect. You will also discover what you are missing really soon as you start using it.

I'm not covering the use of Android Studio or Gradle because this is not the focus of the book, but if you haven't used these tools before, don't panic. I'm sure you'll be able to follow the book and learn the basics in the meanwhile.

Download Android Studio from the official page[8] if you don't have it already.

[5] https://www.jetbrains.com/idea
[6] https://www.jetbrains.com
[7] https://gradle.org/
[8] https://developer.android.com/sdk/index.html

2.2 Install Kotlin plugins

The IDE by itself is not able to understand Kotlin. As I mentioned in the previous section, it was designed to work with Java. But the Kotlin team has created a powerful set of plugins which will make our lives easier. Go to the plugins section inside Android Studio Preferences, and install these two plugins:

- **Kotlin**: This is the basic plugin. It will let you use Android Studio to write and interact with Kotlin code. It's updated every time a new version of the language is released, so that we can make use of the new features and find alternatives and warnings of deprecations. This is the only plugin you need to write Android apps using Kotlin. But we'll be using another one.
- **Kotlin Android Extensions**: the Kotlin team has also released another interesting plugin for Android developers. These Android Extensions will let you automatically inject all the views in an XML into an activity, for instance, without the need of using `findViewById()`. You will get an attribute casted into the proper view type right away. You will need to install this plugin to use this interesting feature. We'll talk deeper about it in next chapters.

Now our environment is ready to understand the language, compile it and execute it just as seamlessly as if we were using Java.

3 Creating a new project

If you are already used to Android Studio and Gradle, this chapter will be quite easy. I don't want to give many details nor screens, because UI changes from time to time and these lines won't be useful anymore.

Our app is consisting on a simple weather app, such as the one used in Google's Beginners Course in Udacity[9]. We'll be probably paying attention to different things, but the idea of the app will be the same, because it includes many of the things you will find in a typical app. If your Android level is low I recommend this course, it's really easy to follow.

3.1 Create the project in Android Studio

First of all, open Android Studio and choose `Create new Project`. It will ask for a name, you can call it whatever you want: `WeatherApp` for instance. Then you need to set a Company Domain. As you are not releasing the app, this field is not very important either, but if you have your own domain, you can use that one. Also choose a location for the project, wherever you want to save it.

In next step, you'll be asked about the minimum API version. We'll select API 15, because one of the libraries we'll be using needs API 15 as minimum. You'll be targeting most Android users anyway. Don't choose any other platform rather than `Phone and Tablet` for now.

Finally, we are required to choose an activity template to start with. We can choose `Add no Activity` and start from scratch (that would be the best idea when starting a Kotlin project), but we'll rather choose `Blank Activity` because I'll show you later an interesting feature in the Kotlin plugin.

Don't worry much about the name of the activities, layouts, etc. that you will find in next screen. We'll change them later if we need to. Press `Finish` and let Android Studio do its work.

3.2 Configure Gradle

The Kotlin plugin includes a tool which does the Gradle configuration for us. But I prefer to keep control of what I'm writing in my Gradle files, otherwise they can get messy rather easily. Anyway, it's a good idea to know how things work before using the automatic tools, so we'll be doing it manually this time.

First, you need to modify the parent `build.gradle` so that it looks like this:

[9] https://www.udacity.com/course/android-development-for-beginners--ud837

```
buildscript {
    ext.kotlin_version = "0.12.613"
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:1.2.3'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

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

As you can see, we are creating a variable which saves current Kotlin version. Check which version is available when you're reading these lines, because there's probably a new version. We need that version number in several places, for instance in the new dependency you need to add for the Kotlin plugin.

You'll need it again in the module `build.gradle`, where we'll specify that this module uses the **Kotlin plugin**, we'll add the dependences to **Kotlin library**, **Anko** library and the **Kotlin Android Extensions plugin**.

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

android {
    ...
}

dependencies {
    compile 'com.android.support:appcompat-v7:22.2.0'
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    compile "org.jetbrains.anko:anko:0.6.2-15s"
}

buildscript {
    repositories {
        jcenter()
    }
```

```
18      dependencies {
19        classpath "org.jetbrains.kotlin:kotlin-android-extensions:$kotlin_version"
20      }
21  }
```

3.3 Convert MainActivity to Kotlin code

An interesting feature the Kotlin plugin includes is the ability to convert from Java to Kotlin code. As any automated process, it won't be perfectly optimised, but it will help a lot in your first days until you start getting used to Kotlin language.

So we are using it in our `MainActivity.java` class. Open the file and select `Code -> Convert Java File to Kotlin File`.

3.4 Test that everything works

We're going to add some code to test Kotlin Android Extensions are working. I'm not explaining much about it yet, but I want to be sure this is working for you. It's probably the trickiest part in this configuration.

First, go to `activity_main.xml` and set an id for the `TextView`:

```
1  <TextView
2      android:id="@+id/message"
3      android:text="@string/hello_world"
4      android:layout_width="wrap_content"
5      android:layout_height="wrap_content"/>
```

Now, add the synthetic import to the activity (don't worry if you don't understand much about it yet):

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

At `onCreate`, you can now get access to that `TextView` directly. With this line, we're testing that both Kotlin Android Extensions and Anko are working:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    message.text = "Hello Kotlin!"
}
```

Now run the app and see everything it's working fine. Check that the message `TextView` is showing the new content. If you have any doubts or want to review some code, take a look at `chapter-3` branch, at Kotlin for Android Developers repository[10].

Next chapters will cover some of the new things you are seeing in the converted `MainActivity`. Once you understand the slight differences between Java and Kotlin, you'll be able to create new code by yourself much easier.

[10] https://github.com/antoniolg/Kotlin-for-Android-Developers

4 Classes and functions

Classes in Kotlin follow a really simple structure. However, there are some slight differences from Java that you will want to know before we continue. You can use try.kotlinlang.org[11] to test this and some other simple examples without the need of a real project and deploy to a device.

4.1 How to declare a class

If you want to declare a class, you just need to use the keyword `class`:

```
1  class MainActivity {
2
3  }
```

Classes have a unique default constructor. We'll see in future lessons that we can create extra constructors for some edgy situations, but keep in mind that most of the times you'll only need a constructor. You write its parameters just after the name. Brackets are not needed in a class if it doesn't have any content:

```
1  class Person(name: String, surname: String)
```

Where's the body of the constructor then? You can declare an `init` block:

```
1  class Person(name: String, surname: String) {
2          init {
3          ...
4          }
5  }
```

4.2 Class inheritance

By default a class always extends from `Any` (similar to Java `Object`), but we can extend any other class. Classes are closed by default (final), so we can only extend a class if it's explicitly declared as `open` or `abstract`:

[11] http://try.kotlinlang.org/

```
open class Animal(name: String)
class Person(name: String, surname: String) : Animal(name)
```

Note that when using the 1-constructor nomenclature, we need to specify the parameters we're using for the parent constructor. That's the substitution to `super()` call in Java.

4.3 Functions

Functions (our methods in Java) are declared just using the **`fun`** keyword:

```
fun onCreate(savedInstanceState: Bundle?) {
}
```

It you don't specify a return value, it will return `Unit`, similar to `void` in Java, though this is really an object. You can, of course, specify any type for the return value:

```
fun add(x: Int, y: Int) : Int {
        return x + y
}
```

Tip: Semi-colons are not necessary

As you can see in the previous example, I'm not using semi-colons at the end of the sentences. While you can use them, semi-colons are not necessary and it's a good practice not to use them. When you get used, you'll find that it saves you a lot of time.

However, if the result can be calculated using a single expression, you can get rid of brackets and use an equal:

```
fun add(x: Int, y: Int) : Int = x + y
```

4.4 Constructor and functions parameters

Parameters in Kotlin are a bit different from Java. As you can see, we first write the name of the parameter and then its type.

```
fun add(x: Int, y: Int) : Int {
        return x + y
}
```

An extremely useful thing about parameters is that we can make them optional by specifying a **default value**. Here it is an example of a function you could create in an activity which uses a toast to show a message:

```
fun toast(message: String, length: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, message, length).show()
}
```

As you can see, the second parameter (`length`) specifies a default value. This means you can specify the second value or not, which avoids the need of overloading the function:

```
toast("Hello")
toast("Hello", Toast.LENGTH_LONG)
```

This would be equivalent to the next code in Java:

```
void toast(String message){
    toast(message, Toast.LENGTH_SHORT);
}

void toast(String message, int length){
    Toast.makeText(this, message, length).show();
}
```

And this can be as complex as you want. Check this other example:

```
fun niceToast(message: String,
              tag: String = javaClass<MainActivity>().getSimpleName(),
              length: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, "[$className] $message", length).show()
}
```

I've added a third parameter that includes a tag which defaults to the class name. The amount of overloads we'd need in Java grows exponentially. You can now make these calls:

```
toast("Hello")
toast("Hello", "MyTag")
toast("Hello", "MyTag", Toast.LENGTH_SHORT)
```

And there is even another option, because you can use **named arguments**, which means you can write the name of the argument preceding the value to specify which one you want:

```
toast(message = "Hello", length = Toast.LENGTH_SHORT)
```

Tip: String templates

You can use template expressions directly in your strings. This will help you write complex strings based on fixed and variable parts in a really simple way. In the previous example, I used `"[$className] $message"`.

As you can see, anytime you want to add an expression, you need to use the `$` symbol. If the expression is a bit more complex, you'll need to add a couple of brackets: `"Your name is ${user.name}"`.

5 Writing your first class

We already have our `MainActivity.kt` class. What we want is to show a list with the forecast for the next days on it, so you're going to need some changes to the current layout.

5.1 Creating the layout

You'll be using a `RecyclerView`, so you need to add a dependency to the `build.gradle`:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile "com.android.support:appcompat-v7:$support_version"
    compile "com.android.support:recyclerview-v7:$support_version"
    ...
}
```

Now, for the `activity_main.xml` :

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

    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecast_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</FrameLayout>
```

In `MainActivity.kt`, remove the line we added to test everything worked (it will be showing an error now). For now, we are still using the good old `findViewById()`:

```
val forecastList = findViewById(R.id.forecast_list) as RecyclerView
forecastList.setLayoutManager(LinearLayoutManager(this))
```

As you can see, we define the variable and cast it to `RecyclerView`. It's a bit different from Java, and we'll see those differences in the next chapter. A `LayoutManager` is also specified. A list will be enough for this layout.

Object instantiation

Object instantiation is a bit different from Java. As you can see, we omit the "`new`" word. The constructor call is still there, but we save four precious characters. `LinearLayoutManager(this)` creates an instance of the object.

5.2 The Recycler Adapter

We need an adapter for the recycler too. I talked about `RecyclerView` in my blog[12] some time ago, so it may help you if your are not used to it.

For the view, I'll just use a `TextView` for now, and a simple list of texts that we'll create manually for now. Create a new Kotlin file called `ForecastListAdapter.kt`, and add this code:

```
1  public class ForecastListAdapter(val items: List<String>) :
2          RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {
3
4      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
5              ForecastListAdapter.ViewHolder? {
6          return ViewHolder(TextView(parent.getContext()))
7      }
8
9      override fun onBindViewHolder(holder: ForecastListAdapter.ViewHolder,
10                                   position: Int) {
11         holder.textView.setText(items.get(position))
12     }
13
14     override fun getItemCount(): Int = items.size()
15
16     class ViewHolder(val textView: TextView) : RecyclerView.ViewHolder(textView)
17 }
```

Back to the `MainActivity`, now just create the list of strings and then assign the adapter:

[12] http://antonioleiva.com/recyclerview/

```
private val items = listOf(
    "Mon 6/23 - Sunny - 31/17",
    "Tue 6/24 - Foggy - 21/8",
    "Wed 6/25 - Cloudy - 22/17",
    "Thurs 6/26 - Rainy - 18/11",
    "Fri 6/27 - Foggy - 21/10",
    "Sat 6/28 - TRAPPED IN WEATHERSTATION - 23/18",
    "Sun 6/29 - Sunny - 20/7"
)

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    val forecastList = findViewById(R.id.forecast_list) as RecyclerView
    forecastList.setLayoutManager(LinearLayoutManager(this))
    forecastList.setAdapter(ForecastListAdapter(items))
}
```

List creation

Tough I'll come back later to talk about collections, I just want to explain for now that you can create constant lists (what we will see as immutable soon) by using the helper function `listOf`. It receives a `vararg` of items of any type and infers the type of the result.

There are many other alternative functions, such as `setOf`, `arrayListOf` or `hashSetOf` among others.

I also organised packages a little bit. You can take a look at the full code by downloading `chapter-5` branch, at Kotlin for Android Developers repository[13].

There were many new things in such a small amount of code, so I'll be covering it now. We can't continue until we get some basics about basic types, variables and properties.

[13] https://github.com/antoniolg/Kotlin-for-Android-Developers

6 Variables and properties

In Kotlin, **everything is an object**. We don't find primitive types as the ones we can use in Java. That's really helpful, because we have an homogeneous way to deal with all the available types.

6.1 Basic types

Of course, basic types such as integers, floats, characters or booleans still exist, but they all act as an object. Most of the names and the way they work are very similar in Java, but there are some differences you might take into account:

- There are no automatic conversions among numbers. For instance, you cannot assign an `Int` to a `Double` variable. An explicit conversion must be done, using one of the many functions available:

```
1  val i: Int = 7
2  val d: Double = i.toDouble()
```

- Characters (`Char`) cannot directly be treated as numbers. We can, however, convert them to a number when we need it:

```
1  val c: Char = 'c'
2  val i: Int = c.toInt()
```

- Bitwise arithmetical operations are a bit different. In Android, we use bitwise `or` quite often for flags, so I'll stick to "`and`" and "`or` " as an example:

```
1  // Java
2  int bitwiseOr = FLAG1 | FLAG2;
3  int bitwiseAnd = FLAG1 & FLAG2;
```

```
// Kotlin
val bitwiseOr = FLAG1 or FLAG2
val bitwiseAnd = FLAG1 and FLAG2
```

There are many other bitwise operations, such as `shl`, `shs`, `ushr`, `xor` or `inv`. When needed, you can take a look at the official Kotlin reference[14].

- Literals can give information about its type. It's not always necessary, but a common practice in Kotlin is to omit the type of the variables (we'll see it soon), so we can specify it to let the compiler infer the type from the literal:

```
val i = 12 // An Int
val iHex = 0x0f // An Int from hexadecimal literal
val l = 3L // A Long
val d = 3.5 // A Double
val f = 3.5F // A Float
```

- A `String` can be accessed as an array and can be iterated:

```
val s = "Example"
val c = s[2] // This is the Char 'a'
```

```
// Iterate over String
val s = "Example"
for (c in s) {
    print(c)
}
```

6.2 Variables

Variables in Kotlin can be easily defined as mutable (`var`) or immutable (`val`). The idea is very similar to using `final` in Java variables. But **immutability** is a very important concept in Kotlin (and many other modern languages).

[14] http://kotlinlang.org/docs/reference/basic-types.html#operations

An immutable object is an object whose state cannot change after instantiation. If you need a modified version of the object, a new object needs to be created. This makes programming much more robust and predictable. In Java, most objects are mutable, which means that any part of the code which has access to the object can modify it, affecting the rest of the application.

Immutable objects are also thread-safe by definition. As they can't change, no special access control must be defined, because all threads will always get the same object.

So the way we think about coding changes a bit in Kotlin, if we want to make use of immutability. **The key concept: just use `val` as much as possible**. There will be situations (specially in Android, where we don't directly use the constructor of many classes) where it won't be possible, but it will most of the time.

Another thing mentioned before is that we usually don't need to specify the object type, it will be inferred from the assignation, which makes the code cleaner and faster to modify. We already have some examples in the previous section.

```
1  val s = "Example" // A String
2  val i = 23 // An Int
3  val actionBar = getSupportActionBar() // An ActionBar
```

However, the type needs to be specified if a more generic type must be used:

```
1  val a: Any = 23
2  val c: Context = activity
```

6.3 Properties

Properties are the fields in Java, but much more powerful. Properties will do the work of a field plus a getter plus a setter. Let's see an example to compare the difference. This is the necessary code in Java:

```
1  public class Person {
2
3      private String name;
4
5      public String getName() {
6          return name;
7      }
8
9      public void setName(String name) {
10         this.name = name;
```

```
    }
}

...

Person person = new Person();
person.setName("name");
String name = person.getName();
```

In Kotlin, only a property is required:

```
public class Person {

    var name: String = ""

}

...

val person = Person()
person.name = "name"
val name = person.name
```

If nothing is specified, the property uses the default getter and setter. It can, of course, be modified to run whatever custom code you need, without having to modify the existing code:

```
public class Person {

    var name: String = ""
        get() = $name.toUpperCase()
        set(value) {
            $name = "Name: $value"
        }

}
```

If the property needs access to its own value in custom getter and setter (as in this case), it requires the creation of a **backing field**. It can be accessed by using the `$` symbol, and will be generated automatically by the compiler if it finds that it's being used. Take into account that if we used the property directly, we would be using the setter and getter, and not doing a direct assignation. The backing field can only be used inside the class where the property is declared.

7 Anko and Extension Functions

7.1 What is Anko?

Anko[15] is a powerful library developed by JetBrains. Its main purpose is the generation of UI layouts by using code instead of XML. This is an interesting feature I recommend you to take a look, but I won't be using it in this project. For me (probably due to years of experience writing UIs) using XML is much easier, but you could like this approach.

But on the other hand, Anko includes a lot of extremely helpful functions and properties that will avoid lots of boilerplate. I'll tell you when we are using something from Anko, but you'll quickly see which kind of things this library will help with. Though Anko is really helpful, I recommend you to understand what it is doing behind the scenes. You always can navigate to Anko source code using `ctrl + click` or `cmd + click`, and learn a lot of Kotlin from it.

7.2 Start using Anko

Before going any further, let's use Anko to simplify some code. One of the key points where Anko can help us is with getters and setters. Anko has properties generated for most of the fields in a view. For instance, we can do this with a `TextView`:

```
1 textView.text = "Hello World!"
2 val text = textView.text
```

As you will see, anytime you use something from Anko, it will include an import with the name of the property or function to the file. This is because Anko is based on **extension functions** to work. We'll see right after what an extension function is and how to write it.

So now you can go to `ForecastListAdapter` and modify `onBindViewHolder`:

```
1 override fun onBindViewHolder(holder: ForecastListAdapter.ViewHolder,
2                                position: Int) {
3         holder.textView.text = items.get(position)
4     }
```

In `MainActivity:onCreate`, another extension function can be used to simplify how to find the `RecyclerView`:

[15] https://github.com/JetBrains/anko

```
val forecastList: RecyclerView = find(R.id.forecast_list)
```

7.3 Extension functions

An extension function is a function that adds a new behaviour to a class, even if we don't have access to the source code of that class. It's a way to extend classes which lack some useful functions. In Java, this is usually implemented in utility classes which include a set of static methods. The advantage when using extension functions in Kotlin is that we don't need to pass the object as an argument. The extension function acts as if it was part of the class, and we can implement it using `this` and all the public methods.

For instance, we can create a `toast` function which doesn't ask for the context in any `Context` class (and its children):

```
fun Context.toast(message: CharSequence, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, message, duration).show()
}
```

Now you could just use, inside an activity for instance:

```
toast("Hello world!")
toast("Hello world!", Toast.LENGTH_LONG)
```

Of course, Anko already includes its own toast, very similar to this. It has functions for both `CharSequence` and resources, and different functions for short and long toasts:

```
toast("Hello world!")
longToast(R.id.hello_world)
```

The extension functions can also be properties. So you can create extension properties too. It's very similar. Taking previous `text` property as an example:

```
public var TextView.text: CharSequence
    get() = getText()
    set(v) = setText(v)
```

Extension functions don't really modify the original class, but the function is added as a static import where it is used. Extension functions can be declared in any file, so a common practice is to create files which include a set of related functions.

And this is the magic behind Anko. From now own, you can create your own magic too.

This chapter includes very little changes, but you can see the code by checking `chapter-7` branch, at Kotlin for Android Developers repository[16].

[16] https://github.com/antoniolg/Kotlin-for-Android-Developers

8 Retrieving data from API

8.1 Performing the request

Now we want some real data for the forecast, which will be used to populate the `RecyclerView`. We'll be using OpenWeatherMap[17] API to retrieve data, and some regular Java classes for the request. As Kotlin interoperability is extremely powerful, you could use any library you want, such as Retrofit[18], for the server requests. But I don't want to add another third party libraries to the project, so I'll keep it simple.

Besides, as you will see, Kotlin adds some extension functions that will make requests much easier. First, we're going to create a new `Request` class:

```
1  public class Request(val url: String) {
2
3      public fun run() {
4          val forecastJsonStr = URL(url).readText()
5          Log.d(javaClass.getSimpleName(), forecastJsonStr)
6      }
7
8  }
```

It simply receives an url, and saves the result in a String. As you can see, we're using `readText`, which is an extension function from the Kotlin library. This method is not recommended for huge responses, but it will be good enough in our case.

If you compare this code with the one you'd need in Java, you will see we've saved a huge amount of overhead just using the standard library. An `HttpURLConnection`, a `BufferedReader` and an iteration over the result would have been necessary to get the same effect, apart from managing the status of the connection and the reader. Obviously, that's what the function is doing behind the scenes, but we have it for free.

You need to add the Internet permission to the `AndroidManifest.xml`:

```
1  <uses-permission android:name="android.permission.INTERNET" />
```

[17] http://openweathermap.org/

[18] https://github.com/square/retrofit

8.2 Moving request out of the main thread

As you may know, HTTP requests cannot be done in main thread, it will throw an exception because blocking the ui thread is a really bad practice. The common solution in Android is to use an `AsyncTask`. But these classes are ugly and it's difficult to make them work properly in all situations. `AsyncTasks` are dangerous if not used carefully, because by the time it reaches `postExecute` the activity could have been destroyed, and the task will crash.

Anko provides a really easy DSL to deal with asynchrony, which will fit most of your needs. Basically you have an `async` function that will execute its code in another thread, and will give the chance of returning main thread using `uiThread`. So we could do something like this:

```
1 async {
2     Request(url).run()
3     uiThread { longToast("Request performed") }
4 }
```

The good part about `uiThread` is that it's implemented differently depending on the class it uses. If we call it from an `Activity`, the `uiThread` code won't be executed if `activity.isFinishing()` returns `true`, and it won't crash in that situation.

You also can use your own executor:

```
1 val executor = Executors.newScheduledThreadPool(4)
2 async(executor) {
3     // Some task
4 }
```

`async` returns a java `Future`, in case you want to work with futures. And if you need it to return a future with a result, you can use `asyncResult`.

Really simple, right? And much more readable than `AsyncTasks`. For now, I'm just sending a static url to the request, to test that we receive the content properly and that we are able to draw it in the activity. I will cover the json parsing and conversion to app data classes soon, but before we continue, you will learn what a data class is.

Check the code to review the url used for the request and some new package organisation I did. You can run the app and check that you can see the json in the log and the toast when the request finishes.

Remember you can see the code by checking `chapter-8` branch, at Kotlin for Android Developers repository[19].

[19] https://github.com/antoniolg/Kotlin-for-Android-Developers

9 Data Classes

Data classes are a powerful kind of classes which avoid the boilerplate we need in Java to create POJO, or classes which are only based on their state, but are very simple in the operations they do. Defining a new data class is very easy:

```
1  data class Forecast(val date: Date, val temperature: Float, val details: String)
```

9.1 Extra functions

With a data class, we get a handful of interesting functions for free, apart from the properties we already talked about, which prevents us from writing getters and setters:

- `equals()`: it compares the properties of both objects to ensure they are identical.
- `hashCode()`: we get a hash code for free, also calculated from the values of the properties.
- `copy()`: you can copy an object, modifying the properties you need. We'll see an example later.
- A set of numbered functions that are useful to map an object into variables. It will also be explained soon.

9.2 Copying a data class

If we use immutability, as talked some chapters ago, we'll find that if we want to change the state of an object, a new instance of the class is required, with one or more of its properties modified. This task can be rather repetitive and far from clean. However, data classes include the `copy()` method, which will make the process really easy and intuitive.

For instance, if we need to modify the temperature of a `Forecast`, we can just do:

```
1  val f1 = Forecast(Date(), 27.5f, "Shiny day")
2  val f2 = f1.copy(temperature = 30f)
```

This way, we copy the first forecast and modify only the `temperature` property and without changing the state of the original object.

Be careful with immutability when using Java classes

If you decide to work with immutability, be aware that Java classes weren't designed with this in mind, and there are still some situations where you will be able to modify the state. In the previous example, you could still access the `Date` object and change its value. The easy (and unsafe) option is to remember the rules of not modifying the state of any object, but copying it when necessary.

Another option is to wrap these classes. You could create an `ImmutableDate` class which wraps a `Date` and doesn't allow to modify its state. It's up to you to decide which solution you take. In this book, I won't be very strict with immutability (as it's not its main goal), so I won't be creating wrappers for every potentially dangerous classes.

9.3 Mapping an object into variables

This process is known as **multi-declaration** and consists of mapping each property inside an object into a variable. That's the reason why the `componentX` functions are automatically created. An example with the previous `Forecast` class:

```
1  val f1 = Forecast(Date(), 27.5f, "Shiny day")
2  val (date, temperature, details) = f1
```

This multi-declaration is compiled down to the following code:

```
1  val date = f1.component1()
2  val temperature = f1.component2()
3  val details = f1.component3()
```

The logic behind this feature is really powerful, and can help simplify the code in many situations. For instance, `Map` class has some extension functions implemented that allow to recover its keys and values in an iteration:

```
1  for ((key, value) in map) {
2      Log.d("map", "key:$key, value:$value")
3  }
```

10 Parsing data

10.1 Converting json to data classes

Now that we know how to create data classes, we are ready to start parsing data. In the `data` package, create a new file called `ResponseClasses.kt`. If you open the url we used in chapter 8, you can see the structure of the json file. It basically consists of an object which contains a city, and a list of forecast predictions. The city has an id, a name, its coordinates and the country it belongs to. Each forecast comes with a good set of information such as the date, different temperatures, and a weather object with the description and an id for an icon, for instance.

In our current UI we're not going to use many of this data. However, we'll parse everything into classes, in case it's of some use in the future. These are the data classes we need:

```
data class ForecastResult(val city: City, val list: List<Forecast>)

data class City(val id: Long, val name: String, val coord: Coordinates,
                val country: String, val population: Int)

data class Coordinates(val lon: Float, val lat: Float)

data class Forecast(val dt: Long, val temp: Temperature, val pressure: Float,
                    val humidity: Int, val weather: List<Weather>,
                    val speed: Float, val deg: Int, val clouds: Int,
                    val rain: Float)

data class Temperature(val day: Float, val min: Float, val max: Float,
                       val night: Float, val eve: Float, val morn: Float)

data class Weather(val id: Long, val main: String, val description: String,
                   val icon: String)
```

As we are using Gson to parse the json to our classes, the properties must have the same name as the ones in the json, or specify a serialised name. A good practice explained in most software architectures is to use different models for the different layers in our app to decouple them from each other. So I prefer to simplify the declaration of these classes, because I'll convert them before being used in the rest of the app. The names of the properties here are exactly the same as the names in the json response.

Now, the `Request` class needs some modifications in order to return the parsed result. It will also receive only the `zipcode` of the city instead of the full url, so that it becomes more readable. For now, the static url will belong to a **companion object**. Maybe we need to extract it later if we create more requests against another endpoint of the API. I I> ### Companion objects I> I> Kotlin allows us to declare objects to define static behaviours. In Kotlin, we can't create static properties or functions, but we need to rely on objects. However, these objects make some well known patterns such as `Singleton` very easy to implement. We'll talk about it in the future. I> I>If we need some static properties, constants or functions in a class, we can use a **companion object**. This object will be shared between all instances of the class, the same as a static field or method would do.

Check the resulting code:

```
public class ForecastRequest(val zipcode: String) {

    companion object {
        private val URL = "http://api.openweathermap.org/data/2.5/" +
                "forecast/daily?mode=json&units=metric&cnt=7&q="
    }

    public fun execute(): ForecastResult {
        val forecastJsonStr = URL(URL + zipcode).readText()
        return Gson().fromJson(forecastJsonStr, javaClass<ForecastResult>())
    }
}
```

Remember you need to add Gson library to the `build.gradle` dependencies:

```
compile "com.google.code.gson:gson:<version>"
```

10.2 Shaping the domain layer

Now we'll create a new package representing the `domain` layer. This layer will basically implement a set of `Commands` in charge of performing the tasks in the app.

First, a definition of a `Command` is required:

```
public interface Command<T> {
    fun execute(): T
}
```

These commands will execute an operation and return an object of the class specified in its generic type. It's interesting to know that **every function in Kotlin returns a value**. By default, if nothing

is specified, it will return an object of the `Unit` class. So if we want our `Command` to return nothing, we can specify `Unit` as its type.

Interfaces in Kotlin are more powerful than Java (prior to version 8), because they can contain code. But for now, we don't need more than what we could do in a Java interface. Future chapters will talk about the differences deeper.

The first command needs to request the forecast to the API and convert it to domain classes. The definition of the domain classes is this:

```
data class ForecastList(val city: String, val country: String,
                        val dailyForecast:List<Forecast>)

data class Forecast(val date: String, val description: String, val high: Int,
                    val low: Int)
```

These classes will probably need to be reviewed in the future, when more features are added. But the data they keep is enough for now.

Classes need to be mapped from the data to the domain model, so the next task will be to create a `DataMapper`:

```
public class ForecastDataMapper {

    public fun convertFromDataModel(forecast: ForecastResult): ForecastList {
        return ForecastList(forecast.city.name, forecast.city.country,
                convertForecastListToDomain(forecast.list))
    }

    private fun convertForecastListToDomain(list: List<Forecast>):
            List<model.Forecast> {
        return list map { convertForecastItemToDomain(it) }
    }

    private fun convertForecastItemToDomain(forecast: Forecast): model.Forecast {
        return model.Forecast(convertDate(forecast.dt),
                forecast.weather[0].description, forecast.temp.max.toInt(),
                forecast.temp.min.toInt())
    }

    private fun convertDate(date: Long): String {
        val df = DateFormat.getDateInstance(DateFormat.MEDIUM,
                Locale.getDefault())
        return df.format(date * 1000)
```

```
23      }
24  }
```

As you can see, when two classes have the same name, Kotlin is able to make the difference by using a relative name of the package, instead of having to write the complete name. But Kotlin also lets us use a keyword, which will rename the class inside that file. I don't find it very useful in this case, because the contracted package name is expressive enough, but it could be helpful in some situations where our classes collide with the classes in a third-party library:

```
1  import com.example.weatherapp.data.Forecast as DataForecast
```

Another interesting thing about this code is the way to convert the forecast list from the data to the domain model:

```
1  return list map { convertForecastItemToDomain(it) }
```

In a single line, we can loop over the collection and return a new list with the converted items. Kotlin provides a good set of functional operations over lists, which apply an operation for all the items in a list and transform them in any way. This is one of the most powerful features in Kotlin for developers used to Java 7. We'll take a look at all the different transformations very soon. It's important to know they exist, because it will be much easier to find places where these functions can save a lot of time and boilerplate.

And now, everything is prepared to write the command:

```
1  public class RequestForecastCommand(val zipCode: String) :
2          Command<ForecastList> {
3      override fun execute(): ForecastList {
4          val forecastRequest = ForecastRequest(zipCode)
5          return ForecastDataMapper().convertFromDataModel(
6                  forecastRequest.execute())
7      }
8  }
```

10.3 Drawing the data in the UI

The `MainActivity` code changes a little, because now we have real data to fill the adapter. The asynchronous call needs to be rewritten:

```
async {
    val result = RequestForecastCommand("94043").execute()
    uiThread {
        forecastList.setAdapter(ForecastListAdapter(result))
    }
}
```

The adapter needs some modifications too:

```
public class ForecastListAdapter(val weekForecast: ForecastList) :
        RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
            ViewHolder? {
        return ViewHolder(TextView(parent.getContext()))
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        with(weekForecast.dailyForecast[position]) {
            holder.textView.text = "$date - $description - $high/$low"
        }
    }

    override fun getItemCount(): Int = weekForecast.dailyForecast.size()

    class ViewHolder(val textView: TextView) : RecyclerView.ViewHolder(textView)
}
```

`with` function

`with` is a useful function included in the standard Kotlin library. It basically receives an object and an extension function as parameters, and makes the object execute the function. This means that all the code we define inside the brackets acts as an extension function of the object we specify in the first parameter, and we can use all the public functions and properties, as well as `this`. Really helpful to simplify code when we do several operations over the same object.

If you have any doubts or want to review some code, take a look at `chapter-10` branch, at Kotlin for Android Developers repository[20].

[20] https://github.com/antoniolg/Kotlin-for-Android-Developers

11 Operator overloading

Kotlin has a fixed number of symbolic operators we can easily use on any class. The way is to create a function with a reserved name that will be mapped to the symbol. Overloading these operators will increment code readability and simplicity.

11.1 Operators tables

Here you can see a set of tables that include an operator and its corresponding function. A function with that name must be implemented to enable the possibility of using the operator in a specific class.

Unary operations

+a	a.plus()
-a	a.minus()
!a	a.not()
a++	a.inc()
a–	a.dec()

Binary operations

a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.mod(b)
a..b	a.rangeTo(b)
a in b	b.contains(a)
a !in b	!b.contains(a)
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.modAssign(b)

Array-like operations

a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

Equals operation

a == b	a?.equals(b) ?: b.identityEquals(null)
a != b	!(a?.equals(b) ?: b.identityEquals(null))

The `equals` operations are a bit different, because they use a more complex translation in order to make a proper equals checking, and because they expect an exact function specification, and not just a specific name. The function must be implemented exactly like this:

```
1  fun equals(other: Any?): Boolean
```

Function invocation

a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

11.2 The example

As you can imagine, Kotlin lists have the array-like operations implemented, so we can access to list items the same way we'd do if it was an array in Java. But it goes beyond: in mutable lists, the item can also be set directly in a very simple way:

```
1  val x = myList[2]
2  myList[2] = 4
```

If you remember, we have a data class called `ForecastList`, which basically consists of a list with some extra info. It'd be interesting to access its items directly instead of having to request its internal list to get an item. On a totally unrelated note, I'm also going to implement a `size()` function, which will simplify the current adapter a little more:

```
data class ForecastList(val city: String, val country: String,
                        val dailyForecast: List<Forecast>) {
    public fun get(position: Int): Forecast = dailyForecast[position]
    public fun size(): Int = dailyForecast.size()
}
```

It makes our `onBindViewHolder` a bit simpler:

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    with(weekForecast[position]) {
        holder.textView.text = "$date - $description - $high/$low"
    }
}
```

As well as the `getItemCount()` function:

```
override fun getItemCount(): Int = weekForecast.size()
```

11.3 Operators in extension functions

We don't need to stick to our own classes, but we could even extend existing classes using extension functions to provide new operations to third party libraries. For instance, we could access to `ViewGroup` views the same way we do with lists:

```
public fun ViewGroup.get(position: Int): View
        = getChildAt(position)
```

Now it's really simple to get a view from a `ViewGroup` by its position:

```
val container: ViewGroup = find(R.id.container)
val view = container[2]
```

Changes can be reviewed at `chapter-11` branch, at Kotlin for Android Developers repository[21].

[21] https://github.com/antoniolg/Kotlin-for-Android-Developers

12 Making the forecast list clickable

The current layout of the items needs some work to be ready for a real app. The first thing is to create a proper XML that can fit our basic needs. We want to show an icon, date, description and high and low temperatures. So let's create a layout called `item_forecast.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/spacing_xlarge"
    android:background="?attr/selectableItemBackground"
    android:gravity="center_vertical"
    android:orientation="horizontal">

    <ImageView
        android:id="@+id/icon"
        android:layout_width="48dp"
        android:layout_height="48dp"
        tools:src="@mipmap/ic_launcher"/>

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_marginLeft="@dimen/spacing_xlarge"
        android:layout_marginRight="@dimen/spacing_xlarge"
        android:orientation="vertical">

        <TextView
            android:id="@+id/date"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textAppearance="@style/TextAppearance.AppCompat.Medium"
            tools:text="May 14, 2015"/>

        <TextView
```

```
            android:id="@+id/description"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textAppearance="@style/TextAppearance.AppCompat.Caption"
            tools:text="Light Rain"/>

    </LinearLayout>

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:orientation="vertical">

        <TextView
            android:id="@+id/maxTemperature"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="@style/TextAppearance.AppCompat.Medium"
            tools:text="30"/>

        <TextView
            android:id="@+id/minTemperature"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="@style/TextAppearance.AppCompat.Caption"
            tools:text="15"/>

    </LinearLayout>

</LinearLayout>
```

The domain model and data mapper must deal with the complete icon url, so that we are able to load it:

```
data class Forecast(val date: String, val description: String,
                    val high: Int, val low: Int, val iconUrl: String)
```

In `ForecastDataMapper`:

```
private fun convertForecastItemToDomain(forecast: Forecast): model.Forecast {
    return model.Forecast(convertDate(forecast.dt),
            forecast.weather[0].description, forecast.temp.max.toInt(),
            forecast.temp.min.toInt(), generateIconUrl(forecast.weather[0].icon))
}

private fun generateIconUrl(iconCode: String): String
        = "http://openweathermap.org/img/w/$iconCode.png"
```

The `icon` code we got from the first request is used to compose the complete url for the icon image. The simplest way to load an image is to make use of an image loader library. Picasso[22] is a really good option. It must be added to `build.gradle` dependencies:

```
compile "com.squareup.picasso:picasso:<version>"
```

The adapter needs a big rework too. A click listener will be necessary, so let's define it:

```
public interface OnItemClickListener {
     fun invoke(forecast: Forecast)
}
```

If you remember from the last lesson, the `invoke` method can be omitted when called. So let's use it as a way of simplification. The listener can be called in two ways:

```
itemClick.invoke(forecast)
itemClick(forecast)
```

The `ViewHolder` now will be responsible of binding the forecast to the new view:

```
class ViewHolder(view: View, val itemClick: OnItemClickListener) :
        RecyclerView.ViewHolder(view) {

    private val iconView: ImageView
    private val dateView: TextView
    private val descriptionView: TextView
    private val maxTemperatureView: TextView
    private val minTemperatureView: TextView

    init {
```

[22] http://square.github.io/picasso/

```
11          iconView = view.find(R.id.icon)
12          dateView = view.find(R.id.date)
13          descriptionView = view.find(R.id.description)
14          maxTemperatureView = view.find(R.id.maxTemperature)
15          minTemperatureView = view.find(R.id.minTemperature)
16      }
17
18      fun bindForecast(forecast: Forecast) {
19          with(forecast) {
20              Picasso.with(itemView.ctx).load(iconUrl).into(iconView)
21              dateView.text = date
22              descriptionView.text = description
23              maxTemperatureView.text = "${high.toString()}"
24              minTemperatureView.text = "${low.toString()}"
25              itemView.onClick { itemClick(forecast) }
26          }
27      }
28  }
```

The constructor of the adapter now receives the `itemClick`. The methods for creation and binding are simpler:

```
1   public class ForecastListAdapter(val weekForecast: ForecastList,
2           val itemClick: ForecastListAdapter.OnItemClickListener) :
3           RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {
4
5       override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
6               ViewHolder? {
7           val view = parent.ctx.layoutInflater.inflate(R.layout.item_forecast,
8               parent, false)
9           return ViewHolder(view, itemClick)
10      }
11
12      override fun onBindViewHolder(holder: ViewHolder, position: Int) {
13          holder.bindForecast(weekForecast[position])
14      }
15      ...
16  }
```

If you use this code, `parent.ctx` won't compile. Anko provides a lot of extension functions to make Android coding simpler. It, for instance, includes a `ctx` property for activities and fragments, among others, which returns the context, but it lacks of the same property for views. So we are going to create a new file called `ViewExtensions.kt` inside `ui.utils`, and add this extension property:

```
public val View.ctx: Context
    get() = getContext()
```

From now on, any view can make use of it. It is not necessary at all, but I think it gives some consistency if we are planning to use `ctx` in the other classes.

Finally, the `MainActivity` call to `setAdapter` results into this:

```
forecastList.setAdapter(ForecastListAdapter(result,
        object : ForecastListAdapter.OnItemClickListener {
            override fun invoke(forecast: Forecast) {
                toast(forecast.date)
            }
        }))
```

As you can see, to create an anonymous class, we create an `object` that implements the interface we created. Not very nice, right? That's because we are not making use of the powers of functional programming, but you'll learn how to convert this code into something much simpler in the next chapter.

In the meanwhile, you can check all this code (I left some little details out of the explanation) and test it in `chapter-12` branch, at Kotlin for Android Developers repository[23]. The UI starts looking sexy!

[23] https://github.com/antoniolg/Kotlin-for-Android-Developers

13 Lambdas

A lambda expression is a simple way to define an anonymous function. Lambdas are very useful because they prevent us from having to write the specification of the function in an abstract class or interface, and then the implementation of the class. In Kotlin, we can use a function as a parameter in another function.

13.1 Simplifying setOnClickListener()

I will explain how this works using a typical example in Android: the `View.setOnClickListener()` method. If we want to implement a click listener behaviour in Java, we first need to write the `OnClickListener` interface:

```
1  public interface OnClickListener {
2      void onClick(View v);
3  }
```

And then we write an anonymous class that implements this interface:

```
1  view.setOnClickListener(new OnClickListener() {
2      @Override
3      public void onClick(View v) {
4          Toast.makeText(v.getContext(), "Click", Toast.LENGTH_SHORT).show();
5      }
6  });
```

This would be the transformation of the code into Kotlin (using Anko `toast` function):

```
1  view.setOnClickListener(object : OnClickListener {
2      override fun onClick(v: View) {
3          toast("Click")
4      }
5  })
```

Luckily, Kotlin allows some optimisations over Java libraries, and any function that receives an interface with a single function can be substituted by the function. It will work as if we had defined `setOnclickListener()` like this:

```
fun setOnClickListener(listener: (View) -> Unit)
```

A lambda expression is defined by the parameters of the function to the left of the arrow (surrounded by parentheses), and the return value to the right. In this case, we get a `View` and return `Unit` (nothing). So with this in mind, we can simplify the previous code a little:

```
view.setOnClickListener({ view -> toast("Click")})
```

Nice difference! While defining a function, we must use brackets and specify the parameters values to the left of the arrow and the code the function will execute to the right. We can even get rid of the left part if the parameters are not being used:

```
view.setOnClickListener({ toast("Click") })
```

If the function is the last one in the parameters of the function, we can move it out of the parentheses:

```
view.setOnClickListener() { toast("Click") }
```

And, finally, if the function is the only parameter, we can get rid of the parentheses:

```
view.setOnClickListener { toast("Click") }
```

More than five times smaller than the original code in Java, and much easier to understand what is doing. Really impressive. Anko gives us a simplified (basically in name) version, which consists of an extension function implemented the way I showed you before. I'll be using that one throughout the project:

```
view.onClick { toast("Click") }
```

13.2 Click listener for ForecastListAdapter

In the previous chapter, I wrote the click listener in the hard way on purpose to have a good context to develop this one. But now it's time to put what you learnt into practice. We are removing the listener interface from the `ForecastListAdapter` and using a lambda instead:

```
public class ForecastListAdapter(val weekForecast: ForecastList,
                                 val itemClick: (Forecast) -> Unit)
```

The function will receive a forecast and return nothing. The same change can be done to the `ViewHolder`:

```
class ViewHolder(view: View, val itemClick: (Forecast) -> Unit)
```

The rest of the code remains untouched. Just a last modification to `MainActivity`:

```
val adapter = ForecastListAdapter(result, { forecast -> toast(forecast.date) })
forecastList.setAdapter(adapter)
```

I could take the function out of the parentheses, as it is the last one in the list of parameters, but it doesn't feel very natural in a constructor, so I prefer to keep it inside in this case. For the record, the alternative would be:

```
val adapter = ForecastListAdapter(result) { forecast -> toast(forecast.date) }
```

We can simplify the last line even more. In functions that only need one parameter, we can make use of the ***it*** reference, which prevents us from defining the left part of the function specifically. So we can do:

```
val adapter = ForecastListAdapter(result, { toast(it.date) })
```

13.3 Extending the language

Thanks to these transformations, we can create our own builders and code blocks. We've already been using some interesting functions such as `with`. A simpler implementation would be:

```
inline fun <T> with(t: T, body: T.() -> Unit) { t.body() }
```

This function gets an object of type `T` and a function that will be used as an extension function. The implementation just takes the object and lets it execute the function. As the second parameter of the function is another function, it can be brought out of the parentheses, so we can create a block of code where we can use `this` and the public properties and functions of the object directly:

```
with(forecast) {
    Picasso.with(itemView.ctx).load(iconUrl).into(iconView)
    dateView.text = date
    descriptionView.text = description
    maxTemperatureView.text = "${high.toString()}"
    minTemperatureView.text = "${low.toString()}"
    itemView.onClick { itemClick(forecast) }
}
```

Inline functions

Inline functions are a bit different from regular functions. An inline function will be substituted by its code during compilation, instead of really calling to a function. It will reduce memory allocations and runtime overhead in some situations. For instance, if we have a function as an argument, a regular function will internally create an object that contains that function. On the other hand, inline functions will substitute the code of the function in the place where its called, so it won ´t require an internal object for that.

Another example: we could create blocks of code that are only executed if the version is Lollipop or newer:

```
inline fun supportsLollipop(code: () -> Unit) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        code()
    }
}
```

It just checks the version and executes the code if it meets the requirements. Now we could do:

```
supportsLollipop {
    getWindow().setStatusBarColor(Color.BLACK)
}
```

For instance, Anko is also based on this idea to implement the DSL for Android layouts. You can also check an example from Kotlin reference, where a DSL to write HTML[24] from code is created.

As usual, you can check code changes in `chapter-13` branch, at Kotlin for Android Developers repository[25].

[24] http://kotlinlang.org/docs/reference/type-safe-builders.html
[25] https://github.com/antoniolg/Kotlin-for-Android-Developers

14 Visibility Modifiers

Until now, I've been using visibility modifiers the same way I'd do in Java. But these modifiers are a bit different in Kotlin. Inside the same module, we will always probably want to use the default internal modifier, and leave `public` modifier for libraries or classes that need to be visible from other modules.

But here it is the long explanation. How do the different visibility modifiers work in Kotlin?

14.1 Modifiers

private

The `private` modifier is the most restrictive we can use. It implies it will only be visible in its own scope and its subscopes inside the same module. So if we declare a class as `private`, the access is restricted to the package and all its subpackages.

On the other hand, if we use `private` inside a class, the access is restricted to that class. Even the classes that extend it won't be able to use it.

So that is the key difference to remember: first level classes, objects, interfaces... (known as package members) declared as `private` are visible in its package and subpackages, while everything defined inside a class or interface will only be visible by that class or interface.

What is a module?

According to Jetbrains definition, a module is a discrete unit of functionality which you can compile, run, test and debug independently. It basically refers to the Android Studio modules we can create to divide our project into different blocks. In Eclipse, these modules would refer to the projects inside a workspace.

protected

This modifier only applies to members inside a class or an interface. A package member cannot be `protected`. Inside a member, it works the same way as in Java: it can be used by the member itself and the members that extend it (for instance, a class and its subclasses).

internal

This is the default modifier, so it can be omitted. An internal member is visible inside the whole module if it's a package member. If it's a member inside another scope, it depends on the visibility of the scope. For instance, if we write a private class, the access to an `internal` function will be limited to the visibility of the class.

This will probably be the visibility modifier you will use the most. The only reasons for using a less restrictive modifier are if we are writing a library, where we must expose some classes and functions, or if we want to divide our app into modules and we need access from another module.

Modules visibility

At the moment of writing these lines, the Kotlin plugin is not able to restrict internal members to its own module, so you may be able to create an internal class and use it from another module. This is highly discouraged, because this functionality will be enabled in a future release and your code will stop compiling.

public

As you may guess, this is the less restrictive modifier, and the member declared as public is visible anywhere, obviously restricted by its scope. A public member defined in a private class won't be visible outside the scope where the class is visible.

All the functions and properties declared as `public` must specify their return type explicitly, so that it's not possible to accidentally change a type that is part of a public API just by altering its implementation. In the rest of visibility scopes, if there is enough information to guess the type, there is no need to specify it.

14.2 Constructors

By default, all constructors are `public`, which means they can be used from any scope where their class is visible. But we can make a constructor private using this specific syntax:

```
1  class C private constructor(a: Int) { ... }
```

14.3 Refactoring the code

If you take a look at the code, you will see we are using a lot of unnecessary `public` modifiers. We are getting rid of them in this refactor, and let them use the default `internal` modifier.

There are many other details we could change. For instance, in `RequestForecastCommand`, the property we create from the `zipCode` constructor parameter could be private.

```
class RequestForecastCommand(private val zipCode: String)
```

The thing is that as we are making use of immutable properties everywhere, the `zipCode` value can only be requested, but not modified. So it is not a big deal to leave it as `internal`, and the code looks cleaner. If, when writing a class, you feel that something shouldn't be visible by any means, feel free to make it `private`.

An example of how we can get rid of the returning types if the modifier is not public:

```
data class ForecastList(...) {
    fun get(position: Int) = dailyForecast[position]
    fun size() = dailyForecast.size()
}
```

The typical situations where we can get rid of the return type are when we assign the value to a function or a property using `equals` (=) instead of writing a code block.

The rest of the modifications are quite straightforward, so you can check them in `chapter-14` branch, at Kotlin for Android Developers repository[26].

[26] https://github.com/antoniolg/Kotlin-for-Android-Developers

15 Kotlin Android Extensions

Another interesting plugin the Kotlin team has developed to make Android development easier is called Kotlin Android Extensions. Currently it only includes a view binder. The plugin automatically creates a set of properties that give direct access to all the views in the XML. This way we don't need to explicitly find all the views in the layout before starting using them.

The name of the properties are taken from the ids of the views, so we must be careful when choosing those names because they will be an important part of our classes. The type of these properties is also taken from the XML, so there is no need to do any extra castings.

The good part about Kotlin Android Extensions is that it doesn't add any extra libraries to our code. It just consists of a plugin that generates the code it needs to work only when it's required, just by using the standard Kotlin library.

How does it works under the hood? The plugin substitutes any property call into a function that requests the view, and a caching function that prevents from having to find the view every time a property is called. Be aware that this caching mechanism only works if the receiver is an Activity or a Fragment. If it's used in an extension function, the caching will be skipped, because it could be used in an activity the plugin is not able to modify, so it won't be able to add the caching function.

15.1 How to use Kotlin Android Extensions

If you remember, the project is already prepared to use Kotlin Android Extensions. When we were creating the project, we already added the dependency in the `build.gradle`:

```
buildscript {
    repositories {
      jcenter()
    }
    dependencies {
      classpath "org.jetbrains.kotlin:kotlin-android-extensions:$kotlin_version"
    }
}
```

The only thing required by the plugin is the addition of a special "synthetic" `import` in the class which will make use of this feature. We have a couple of ways to use it:

Android Extensions for Activities or Fragments

This is the most typical way to use it. The views can be accessed as if they were properties of the activity or fragment. The name of the properties are the ids of the views in the XML.

The `import` we need to use will start with `kotlin.android.synthetic` plus the name of the XML we want to bind to the activity:

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

From that moment, we can access the views after `setContentView` is called.

Android Extensions for Views

The previous usage is rather restrictive, because we have many other situations where we could need to access the views inside an XML. For example, a custom view or an adapter. For these cases, there is an alternative which will bind the views of the XML to another view. The only difference is the required `import`:

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

If we were in an adapter, for instance, we could now access the properties from the inflated views:

```
1 view.textView.text = "Hello"
```

15.2 Refactoring our code

Now it's time to change our code so that we can start making use of Kotlin Android Extensions. The modifications are fairly simple.

Let's start with `MainActivity`. We are currently only using a `forecastList` view, which is in fact a `RecyclerView`. But we can simplify the code a little bit. First, add the synthetic import for the `activity_main` XML:

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

As said before, we use the id to access the views, so I'm changing the id of the `RecyclerView` so that it doesn't use underscores, but a more appropriate name for a Kotlin variable. The XML results into this:

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

    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecastList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</FrameLayout>
```

And now we can just get rid of the `find` line:

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

    forecastList.setLayoutManager(LinearLayoutManager(this))
    ...
}
```

The simplification was minimal. But the `ForecastListAdapter` can also benefit from the use of this plugin. Here, we can use the mechanism to bind the properties into a view, which will help us remove all the `find` code inside the `ViewHolder`.

First, add the synthetic import for `item_forecast`:

```
import kotlinx.android.synthetic.item_forecast.view.*
```

And now we can use the properties in `itemView` property inside the `ViewHolder`. In fact you can use those properties over any view, but it will obviously crash if the view doesn't contain the requested sub-views.

We don't need the properties for the views anymore, we now have a more direct access:

```
class ViewHolder(view: View, val itemClick: (Forecast) -> Unit) :
        RecyclerView.ViewHolder(view) {

    fun bindForecast(forecast: Forecast) {
        with(forecast) {
            Picasso.with(itemView.ctx).load(iconUrl).into(itemView.icon)
            itemView.date.text = date
            itemView.description.text = description
            itemView.maxTemperature.text = "${high.toString()}□□"
            itemView.minTemperature.text = "${low.toString()}□□"
            itemView.onClick { itemClick(forecast) }
        }
    }
}
```

The Kotlin Android Extensions plugin helps us reduce some more boilerplate and simplify the way we access our views. Check `chapter-15` branch at Kotlin for Android Developers repository[27].

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

16 Application Singleton and Delegated Properties

We are going to implement a database soon and, if we want to keep our code simple and our app in separated modules (instead of everything added to our activity), we'll need to have an easier access to the application context.

16.1 Application Singleton

The simplest way is to just create a singleton the way we'd do in Java:

```
class App : Application() {

    companion object {
        private var instance: Application? = null
        fun instance() = instance!!
    }

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

Remember you need to add this `App` class to the `AndroidManifest.xml` in order to be used as the application instance:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme"
    android:name=".ui.App">
    ...
</application>
```

The problem with Android is that we don't have control over many class constructors. For instance, we cannot initialise a non-nullable property, because its value needs to be defined in the constructor. So we need a nullable variable and then a function that returns a non-nullable value. We know we always have an `App` instance, and that nothing under our control can be done before application `onCreate`, so we are safe by assuming `instance()` function will always be able to return a non-nullable `App` instance.

But this solution seems a bit unnatural. We need to define a property (which already has a getter and a setter) and then a function to return that property. Do we have another way to get a similar result? Yeah, we can delegate the value of a property to another class. This is commonly know as **delegated properties**.

16.2 Delegated Properties

There are some kind of common behaviours we may need in a property that would be interesting to be reused, such as lazy values or observable properties. Instead of having to declare the same code over and over again, Kotlin provides a way to delegate the code a property needs to another class. This is know as a **delegated property**.

When we use `get` or `set` from a property, the `get` and `set` of the delegated property are called.

The structure of a property delegate is:

```
1  class Delegate<T> {
2    fun get(thisRef: Any?, prop: PropertyMetadata): T {
3      return ...
4    }
5
6    fun set(thisRef: Any?, prop: PropertyMetadata, value: T) {
7      ...
8    }
9  }
```

The `T` is the type of the property that is delegating its behaviour. The `get` function receives a reference to the class and the metadata of the property. The `set` function also receives the value that is being assigned. If the property is immutable (`val`), it will only require the `get` function.

This is how the property delegate is assigned:

```
1  class Example {
2    var p: String by Delegate()
3  }
```

It uses **`by`** reserved word to specify the delegation.

16.3 Standard Delegates

There exists a set of standard delegates included in the Kotlin standard library. These are the most common situations where a delegate is really useful, but we could also create our own.

Lazy

It takes a lambda that is executed the first time `get` is called, so the initialisation of the property is delayed up to that moment. Subsequent calls will return the same value. This is very interesting for things that are not always necessary and/or require some other parts to be ready before this one is used. We can save memory and skip the initialisation until the property is required.

```
class App : Application() {
    val database: SQLiteOpenHelper = Delegates.lazy {
            MyDatabaseHelper(getApplicationContext())
    }

    override fun onCreate() {
        super.onCreate()
        val db = database.getWritableDatabase()
    }
}
```

In this example, the database is not really initialised until it's called first time in `onCreate`. At that moment, we are sure the application context exists and is ready to be used.

You can also use `Delegates.blockingLazy` if you need thread safety.

Observable

This delegate will help us detect changes on any property we need to observe. It will execute the lambda expression we specify, every time the `set` function is called. So before the new value is assigned, we receive the delegated property, the old value and the new one.

```
class ViewModel(val db: MyDatabase) {

    var myProperty = Delegates.observable("") {
        d, old, new ->
        db.saveChanges(this, new)
    }

}
```

This example represents some kind of `ViewModel` class which is aware of `myProperty` changes, and saves them to the database every time a new value is assigned

Vetoable

This is a special kind of observable that lets you decide whether the value must be saved or not. It can be used to check some conditions before saving a value.

```
var positiveNumber = Delegates.vetoable(0) {
    d, old, new ->
    new >= 0
}
```

The previous delegate will only allow the new value to be saved if it's a positive number. Inside lambdas, the latest line represents the return value. You don't need to use the `return` word (it won't compile indeed).

Not Null

Sometimes we need something else to initialise a property, but we don't have the necessary state available in the constructor, or we even are not able to do anything in constructors. This second case happens all the time in Android, in activities, fragments, services, broadcast receivers... However, a non abstract property needs a value before the constructor finishes executing. We cannot just wait until we want, in order to assign a value to the property. We have at least a couple of options.

The first one is to use a nullable type and set it to null, until we have the real value. But we then need to check everywhere in the code whether the property is null or not. If we are sure this property is not going to be null at any moment before using it, this may make us write some unnecessary code.

The second option is to use a Not-Null delegate. It will internally save a nullable variable and assign the real value when it is set to the property. If the value is requested before it is assigned, it will throw an exception.

This could be helpful in the App singleton example:

```
class App : Application() {

    companion object {
        var instance: App by Delegates.notNull()
    }

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

Values from a map

Another way to delegate the values of a property is to get them from a map, using the name of the property as the key of the map. This delegate let us do really powerful things, because we can easily create an instance of an object from a dynamic map. If we use `Delegates.mapVal()`, the constructor will get an immutable map and we can assign the values to `val` properties. If we want modifiable map and properties, we can make use of `Delegates.mapVar()`. The class will require a `MutableMap` as constructor parameter instead.

Imagine a configuration class we load from a Json, and assign those key and values to a map. We could just create an instance of a class by passing this map to the constructor:

```
class Configuration(map: Map<String, Any?>) {
    val width: Int by Delegates.mapVal(map)
    val height: Int by Delegates.mapVal(map)
    val dp: Int by Delegates.mapVal(map)
    val deviceName: String by Delegates.mapVal(map)
}
```

As a reference, here it is how we could create the necessary map for this class:

```
val conf = Configuration(mapOf(
        "width" to 1080,
        "height" to 720,
        "dp" to 240,
        "deviceName" to "mydevice"
))
```

16.4 How to create a custom delegate

Let's say we want, for instance, to create a `Not-Null` delegate that can be only assigned once. Second time it's assigned, it will throw an exception.

Kotlin library provides a couple of interfaces our delegates must implement: `ReadOnlyProperty` and `ReadWriteProperty`. The one that should be used depends on whether the delegate property is `val` or `var`.

The first thing we can do is to create a class that extends `ReadWriteProperty`:

```
private class NotNullSingleValueVar<T: Any>() : ReadWriteProperty<Any?, T> {

    override fun get(thisRef: Any?, desc: PropertyMetadata): T {
        throw UnsupportedOperationException()
    }

    public override fun set(thisRef: Any?, desc: PropertyMetadata, value: T) {

    }
}
```

This delegate can work over any non-nullable type (`<T:Any>`). It will receive a reference of an object of any type, and use `T` as the type of the getter and the setter. Now we need to implement the methods.

- The getter will return a value if it's assigned, otherwise it will throw an exception.
- The setter will assign the value if it is still null, otherwise it will throw an exception.

```
private class NotNullSingleValueVar<T : Any>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null

    public override fun get(thisRef: Any?, desc: PropertyMetadata): T {
        return value ?: throw IllegalStateException("${desc.name} " +
                "not initialized")
    }

    public override fun set(thisRef: Any?, desc: PropertyMetadata, value: T) {
        this.value = if (this.value == null) value
        else throw IllegalStateException("${desc.name} already initialized")
    }
}
```

Now you can create an object with a function that provides your new delegates:

```
object DelegatesExt {
    fun notNullSingleValue<T : Any>():
            ReadWriteProperty<Any?, T> = NotNullSingleValueVar()
}
```

16.5 Reimplementing App Singleton

Delegates can help us in this situation. We know that our singleton is not going to be null, but we can't use the constructor to assign the property. So we can make use of a Not-Null delegate:

```
class App : Application() {

    companion object {
        var instance: App by Delegates.notNull()
    }

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

The problem with this solution is that we could change the value of this instance from anywhere in the App, because a `var` property is required if we want to use `Delegates.notNull()`. But we can protect it a little more, by using the delegate we created just before. That way, we can only change its value the first time:

```
companion object {
    var instance: App by DelegatesExt.notNullSingleValue()
}
```

Though, in this case, the initial way of doing a singleton is probably the most simple option, I wanted to show you how to create a custom property delegate and use it in your code. If you want to see the code, check `chapter-16` branch at Kotlin for Android Developers repository[28].

[28] https://github.com/antoniolg/Kotlin-for-Android-Developers

17 Creating an SQLiteOpenHelper

As you may know, Android uses SQLite as database management system. SQLite is a database which goes embedded into the App, and it's really lightweight. That's why it is a good option for mobile Apps.

However, the API to work with databases in Android is quite raw. You'll see you need to write many SQL sentences and parse your objects into `ContentValues` or from `Cursors`. Thankfully, by using a mix of Kotlin and Anko, we are simplifying this task a lot.

Of course, there are many libraries to work with databases in Android, and all of them work with Kotlin thanks to its interoperability. But it's possible you don't want to use them anymore after reading this and next chapters.

17.1 ManagedSqliteOpenHelper

Anko provides a powerful `SqliteOpenHelper` which simplifies things a lot. When we use a regular `SqliteOpenHelper`, we need to call `getReadableDatabase()` or `getWritableDatabase()`, and then we can perform our queries over the object we get. After that, we shouldn't forget calling `close()`. With a `ManagedSqliteOpenHelper` we just do:

```
1  forecastDbHelper.use {
2  ...
3  }
```

Inside the function we can use the `SqliteDatabase`. How it works? It's really interesting reading the implementation of Anko functions, you can learn a lot of Kotlin from it:

```
1  public fun <T> use(f: SQLiteDatabase.() -> T): T {
2      try {
3          return openDatabase().f()
4      } finally {
5          closeDatabase()
6      }
7  }
```

First, `use` receives a function that will be used as an extension function by `SQLiteDatabase`. This means we can use `this` inside the brackets, and we'll be referring to the `SQLiteDatabase` object. This extension function can return a value, so we could do something like this:

```
val result = forecastDbHelper.use {
    val queriedObject = ...
    queriedObject
}
```

Remember that inside a function, the last line represents the returned value. As `T` doesn't have any restrictions, we can return any value. Even `Unit` if we don't want to return anything.

By using a `try-finally`, the `use` function makes sure that the database is closed no matter the extended function succeeds or crashes.

Besides, we have a lot of other really useful extension functions over `SqliteDatabase` that we'll be using later. But for now let's define our tables and implement the `SqliteOpenHelper`.

17.2 Tables definition

The creation of a couple of `objects` that represent our tables will be helpful to avoid misspelling table or column names and repetition. We need two tables: one will save the info of the city and another one the forecast of a day. This second table will have a relationship field to the first one.

`CityForecastTable` first provides the name of the table and then the set of columns it needs: an `id` (which will be the zipCode of the city), the name of the city and the country.

```
private object CityForecastTable {
    val NAME = "CityForecast"
    val ID = "_id"
    val CITY = "city"
    val COUNTRY = "country"
}
```

`DayForecast` has some more info, so it will need the set of columns you can see below. The last column, `cityId`, will keep the id of the `CityForecast` this forecast belongs to.

```
private object DayForecastTable {
    val NAME = "DayForecast"
    val ID = "_id"
    val DATE = "date"
    val DESCRIPTION = "description"
    val HIGH = "high"
    val LOW = "low"
    val ICON_URL = "iconUrl"
    val CITY_ID = "cityId"
}
```

17.3 Implementing SqliteOpenHelper

Our `SqliteOpenHelper` will basically manage the creation and upgrade of our database, and will provide the `SqliteDatabase` so that we can work with it. The queries will be extracted to another class:

```
private class ForecastDbHelper() : ManagedSQLiteOpenHelper(App.instance,
        ForecastDbHelper.DB_NAME, null, ForecastDbHelper.DB_VERSION) {
    ...
}
```

We are using the `App.instance` we created in the previous chapter, as well as a database name and version. These values will be defined in the companion object, together with the helper single instance:

```
companion object {
    val DB_NAME = "forecast.db"
    val DB_VERSION = 1
    val instance by Delegates.blockingLazy() { ForecastDbHelper() }
}
```

The `instance` property uses a `blockingLazy` delegate, which means the object won't be created until it's used. That way, if the database is never used, we don't create unnecessary objects. And it's blocking to prevent the creation of several instances from different threads. This only would happen if two threads try to access the `instance` at the same time, which is difficult but it could happen depending on the type of App you are implementing. But `blockingLazy` is thread safe.

In order to define the creation of the tables, we are required to provide an implementation of `onCreate` function. When no libraries are used, the creation of the tables is done by writing a raw `CREATE TABLE` query where we define all the columns and their types. However, Anko provides a simple extension function which receives the name of the table and a set of `Pair` objects that identify the name and the type of the column:

```
db.createTable(CityForecastTable.NAME, true,
        Pair(CityForecastTable.ID, INTEGER + PRIMARY_KEY),
        Pair(CityForecastTable.CITY, TEXT),
        Pair(CityForecastTable.COUNTRY, TEXT))
```

- The first parameter is the name of the table.
- The second parameter, when set to `true`, will check if the table doesn't exist before trying to create it.

- The third parameter is a `vararg` of `Pairs`. The `varargs` also exist in Java, and it's a way to pass a variable number of arguments of the same type to a function. The function will receive an array with the objects.

The types are from a special Anko class called `SqlType`, which can be mixed with `SqlTypeModifiers`, such as `PRIMARY_KEY`. The `+` operation is overloaded the same way we saw in chapter 11. This `plus` function will concatenate both values in a proper way returning a new special `SqlType`:

```
public fun SqlType.plus(m: SqlTypeModifier) : SqlType {
    return SqlTypeImpl(name, if (modifier == null) m.toString()
            else "$modifier $m")
}
```

As you can see, it can also concatenate several modifiers.

But returning to our code, we can do it better. The Kotlin Standard library includes a function called `to` which, once more, shows the power of Kotlin to let us model our own language. It acts as an extension function for the first object and receives another object as parameter, returning a `Pair` object with them.

```
public fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

As functions with one parameter can be used inline, the result is quite clean:

```
val pair = object1 to object2
```

And this, applied to the creation of our tables:

```
db.createTable(CityForecastTable.NAME, true,
        CityForecastTable.ID to INTEGER + PRIMARY_KEY,
        CityForecastTable.CITY to TEXT,
        CityForecastTable.COUNTRY to TEXT)
```

This is how the whole method looks:

```
override fun onCreate(db: SQLiteDatabase) {
    db.createTable(CityForecastTable.NAME, true,
            CityForecastTable.ID to INTEGER + PRIMARY_KEY,
            CityForecastTable.CITY to TEXT,
            CityForecastTable.COUNTRY to TEXT)

    db.createTable(DayForecastTable.NAME, true,
            DayForecastTable.ID to INTEGER + PRIMARY_KEY + AUTOINCREMENT,
            DayForecastTable.DATE to INTEGER,
            DayForecastTable.DESCRIPTION to TEXT,
            DayForecastTable.HIGH to INTEGER,
            DayForecastTable.LOW to INTEGER,
            DayForecastTable.ICON_URL to TEXT,
            DayForecastTable.CITY_ID to INTEGER)
}
```

We have a similar function to drop a table. `onUpgrade` will just delete the tables so that they are recreated. We are using our database just as a cache, so it's the easiest and safest way to be sure the tables are recreated as expected. If we had important data to be kept, we'd need to improve `onUpgrade` code by doing the corresponding migration depending on the database version.

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
    db.dropTable(CityForecastTable.NAME, true)
    db.dropTable(DayForecastTable.NAME, true)
    onCreate(db)
}
```

17.4 Dependency injection

Although I try not to add much complexity to the code regarding architectures, clean testable code or good practices, I thought it'd be a good idea to show another way to simplify our code using Kotlin. If you want to know a little more about topics like dependency inversion or injection, you can check my set of articles about dependency injection in Android using Dagger[29]. The first article covers a simple explanation about these terms.

In a simple way, if we want to have classes that are independent of other classes, way more testable, and write code easier to extend and maintain, we need to make use of dependency inversion. Instead of instantiating the collaborators inside the class, we provide them (usually via constructor) and instantiate them somewhere else. That way, we can substitute them by other objects that, for instance, implement the same interface, or make use of mocks in tests.

[29] http://antonioleiva.com/dependency-injection-android-dagger-part-1/

But now those dependencies must be provided from somewhere, so the dependency injection consists of providing the collaborators required by the classes. The most common way of doing that is by using a dependency injector. Dagger[30] is probably the most used dependency injector in Android. It's, of course, a very good alternative when we need some complexity to provide those dependencies.

But a simplest alternative is to make use of the default values in a constructor. We can provide the dependency by assigning a default value to the constructor parameters, and then provide a different instance if we need it in other situations. For example, in our `ForecastDbHelper` we can provide the context in a smarter way:

```
1 private class ForecastDbHelper(ctx: Context = App.instance) :
2         ManagedSQLiteOpenHelper(ctx, ForecastDbHelper.DB_NAME, null,
3         ForecastDbHelper.DB_VERSION) {
4         ...
5 }
```

Now we have two ways to create this class:

```
1 val dbHelper1 = ForecastDbHelper()  // It will use App.instance
2 val dbHelper2 = ForecastDbHelper(mockedContext) // For tests, for example
```

I'll be using this mechanism here and there, so I didn't want to continue without explaining the reason. We already have the tables, so it's time to start adding and requesting data from them. But before that, I want to talk about collections and functional operations. Don't forget checking `chapter-17` branch at Kotlin for Android Developers repository[31].

[30] http://square.github.io/dagger/

[31] https://github.com/antoniolg/Kotlin-for-Android-Developers

18 Collections and functional operations

We've been using collections before in this project, but now it's time to show how powerful they are in combination with functional operations. The good part about functional programming is that instead of explaining how we do things, we just say what we want to do. For instance, if we want to filter a list, instead of creating a list, iterating over the original one and add the items to the new if they satisfy a condition, we just use a filter function and specify which filter we want to use. That way, we can say a lot more using less code.

Although we can just use Java collections, Kotlin provides a good set of native interfaces you will want to use:

- **Iterable**: The parent class. Any classes that inherit from this interface represent a sequence of elements we can iterate over.
- **MutableIterable**: Iterables that support removing items during iteration.
- **Collection**: This class represents a generic collection of elements. We get access to functions that return the size of the collection, whether the collection is empty, contains an item or a set of items. All the methods for this kind of collections are only to request data, because collections are immutable.
- **MutableCollection**: a `Collection` that supports adding and removing elements. It provides extra functions such as `add`, `remove` or `clear` among others.
- **List**: Probably the most used collection. It represents a generic ordered collection of elements. As it's ordered, we can request an item by its position, using the `get` function.
- **MutableList**: a `List` that supports adding and removing elements.
- **Set**: an unordered collection of elements that doesn't support duplicate elements.
- **MutableSet**: a `Set` that supports adding and removing elements.
- **Map**: a collection of key-value pairs. The keys in a map are unique, which means we cannot have two pairs with the same key in a map.
- **MutableMap**: a `Map` that supports adding and removing elements.

This is the set of functional operations we have available over the different collections. I want to show you a little definition and example. It is useful to know what the options are, because that way it's easier to identify where these functions can be used.

18.1 Aggregate operations

any

Returns `true` if at least one element matches the given predicate.

```
val list = listOf(1, 2, 3, 4, 5, 6)
assertTrue(list any { it % 2 == 0 })
assertFalse(list any { it > 10 })
```

all

Returns `true` if all the elements match the given predicate.

```
assertTrue(list all { it < 10 })
assertFalse(list all { it % 2 == 0 })
```

count

Returns the number of elements matching the given predicate.

```
assertEquals(3, list count { it % 2 == 0 })
```

fold

Accumulates the value starting with an initial value and applying an operation from the first to the last element in a collection.

```
assertEquals(25, list.fold(4) { total, next -> total + next })
```

foldRight

Same as `fold`, but it goes from the last element to first.

```
assertEquals(25, list.foldRight(4) { total, next -> total + next })
```

forEach

Performs the given operation to each element.

```
list forEach { println(it) }
```

forEachIndexed

Same as `forEach`, though we also get the index of the element.

```
list forEachIndexed { index, value
       -> println("position $index contains a $value") }
```

max

Returns the largest element or `null` if there are no elements.

```
assertEquals(6, list.max())
```

maxBy

Returns the first element yielding the largest value of the given function or `null` if there are no elements.

```
// The element whose negative is greater
assertEquals(1, list.maxBy { -it })
```

min

Returns the smallest element or `null` if there are no elements.

```
assertEquals(1, list.min())
```

minBy

Returns the first element yielding the smallest value of the given function or `null` if there are no elements.

```
// The element whose negative is smaller
assertEquals(6, list.minBy { -it })
```

none

Returns `true` if no elements match the given predicate.

```
// No elements are divisible by 7
assertTrue(list none { it % 7 == 0 })
```

reduce

Same as `fold`, but it doesn't use an initial value. It accumulates the value applying an operation from the first to the last element in a collection.

```
assertEquals(21, list reduce { total, next -> total + next })
```

reduceRight

Same as `reduce`, but it goes from the last element to first.

```
assertEquals(21, list reduceRight { total, next -> total + next })
```

sumBy

Returns the sum of all values produced by the transform function from the elements in the collection.

```
assertEquals(3, list sumBy { it % 2 })
```

18.2 Filtering operations

drop

Returns a list containing all elements except first n elements.

```
assertEquals(listOf(5, 6), list drop(4))
```

dropWhile

Returns a list containing all elements except first elements that satisfy the given predicate.

```
assertEquals(listOf(3, 4, 5, 6), list dropWhile { it < 3 })
```

dropLastWhile

Returns a list containing all elements except last elements that satisfy the given predicate.

```
assertEquals(listOf(1, 2, 3, 4), list dropLastWhile { it > 4 })
```

filter

Returns a list containing all elements matching the given predicate.

```
assertEquals(listOf(2, 4, 6), list filter { it % 2 == 0 })
```

filterNot

Returns a list containing all elements not matching the given predicate.

```
assertEquals(listOf(1, 3, 5), list filterNot { it % 2 == 0 })
```

filterNotNull

Returns a list containing all elements that are not `null`.

```
assertEquals(listOf(1, 2, 3, 4), listWithNull.filterNotNull())
```

slice

Returns a list containing elements at specified indices.

```
assertEquals(listOf(2, 4, 5), list.slice(listOf(1, 3, 4)))
```

take

Returns a list containing first n elements.

```
assertEquals(listOf(1, 2), list.take(2))
```

takeLast

Returns a list containing last n elements.

```
assertEquals(listOf(5, 6), list.takeLast(2))
```

takeWhile

Returns a list containing first elements satisfying the given predicate.

```
assertEquals(listOf(1, 2), list takeWhile { it < 3 })
```

18.3 Mapping operations

flatMap

Iterates over the elements creating a new collection for each one, and finally flattens all the collections into a unique list containing all the elements.

```
assertEquals(listOf(1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7), list flatMap { listOf(i\
t, it + 1) })
```

groupBy

Returns a map of the elements in original collection grouped by the result of given function

```
assertEquals(mapOf("odd" to listOf(1, 3, 5), "even" to listOf(2, 4, 6)),
            list groupBy { if (it % 2 == 0) "even" else "odd" })
```

map

Returns a list containing the results of applying the given transform function to each element of the original collection.

```
assertEquals(listOf(2, 4, 6, 8, 10, 12), list map { it * 2 })
```

mapIndexed

Returns a list containing the results of applying the given transform function to each element and its index of the original collection.

```
assertEquals(listOf (0, 2, 6, 12, 20, 30), list mapIndexed { index, it
        -> index * it })
```

mapNotNull

Returns a list containing the results of applying the given transform function to each non-null element of the original collection.

```
1 assertEquals(listOf(2, 4, 6, 8), listWithNull mapNotNull { it * 2 })
```

18.4 Elements operations

contains

Returns `true` if the element is found in the collection.

```
1 assertTrue(list.contains(2))
```

elementAt

Returns an element at the given index or throws an `IndexOutOfBoundsException` if the index is out of bounds of this collection.

```
1 assertEquals(2, list.elementAt(1))
```

elementAtOrElse

Returns an element at the given index or the result of calling the default function if the index is out of bounds of this collection.

```
1 assertEquals(20, list.elementAtOrElse(10, { 2 * it }))
```

elementAtOrNull

Returns an element at the given index or `null` if the index is out of bounds of this collection.

```
1 assertNull(list.elementAtOrNull(10))
```

first

Returns the first element matching the given predicate

```
1 assertEquals(2, list first { it % 2 == 0 })
```

firstOrNull

Returns the first element matching the given predicate, or `null` if no element was found.

```
assertNull(list firstOrNull { it % 7 == 0 })
```

indexOf

Returns the first index of element, or -1 if the collection does not contain element.

```
assertEquals(3, list.indexOf(4))
```

indexOfFirst

Returns index of the first element matching the given predicate, or -1 if the collection does not contain such element.

```
assertEquals(1, list.indexOfFirst { it % 2 == 0 })
```

indexOfLast

Returns index of the last element matching the given predicate, or -1 if the collection does not contain such element.

```
assertEquals(5, list.indexOfLast { it % 2 == 0 })
```

last

Returns the last element matching the given predicate.

```
assertEquals(6, list last { it % 2 == 0 })
```

lastIndexOf

Returns last index of element, or -1 if the collection does not contain element.

```
val listRepeated = listOf(2, 2, 3, 4, 5, 5, 6)
assertEquals(5, listRepeated.lastIndexOf(5))
```

lastOrNull

Returns the last element matching the given predicate, or `null` if no such element was found.

```
val list = listOf(1, 2, 3, 4, 5, 6)
assertNull(list lastOrNull { it % 7 == 0 })
```

single

Returns the single element matching the given predicate, or throws exception if there is no or more than one matching element.

```
assertEquals(5, list single { it % 5 == 0 })
```

singleOrNull

Returns the single element matching the given predicate, or `null` if element was not found or more than one element was found.

```
assertNull(list singleOrNull { it % 7 == 0 })
```

18.5 Generation operations

merge

Returns a list of values built from elements of both collections with same indexes using the provided transform function. The list has the length of shortest collection.

```
val list = listOf(1, 2, 3, 4, 5, 6)
val listRepeated = listOf(2, 2, 3, 4, 5, 5, 6)
assertEquals(listOf(3, 4, 6, 8, 10, 11), list.merge(listRepeated) { it1, it2 ->
        it1 + it2 })
```

partition

Splits original collection into pair of collections, where the first collection contains elements for which the predicate returned `true`, while the second collection contains elements for which the predicate returned `false`.

```
assertEquals(Pair(listOf(2, 4, 6), listOf(1, 3, 5)),
        list partition { it % 2 == 0 })
```

plus

Returns a list containing all elements of the original collection and then all elements of the given collection. Because of the name of the function, we can use the ‘+’ operator with it.

```
assertEquals(listOf(1, 2, 3, 4, 5, 6, 7, 8), list + listOf(7, 8))
```

zip

Returns a list of pairs built from the elements of both collections with the same indexes. The list has the length of the shortest collection.

```
assertEquals(listOf(Pair(1, 7), Pair(2, 8)), list.zip(listOf(7, 8)))
```

18.6 Ordering operations

reverse

Returns a list with elements in reversed order.

```
val unsortedList = listOf(3, 2, 7, 5)
assertEquals(listOf(5, 7, 2, 3), unsortedList.reverse())
```

sort

Returns a sorted list of all elements.

```
assertEquals(listOf(2, 3, 5, 7), unsortedList.sort())
```

sortBy

Returns a list of all elements, sorted by the specified comparator.

```
assertEquals(listOf(3, 7, 2, 5), unsortedList sortBy { it % 3 })
```

sortDescending

Returns a sorted list of all elements, in descending order.

```
assertEquals(listOf(7, 5, 3, 2), unsortedList.sortDescending())
```

sortDescendingBy

Returns a sorted list of all elements, in descending order by the results of the specified order function.

```
assertEquals(listOf(2, 5, 7, 3), unsortedList sortDescendingBy { it % 3 })
```

19 Saving and requesting data from database

A previous chapter covered the creation of a `SQLiteOpenHelper`, but now we need a way to use it to persist our data into the database and recover it when necessary. Another class, called `ForecastDb`, will make use of it.

19.1 Creating database model classes

But first, we are going to create the model classes for the database. Remember the `mapVar` delegates we saw? We are using them to map those fields directly to the database and viceversa.

Let's take a look at the `CityForecast` class first:

```
1  private class CityForecast(val map: MutableMap<String, Any?>,
2                             val dailyForecast: List<DayForecast>) {
3
4      var _id: Long by Delegates.mapVar(map)
5      var city: String by Delegates.mapVar(map)
6      var country: String by Delegates.mapVar(map)
7
8      constructor(id: Long, city: String, country: String,
9          dailyForecast: List<DayForecast>) : this(HashMap(), dailyForecast) {
10         this._id = id
11         this.city = city
12         this.country = country
13     }
14 }
```

The default constructor is getting a map, presumably filled with the values of the properties, and a `dailyForecast`. Thanks to the delegates, the values will be mapped to the corresponding properties based on the name of the key. If we want to make the mapping work perfectly, the names of the properties must be the same as the names of the columns in the database. We'll see why later.

But then, a second constructor is necessary. This is because we'll be mapping classes from the domain to classes for the database, so instead of using a map, extracting the values from the properties will be more convenient. We pass an empty map, but again, thanks to the delegate, when we set a value

to a property, it automatically adds a new value to the map. That way, we'll have our map ready to be added to the database. After some useful code, you will see it works like magic.

Now we need a second class, the `DayForecast`, which corresponds to the second table. This one will basically have one property per column, and will also use a secondary constructor. The only difference is that we are not assigning an id, because it will be auto-generated by SQLite.

```
private class DayForecast(var map: MutableMap<String, Any?>) {
    var _id: Long by Delegates.mapVar(map)
    var date: Long by Delegates.mapVar(map)
    var description: String by Delegates.mapVar(map)
    var high: Int by Delegates.mapVar(map)
    var low: Int by Delegates.mapVar(map)
    var iconUrl: String by Delegates.mapVar(map)
    var cityId: Long by Delegates.mapVar(map)

    constructor(date: Long, description: String, high: Int, low: Int,
            iconUrl: String, cityId: Long)
    : this(HashMap()) {
        this.date = date
        this.description = description
        this.high = high
        this.low = low
        this.iconUrl = iconUrl
        this.cityId = cityId
    }
}
```

These classes will help us map the data between objects and SQLite tables, in both directions.

19.2 Writing and requesting data

The `SqliteOpenHelper` is just the tool, the channel between object oriented and SQL worlds. We'll use it in a new class, to request data already saved in the database, and to save fresh data. The definition of the class will be using a `ForecastDbHelper` and a `DataMapper` that will convert classes from database to domain models. I'm still using default values as an easy way of dependency injection:

```
class ForecastDb(
    val forecastDbHelper: ForecastDbHelper = ForecastDbHelper.instance,
    val dataMapper: DbDataMapper = DbDataMapper()) {
         ...
}
```

Both functions will make use of the `use()` function we saw in the previous chapter. The value that the lambda returns will be used as the result of our function. So let's define a function that requests a forecast based on a zip code and a date:

```
fun requestForecastByZipCode(zipCode: Long, date: Long) = forecastDbHelper.use {
    ....
}
```

Not much to explain here: we return the result of the `use` function as the result of our function.

Requesting a forecast

The first request that needs to be done is the daily forecast, because we need the list to create the city object. Anko provides an simple request builder, so let's take advantage of it:

```
val dailyRequest = "${DayForecastTable.CITY_ID} = ? " +
    "AND ${DayForecastTable.DATE} >= ?"

val dailyForecast = select(DayForecastTable.NAME)
        .whereSupport(dailyRequest, zipCode.toString(), date.toString())
        .parseList { DayForecast(HashMap(it)) }
```

The first line, `dailyRequest`, is the `where` part of the query. This is the first parameter the `whereSupport` function needs, and it's very similar to what we'd do in a regular use of the helper. There is another function called simply `where`, which takes some tags and values and match them. I don't like it very much because I think it adds more boilerplate, though it has the advantage of parsing the values to the `String`s we need. This is how it would look with it:

```
val dailyRequest = "${DayForecastTable.CITY_ID} = {id}" +
"AND ${DayForecastTable.DATE} >= {date}"

val dailyForecast = select(DayForecastTable.NAME)
        .where(dailyRequest, "id" to zipCode, "date" to date)
        .parseList { DayForecast(HashMap(it)) }
```

You can choose your preferred one. The `select` function is simple, it just asks for the name of the table. The `parse` methods are where the magic happens. In this case we are using the function `parseList`, which assumes we are requesting a list of items. It uses a `RowParser` or `MapRowParser` to convert the cursor into a list of object. The difference between both is that the `RowParser` relies on the order of the columns, while the `MapRowParser` uses the name of the column as the key of the map.

These two overloads conflict between them, so we can't directly use the simplification that prevents from the need of creating an object explicitly. But nothing that can't be solved with an extension function. I'm creating a function that receives a lambda and returns a `MapRowParser`. The parser will use that lambda to create the object:

```
public fun <T : Any> SelectQueryBuilder.parseList(
    parser: (Map<String, Any>) -> T): List<T> =
        parseList(object : MapRowParser<T> {
            override fun parseRow(columns: Map<String, Any>): T = parser(columns)
        })
```

This function helps simplify the `parseList` request to:

```
parseList { DayForecast(HashMap(it)) }
```

The immutable map that the parser receives is converted into a mutable map (we need it to be mutable in our database model) by using the corresponding constructor from the `HashMap`. That `HashMap` is used by the constructor of `DayForecast`.

So, to understand what is happening behind the scenes, the request returns a `Cursor`. `parseList` iterates over it and gets the rows from the `Cursor` until it reaches the last one. For each row, it creates a map with the columns as keys and assigns the value to the corresponding key. The map is then returned to the parser.

If there were no results for the request, `parseList` returns an empty list.

The next step is to request the city, in a similar way:

```
val city = select(CityForecastTable.NAME)
        .whereSupport("${CityForecastTable.ID} = ?", zipCode.toString())
        .parseOpt { CityForecast(HashMap(it), dailyForecast) }
```

The difference here: we are using `parseOpt` instead. This function returns a nullable object. The result can be null or a single object, depending on whether the request finds something in the database or not. There is another function called `parseSingle`, which does essentially the same, but returns a non-nullable object. So if it doesn't find a register in the database, it throws an exception. In our case, the first time we query a city it won't be there, so using `parseOpt` is safer. I also created a handy function to prevent the need of an object creation:

```
public fun <T : Any> SelectQueryBuilder.parseOpt(
    parser: (Map<String, Any>) -> T): T? =
        parseOpt(object : MapRowParser<T> {
            override fun parseRow(columns: Map<String, Any>): T = parser(columns)
        })
```

Finally, if the returned city is not null, we convert it to a domain object and return it, using the `dataMapper`. Otherwise, we just return a null. As you may remember, the last line inside a lambda represents what the lambda returns. So it will return an object of the type `CityForecast?`:

```
if (city != null) dataMapper.convertToDomain(city) else null
```

The `DataMapper` function is easy:

```
fun convertToDomain(forecast: CityForecast) = with(forecast) {
    val daily = dailyForecast map { convertDayToDomain(it) }
    ForecastList(_id, city, country, daily)
}

private fun convertDayToDomain(dayForecast: DayForecast) = with(dayForecast) {
    Forecast(date, description, high, low, iconUrl)
}
```

So this is how the complete function looks like:

```
fun requestForecastByZipCode(zipCode: Long, date: Long) = forecastDbHelper.use {

    val dailyRequest = "${DayForecastTable.CITY_ID} = ? AND " +
        "${DayForecastTable.DATE} >= ?"
    val dailyForecast = select(DayForecastTable.NAME)
            .whereSupport(dailyRequest, zipCode.toString(), date.toString())
            .parseList { DayForecast(HashMap(it)) }

    val city = select(CityForecastTable.NAME)
            .whereSupport("${CityForecastTable.ID} = ?", zipCode.toString())
            .parseOpt { CityForecast(HashMap(it), dailyForecast) }

    if (city != null) dataMapper.convertToDomain(city) else null
}
```

Another interesting functionality from Anko I'm not showing here is that you can make use of a `classParser()` instead of the `MapRowParser` we are using, which uses reflection to fill a class based on the names of the columns. I prefer the other way because we don't need reflection and have more control over the transformations, but it can be of use for you at some time.

Saving a forecast

The `saveForecast` function just clears the data from the database so that we save fresh data, converts the domain forecast model to database model, and inserts each daily forecast and the city forecast. The structure is similar to the previous one: it returns the value from the `use` function from the database helper. In this case we don't need a result, so it'll return `Unit`.

```
fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {
    ...
}
```

First, we clear both tables. Anko doesn't provide any beautiful way to do it, but it doesn't mean we can't. So we are creating an extension function for `SQLiteDatabase` that will execute the proper SQL query for us:

```
public fun SQLiteDatabase.clear(tableName: String) {
    execSQL("delete from $tableName")
}
```

The function is applied to both tables:

```
clear(CityForecastTable.NAME)
clear(DayForecastTable.NAME)
```

The next step is to convert the data, and use the result to execute the `insert` queries. At this point you probably know I'm a fan of the `with` function:

```
with(dataMapper.convertFromDomain(forecast)) {
     ...
}
```

The conversion from the domain model is straightforward too:

```
fun convertFromDomain(forecast: ForecastList) = with(forecast) {
    val daily = dailyForecast map { convertDayFromDomain(id, it) }
    CityForecast(id, city, country, daily)
}

private fun convertDayFromDomain(cityId: Long, forecast: Forecast) =
    with(forecast) {
        DayForecast(date, description, high, low, iconUrl, cityId)
}
```

Inside the block, we can use `dailyForecast` and `map` without the need of referring to a variable, just like if we were inside the class. We are using another Anko function for the insertion, which asks for a table name and a `vararg` of `Pair<String, Any>`. The function will convert the `vararg` to the `ContentValues` object the Android SDK needs. So our task consists of transforming the `map` into a `vararg` array. We are creating another extension function for `MutableMap` to do that:

```
public fun MutableMap<K, V?>.toVarargArray<K : Any?, V : Any>():
    Array<out Pair<K, V>> = map({ Pair(it.key, it.value!!) }).toTypedArray()
```

It works over a `MutableMap` with nullable values (this was a condition from the `mapVar` delegate), and converts it to an `Array` with non-nullable values (`select` function requisite) of pairs. Don't worry if you don't understand this function completely, I will be covering nullity really soon.

So, with this new function we can do:

```
insert(CityForecastTable.NAME, *map.toVarargArray())
```

It inserts a new row in the `CityForecast` table. the '`*`' used before the result of `toVarargArray` indicates that the array will be decomposed in a `vararg` parameter. This is done automatically in Java, but we need to make it explicit in Kotlin.

And the same for each daily forecast:

```
dailyForecast forEach { insert(DayForecastTable.NAME, *it.map.toVarargArray()) }
```

So, with the use of maps, we've been able to convert classes to database registers and viceversa in a very simple way. Once we have these extension functions ready, we can use them for other projects, so it's a really well paid effort.

The complete code of this function:

```
fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {

    clear(CityForecastTable.NAME)
    clear(DayForecastTable.NAME)

    with(dataMapper.convertFromDomain(forecast)) {
        insert(CityForecastTable.NAME, *map.toVarargArray())
        dailyForecast forEach {
            insert(DayForecastTable.NAME, *it.map.toVarargArray())
        }
    }
}
```

A lot of new code was involved in this chapter, so you can take a look at the repository to review it. The changes are in `chapter-17` branch at Kotlin for Android Developers repository[32].

[32] https://github.com/antoniolg/Kotlin-for-Android-Developers

20 Null safety in Kotlin

Null safety is one of the most interesting features about Kotlin if you are currently working with Java 7. But as you have seen during this book, it's so implicit in the language we hardly had to worry about it until the previous chapter.

Being considered the billion-dollar mistake by its own creator[33], it's true that we sometimes need to define whether a variable contains a value or not. In Java, though annotations and IDEs are helping a lot these days, we can still do something like:

```
1 Forecast forecast = null;
2 forecast.toString();
```

This code will perfectly compile (you may get a warning from the IDE), and when it runs, it will obviously throw a `NullPointerException`. This is really unsafe, and as we can think we should be able to have everything under control, as the code grows we'll start losing track of the things that could be null. So we end up with lots of `NullPointerExceptions` or lots of nullity checks (probably a mix of both).

20.1 How Null types work

Most modern languages solve this issue in some way, and the Kotlin way is quite peculiar and different from the rest of similar languages. But the golden rule is the same: if a variable can be null, the compiler will force us deal with it in some way.

The way to specify that a variable can be null is by **adding a question mark to the end of its type**. As everything is an object in Kotlin (even Java primitive types), everything can be null. So, of course, we can have a nullable integer:

```
1 val a: Int? = null
```

You can't work directly with a nullable type without doing some checks before. This code won't compile:

[33] https://en.wikipedia.org/wiki/Tony_Hoare

```
val a: Int? = null
a.toString()
```

The previous code could be null, and **the compiler is aware of that**, so until the nullity is checked, you won't be able to use it. Here it is when another feature of the Kotlin compiler comes into action: the smart cast. If we check the nullity of an object, from that moment the object is automatically casted to its non-nullabe type. Let's see an example:

```
val a: Int? = null
...
if (a != null) {
    a.toString()
}
```

Inside the `if`, `a` becomes `Int` instead of `Int?`, so we can use it without checking nullity anymore. The code outside the `if` context, of course, will have to deal with it. This only works if a variable is immutable (val), because otherwise the value could've been changed from another thread and the previous check would be false at that moment.

This can sound like a lot of work. Do we have to fill all our code with nullity checks? Of course not. First, because most of the time you won't need null objects. Null references are more unused that one could think, you'll realise when you start figuring out whether a variable should be null or not. But Kotlin also has its own mechanisms to do this task easier. We can, for instance, simplify the previous code to:

```
val a: Int? = null
...
a?.toString()
```

Here we are using the **safe call operator (?.)**. The previous line will only be executed if the variable is not null. Otherwise, it will do nothing. And we can even provide an alternative for the null case using the **Elvis operator (?:)**:

```
val a: Int? = null
...
val myString = a?.toString() ?: ""
```

Since `throw` and `return` are also expressions in Kotlin, they can be used in the right side of the Elvis operator:

```
val myString = a?.toString() ?: return false
```

```
val myString = a?.toString() ?: throw IllegalStateException()
```

When we are dealing with some Java libraries though, there can be situations when we know for sure we are dealing with a non-nullable variable, but the type is nullable. We can force the compiler to deal with nullable types skipping the restriction by using the **!! operator**:

```
val a: Int? = null
a!!.toString()
```

The previous code will compile, but will obviously crash. So we must make sure we only use it in very specific situations. Most of the time we can choose alternative solutions. A code full of **!!** will be a smell of something not being done properly.

20.2 Nullity and Java libraries

Ok, so the previous explanation works perfectly well with Kotlin code. But what happens with Java libraries in general and Android SDK in particular? In Java, every object can be null by definition. So we would have to deal with a lot potentially null variables which in real life are never null. So our code could end up with hundreds of `!!` operators, which is not a good idea at all.

When you are dealing with the Android SDK, you'll probably see that all the parameters are marked with a single '**!**' when any methods are used. For instance, something that gets an `Object` in Java will be represented as `Any!` in Kotlin. This means that it's up to the developer to decide whether that variable should be null or not.

Luckily, latest versions of Android are starting using the `@Nullable` and `@NonNull`annotations to identify the parameters that can be null or the functions that can return null. So when in doubt, we can go to the source and check if there's a chance to receive a null object there. My guess is that in the future, the compiler will be able to read those annotations and force (or at least suggest) the better approach.

As of today, a warning is shown when the code is marked with a Jetbrains `@Nullable` annotation (it's not the same as the one from Android annotations) and we use a non-nullable variable. The opposite doesn't happen with a `@NotNull` annotation.

So for example, if we create a test class in Java:

```
import org.jetbrains.annotations.Nullable;

public class NullTest {

    @Nullable
    public Object getObject(){
        return "";
    }
}
```

And then we use it from Kotlin:

```
val test = NullTest()
val myObject: Any = test.getObject()
```

We'll find that the compiler shows a warning in `getObject` function. But that's the only check the compiler is doing nowadays, and it doesn't understand about Android annotations, so we'll probably have to wait some more time until this is dealt in a smarter way. However, with the help of the annotated source code and some knowledge about the Android SDK, it's really difficult to make a mistake.

Said that, if we are for instance overriding `onCreate` for an `Activity`, it's our decision to make the `savedInstanceState` nullable or not:

```
override fun onCreate(savedInstanceState: Bundle?) {
}
```

```
override fun onCreate(savedInstanceState: Bundle) {
}
```

Both ways will compile, but the second one is wrong, because an activity can perfectly receive a null bundle. Just a little of care will be enough. And when in doubt, you can just use a nullable object and deal with it properly. Remember, if you use `!!` it's because you are sure that the object can't be null, so just declare it as non-nullable.

This flexibility is really necessary to work with Java libraries, and as the compiler evolves, we'll probably see better interaction (probably based on annotations), but for now this mechanism is good and flexible enough.

21 Creating the business logic to data access

After implementing the access to the server and a way to interact with the database, it's time to put things together. The logical steps would be:

1. Request the required data from the database
2. Check if there is data for the corresponding week
3. If the required data is found, it is returned to the UI to be rendered
4. Otherwise, the data is requested to the server
5. The result is saved in the database and returned to the UI to be rendered

But our commands shouldn't need to deal with all this logic. The source of the data is an implementation detail that could easily be changed, so adding some extra code that abstracts the commands from the access to the data sounds like a good idea. In our implementation, it will iterate over a list of sources until a proper result is found.

So let's start by specifying the interface any data source that wants to be used by our provider should implement:

```
interface ForecastDataSource {
    fun requestForecastByZipCode(zipCode: Long, date: Long): ForecastList?
}
```

The provider will require a function that receives a zip code and a date, and it should return a weekly forecast from that day.

```
class ForecastProvider(val sources: List<ForecastDataSource> =
        ForecastProvider.SOURCES) {

    companion object {
        val DAY_IN_MILLIS = 1000 * 60 * 60 * 24
        val SOURCES = listOf(ForecastDb(), ForecastServer())
    }
...
}
```

The forecast provider receives a list of sources, that once again can be specified through the constructor (for test purposes for instance), but I'm defaulting it to a `SOURCES` list defined in the companion object. It will use a database source and a server source. The order is important, because it will iterate over the sources, and the search will be stopped when any of the sources returns a valid result. The logical order is to search first locally (in the database) and then through the API.

So the main method looks like this:

```
fun requestByZipCode(zipCode: Long, days: Int): ForecastList
            = sources firstResult { requestSource(it, days, zipCode) }
```

It will get the first result that is not `null`. When searching through the list of functional operators explained in chapter 18, I couldn't find one that did exactly what I was looking for. So, as we have access to Kotlin sources, I just copied `first` function and modified to behave as expected:

```
public inline fun <T, R : Any> Iterable<T>.firstResult(predicate: (T) -> R?): R {
    for (element in this) {
        val result = predicate(element)
        if (result != null) return result
    }
    throw NoSuchElementException("No element matching predicate was found.")
}
```

The function receives a predicate which gets an object from type `T` and returns a value of type `R?`. This means that the predicate can return null, but our `firstResult` function can't. That's the reason why it returns a value of type `R`.

How it works? It will iterate and execute the predicate over the elements in the `Iterable` collection. When the result of the predicate is not null, this result will be returned.

If we wanted to include the case where all the sources can return `null`, we could have derived from `firstOrNull` function instead. The difference would consist of returning `null` instead of throwing an exception in the last line. But I'm not dealing with those details in this code.

In our example `T = ForecastDataSource` and `R = ForecastList`. But remember the function specified in `ForecastDataSource` returned a `ForecastList?`, which equals `R?`, so everything matches perfectly. The function `requestSource` just makes the previous function look more readable:

```
private fun requestSource(source: ForecastDataSource, days: Int, zipCode: Long):
        ForecastList? {
    val res = source.requestForecastByZipCode(zipCode, todayTimeSpan())
    return if (res != null && res.size() >= days) res else null
}
```

The request is executed and only returns a value if the result is not null and the number of days matches the parameter. Otherwise, the source doesn't have enough up-to-date data to return a successful result.

The function `todayTimeSpan()` calculates the time in milliseconds for the current day, eliminating the "time" offset. Some of the sources (in our case the database) may need it. The server defaults to today if we don't send more information, so it won't used there.

```
private fun todayTimeSpan() = System.currentTimeMillis() /
    DAY_IN_MILLIS * DAY_IN_MILLIS
```

The complete code of this class would be:

```
class ForecastProvider(val sources: List<ForecastDataSource> =
        ForecastProvider.SOURCES) {

    companion object {
        val DAY_IN_MILLIS = 1000 * 60 * 60 * 24;
        val SOURCES = listOf(ForecastDb(), ForecastServer())
    }

    fun requestByZipCode(zipCode: Long, days: Int): ForecastList
            = sources firstResult { requestSource(it, days, zipCode) }

    private fun requestSource(source: RepositorySource, days: Int,
            zipCode: Long): ForecastList? {
        val res = source.requestForecastByZipCode(zipCode, todayTimeSpan())
        return if (res != null && res.size() >= days) res else null
    }

    private fun todayTimeSpan() = System.currentTimeMillis() /
            DAY_IN_MILLIS * DAY_IN_MILLIS
}
```

We already defined `ForecastDb`. It just now needs to implement `ForecastDataSource`:

```
class ForecastDb(val forecastDbHelper: ForecastDbHelper =
        ForecastDbHelper.instance, val dataMapper: DbDataMapper = DbDataMapper())
        : ForecastDataSource {

    override fun requestForecastByZipCode(zipCode: Long, date: Long) =
            forecastDbHelper.use {
    ...
    }
    ...
}
```

The `ForecastServer` is not implemented yet, but it's really simple. It will make use of a `ForecastDb` to save the response once it's received from the server. That way, we can keep it cached into the database for future requests.

```
class ForecastServer(val dataMapper: ServerDataMapper = ServerDataMapper(),
        val forecastDb: ForecastDb = ForecastDb()) : ForecastDataSource {

    override fun requestForecastByZipCode(zipCode: Long, date: Long):
            ForecastList? {
        val result = ForecastByZipCodeRequest(zipCode).execute()
        val converted = dataMapper.convertToDomain(zipCode, result)
        forecastDb.saveForecast(converted)
        return converted
    }

}
```

It also makes use of a data mapper, the first one we created, though I modified the name of some methods to make it similar to the data mapper we used for the database model. You can take a look at the provider to see the details.

The overridden function makes the request to the server, converts the result to domain objects and saves them into the database. It finally returns the converted result.

With these last steps, the provider is already implemented. Now we need to start using it. The `ForecastCommand` no longer should interact directly with server requests, nor convert the data to the domain model.

```
class RequestForecastCommand(val zipCode: Long,
        val forecastProvider: ForecastProvider = ForecastProvider()) :
        Command<ForecastList> {

    companion object {
        val DAYS = 7
    }

    override fun execute(): ForecastList {
        return forecastProvider.requestByZipCode(zipCode, DAYS)
    }
}
```

The rest of code modifications consist of some renames and package organisation here and there. Take a look at `chapter-21` branch at Kotlin for Android Developers repository[34].

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

22. Control flow and ranges

I've been using some conditional expressions in our code, but now it's time to explain them more deeply. Though we'll usually use less mechanisms to control the flow of the code that we'd normally use in a completely procedural programming language (some of them even practically disappear), they are still useful. There are also new powerful ideas that will solve some particular problems much easier.

22.1 If Expression

Almost everything in Kotlin is an expression, which means it returns a value. `If` conditions are not an exception in this, so though we can use `if` as we are used to do it:

```
1  if (x > 0) {
2          toast("x is greater than 0")
3  } else if (x == 0) {
4          toast("x equals 0")
5  } else {
6          toast("x is smaller than 0")
7  }
```

We can also assign its result to a variable. We've used it like that several times in our code:

```
1  val res = if (x != null && x.size() >= days) x else null
```

This also implies we don't need a ternary operation similar to the Java one, because we can solve it easily with:

```
1  val z = if (condition) x else y
```

So the `if` expression always returns a value. If one of the branches returns `Unit`, the whole expression will return `Unit`, which can be ignored, and it will work as a regular Java `if` condition in that case.

22.2 When expression

`When` expressions are similar to `switch/case` in Java, but far more powerful. This expression will try to match its argument against all possible branches until it finds one that is satisfied. It will then apply the right side of the expression. The difference with a `switch/case` in Java is that the argument can be literally anything, and the conditions for the branches too.

For the default option, we can add an `else` branch that will be executed if none of the previous conditions is satisfied. The code executed when a condition is satisfied can be a block too:

```
1  when (x) {
2      1 -> print("x == 1")
3      2 -> print("x == 2")
4      else -> {
5          print("I'm a block")
6          print("x is neither 1 nor 2")
7      }
8  }
```

As it's an expression, it can return a result too. Take into consideration that when used as an expression, it must cover all the possible cases or implement the `else` branch. It won't compile otherwise:

```
1  val result = when (x) {
2      0, 1 -> "binary"
3      else -> "error"
4  }
```

As you can see, the condition can be a set of values separated with commas. But it can be many more things. We could, for instance, check the type of the argument and take decisions based on this:

```
1  when(view) {
2      is TextView -> view.setText("I'm a TextView")
3      is EditText -> toast("EditText value: ${view.getText()}")
4      is ViewGroup -> toast("Number of children: ${view.getChildCount()} ")
5      else -> view.visibility = View.GONE
6  }
```

The argument is automatically casted in the right part of the condition, so you don't need to do any explicit casting.

It's possible to check whether the argument is inside a range (I'll explain ranges later in this chapter), or even inside a collection:

```
val cost = when(x) {
    in 1..10 -> "cheap"
    in 10..100 -> "regular"
    in 100..1000 -> "expensive"
    in specialValues -> "special value!"
    else -> "not rated"
}
```

Or you could even get rid of the argument and do any crazy check you may need. It could easily substitute an `if / else` chain:

```
val res = when {
    x in 1..10 -> "cheap"
    s.contains("hello") -> "it's a welcome!"
    v is ViewGroup -> "child count: ${v.getChildCount()}"
    else -> ""
}
```

22.3 For loops

Though you won't probably use them too much if you make use of functional operators in collections, `for` loops can be useful in some situations, so they are still available. It works with anything that provides an iterator:

```
for (item in collection) {
        print(item)
}
```

If we want to get a more typical iteration over indices, we can also do it using ranges (there are usually smarter solutions anyway):

```
for (index in 0..viewGroup.getChildCount() - 1) {
    val view = viewGroup.getChildAt(index)
    view.visibility = View.VISIBLE
}
```

When iterating over an array or a list, a set of indices can be requested to the object, so the previous artifact is not necessary:

```
for (i in array.indices)
    print(array[i])
```

22.4 While and do/while loops

You can keep using while loops too, though it won't be very common either. There are usually simpler and more visual ways to resolve a problem. A couple of examples:

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

22.5 Ranges

It's difficult to explain control flow without talking about ranges. But their scope is much wider. Range expressions make use of an operator in the form of ". . " that is defined implementing a `RangeTo` function.

Ranges help simplify our code in many creative ways. For instance we can convert this:

```
if (i >= 0 && i <= 10)
    println(i)
```

Into this:

```
if (i in 0..10)
    println(i)
```

Range is defined by any type that can be compared, but for numerical types the compiler will optimise it by converting it to simpler analogue code in Java, to avoid the extra overhead. The numerical ranges can also be iterated, and the loops are optimised too by converting them to the same bytecode a `for` with indices would use in Java:

```
for (i in 0..10)
    println(i)
```

Ranges are incremental by default, so something like:

```
for (i in 10..0)
    println(i)
```

Would do nothing. You can, however, use the function `downTo`:

```
for (i in 10 downTo 0)
    println(i)
```

What if you want to iterate over a double? You can do it too, though the default step is `1`, which means that a range like `1.0..2.0` will return only two values: `1.0` and `2.0`. However, we can define whatever step we want, so we could do:

```
for (i in 1.0..2.0 step 0.1) print("$i ")
```

That would print all the intermediate values with a `0.1` step: `1.0`, `1.1`, ..., `1.9`, `2.0`. The step can be used on any numerical value. Integers can make use of it too:

```
for (i in 1..4 step 2) println(i)

for (i in 4 downTo 1 step 2) println(i)
```

As mentioned before, there are really creative ways to use ranges. For instance, an easy way to get the list of `Views` inside a `ViewGroup` would be:

```
val views = (0..viewGroup.getChildCount() - 1) map { viewGroup.getChildAt(it) }
```

The mix of ranges and functional operators prevents from having to use an explicit loop to iterate over the collection, and the creation of an explicit list where we add the views. Everything is done in a single line.

If you want to know more about how ranges are implemented and a lot more examples and useful information, you can go to Kotlin reference[35].

[35] http://kotlinlang.org/docs/reference/ranges.html

23 Creating a Detail Activity

When we click on an item at home, we would expect to navigate to a detail activity and see some extra info about the forecast for that day. We are currently showing a toast when an item is clicked, but it's time to change that.

23.1 Preparing the request

As we need to know which item we are going to show in the detail activity, logic tells we'll need to send the id of the forecast to the detail. So the domain model needs a new id property:

```
data class Forecast(val id: Long, val date: Long, val description: String,
    val high: Int, val low: Int, val iconUrl: String)
```

The `ForecastProvider` also needs a new function, which returns the requested forecast by id. The `DetailActivity` will need it to recover the forecast based on the id it will receive. As all the requests always iterate over the sources and return the first non-null result, we can extract that behaviour to another function:

```
private fun requestToSources<T : Any>(f: (ForecastDataSource) -> T?): T
         = sources firstResult { f(it) }
```

The function is generified using a non-nullable type. It will receive function which uses a `Forecast-DataSource` to return an nullable object of the generic type, and will finally return a non-nullable object. We can rewrite the previous request and write the new one this way:

```
fun requestByZipCode(zipCode: Long, days: Int): ForecastList = requestToSources {
    val res = it.requestForecastByZipCode(zipCode, todayTimeSpan())
    if (res != null && res.size() >= days) res else null
}

fun requestForecast(id: Long): Forecast = requestToSources {
    it.requestDayForecast(id)
}
```

Now the data sources need to implement the new function:

```
fun requestDayForecast(id: Long): Forecast?
```

The `DbProvider` will always have the required value already cached from previous requests, so we can get it from there this way:

```
override fun requestDayForecast(id: Long): Forecast? = forecastDbHelper.use {
    val forecast = select(DayForecastTable.NAME).byId(id).
            parseOpt { DayForecast(HashMap(it)) }

    if (forecast != null) dataMapper.convertDayToDomain(forecast) else null
}
```

The `select` query is very similar to the previous one. I created another utility function called `byId`, because a request by id is so common that a function like that simplifies the process and is easier to read. The implementation of the function is quite simple:

```
fun SelectQueryBuilder.byId(id: Long): SelectQueryBuilder
        = whereSupport("_id = ?", id.toString())
```

It just makes use of the `whereSupport` function and implements the search over the `_id` field. This function is quite generic, but as you can see, you could create as many extension functions as you need based on the structure of your database, and hugely simplify the readability of your code. The `DbDataMapper` has some slight changes not worth mentioning. You can check them in the repository.

On the other hand, the `ForecastServer` will never be used, because the info will be always cached in the database. We could implement it to defend our code from strange situations, but we're not doing it in this case, so it will just throw an exception if it's called:

```
override fun requestDayForecast(id: Long): Forecast?
        = throw UnsupportedOperationException()
```

`try` and `throw` are expressions

In Kotlin, almost everything is an expression, which means it returns a value. This is really important for functional programming, and particularly useful when dealing with edge cases with `try-catch` or when throwing exceptions. For instance, in the previous example we can assign an exception to the result even if they are not of the same type, instead of having to create a full block of code. This is very useful too when we want to throw an exception in one of `when` branches:

```
1  val x = when(y){
2      in 0..10 -> 1
3      in 11..20 -> 2
4      else -> throw Exception("Invalid")
5  }
```

The same happens with `try-catch`, we can assign a value depending on the result of the try:

```
1  val x = try { doSomething() } catch { null }
```

The last thing we need to be able to perform the request from the new activity is to create a command. The code is really simple:

```
1  class RequestDayForecastCommand(
2          val id: Long,
3          val forecastProvider: ForecastProvider = ForecastProvider()) :
4          Command<Forecast> {
5
6      override fun execute() = forecastProvider.requestForecast(id)
7  }
```

The request returns a `Forecast` result that will be used by the activity to draw its UI.

23.2 Providing a new activity

We are now prepared to create the `DetailActivity`. Our detail activity will receive a couple of parameters from the main one: the forecast id and the name of the city. The first one will be used to request the data from the database, and the name of the city will fill the toolbar. So we first need a couple of names to identify the parameters in the bundle:

```
public class DetailActivity : AppCompatActivity() {

    companion object {
        val ID = "DetailActivity:id"
        val CITY_NAME = "DetailActivity:cityName"
    }
    ...
}
```

In onCreate function, the first step is to set the content view. The UI will be really simple, but more than enough for this app example:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center_vertical"
        tools:ignore="UseCompoundDrawables">

        <ImageView
            android:id="@+id/icon"
            android:layout_width="64dp"
            android:layout_height="64dp"
            tools:src="@mipmap/ic_launcher"
            tools:ignore="ContentDescription"/>

        <TextView
            android:id="@+id/weatherDescription"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/spacing_xlarge"
```

```
            android:textAppearance="@style/TextAppearance.AppCompat.Display1"
            tools:text="Few clouds"/>

    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/maxTemperature"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_margin="@dimen/spacing_xlarge"
            android:gravity="center_horizontal"
            android:textAppearance="@style/TextAppearance.AppCompat.Display3"
            tools:text="30"/>

        <TextView
            android:id="@+id/minTemperature"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_margin="@dimen/spacing_xlarge"
            android:gravity="center_horizontal"
            android:textAppearance="@style/TextAppearance.AppCompat.Display3"
            tools:text="10"/>

    </LinearLayout>

</LinearLayout>
```

Then assign it from `onCreate` code. Use the city name to fill the toolbar title. I'm also using an extension property from Anko, `intent`, which basically replaces `getIntent()` function:

```
setContentView(R.layout.activity_detail);
setTitle(intent.getStringExtra(CITY_NAME))
```

The other part in `onCreate` implements the call to the command. It's very similar to the call we previously did:

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

When the result is recovered from the database, the `bindForecast` function is called in the UI thread. I'm using Kotlin Android Extensions plugin again in this activity, to get the properties from the XML without using `findViewById`:

```
import kotlinx.android.synthetic.activity_detail.*

...

private fun bindForecast(forecast: Forecast) = with(forecast) {
    Picasso.with(ctx).load(iconUrl).into(icon)
    getSupportActionBar().setSubtitle(date.toDateString(DateFormat.FULL))
    weatherDescription.text = description
    bindWeather(high to maxTemperature, low to minTemperature)
}
```

There are some interesting things here. For instance, I'm creating another extension function able to convert a `Long` object into a visual date string. Remember we were using it in the adapter too, so it's a good moment to extract it into a function:

```
public fun Long.toDateString(dateFormat: Int = DateFormat.MEDIUM): String {
    val df = DateFormat.getDateInstance(dateFormat, Locale.getDefault())
    return df.format(this)
}
```

It will get a date format (or use the default `DateFormat.MEDIUM`) and convert the `Long` into a `String` that is understandable by the user.

Another interesting function is `bindWeather`. It will get a `vararg` of pairs of `Int` and `TextView`, and assign a text and a text color to the `TextViews` based on the temperature.

```
private fun bindWeather(vararg views: Pair<Int, TextView>) = views forEach {
    it.second.text = "${it.first.toString()}□□"
    it.second.textColor = color(when (it.first) {
        in -50..0 -> android.R.color.holo_red_dark
        in 0..15 -> android.R.color.holo_orange_dark
        else -> android.R.color.holo_green_dark
    })
}
```

For each pair, it assigns the text that will show the temperature and a color based on the value of the temperature: red for low temperatures, orange for medium ones and green for the rest. The values are taken quite randomly, but it's a good representation of what we can do with a `when` expression, how clean and short the code becomes.

`color` is an extension function I miss from Anko, which simplifies the way to get a color from resources, similar to the `dimen` one we've used in some other places. As the time of writing this lines, current support library relies on the class `ContextCompat` to get a color in a compatible way in all Android versions:

```
public fun Context.color(res: Int): Int = ContextCompat.getColor(this, res)
```

The `AndroidManifest` also needs to be aware that a new activity exists:

```
<activity
    android:name=".ui.activities.DetailActivity"
    android:parentActivityName=".ui.activities.MainActivity" >
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.antonioleiva.weatherapp.ui.activities.MainActivity" />
</activity>
```

23.3 Start an activity: reified functions

The last step consists of starting the detail activity from the main activity. We can rewrite the adapter instantiation this way:

```
val adapter = ForecastListAdapter(result) {
    val intent = Intent(MainActivity@this, javaClass<DetailActivity>())
    intent.putExtra(DetailActivity.ID, it.id)
    intent.putExtra(DetailActivity.CITY_NAME, result.city)
    startActivity(intent)
}
```

But this is too verbose. As usual, Anko provides a much simpler way to start an activity by using a **reified function**:

```
val adapter = ForecastListAdapter(result) {
    startActivity<DetailActivity>(DetailActivity.ID to it.id,
            DetailActivity.CITY_NAME to result.city)
}
```

What's the magic behind reified functions? As you may know, when we create a generic method in Java, there is no way to get the class from the generic type. The typical workaround is passing the class as a parameter. In Kotlin, an inline function can be reified, which means we can get and use the class of the generic type inside the function. In this case, we can create the intent inside the function, by calling `javaClass<T>()`. A simplified version of what Anko really does would be the next (I'm only using `String` extras in this example):

```
public inline fun <reified T: Activity> Context.startActivity(
        vararg params: Pair<String, String>) {

    val intent = Intent(this, javaClass<T>())
    params forEach { intent.putExtra(it.first, it.second) }
    startActivity(intent)
}
```

The real implementation is a bit more complex because it uses a long and boring `when` expression to add the extras depending on the type, but it doesn't add much useful knowledge to the concept.

Reified functions are, once more, a syntactic sugar that simplifies the code and improves its comprehension. In this case, it creates an intent by getting the `javaClass` from the generic type, iterates over params and adds them to the intent, and start the activity using the intent. The reified type is limited to be an `Activity` descendant.

The rest of little details are covered in `chapter-23` branch at Kotlin for Android Developers repository[36]. We now have a very simple (but complete) master-detail App implemented in Kotlin without using a single line of Java.

[36] https://github.com/antoniolg/Kotlin-for-Android-Developers