KOTLIN

FOR ANDROID DEVELOPERS

Learn Kotlin the easy way while developing an Android App

6th edition
Updated to Kotlin 1.2.31

Antonio Leiva

Kotlin for Android Developers

Learn Kotlin the 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 2018-04-05

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:

#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 that this book was a necessary step forward.

Special mention goes to Luis Herrero, who designed the excellent cover of this book, and to Gautier Mechling for helping me so much by reviewing this book. It is thanks to him that these pages are not full of typos and mistakes.

And, of course, this is specially dedicated to you. With your support and your help, this book is growing and becoming a reference. So any suggestions to improve the quality of this book will be welcomed. Feel free to write anytime to contact@antonioleiva.com.

Contents

I. About this book

Hey! Thanks so much for your interest in this book, I am thrilled that you decided to become a Kotlin for Android expert.

Many things have happened since I started writing "Kotlin for Android Developers" in mid-2015. At that point, Kotlin was still in an early access edition. However, I felt so much power in the language, that I decided to explain everything that I was learning in a book.

Since then, the language has kept growing to the point that Google announced that they would start supporting Kotlin as an official language to develop Android apps.

These were amazing news. But it was even more amazing that both Google and JetBrains suggested this book to Android developers who want to learn the language.

As you may know, this is a lean publication. This book grew and progressed thanks to the readers' comments. So I can only thank you for helping me bring this dream to life.

What is "Kotlin for Android Developers" about

Throughout this book, I create an Android app from the ground up using Kotlin as the primary language. The idea is to learn the language by example, instead of following a regular reference book structure. I will be stopping to explain the most useful concepts and ideas about Kotlin, comparing them to Java 6. This way, you can see what the differences are and which parts of the language can 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 projects by themselves. The examples are meant to overcome the most recurring problems we have to face in our daily lives as app developers, by making use of Kotlin's expressiveness and some other exciting tools and libraries. However, this text covers most of the features of Kotlin, so by the end of the reading, you will have an in-depth knowledge of the language.

The content is very practical, so I recommend that you follow the examples and the code in front of a computer and try everything it suggests. You could, however, take one first read to get a broad idea and then dive into practice.

Even though this book now finished, I will review it from time to time to keep it up to date with new Kotlin versions. So feel free to write and tell me what you think about the book, or what could be improved. I want this book to be the perfect tool for Android developers, and as such, help and ideas will be welcomed.

If you are reading a printed copy and want to receive the latest updates, please feel free to email me at *contact@antonioleiva.com* with a proof of purchase, and I will give you access to the digital copy. That way, you will keep receiving the updates. Do the same if you got it from any bookstores (Kindle, Kobo…) and you want to get access to the PDF version.

Thanks for becoming part of this exciting project!

II. Is this book for you?

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

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

- You have some basic knowledge of Android Development and the Android SDK, as well as the Java language.
- 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 common challenges Android developers face every day, by using a cleaner and more expressive language.

On the other hand, this book may not be for you. These are the topics that you will not find in it:

- The content of these pages is not a Kotlin Bible. I shall explain all language basics, and even more complicated 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 do not need a deep understanding of the platform, but at least 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!
- The book is not a guide to functional programming. Of course, I am showing what you need, as Java 6 is not functional at all, but I will not dive deep into the programming paradigm.

III. About the author

Antonio Leiva is an Android Engineer who spends time learning about new ways to get the most out of Android and then writes about it. He writes a blog at antonioleiva.com[1], focused on helping other Android developers learn Kotlin.

He also leads intensive live workshops, where the attendants put all the content from this book into practice. In 10 hours, people steps from no Kotlin knowledge to being able to create their apps from scratch.

Antonio started as a consultant in CRM technologies, but after some time, looking for his real passion, he discovered the Android platform. After getting some experience, he started a new adventure at a mobile company, where he led several projects for well-known companies in Spain.

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

You can find Antonio on Twitter as @lime_cl[3] or Google+ as +AntonioLeivaGordillo[4].

[1]http://antonioleiva.com

[2]http://plex.tv

[3]https://twitter.com/lime_cl

[4]http://plus.google.com/+AntonioLeivaGordillo‘

1 Introduction

Things are changing for good for Android Developers. In Google I/O 2017, the Android team announced that Kotlin was becoming an official language to develop Android apps.

This means that, while it is still possible to develop Android apps using Java, from now on Kotlin is fully supported, and Google is making sure that every new Android feature, the framework, the IDE and all their libraries work seamlessly with the new language.

Google listened to the community, who was asking for years that Kotlin became a first-party language. So you can now take advantage of all the features of a modern language while developing for Android.

Throughout this book, I will show you how, so I hope that I can help you understand the various ways that Kotlin can take you one step ahead and make your code much better.

However, before diving into the features of the language, let me tell you just a little bit of background.

1.1 What is Kotlin?

Kotlin is language developed by JetBrains[5], a company known for building the IntelliJ IDEA, a powerful IDE for Java development. Android Studio, the official Android IDE, is based on IntelliJ. It was initially implemented to run on the Java Virtual Machine.

JetBrains designed Kotlin with Java developers in mind, and with IntelliJ as its primary development IDE. These two factors are breaking points that made Android developers quickly adopt the language:

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

- **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 can be mastered in no time.
- **We have total integration with our daily IDE for free**. Android Studio can understand, compile and run Kotlin code. Moreover, the support for this language comes from the company who develops the IDE, so we Android developers are first-class citizens.

However, this is only related to how the language integrates with our tools. What are the advantages of the language when compared to Java 6?

- ** It is more expressive**: this is one of its main points. You can write more with much less code.
- ** It is safer**: Kotlin is null safe, which means that we deal with possible null situations at compile time, to prevent execution time exceptions. We need to specify that an object can be null explicitly, and then check its nullity before using it. You can save much time debugging null pointer exceptions and fixing nullity bugs.
- ** It is functional**: Kotlin is fundamentally an object-oriented language, not a pure functional language. However, like many other modern languages, it uses many concepts from functional programming, such as lambda expressions, to solve some problems more naturally. Another nice feature 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 do not have access to the source code.
- ** It is highly interoperable**: You can continue using most libraries and code written in Java because the interoperability between both languages is excellent. It is even possible to create mixed projects, with both Kotlin and Java files coexisting.

However, this is only the tip of the iceberg: - Since Kotlin 1.1, the final version of Kotlin JS[6] was released. This new variant allows you to develop web apps using

[6]https://kotlinlang.org/docs/reference/js-overview.html

Kotlin. - Since Kotlin 1.2, you can also create multiplatform projects[7]. With it, you can share code between JVM and Javascript. - As an experimental feature, the JetBrains team has also released Kotlin/Native[8], a project that finally takes Kotlin out of the JVM. In the future, we will be able to implement the server, the web and the Android and iOS Apps using Kotlin for most of the code base. Also, you can try it today. - Gradle is adding support to use Kotlin Script (a simplified version of Kotlin) to write Gradle files instead of Groovy. The project is quickly approaching its 1.0 release.

So you can see that the future of Kotlin is pretty promising. Learning Kotlin can become the language of reference in many other platforms, and sharing code among all them is undoubtedly a high selling point.

1.2 What do we get with Kotlin?

Without diving too deep into the Kotlin language (we will learn everything about it throughout this book), these are some interesting features we miss in Java:

Expressiveness

With Kotlin, it is much easier to avoid boilerplate because the language covers the most common patterns by default. For instance, in Java, if we want to create a data class, we need to write (or at least generate) this code:

```java
public class Artist {
    private long id;
    private String name;
    private String url;
    private String mbid;

    public long getId() {
        return id;
    }

    public void setId(long id) {
```

[7] https://kotlinlang.org/docs/reference/whatsnew12.html#multiplatform-projects-experimental
[8] https://kotlinlang.org/docs/reference/native-overview.html

```
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getMbid() {
        return mbid;
    }

    public void setMbid(String mbid) {
        this.mbid = mbid;
    }

    @Override public String toString() {
        return "Artist{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", url='" + url + '\'' +
                ", mbid='" + mbid + '\'' +
                '}';
    }
}
```

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

```kotlin
data class Artist(
    var id: Long,
    var name: String,
    var url: String,
    var mbid: String)
```

This data class auto-generates all the fields and property accessors, as well as some useful methods such as `toString()`. You also get `equals()` and `hashCode()` for free, which are very verbose and can be dangerous if they are incorrectly implemented.

Null Safety

When we use Java, a significant amount of our code is defensive. We need to check once and another whether something is null before using it to prevent unexpected *NullPointerException*. Kotlin, like many other modern languages, is null-safe because the type explicitly defines whether an object can be null by using the safe call operator (written `?`).

We can do things like this:

```kotlin
// This does not compile. Artist cannot be null
var notNullArtist: Artist = null

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

// Will not 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 not need to use safe call operator if we previously
// checked nullity
if (artist != null) {
    artist.print()
}

// Only use it when we are sure it is not null. It throws 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

Thanks to extension functions, you can add new functions to any class. It is a cleaner substitute for the common utility classes we all have in our projects. You 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()
}
```

And then use it like this:

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

Functional support (Lambdas)

What if, instead of having to declare an anonymous class every time we need to implement a click listener, we could just define what we want to do? We can indeed do it. This (and many other interesting things) is what we get thanks to lambdas:

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

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

2 Getting ready

Now that you know some little examples of what you may do with Kotlin, I am 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

The first thing you need is to install Android Studio. As you may know, Android Studio is the official Android IDE, which was presented to the public in 2013 as a preview and finally released in 2014.

Android Studio is implemented as a plugin for IntelliJ IDEA[9], a Java IDE created by Jetbrains[10], the company which is also behind Kotlin. So, as you can see, everything is tightly connected.

The adoption of Android Studio was a significant step forward for Android developers. First, because we left the buggy Eclipse behind and moved to a set of tools specially designed for Java developers, which gives us a whole interaction with the language. We enjoy fantastic features such as fast and impressively smart code completion, and powerful analyzing and refactoring tools among others.

And second, Gradle[11] became the official build system for Android, which means a whole bunch of new possibilities relating to versioning, building, and deployment. Two of the most exciting functions are build types and flavors, which let you create infinite versions of the app (or even different apps) in a straightforward way while using the same code base.

If you are still using Eclipse, I am afraid you need to switch to Android Studio if you want to follow this book. The Kotlin team is creating a plugin for Eclipse, but it

[9] https://www.jetbrains.com/idea
[10] https://www.jetbrains.com
[11] https://gradle.org/

may be far behind the one for Android Studio, and the integration is probably less smooth. You will also discover what you are missing as soon you start using it.

I am not covering the use of Android Studio or Gradle because this is not the focus of the book. But if this is the first time you use these tools, I am sure that you will be able to follow the content and learn the basics in the meanwhile.

Download Android Studio from the official page[12] if you do not have it already.

2.2 Install Kotlin plugin

Since IntelliJ 15 was released, the Kotlin plugin is bundled with the IDE. For Android Studio, it depends on the version you are using.

If you have Android Studio 3.0 or above, you already have everything you need. If your Android Studio is 2.3 or below, you need to install the Kotlin plugin manually.

So go to the plugins section inside *Android Studio Preferences*, and install the Kotlin plugin. Use the search tool if you cannot find it.

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

[12] https://developer.android.com/sdk/index.html

3 Creating a new project

So now it is time to start creating our App. I want to keep the explanation simple on purpose, because Android Studio wizard changes from time to time and these lines can become deprecated pretty soon.

Our app consists of a simple weather app, such as the one used in Google's Beginners Course from Udacity[13]. We will be paying attention to different things, but the idea of the app is the same because it includes many of the things you find in a regular app. If you are starting with Android development, I recommend you this course.

3.1 Create a new project in Android Studio

Again, there is a small difference depending on the Android Studio version you are using.

For Android Studio 3, the steps are pretty straightforward. Previous versions require a little more work, but it is far from complicated too. Even if you are already using new versions of Android Studio, I recommend you to take a look to the whole process, because it is what you would need to do if you want to start using Kotlin on a Java project.

First of all, open Android Studio and choose `Create New Project`. The first step is to decide a name for your app; you can use the name you want: `WeatherApp` for instance. Then the wizard asks for a Company Domain. As you are not releasing the app, this field is not very important either, but if you own a domain, you can use that one. Also, choose the location where you want to save the project.

In Android Studio 3, at this point, you find a checkbox that enables Kotlin support. So select it if that is your case.

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

For the next step, the wizard asks for the minimum API version. I selected API 15 for the sample app, but Kotlin does not have any restrictions regarding the minimum API. Skip any other platform different to `Phone and Tablet` for now.

Finally, we are required to choose an activity template. Select `Empty Activity`. This template generates little code, and it is easier for us to add new logic.

Keep the name of the activities or layouts in the next screen. We will change them later if we need to. Press `Finish` and let Android Studio do its work.

With Android Studio 3 you are done, skip to point 3.4. The next two steps are only required if you had to install the plugin manually.

3.2 Convert MainActivity to Kotlin code

An interesting feature that the Kotlin plugin includes is the ability to convert from Java to Kotlin code. As an automated process, it is not perfect, but it may help a lot during your first days until you start getting used to the Kotlin language.

So we are using this in our `MainActivity.java` class. Open the file and select `Code -> Convert Java File to Kotlin File`. Take a look at the differences, so that you start becoming familiar with the language.

You can also copy any Java code into a Kotlin file, and the plugin also converts it.

3.3 Configure Kotlin in project

The Kotlin plugin also includes a tool that configures the Gradle files for us.

Just go to `Tools -> Project -> Configure Kotlin in Project`. Choose the latest Kotlin version, and press OK.

Review the changes. There should be something like this:

build.gradle

```
buildscript {
    ext.kotlin_version = '1.2.31'
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.1.0'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

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

As you can see, it is creating a variable that saves the current Kotlin version. We need that version number in several places, for instance in the new dependency to the Kotlin plugin.

It will be useful again in the module `build.gradle`, where the configuration has also added a dependency to the **Kotlin standard library**. Go check it now:

app/build.gradle

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

android {
    ...
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

3.4 Include some other useful configuration

There are some other extra libraries that we will be using, so take the chance to add them to the `build.gradle`.

An important one is Anko, a library that uses the power of Kotlin to simplify some tasks in Android. We will use more modules of Anko later on, but for now, it is enough to add `anko-common`. This library is split into several smaller ones so that we include in the project only the parts we require.

Let's go to the main one first, and update the buildscript by adding two new variables for the support libraries as well as Anko library (you can also check the latest version here[14]). This way, it is easier to modify all the versions in a row, as well as adding new libraries that use the same version without having to change it everywhere:

build.gradle

```
buildscript {
    ext.support_version = '27.1.0'
    ext.kotlin_version = '1.2.31'
    ext.anko_version = '0.10.4'
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.1.0'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}

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

If you are using Android Studio 2.3, you also need to add the Google repository as you can see above. Android Studio 3 does it for you.

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

Let's then add the **Anko** library dependency to the module file, update the **AppCompat** to use our variable, and apply the **Kotlin Android Extensions** plugin:

app/build.gradle

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

android {
    ...
}

dependencies {
    implementation "com.android.support:appcompat-v7:$support_version"
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation "org.jetbrains.anko:anko-common:$anko_version"
}
```

With these changes, we should be ready to start.

3.5 Test that everything works

We are going to add some code to check that Kotlin Android Extensions are fully functional. I am not explaining much about it yet, but I want to be sure this is working for you.

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

activity_main.xml

```
<TextView
    android:id="@+id/message"
    android:text="@string/hello_world"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

At `onCreate`, try to write the id you previously chose (`message` in this example) for the `TextView`:

MainActivity.kt

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

Do you see the magic? You could access the view without finding it or using third-party libraries. *Kotlin Android Extensions* is a plugin that comes included in the main Kotlin one, and that is its primary purpose. Be patient; I shall show you how to use it properly soon.

You will also see that a synthetic import was added automatically to the activity, but we are not covering it yet:

MainActivity.kt

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

Thanks to Kotlin interoperability with Java, we can use setters and getters methods from Java libraries as a property in Kotlin. We will talk about properties later too, but just notice that we can use `message.text` instead of `message.setText` for free. The compiler uses the real Java methods, so there is no performance overhead when using it.

Now run the app and check that everything is working fine. Review that the message `TextView` is showing the new content. If you have any doubts or want to revise some code, take a look at Kotlin for Android Developers repository[15]. I created a branch for each chapter when the new content involves modifications in code, so be sure to review it to check all the changes.

Next chapters cover some of the new things you could see in the converted `MainActivity`. Once you understand the slight differences between Java and Kotlin, you will be able to write new code by yourself without any hassle.

[15] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-3

4 Classes and functions

Classes in Kotlin follow a straightforward structure. However, there are some slight differences from Java that you want to know before we continue. You can use try.kotlinlang.org[16] to test this and some other simple examples without the need for a real project.

You could also use the REPL that comes bundled with the Kotlin plugin. You can find it in *Tools -> Kotlin -> Kotlin REPL*.

4.1 How to declare a class

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

```kotlin
class MainActivity {

}
```

Classes have a unique default constructor. We will see that we can create extra constructors for some exceptional cases, but keep in mind that most situations only require a single constructor. Parameters are written just after the name. Braces are not required if the class is empty:

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

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

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

```
class Person(name: String, surname: String) {
    init {
    ...
    }
}
```

4.2 Class inheritance

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

```
open class Animal(name: String)
class Person(firstName: String, lastName: String) : Animal(firstName)
```

Note that when using the single constructor structure, we need to specify the parameters we are using for the parent constructor. That is equivalent to calling `super()` in Java.

4.3 Functions

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

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

Functions in Kotlin always return a value. If you skip the return value, the function is indeed returning `Unit`.

`Unit` is similar to `void` in Java, though this is, in fact, an object. You can, of course, specify any type as a return value:

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

Tip: Semicolons are not necessary

As you can see in the example above, I am not using semicolons at the end of the sentences. While you can use them, semicolons are not necessary, and it is a good practice to avoid them (the IDE warns you if you write them). When you get used, you will find that it saves you loads of time.

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

```kotlin
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.

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

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

```kotlin
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, which allows you to write or omit the second value. This feature prevents the need for function overloading:

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

The previous function 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 = MainActivity::class.java.simpleName,
              length: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(this, "[$tag] $message", length).show()
}
```

I added a third parameter that includes a tag which defaults to the class name. The number of overloads we would need in Java grows exponentially. You can now write these calls:

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

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:

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

Tip: String templates

You can use template expressions directly in your strings, which simplifies writing complex strings based on static and variable parts. In the previous example, I used `"[$className] $message"`.

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

5 Writing your first class

We already have our `MainActivity.kt` class. This activity will render a list of daily forecasts for the next seven days, so the layout requires some changes.

5.1 Creating the layout

The main view that will render the forecast list will be a `RecyclerView`, so a new dependency is required. Modify the `build.gradle` file:

app/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, in `activity_main.xml` :

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 that everything worked (it will be showing an error now). We will continue using the good old `findViewById()` for the time being:

MainActivity.kt

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

As you can see, we define the variable and cast it to `RecyclerView`. It is a bit different from Java; we are reviewing these differences in the next chapter. A `LayoutManager` is also specified, using the property naming instead of the setter. A list is enough for this layout, so let's make use of a `LinearLayoutManager`.

Object instantiation

Object instantiation presents some differences from Java too. You might have realized I omitted the "`new`" keyword. 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` on my blog[17] some time ago, so it may help you if you are not used to it.

The views used for `RecyclerView` adapter will be just `TextViews` for now, and a simple list of texts that we will create manually. Add a new Kotlin file called `ForecastListAdapter.kt`, and include this code:

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

ForecastListAdapter.kt

```kotlin
class ForecastListAdapter(val items: List<String>) :
        RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

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

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.textView.text = items[position]
    }

    override fun getItemCount(): Int = items.size

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

Again, we can access the context and the text as properties. You can keep doing it as usual (using getters and setters), but you will get a warning from the compiler. This check can be disabled if you prefer to keep using the Java way. However, once you get used to properties, you will never look back.

Default visibility is `public`

Unless a visibility modifier is applied, classes, functions or properties are `public` by default. You can write it, but the compiler will show a warning, as it is not required.

If you check the previous code, you may find there are warnings for a couple of things. The first one is recommending to set the `items` argument to private, as it is being used only inside the class.

```kotlin
class ForecastListAdapter(private val items: List<String>)
```

The other is recommending to write single-lined functions using an expression body. Let’s follow the recommendations here too:

```kotlin
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        ViewHolder(TextView(parent.context))
```

So the complete resulting code is:

ForecastListAdapter.kt

```kotlin
class ForecastListAdapter(private val items: List<String>)
    : RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
            ViewHolder(TextView(parent.context))

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.textView.text = items[position]
    }

    override fun getItemCount(): Int = items.size

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

Back to the `MainActivity`, let's create the list of strings and then an instance of the adapter:

MainActivity.kt

```kotlin
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<RecyclerView>(R.id.forecast_list)
    forecastList.layoutManager = LinearLayoutManager(this)
    forecastList.adapter = ForecastListAdapter(items)
}
```

List creation

Though I will talk about collections later in this book, I just want to explain for now that you can create constant lists (called immutable, we will see this concept 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`, `mutableListOf` or `hashSetOf`, among others.

I have also moved some classes to new packages to achieve a better project structure, so check the corresponding branch[18] if you want to follow the same organization.

We examined numerous new ideas in such a small amount of code, so let's cover them in the next chapter. We must stop here to learn some crucial concepts regarding basic types, variables, and properties.

[18] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-5

6 Variables and properties

In Kotlin, **everything is an object**. There are no primitive types as the ones we can use in Java. That is helpful because we have a uniform 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 like an object. The name of the basic types and the way they work are very similar to Java, but there are some differences you might take into account:

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

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

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

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

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

```
// Java
int bitwiseOr = FLAG1 | FLAG2;
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`. You can take a look at the official Kotlin reference[19] for more information.

- Literals can give information about its type. It is not a requirement, but a common practice in Kotlin is to omit variable types (we will see it soon), so we can give some clues to the compiler to let it 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'
```

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

```kotlin
// 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. However,**immutability** is a fundamental concept in Kotlin (and many other modern languages).

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

Immutable objects are also thread-safe by definition. As the value is constant, you do not need to implement any particular synchronized mechanism, because all threads 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 are situations (especially in Android, where the constructor is called by the system for several classes) where immutability becomes hard to implement, but most of the time it will be possible.

Another thing mentioned before is that we can usually avoid specifying object types, the compiler infers them from the value, which makes the code cleaner and faster to modify. We already have some examples from the section above.

```kotlin
val s = "Example" // A String
val i = 23 // An Int
val actionBar = supportActionBar // An ActionBar in an Activity context
```

However, a type needs to be specified if we want to use a more generic type:

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

6.3 Properties

Properties are the equivalent of fields in Java, but much more powerful. Properties do the work of a field plus a getter plus a setter. Let's see an example to compare the difference. The following snippet shows the code required in Java to access and modify a field safely:

```
public class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

...

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

In Kotlin, you only need a property to achieve the same behavior:

```
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 behavior you need, without having to change the existing code:

```
class Person {

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

}
```

If the property needs access to its value in a custom getter or setter (as in this case), it requires the creation of a **backing field**. It can be accessed by using `field`, a reserved word, and it is automatically created when the compiler finds that it is required. Take into account that if we used the property directly, we would be using the setter and getter, and not doing a direct assignment. You can only use the backing field inside the accessors of the property.

As mentioned in some previous chapters, when dealing with code written in Java, Kotlin allows using the property syntax where a getter, and optionally a setter, are defined in Java. The compiler just links to the original getters and setters, so there are no performance penalties when using these mapped properties.

7 Anko and Extension Functions

The Kotlin team has developed some great tools to make Android development more accessible. In this chapter, I talk you about them and how you can start using them.

7.1 What is Anko?

Anko[20] is a robust library developed by JetBrains. Its primary purpose is the generation of UI layouts by using code instead of XML. This feature is interesting; try it out if you have the chance, but I will not be using it in this project. To me (probably due to years of experience writing user interfaces) using XML is more comfortable, but you might think differently.

However, this is not the only feature we can get from this library. Anko includes many helpful functions and properties that avoid lots of boilerplate. There are several examples covered in this book, but you will quickly see which kind of problems this library solves.

Since Anko is a library written specifically for Android, understanding what it does behind the scenes is a good exercise. You can navigate at any moment to Anko's source code using `ctrl + click` (Windows or Linux) or `cmd + click` (Mac). Anko's implementation is an excellent example to learn useful ways to get the most out of Kotlin language.

7.2 Start using Anko

Before going any further, let's use Anko to improve a couple of things. As you will see, anytime you use something from Anko, it includes an import with the name of the property or function to the file. Anko uses **extension functions** to add new

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

features to the Android framework, and the import links to them. We will see right below what an extension function is and how to write it.

In `MainActivity::onCreate`, an Anko extension function can be used instead of `findViewById` to find the `RecyclerView`. Since API 26, it is not necessary anymore, as `findViewById` is now generic, but it will serve us as an example:

activities/MainActivity.kt

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

Take a look at the imports, and see that the function `find` was added there:

```
import org.jetbrains.anko.find
```

This is all we can use from the library for now, but Anko can help us simplify, among others, the instantiation of intents, the navigation between activities, the creation of fragments, database access, or building alerts. We will find several showcase examples while we implement the app.

7.3 Extension functions

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

For instance, we can create a `toast` function that could be used by any `Context` objects and those whose type extends `Context`, such as `Activity` or `Service`:

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

Now you can use this extension inside an activity, for instance:

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

Of course, Anko already includes its own `toast` extension function, very similar to this one. The library provides functions for both `CharSequence` and resources, and different names for short and long toasts:

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

Extensions can also be properties. So you can create extension properties in a very similar way. Thanks to this code below, you could add an extra property to `ViewGroup` to get a list of its child views:

```
val ViewGroup.childViews: List<View>
    get() = (0 until childCount).map { getChildAt(it) }
```

Do not pay much attention to the implementation yet, as we still need to cover some of the concepts here.

Extension functions do not modify the original class. You may find a static import added to the file where you used it. You can declare extension functions anywhere, so a common practice is to create files that hold a set of related functions.

Extension functions are the magic behind many Anko features. From now on, you can create your magic too.

This chapter has little changes; you can find the source code here[21].

[21] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-7

8 Retrieving data from API

8.1 Performing a request

Our current placeholder texts are an excellent place to start getting an idea of what we want to achieve, but now it is time to request some real data, which will be used to populate the `RecyclerView`. We are going to make use of the OpenWeatherMap[22] API to retrieve data and some regular classes for the request. As Kotlin interoperability is extremely powerful, you could use any library you want, such as Retrofit[23], for server requests. However, as we are just performing a simple API request, we can achieve our goal much easier without adding another third-party library.

Besides, as you will see, Kotlin provides some extension functions that will make requests much simpler. First, we are going to create a new `Request` class:

data/Request.kt

```
class Request(private val url: String) {

    fun run() {
        val forecastJsonStr = URL(url).readText()
        Log.d(javaClass.simpleName, forecastJsonStr)
    }

}
```

The constructor receives a URL. Then the `run` function reads the result and outputs the JSON in the Logcat.

The implementation is straightforward when using `readText`, an extension function from the Kotlin standard library. This method is not recommended for large responses, but it is good enough in our case.

[22] http://openweathermap.org/
[23] https://github.com/square/retrofit

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

In order to be able to perform the request, the app must use the Internet permission. So it must be added to the `AndroidManifest.xml`:

AndroidManifest.xml

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

8.2 Performing the request out of the main thread

As you may know, HTTP requests are not allowed in the main thread, or the app will throw an exception. This limitation prevents from blocking the UI thread, which is a bad practice. The standard solution for Android is to use an `AsyncTask`. However, these classes are ugly and difficult to implement without any side effects. `AsyncTasks` are dangerous if not used carefully because by the time it reaches `postExecute`, the system could have destroyed the activity, and the task will crash.

Anko provides a very easy DSL to deal with asynchrony, which fits most basic needs. It provides a `doAsync` function that executes its code in another thread, with the option to return to the main thread by calling `uiThread`. Executing the request in a secondary thread is as easy as this:

activities/MainActivity.kt

```
val url = "http://api.openweathermap.org/data/2.5/forecast/daily?" +
            "APPID=15646a06818f61f7b8d7823ca833e1ce&zip=94043&mode=json&units=met\
ric&cnt=7"

doAsync() {
    Request(url).run()
    uiThread { longToast("Request performed") }
}
```

You can get the url from the repository branch[24] for this lesson.

A beautiful thing about `uiThread` is that it has a different implementation depending on the caller object. When an `Activity` uses it, the `uiThread` code is not executed if `activity.isFinishing()` returns `true`, and it will not crash if the activity is no longer valid.

You also can use your own executor:

```
val executor = Executors.newScheduledThreadPool(4)
doAsync(executor) {
    // Some task
}
```

`doAsync` returns a java `Future`, in case you want to work with futures. If you need it to return a `Future` with a result, you can use `doAsyncResult`.

It is simple and more straightforward to read than `AsyncTasks`. For now, I am just sending a static URL to the request, to test that we receive the content properly and that we can draw it in the activity. I will cover the JSON parsing and conversion to app data classes soon, but before we continue, it is essential to learn what a data class is.

You can run the app and check that you can see the JSON in the log and the toast when the request finishes.

[24]https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-8

9 Data Classes

Data classes are a powerful kind of classes that avoid the boilerplate we need in Java to create POJO: classes which are used to keep state but are elementary in the operations they do. They usually only provide plain getters and setters to access to their fields. Defining a new data class is very easy:

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

9.1 Extra functions

Along with a data class, we get a handful of new functions for free, apart from the properties we already talked about (which prevent us from writing the accessors):

- `equals()`: it compares the properties from 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 will see an example later.
- A set of numbered functions that are used to map an object into variables. I will also explain this soon.

9.2 Copying a data class

If we use immutability, as talked some chapters ago, we 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 slightly repetitive and far from clean. However, data classes include the `copy()` method, which makes the process easy and intuitive.

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

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

This way, we copy the first forecast and modify only the `temperature` property 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 were not designed with this in mind, and there are still some circumstances where you are able to modify the state. In the previous example, you could always access the `Date` object and change its value. The easy (and unsafe) option is to remember the rules of not modifying the state of an object, but copying it when necessary.

Another option is to wrap these classes. You could create an `ImmutableDate` class which wraps a `Date` and does not allow to modify its state. It is up to you to decide which solution you take. In this book, I am not very strict with immutability. So I will not create wrappers for every potentially dangerous class.

9.3 Mapping an object into variables

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

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

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

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

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

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

10 Parsing data

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 in a browser the URL we used in chapter 8, you can see the structure of the JSON file. It 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 where it belongs. 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.

10.1 Converting JSON to data classes

In our current UI, we are not going to use all this data. However, we will parse everything down to classes, in case it is of some use in the future. These are the data classes we need:

data/ResponseClasses.kt

```kotlin
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[25] to parse the JSON to our classes, the properties must have the same name as the ones in the JSON or specify a serialized 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 will convert them before being used in the rest of the app. The names of the properties used in this class are the same as the names in the JSON response.

Now, the `Request` class needs some modifications to return the parsed result. It also receives only the `zipcode` of the city instead of the complete URL so that it becomes more readable. For now, the static URL is declared in a **companion object**.

Companion objects

Kotlin allows declaring objects to define static behaviors. In Kotlin, we cannot 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.

If we need some static properties, constants or functions in a class, we can use a **companion object**. This object is shared among all instances of the class, the same as a static field or method would do in Java. Use the reserved word `const` for your compile-time constants. They are more efficient and friendly to use from Java code.

Check the resulting code:

[25] https://github.com/google/gson

data/ForecastRequest.kt

```
class ForecastRequest(private val zipCode: String) {

    companion object {
        private const val APP_ID = "15646a06818f61f7b8d7823ca833e1ce"
        private const val URL = "http://api.openweathermap.org/data/2.5/" +
                "forecast/daily?mode=json&units=metric&cnt=7"
        private const val COMPLETE_URL = "$URL&APPID=$APP_ID&zip="
    }

    fun execute(): ForecastResult {
        val forecastJsonStr = URL(COMPLETE_URL + zipCode).readText()
        return Gson().fromJson(forecastJsonStr, ForecastResult::class.java)
    }
}
```

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

app/build.gradle

```
compile "com.google.code.gson:gson:2.8.1"
```

10.2 Shaping the domain layer

Now let's create a new package representing the `domain` layer. It contains some `Commands` in charge of performing the use cases of the app.

Firstly, a definition of a `Command` is required. There is a generic declaration in the following example. We will talk about this topic in next chapters.

domain/commands/Command.kt

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

These commands execute an operation and return an object of the class specified in its generic type. Remember that **every function in Kotlin returns a value**. By default, if nothing is specified, it returns 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 potent than Java (before Java 8) because they can contain code. However, for now, we do not need that feature.

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

domain/model/DomainClasses.kt

```kotlin
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 we add more features. For now, the data they keep is enough.

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

domain/mappers/ForecastDataMapper.kt

```kotlin
class ForecastDataMapper {

    fun convertFromDataModel(forecast: ForecastResult): ForecastList =
        ForecastList(forecast.city.name, forecast.city.country,
            convertForecastListToDomain(forecast.list))

    private fun convertForecastListToDomain(list: List<Forecast>)
            : List<ModelForecast> {
        return list.mapIndexed { i, forecast ->
            val dt = Calendar.getInstance().timeInMillis +
                TimeUnit.DAYS.toMillis(i.toLong())
            convertForecastItemToDomain(forecast.copy(dt = dt))
        }
    }

    private fun convertForecastItemToDomain(forecast: Forecast): ModelForecast {
        return ModelForecast(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)
    }
}
```

As we are using two classes with the same name, give a specific name to one of them so that we can avoid writing the complete package:

```
import com.antonioleiva.weatherapp.domain.model.Forecast as ModelForecast
```

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

```
return list.mapIndexed { i, forecast -> ... }
```

We can loop over the collection that easily 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. Collection operations are one of the most powerful features in Kotlin for developers used to Java 7. We shall take a look at all the different transformations very soon. It is important to know they exist because it makes it easier to find places where these functions can save much time and boilerplate.

You may have noticed too that I am not using the date, but generating a new one. That is because the request returns a date that is difficult to convert correctly to current date while not having issues with timezone. As we know that the first day we receive is today, we can iterate and build the proper dates this way.

And now, everything is ready to write the command:

domain/commands/RequestForecastCommand.kt

```kotlin
class RequestForecastCommand(private val zipCode: String) :
        Command<ForecastList> {
    override fun execute(): ForecastList {
        val forecastRequest = ForecastRequest(zipCode)
        return ForecastDataMapper().convertFromDataModel(
                forecastRequest.execute())
    }
}
```

10.3 Drawing the data in the UI

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

ui/activities/MainActivity.kt

```kotlin
doAsync() {
    val result = RequestForecastCommand("94043").execute()
    uiThread {
        forecastList.adapter = ForecastListAdapter(result)
    }
}
```

The adapter requires some modifications too:

ui/adapters/ForecastListAdapter.kt

```kotlin
class ForecastListAdapter(private  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 receives an object and an extension function as parameters and makes the object execute the function. That way, all the code that we define inside the brackets acts as an extension function for the object provided in the first parameter, and we can use all its public functions and properties, as well as `this`. The `with` function is helpful to simplify code when we do several operations over the same object.

There is some new code in this chapter, so feel free to check it out on the repository[26].

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

11 Operator overloading

Kotlin has a fixed number of symbolic operators we can use in any class. These symbols map to a function, which is the one that provides the logic that the operator uses. Overloading these operators may increment the readability and simplicity of your code.

To notify the compiler that you want to overload an operator, you must annotate the functions with the `operator` modifier.

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.unaryPlus()
-a	a.unaryMinus()
!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 === null
a != b	!(a?.equals(b) ?: b === null)

The `equals` operations are a bit different, because they use a more complex translation 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:

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

Operators === and `!==` do identity checks (they are == and `!=` in Java respectively) and cannot be overloaded.

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 An example

As you can imagine, Kotlin lists have the array-like operations implemented in the standard library, so we can access the items of the list the same way we would do in Java arrays. But it goes beyond: in mutable lists, you can also assign a value to a position by using the same operator:

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

If you remember, we have a data class called `ForecastList`, which consists of a list with some extra info. It would be interesting to access its items directly instead of having to request its internal list to get an item. On an unrelated note, I am also going to implement a `size` property, which simplifies the current adapter a little more:

domain/model/DomainClasses.kt

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

    val size: Int
        get() = dailyForecast.size

    operator fun get(position: Int): Forecast = dailyForecast[position]
}
```

It makes our `onBindViewHolder` a bit simpler:

ui/adapters/ForecastListAdapter.kt

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

As well as the `getItemCount()` function:

ui/adapters/ForecastListAdapter.kt

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

11.3 Operators in extension functions

Apart from using extension functions for our classes, we can also extend existing classes to provide new operations to third-party libraries. For instance, we could access to `ViewGroup` views the same way we do with lists:

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

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

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

Take a look at the branch for this chapter[27], and then you can go on with the next one.

[27] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-11

12 Making the forecast list clickable

Current items layout 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`:

layout/item_forecast.xml

```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 generate the complete icon url, so that we can load it:

model/DomainClasses.kt

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

In `ForecastDataMapper`:

domain/mappers/ForecastDataMapper.kt

```
private fun convertForecastItemToDomain(forecast: Forecast): ModelForecast {
    return ModelForecast(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 by making use of an image loader library. Picasso[28] is an excellent option. You must add it to `build.gradle` dependencies:

app/build.gradle

```
compile "com.squareup.picasso:picasso:2.5.2"
```

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

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

ui/adapters/ForecastListAdapter.kt

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

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

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

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

ui/adapters/ForecastListAdapter.kt

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

    private val iconView = view.find<ImageView>(R.id.icon)
    private val dateView = view.find<TextView>(R.id.date)
    private val descriptionView =
            view.find<TextView>(R.id.description)
    private val maxTemperatureView =
            view.find<TextView>(R.id.maxTemperature)
    private val minTemperatureView =
            view.find<TextView>(R.id.minTemperature)

    fun bindForecast(forecast: Forecast) {
        with(forecast) {
            Picasso.with(itemView.ctx).load(iconUrl).into(iconView)
            dateView.text = date
            descriptionView.text = description
            maxTemperatureView.text = "$high"
            minTemperatureView.text = "$low"
            itemView.setOnClickListener { itemClick(this) }
        }
    }
}
```

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

ui/adapters/ForecastListAdapter.kt

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

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
            ViewHolder {
        val view = LayoutInflater.from(parent.ctx)
            .inflate(R.layout.item_forecast, parent, false)

        return ViewHolder(view, itemClick)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bindForecast(weekForecast[position])
    }
    ...
}
```

If you use this code, `parent.ctx` will not compile. Anko provides many 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 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:

```
val View.ctx: Context
    get() = context
```

From now on, any view can make use of it. It is not necessary at all, because you can use `context` synthetic property, but I think it gives some consistency if we are planning to use `ctx` in the other classes. Besides, it is another example of how to use extension properties.

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

ui/activities/MainActivity.kt

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

As you can see, to implement an anonymous class we, in fact, create an `object` that implements the interface we created. That is because we are not making use of the power of functional programming, but you will learn how to convert this code into something cleaner in the next chapter.

Try the new changes from the repository[29].

[29] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-12

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, lambdas are first class citizens, which means that a function behaves as a type: it can be passed as an argument to another function, can be returned by a function, saved into a variable or a property, etc.

13.1 Simplifying setOnClickListener()

Let's see how this works using a typical example in Android: the click listener method from a `View`. If we want to implement a click listener behavior in Java, we first need to write the `OnClickListener` interface:

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

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

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

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

```
view.setOnClickListener(object : OnClickListener {
    override fun onClick(v: View) {
        toast("Click")
    }
})
```

Luckily, Kotlin allows some optimizations over Java libraries, and lambdas can substitute any function that receives an interface with a single function. It works as if we had defined `setOnclickListener()` like this:

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

To define a lambda expression, specify the function input arguments to the left of the arrow (surrounded by parentheses), and the return type 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 braces and specify the argument values to the left of the arrow and the body of the function to the right. We can even get rid of the left part if the input values are not used:

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

If the last argument of a function is also a 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.

13.2 Click listener for ForecastListAdapter

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

ui/adapters/ForecastListAdapter.kt

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

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

ui/adapters/ForecastListAdapter.kt

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

The rest of the code remains unmodified. Just a last change to `MainActivity`:

ui/activities/MainActivity.kt

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

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

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

13.3 Extending the language

Thanks to these transformations, we can create our builders and code blocks. We have already been using some interesting functions such as `with`. Though the one included in the standard library is a little more complex, 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 lambda that behaves as an extension function. The implementation 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 = "$high"
    minTemperatureView.text = "$low"
    itemView.setOnClickListener { itemClick(this) }
}
```

Inline functions

Inline functions are a bit different from regular functions. An inline function is substituted by its code during compilation, instead of doing the real call to a function. That reduces memory allocations and runtime overhead in some situations. For instance, if we have a function as an argument, a regular function internally creates an object that contains that function. On the other hand, inline functions substitute the code of the function in the place where it is called, so it does not require an internal object for that.

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

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

The body of the function checks the version and executes the code if it meets the requirements. Now we could do:

```
supportsLollipop {
    window.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, a showcase of a DSL to generate HTML[30] from code.

The changes from this chapter are in its corresponding branch[31].

[30]http://kotlinlang.org/docs/reference/type-safe-builders.html
[31]https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-13

14 Visibility Modifiers

Modifiers are a bit different in Kotlin from how we use them in Java. The default modifier in this language is `public`, which saves a lot of time and characters. Here it is the long explanation: how do visibility modifiers work in Kotlin?

14.1 Modifiers

private

The `private` modifier is the most restrictive we can use. It indicates it is visible in its file. So if we declare a class as `private`, we are not able to use it outside the file where it was defined.

On the other hand, if we use `private` inside a class, the access is restricted to that class. Even classes that extend it cannot use it.

So first level classes, objects, interfaces… (known as package members) declared as `private` are only visible inside the file where they are declared, while everything defined inside a class or interface is only visible by that class or interface.

protected

This modifier only applies to members of 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

An internal member is visible inside the whole module if it is a package member. If it is a member of another scope, it depends on the visibility of the scope. For instance, if we write a private class, the access to an `internal` function is limited to the visibility of the class.

We can use internal classes from any other class in the same module, but not from another module.

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 refers to the Android Studio modules we can create to split our project into different blocks. In Eclipse, these modules would refer to the projects inside a workspace.

public

As you may guess, this is the less restrictive modifier. ** It is the default modifier**, and a member declared as `public` is visible anywhere, only restricted by its scope. A public member defined in a private class will not be visible outside the scope where the class is visible.

14.2 Constructors

By default, all constructors are `public`, which means they can be used from any scope where their class is visible. If we want to restrict the visibility of a constructor, we can apply modifiers to it with this specific syntax:

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

14.3 Reviewing our code

We already made use of the `public` default modifier, and latest versions of the Kotlin plugin also detect when an argument can be private. For instance, in `RequestForecastCommand`, we made the `zipCode` property private.

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

Besides, in Kotlin we do not need to specify the return type of a function if the compiler can infer it. An example of how we can get rid of the returning types:

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

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 pretty straightforward. You can review them in the repository[32].

[32]https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-14

15 Kotlin Android Extensions

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

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

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

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

15.1 How to use Kotlin Android Extensions

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

app/build.gradle

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

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

Recovering views from the XML

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

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

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

</FrameLayout>
```

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

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

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

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

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

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

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

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

The magic behind Kotlin Android Extensions

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

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

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

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

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

The interesting part is this one:

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

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

   return var2;
}

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

}
```

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

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

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

is converted into this:

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

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

Kotlin Android Extensions on fragments

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

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

```
class Fragment : Fragment() {

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

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

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

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

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

Kotlin Android extensions on a Custom View

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

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

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

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

</merge>
```

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

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

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

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

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

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

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

Recovering views from another view

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

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

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

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

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

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

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

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

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

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

15.2 Kotlin Android Extensions in 1.1.4

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

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

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

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

View cache on a ViewHolder (or any custom classes)

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

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

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

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

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

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

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

Kotlin Android Extensions to implement Parcelable

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

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

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

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

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

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

Customize the cache build

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

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

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

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

Currently, the existing options are these:

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

    ...
}
```

Now you know all the features that the Kotlin Android Extensions plugin provides. Now, let's use some of these features in our code.

15.3 Refactoring our code

The modifications we are doing to our sample app to start using Kotlin Android Extensions are fairly simple.

Let's start with `MainActivity`. We are currently only using a `forecast_list` view, which is an instance of a `RecyclerView`. Let's clean up this code.

As said before, we use the id to access the views, so I am changing the id of the `RecyclerView` so that it does not use underscores, but a more appropriate name for a Kotlin variable. The XML now looks like this:

layout/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/forecastList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</FrameLayout>
```

And now we can just get rid of the `find` line. Start writing the id of the view, and the autocomplete should help you add the import. Otherwise, you will need to add this:

ui/activities/MainActivity.kt

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

This is the resulting code:

ui/activities/MainActivity.kt

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

    forecastList.layoutManager = LinearLayoutManager(this)
    ...
}
```

There is not much improvement because this layout only had one view. 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 to remove all the `find` code inside the `ViewHolder`.

This is the synthetic import you will need for `item_forecast`:

ui/adapters/ForecastListAdapter.kt

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

Now we can find the views from `itemView` property inside the `ViewHolder`. As I mentioned before, you can use those properties over any other views, but it will crash if the view does not contain the requested sub-views.

There is no need to declare properties for the views anymore, you can use them directly:

ui/adapters/ForecastListAdapter.kt

```kotlin
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 = "$high"
            itemView.minTemperature.text = "$low"
```

```
                itemView.setOnItemClickListener { itemClick(this) }
            }
        }
}
```

Our layout is pretty simple, and it will not affect performance. In any case, remember what we talked in the section above: this is not caching the views, which implies that each call to the bind function is doing `findViewById` under the hood for all its views.

To solve it, you can use the new features in Kotlin 1.1.4. First, enable the *experimental* flag in the `build.gradle`:

app/build.gradle

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

Now update the `ViewHolder`. You will not need to access the views through the `itemView` anymore, as now the views behave as `ViewHolder` properties. Check that the import is different too:

ui/adapters/ForecastListAdapter.kt

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

class ForecastListAdapter(private val weekForecast: ForecastList,
        private val itemClick: (Forecast) -> Unit) :
        RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {

    ...

    class ViewHolder(override val containerView: View,
         private val itemClick: (Forecast) -> Unit)
        : RecyclerView.ViewHolder(containerView), LayoutContainer {

        fun bindForecast(forecast: Forecast) {
            with(forecast) {
                Picasso.with(itemView.ctx).load(iconUrl).into(icon)
                dateText.text = date
                descriptionText.text = description
                maxTemperature.text = "${high}º"
                minTemperature.text = "${low}º"
                itemView.setOnClickListener { itemClick(this) }
            }
        }
    }
}
```

I did a change in the XML too, so that the names of the views are different from the properties in `Forecast`. Otherwise you will have issues to refer to them:

layout/item_forecast.xml

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

Our app is now using Kotlin Android Extensions to find the views. Run it and check that everything is working as expected.

Kotlin Android Extensions plugin helps us reduce some more boilerplate and minimize the code required to access our views. Check the latest changes in the repository[33].

[33] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-15

16 Application Singleton and Delegated Properties

We are implementing a database soon and if we want to keep our code simple and our app in independent layers (instead of everything added to our activity), we need to have easier access to the application context.

16.1 Application Singleton

The simplest way is to create a singleton the way we would do in Java:

ui/App.kt

```kotlin
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:

AndroidManifest.xml

```xml
<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 do not have control over many class constructors. For instance, we cannot initialize a non-nullable property, because its value must be declared 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 the first code that executes is application `onCreate`, so we are safe by assuming `instance()` function can always return a non-nullable `App` instance.

However, this solution seems strange. 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? Yes, we can delegate the value of a property to another class.

16.2 Delegated Properties

There are some typical behaviors 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 that a property needs to another class. These members are known as **delegated properties**.

When we use `get` or `set` from a property, the delegate calls to its corresponding `getValue` and `setValue`, which can decide if, when and how the value is set or returned.

The structure of a property delegate is:

```kotlin
class Delegate<T> {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return ...
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        ...
    }
}
```

The `T` is the type of the property that is delegating its behavior. The `getValue` function receives a reference to the class and the metadata of the property. The `setValue` function also receives the assigned value. If the property is immutable (`val`), it only requires the `getValue` function.

The snippet below is an example of how the property delegate is assigned:

```kotlin
class Example {
  var p: String by Delegate()
}
```

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

16.3 Standard Delegates

You can find a set of delegates included in the Kotlin standard library. These are the most common circumstances where a delegate may be useful, but we could also create our own.

Lazy

It takes a lambda that runs after the first call to `getValue`, so the initialization of the property is delayed up to that moment. Subsequent calls return the same value. This delegate is very interesting for things that are not always used or require some other parts to be ready before this one can be declared. We can save memory and skip the initialization until the property is required.

```
class App : Application() {
    val database: SQLiteOpenHelper by lazy {
        MyDatabaseHelper(applicationContext)
    }

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

In this example, the database does not initialize until it is called first time in `onCreate`. At that moment, we are sure the application context exists and is ready to be used. The `lazy` operation is thread-safe.

You can also use `lazy(LazyThreadSafetyMode.NONE) { ... }` if you are not worried about multi-threading and want to get some extra performance.

Observable

This delegate helps us detect changes on any property we need to observe. It executes the declared lambda expression every time the `set` function is called. So after the new value is assigned, we receive the delegated property, the old value, and the new one.

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

    var myProperty by Delegates.observable("") {
        _, _, new ->
        db.saveChanges(this, new)
    }

}
```

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

Underscores for unused parameters

Since Kotlin 1.1, you can use underscores to avoid giving a name to a lambda argument that you are not using. This simplification helps both the compiler and the reader. The compiler can prevent saving memory for the ignored input arguments, and the reader does not need to parse which parameters are being used and discard the ones that are not.

Vetoable

This delegate is similar to observable, with the difference that it runs before the value is assigned, and 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) {
    _, _, new ->
    new >= 0
}
```

The former delegate only allows saving the new value if it is a positive number. Inside lambdas, the latest line represents the return value. You cannot use the `return` word because it will not compile.

lateinit

Sometimes we need something else to initialize a property, but we lack the necessary state available in the constructor, or we are even not able to retrieve it at that point. This second case happens now and then in Android: in activities, fragments, services or broadcast receivers for example. However, a non-abstract property needs a value before the constructor finishes executing. We cannot just wait until we want 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. With this solution, we then need to check everywhere throughout the code whether the property is null or not. If we are sure that the property will not be null the first time we use it, this may make us write some unnecessary code.

The second alternative is **`lateinit`**, which can be used to identify that the property should have a non-nullable value, but we are delaying the assignment. If the value is requested before it is assigned, it will throw an exception that identifies the accessed property.

`lateinit` is not exactly a delegate, but a property modifier, and that is why it must be written before the property.

This modifier is helpful in the App singleton example:

```
class App : Application() {

    companion object {
        lateinit var instance: App
    }

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

`lateinit` is also indispensable when working with a dependency injector such as Dagger, and very useful for tests too.

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 lets create an instance of an object from a dynamic map. If we are using immutable properties, the map can be immutable too. For mutable properties, the class requires a `MutableMap` as constructor parameter instead.

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

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

As a reference, here it is how we could create the map that this class would require:

```kotlin
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 non-nullable delegate that can be only assigned once. The second time it is assigned, it will throw an exception.

To define a delegate class, you only need to write a couple of operator functions: `getValue` and `setValue`. They have a very specific set of arguments that look like this:

ui/utils/DelegatesExt.kt

```kotlin
class NotNullSingleValueVar<T> {

    operator fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return ...
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        ...
    }
}
```

This delegate can work with any non-nullable type. It receives 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 returns a value if it is assigned; otherwise, it throws an exception.
- The setter assigns the value if it is still null; otherwise, it throws an exception.

ui/utils/DelegatesExt.kt

```kotlin
class NotNullSingleValueVar<T> {

    private var value: T? = null

    operator fun getValue(thisRef: Any?, property: KProperty<*>): T =
        value ?: throw IllegalStateException("${property.name} " +
                "not initialized")
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        this.value = if (this.value == null) value
        else throw IllegalStateException("${property.name} already initialized")
    }
}
```

Now let's create an object with a function that provides your new delegate:

ui/utils/DelegatesExt.kt

```kotlin
object DelegatesExt {
    fun <T> notNullSingleValue() = NotNullSingleValueVar<T>()
}
```

This last part is not necessary, but it can help you aggregate all the delegates you implement.

16.5 Reimplementing the App Singleton

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

ui/App.kt

```kotlin
class App : Application() {

    companion object {
        lateinit var instance: App
    }

    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 `lateinit`. That is easy to solve by using a private `set`:

```kotlin
companion object {
    lateinit var instance: App
        private set
}
```

Now, the value of the instance can only be modified inside the `App` class.

But we will make use of our custom delegate instead:

ui/App.kt

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

Though, in this case, `lateinit` is probably the simplest option, I wanted to show you how to create a custom property delegate and use it in your code.

Take a look at the specific branch of this chapter[34] if you have any doubts.

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

17 Creating an SQLiteOpenHelper

As you may know, Android uses SQLite as a database management system. SQLite is a database embedded into the app, and it is lightweight. That is why it is a good option for mobile apps.

However, the API to work with databases in Android is somewhat raw. You will see you need to write many SQL sentences and map 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 in Kotlin thanks to its interoperability. However, it is possible you can avoid using complicated frameworks if the database is simple, as we are going to see in a minute.

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 should not forget to call `close()`. With a `ManagedSqliteOpenHelper` we just do:

```
forecastDbHelper.use {
...
}
```

Inside the lambda, we can use `SqliteDatabase` functions directly. How does it work? Take a look at the implementation of Anko functions; you can learn a good deal of Kotlin from them:

```
public fun <T> use(f: SQLiteDatabase.() -> T): T {
    try {
        return openDatabase().f()
    } finally {
        closeDatabase()
    }
}
```

First, `use` receives a function and uses it as an extension function for `SQLiteDatabase`. Thanks to extensions, we can use `this` inside the braces, and this is 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
}
```

Keep in mind that, inside a function, the last line represents the returned value. As `T` does not have any restrictions, we can return any value. Even `Unit` if we do not 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 handy extension functions over `SqliteDatabase` that we will use later. For now, let's define the database tables and implement the `SqliteOpenHelper`.

17.2 Tables definition

Let's create a couple of objects that represent our tables. These objects will be helpful to avoid misspelling table or column names and repetition. We need two tables: one that saves the info of the city, and the other one the forecast for individual days. This second table has a foreign key to the first one.

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

data/db/Tables.kt

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

`DayForecast` has some more info, so it requires the set of columns you can see below. The last column, `cityId`, keeps the id of the `CityForecast` where this forecast belongs.

data/db/Tables.kt

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

17.3 Implementing SqliteOpenHelper

If you remember, Anko is divided into several libraries to be more lightweight. We already added `anko-common`, but we also need `anko-sqlite` if we want to use database features:

app/build.gradle

```
dependencies {
    ...
    compile "org.jetbrains.anko:anko-sqlite:$anko_version"
}
```

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:

data/db/ForecastDbHelper.kt

```
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. Define these values in the companion object, along with the helper instance:

data/db/ForecastDbHelper.kt

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

The `instance` property uses a `lazy` delegate, which means the object is not created until its first use. That way, if the database is never employed, we avoid creating unnecessary objects. The regular `lazy` delegate is blocking in order to prevent the creation of several instances from different threads. This situation can only 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.

In order to create the database schema, we are required to provide an implementation of the onCreate function. When no libraries are used, tables creation 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 that receives the name of the table and a set of Pair objects that identify the name and the type of the column:

data/db/ForecastDbHelper.kt

```
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, checks if the table does not exist before trying to create it.
- The third parameter is a vararg of Pairs. The vararg type also exists in Java, and it is a way to pass a variable number of arguments of the same type to a function. The function then receives an array with the objects.

Column types are using a special Anko class called SqlType, which can be used together with SqlTypeModifiers, such as PRIMARY_KEY. The + operation is overloaded the same way we saw in chapter 11. This plus function properly concatenates both values returning a new special SqlType:

```
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.

Coming back to our code, we can do it better. Kotlin standard library includes a function called to which, once more, shows the ability of Kotlin to let us model the language to our needs. It acts as an extension function for the first object and receives another object as a parameter, returning a Pair object with them.

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

Functions with one parameter that use the `infix` modifier can be used inline, so 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 complete method looks:

data/db/ForecastDbHelper.kt

```
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` just deletes the tables so that they are recreated. We are using our database just as a cache, so it is the easiest and safest way to be sure the tables are rebuilt as expected. If we had stored persistent data, we would 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 am trying not to add much complexity to the code regarding architectures, clean testable code or good practices, I thought it would 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[35]. The first article covers a light explanation of these terms.

To put it simply, if we want to have classes that are independent one from another, way more testable, and write code that is easy 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.

However, now those dependencies must be provided from somewhere, so the dependency injection consists of providing the collaborators required by the classes. Developers usually delegate this task to a dependency injector. Dagger[36] is probably the most famous dependency injector for Android. It is, of course, an excellent alternative when we need some complexity to provide those dependencies.

But a simpler alternative is to make use of default values in constructors. We can provide the dependency by assigning a default value to constructor arguments, 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:

[35]http://antonioleiva.com/dependency-injection-android-dagger-part-1/
[36]https://google.github.io/dagger/

```kotlin
class ForecastDbHelper(ctx: Context = App.instance) :
    ManagedSQLiteOpenHelper(ctx, ForecastDbHelper.DB_NAME, null,
    ForecastDbHelper.DB_VERSION) {
    ...
}
```

Now we have two ways to create this class:

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

I will be using this mechanism here and there, so I did not want to continue without explaining the motivation. We already have created the tables, so it is time to start adding and requesting data from them. But before that, I want to talk about collections and functional operations. Remember to check the repository[37] to find the latest changes.

[37] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-17

18 Collections and functional operations

We have been using collections before in this project, but now it is 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 simply 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 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 meant 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 popular collection type. It represents a generic ordered collection of elements. As it is 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 ignores 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.

These collections can apply a set of functional operations. I want to show you a little definition and an example of several operations. It is useful to know what the options are. That way it is 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 does not 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`.

```
val listWithNull = listOf(1, null, 2, 3, null, 4)
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(it, 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.

```
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.

```
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.

```
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.

```
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.

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

first

Returns the first element matching the given predicate. It will throw a `NoSuchElementException` if no elements are found.

```
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 the element, or -1 if the collection does not contain the 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. It throws a `NoSuchElementException` if no elements are found.

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

lastIndexOf

Returns the last index of the element, or -1 if the collection does not contain the 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 an 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 the 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 the 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 a 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)))
```

unzip

Generates a `Pair` of `Lists` from a `List` of `Pairs`

```
assertEquals(Pair(listOf(5, 6), listOf(7, 8)), listOf(Pair(5, 7), Pair(6, 8)).unzip())
```

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 the database

A previous chapter covered the creation of an `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`, is making use of it.

19.1 Creating database model classes

First, we are going to create the model classes for the database. Do you remember the map delegates we saw? We are using them to map those fields directly to the database and vice-versa.

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

data/db/DbClasses.kt

```
class CityForecast(val map: MutableMap<String, Any?>,
                   val dailyForecast: List<DayForecast>) {
    var _id: Long by map
    var city: String by map
    var country: String by map

    constructor(id: Long, city: String, country: String,
                dailyForecast: List<DayForecast>)
    : this(HashMap(), dailyForecast) {
        this._id = id
        this.city = city
        this.country = country
    }
}
```

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

Besides, a second constructor is necessary. This is because we are mapping classes from the domain back to the database. So instead of using a map, extracting the values from the properties is 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 have our map ready to be added to the database. After writing some extra code, you will see that it works like magic.

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

data/db/DbClasses.kt

```kotlin
class DayForecast(var map: MutableMap<String, Any?>) {
    var _id: Long by map
    var date: Long by map
    var description: String by map
    var high: Int by map
    var low: Int by map
    var iconUrl: String by map
    var cityId: Long by 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 helps 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 are using it in a new class, to request data already saved in the database, and to save fresh data. The definition of the class is using a `ForecastDbHelper` and a `DataMapper` that converts classes from database to domain models. I am still using default values as an easy way of dependency injection:

data/db/ForecastDb.kt

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

Both functions will call `use()`, the 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:

data/db/ForecastDb.kt

```
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 a simple request builder, so let's take advantage of it:

data/db/ForecastDb.kt

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

val dailyForecast = select(DayForecastTable.NAME)
        .whereSimple(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 `whereSimple` function needs, and it is very similar to what we would do in regular use of the helper. There is another function called simply `where`, which takes some tags and values and match them. I do not find it very useful because I think it adds more boilerplate, though it has the advantage of parsing the values to the `Strings` we need. The following snippet shows how it would look with it:

data/db/ForecastDb.kt

```
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 cannot directly use the simplification that prevents from the need of creating an object explicitly. T This conflict can be solved by implementing an extension function. I am creating a function that receives a lambda and returns a `MapRowParser`. The parser will use that lambda to create the object:

extensions/DatabaseExtensions.kt

```kotlin
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)
        })
```

If you do not fully understand it yet, you can come back once we dive deep into generics. The previous function helps simplify the `parseList` request to:

```kotlin
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`. This `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 are no results for the request, `parseList` returns an empty list.

Next step will request the city in a similar way:

data/db/ForecastDb.kt

```kotlin
val city = select(CityForecastTable.NAME)
        .whereSimple("${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 does not find a row in the database, it throws an exception. In our case, the first time we query a city, it will not be there, so using `parseOpt` is safer. I also created a handy function to prevent the need for object creation:

extensions/DatabaseExtensions.kt

```
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 `null`. As you may remember, last line inside a lambda represents what the lambda returns. So it will return an object from the type `CityForecast?`:

data/db/ForecastDb.kt

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

`DataMapper` function is easy:

data/db/DbDataMapper.kt

```
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:

data/db/ForecastDb.kt

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

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

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

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

Another interesting functionality from Anko I am 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 do not need reflection and have more control over the transformations, but it can be of use for you at some point.

Saving a forecast

The `saveForecast` function 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 do not need a result, so it will return `Unit`.

data/db/ForecastDb.kt

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

First, we clear both tables. Anko does not provide any beautiful way to do it, so we are creating an extension function for `SQLiteDatabase` that executes the proper SQL query for us:

extensions/DatabaseExtensions.kt

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

The function is applied to both tables:

data/db/ForecastDb.kt

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

Now it is time to convert the data, and use the result to execute the `insert` queries. The `with` function may help us:

data/db/ForecastDb.kt

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

The conversion from the domain model is straightforward too:

data/db/DbDataMapper.kt

```kotlin
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 `Map` to do that:

extensions/CollectionExtensions.kt

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

It works over a `Map` with nullable values (this was a condition from the map delegate) and converts it to an `Array` with non-nullable values (`select` function requisite) of pairs. Another complex one you do not need to understand yet. I will be covering nullity soon.

So, with this new function we can do:

data/db/ForecastDb.kt

```
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 is decomposed to 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:

data/db/ForecastDb.kt

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

So, with the use of maps, we have been able to convert classes to database registers and vice-versa in an effortless way. Once we have these extension functions ready, we can use them for other projects, so it is a well-paid effort.

The complete code of this function:

data/db/ForecastDb.kt

```
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())
        }
    }
}
```

Much new code was involved in this chapter, so you can take a look at the repository[38] to review it.

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

20 Null safety in Kotlin

Null-safety is one of the most exciting features in Kotlin when you come from Java 7. However, as you have seen during this book, it is so implicit in the language we hardly had to worry about it until the previous chapter.

Being `null` considered the billion-dollar mistake by its creator[39], it is 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:

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

This code compiles correctly (you may get a warning from the IDE), and when it runs, it naturally throws a `NullPointerException`. This is unsafe, and even if you think that you could be able to have everything under control, as the code grows you will 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. Kotlin, the same as other similar languages such as Swift, **make use of question marks to identify nullable types**. That way, if a variable can be null, the compiler forces to deal with it in some way.

As everything is an object in Kotlin (even Java primitive types), everything can be null. So, of course, we can have a nullable integer:

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

You cannot work directly with a nullable type without doing some checks before. This code does not compile:

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

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

That variable could be null and **the compiler is aware of that**, so until the nullity is checked, you cannot 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 cast to its non-nullable type. Let's see an example:

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

Inside the `if`, `a` becomes `Int` instead of `Int?`, so we can use it without checking nullity anymore or doing any castings. The code outside the `if` context, of course, has to deal with it.

This smart cast only works if a variable cannot be concurrently modified because otherwise the value could have been changed from another thread, and the previous check would be false at that moment. The compiler supports smart casting for `val` properties or local (`val` or `var`) variables.

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

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

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

```
val a: Int? = null
...
val myLong = a?.toLong() ?: 0L
```

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

```
val myLong = a?.toLong() ?: return false
```

```
val myLong = a?.toLong() ?: throw IllegalStateException()
```

However, there may 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!!.toLong()
```

The previous code compiles, but it will crash. So we must make sure we only use it in a reduced number of situations. Usually, we can choose alternative solutions. A code full of !! is a smell, because it brings back the possible exceptions we are trying to avoid with nullable types.

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. This means 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 will probably see that some arguments are marked with a single '`!`' when a Java method is used. For instance, something that gets an `Object` in Java is represented as `Any!` in Kotlin. This special operator represents that it is up to the developer to decide whether that variable should be null or not.

Luckily, latest versions of the Android framework and the support library are starting using the `@Nullable` and `@NonNull` annotations to identify the parameters that can be null or the functions that can return null, and the Kotlin compiler can detect that and choose the appropriate translation into Kotlin language.

Said that, if we are for instance overriding `onCreate` for an `Activity`, we need to mark `savedInstanceState` as nullable:

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

Otherwise it will show an error. We cannot use this implementation:

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

This is great because an activity can receive a null bundle, and we get the right implementation for free. However, there are parts of the SDK that are missing annotations yet. So when in doubt, you can just use a nullable object and deal with the possible `null`. Remember, if you use `!!` it is because you are sure that the object cannot be null, so you can declare it as non-nullable directly.

21 Creating the business logic to data access

After implementing the access to the server and a way to interact with the database, it is 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 it finds the required data, it is returned so that the UI can render it
4. Otherwise, it requests the data to the server
5. The result is saved in the database and returned so that the UI can render it

Our commands should not need to deal with all this logic. The source of the data is an implementation detail that could easily require a logic modification, 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 it finds a proper result.

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 requires a function that receives a zip code and a date, and it should return a weekly forecast from that day.

domain/datasource/ForecastProvider.kt

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

    companion object {
        const 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 am defaulting it to a `SOURCES` list defined in the companion object. It will use a database source and a server source. The order is essential, 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:

domain/datasource/ForecastProvider.kt

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

It gets the first result that is not `null`. When searching through the list of functional operators explained in chapter 18, I could not 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 it to behave as expected:

extensions/CollectionsExtensions.kt

```kotlin
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 of type `T` and returns a value of type `R?`. The generic type specifies that the predicate can return null, but our `firstResult` function cannot. That is the reason why it returns a value of type `R`.

How does it work? It iterates and executes the predicate over the elements in the `Iterable` collection. When the result of the predicate is not null, this result is 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 am 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:

domain/datasource/ForecastProvider.kt

```kotlin
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 does not 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, and keeping only the day. Some of the sources (in our case the database) may need it. The server defaults to today unless we specify it, so it will be ignored there.

domain/datasource/ForecastProvider.kt

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

The complete code of this class would be:

domain/datasource/ForecastProvider.kt

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

    companion object {
        const 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`:

data/db/ForecastDb.kt

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

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

We have not implemented `ForecastServer` yet, but it is straightforward. It makes use of a `ForecastDb` to save the response once it is received from the server. That way, we can keep it cached into the database for future requests.

data/server/ForecastServer.kt

```kotlin
class ForecastServer(
    private val dataMapper: ServerDataMapper = ServerDataMapper(),
    private 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 forecastDb.requestForecastByZipCode(zipCode, date)
    }

}
```

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 values from the database, because we need the row ids that were self-generated by the insert query.

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.

domain/commands/RequestForecastCommand.kt

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

    companion object {
        const val DAYS = 7
    }

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

The rest of code modifications consist of some renames and changes in package structure here and there. Take a look at the corresponding branch at Kotlin for Android Developers repository[40].

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

22. Flow control and ranges

I have been using some conditional expressions in our code, so you are not new to some of the concepts of this chapter. But now it is time to explain them in more depth. Though in Kotlin we usually need fewer mechanisms to control the flow of the code that we would typically use in an entirely procedural programming language (some of them even practically disappear), they are still useful. There are also new powerful ideas that 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, so though we can use `if` as we are used to doing:

```
if (x > 0) {
    toast("x is greater than 0")
} else if (x == 0) {
    toast("x equals 0")
} else {
    toast("x is smaller than 0")
}
```

We can also assign its result to a variable. We have used it like that several times in our code:

```
val res = if (x != null && x.size() >= days) x else null
```

This also implies that we do not need a ternary operation similar to the Java one because we can solve it easily with:

```kotlin
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 returns `Unit`, which can be ignored, and it will behave like 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 tries to match its argument against all possible branches sequentially until it finds one that is satisfied. It then applies 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 are satisfied. The code executed when a condition is satisfied can be a block too:

```kotlin
when (x) {
    1 -> println("x == 1")
    2 -> println("x == 2")
    else -> {
        println("I'm a block")
        println("x is neither 1 nor 2")
    }
}
```

As it is an expression, it can return a result too. Take into consideration that when it is used as an expression, it must cover all the possible cases or implement the `else` branch. It will not compile otherwise:

```
val result = when (x) {
    0, 1 -> "binary"
    else -> "error"
}
```

As you can see, the condition can be a set of values separated by commas. However, it can be many more things. We could, for instance, check the type of the argument and make decisions based on this:

```
when(view) {
    is TextView -> view.text = "I'm a TextView"
    is EditText -> toast("EditText value: ${view.getText()}")
    is ViewGroup -> toast("Number of children: ${view.getChildCount()} ")
    else -> view.visibility = View.GONE
}
```

The argument is automatically converted in the right part of the condition, so you do not need to do any explicit casting.

It is possible to check whether the argument is inside a range (I will 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"
}
```

Alternatively, you could also get rid of the argument and do any crazy checks 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 will not 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 achieve the regular iteration over indices, we can also do it using ranges:

```
for (index in 0..viewGroup.getChildCount() - 1) {
    val view = viewGroup.getChildAt(index)
    view.visibility = View.VISIBLE
}
```

When iterating over an array or a list, we can make uses of existing properties that return a set of indices, 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 they are not very common in Kotlin either. There are usually simpler and more visual ways to resolve this problem. A couple of examples:

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

22.5 Ranges

It is challenging to explain flow control without talking about ranges. But their scope is broader. 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)
```

Kotlin defines ranges for any types that can be compared, but for numerical types the compiler optimizes it by converting it to simpler analog code in Java, to avoid the extra overhead. The numerical ranges are also iterable, and the loops are optimized 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)
```

We can define a spacing different from 1 among the values in a range by using `step`:

```
for (i in 1..4 step 2) println(i)

for (i in 4 downTo 1 step 2) println(i)
```

If you want to create an open range (which excludes the last item), you can use the function `until`:

```
for (i in 0 until 4) println(i)
```

This previous line prints from 0 to 3, but skips the last value. This means that `0 until 4 == 0..3`. For iterations over lists, it could be easier to understand if we use `for (i in 0 until list.size)` instead of `for (i in 0..list.size - 1)`.

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 until viewGroup.childCount).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. You can do everything on a single line.

If you want to know more about how to implement ranges and see more examples and useful information, you can go to Kotlin reference[41].

[41]https://kotlinlang.org/docs/reference/ranges.html

23 Creating a Detail Activity

When we click on an item from the home screen, 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 after an item click, but it is 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 need to send the *id* of the forecast to the detail. So the domain model needs a new *id* property:

domain/model/DomainClasses.kt

```kotlin
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 use 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:

domain/datasource/ForecastProvider.kt

```kotlin
private fun <T : Any> requestToSources(f: (ForecastDataSource) -> T?): T
        = sources.firstResult { f(it) }
```

The function is generified using a non-nullable type. It will receive a function which uses a `ForecastDataSource` to return a 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:

domain/datasource/ForecastProvider.kt

```
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:

domain/datasource/ForecastDataSource.kt

```
fun requestDayForecast(id: Long): Forecast?
```

The `ForecastDb` will always have the required value already cached from previous requests, so we can get it from there this way:

data/db/ForecastDb.kt

```
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:

extensions/DatabaseExtensions.kt

```
fun SelectQueryBuilder.byId(id: Long)
        = whereSimple("_id = ?", id.toString())
```

It makes use of the `whereSimple` 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` is never used, because the info is always cached in the database. We could implement it to defend our code from uncommon edge cases, but we are not doing it in this case, so it will just throw an exception if the function is called:

data/server/ForecastServer.kt

```
override fun requestDayForecast(id: Long)
        = 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, the example above shows how 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:

```
val x = when(y) {
    in 0..10 -> 1
    in 11..20 -> 2
    else -> throw Exception("Invalid")
}
```

The same happens with `try-catch`, we can assign a value depending on the result of the try:

```
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:

domain/commands/RequestDayForecastCommand.kt

```
class RequestDayForecastCommand(
        val id: Long,
        private val forecastProvider: ForecastProvider = ForecastProvider()) :
        Command<Forecast> {

    override fun execute() = forecastProvider.requestForecast(id)
}
```

The request returns a `Forecast` result that is used by the activity to draw its UI.

23.2 Providing a new activity

We are now prepared to create the `DetailActivity`.

In Android Studio 2.3 and below, you can select the package where you want to create the activity, right click and select `New -> Kotlin Activity`.

For Android Studio 3.0, the wizard already allows selecting the language. Choose `New -> Activity` and the type of activity you want to create. There is a `Source Language` drop-down at the end of the screen that you can use to change the language to Kotlin.

Our detail activity receives a couple of parameters from the main one: the forecast id and the name of the city. The first one is used to request the data from the database, and the name of the city fills the toolbar. So we first need a couple of names to identify the parameters in the bundle:

ui/activities/DetailActivity.kt

```
public class DetailActivity : AppCompatActivity() {

    companion object {
        const val ID = "DetailActivity:id"
        const 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:

layout/activity_detail.xml

```xml
<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. The methods for `intent` and `title` are automatically mapped to a property:

ui/activities/DetailActivity.kt

```
setContentView(R.layout.activity_detail)
title = intent.getStringExtra(CITY_NAME)
```

The other part in `onCreate` implements the call to the command. It is very similar to the call we previously did:

ui/activities/DetailActivity.kt

```kotlin
doAsync {
    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 am using Kotlin Android Extensions plugin again in this activity, to get the properties from the XML without using `findViewById`:

ui/activities/DetailActivity.kt

```kotlin
import kotlinx.android.synthetic.main.activity_detail.*

...

private fun bindForecast(forecast: Forecast) = with(forecast) {
    Picasso.with(ctx).load(iconUrl).into(icon)
    supportActionBar?.subtitle = date.toDateString(DateFormat.FULL)
    weatherDescription.text = description
    bindWeather(high to maxTemperature, low to minTemperature)
}
```

There are some interesting things here. For instance, I am 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 is a good moment to extract it into a function:

extensions/ExtensionUtils.kt

```kotlin
fun Long.toDateString(dateFormat: Int = DateFormat.MEDIUM): String {
    val df = DateFormat.getDateInstance(dateFormat, Locale.getDefault())
    return df.format(this)
}
```

It gets a date format (or use the default `DateFormat.MEDIUM`) and converts the `Long` into a `String` that is understandable by the user.

Another interesting function is `bindWeather`. It gets a `vararg` of pairs of `Int` and `TextView`, and assigns a text and a text color to the `TextViews` based on the temperature.

ui/activities/DetailActivity.kt

```
private fun bindWeather(vararg views: Pair<Int, TextView>) = views.forEach {
    it.second.text = "${it.first}"
    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 shows the temperature and a color based on the value of the temperature: red for low temperatures, orange for mild ones and green for the rest. The values are random, but it is a good illustration 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 have used in some other places. At the time of writing this lines, current support library relies on the class `ContextCompat` to get a color in a compatible way for all Android versions:

extensions/ContextExtensions.kt

```
public fun Context.color(res: Int): Int = ContextCompat.getColor(this, res)
```

I was missing a property representation for `textColor`. The thing is `TextView` lacks `getTextColor()` method, so it is not automatically parsed. A definition could be this one:

extensions/ViewExtensions.kt

```
var TextView.textColor: Int
    get() = currentTextColor
    set(v) = setTextColor(v)
```

There is an implementation in another Anko package (it returns an exception in `get`, it could be an alternative), the one related to the creation of views using a DSL. If you are implementing your views using regular XML, I recommend not to add this library if you are using one or two functions. That library is big, and you will waste a good part of method count.

The `AndroidManifest` also needs to be aware that a new activity exists:

AndroidManifest.xml

```
<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 final 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**:

ui/activities/MainActivity.kt

```
val adapter = ForecastListAdapter(result) {
    startActivity<DetailActivity>(DetailActivity.ID to it.id,
            DetailActivity.CITY_NAME to result.city)
}
```

What is 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. A popular workaround is to pass 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 `T::class.javaClass`. A simpler version of what Anko does would be the next (I am only using `String` extras in this example):

```
public inline fun <reified T: Activity> Context.startActivity(
        vararg params: Pair<String, String>) {

    val intent = Intent(this, T::class.java)
    params.forEach { intent.putExtra(it.first, it.second) }
    startActivity(intent)
}
```

The real implementation is a bit more complicated because it uses a long and a boring `when` expression to add the extras depending on the type, but it does not 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 starts the activity using the intent. The reified type is limited to be an `Activity` descendant.

The rest of little details are covered in the repository[42]. We now have a simple (but complete) master-detail App implemented in Kotlin without using a single line of Java.

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

24 Interfaces and Delegation

24.1 Interfaces

Interfaces in Kotlin are more powerful than in Java 7. If you have worked with Java 8, similarities are much closer there. In Kotlin, we can use interfaces the same way as in Java. Imagine we have some animals, and some of them can fly. This is the interface we could have for flying animals:

```kotlin
interface FlyingAnimal {
    fun fly()
}
```

Both birds and bats can fly by moving their wings. So let's create a couple of classes for them:

```kotlin
class Bird : FlyingAnimal {
    val wings: Wings = Wings()
    override fun fly() = wings.move()
}

class Bat : FlyingAnimal {
    val wings: Wings = Wings()
    override fun fly() = wings.move()
}
```

When a couple of classes extend from an interface, it is very typical they both share the same implementation. However, Java 7 interfaces can only define the behavior, but not implement it.

Kotlin interfaces, on the other hand, can implement functions. The only difference from a class is that they are stateless, so the properties that need a backing field need to be overridden by the class. The class is in charge of saving the state of interface properties.

We can make the interface implement the `fly` function:

```
interface FlyingAnimal {
    val wings: Wings
    fun fly() = wings.move()
}
```

As mentioned, classes need to override the property:

```
class Bird : FlyingAnimal {
    override val wings: Wings = Wings()
}

class Bat : FlyingAnimal {
    override val wings: Wings = Wings()
}
```

And now both birds and bats can fly:

```
val bird = Bird()
val bat = Bat()

bird.fly()
bat.fly()
```

24.2 Delegation

The delegation[43] is a handy pattern that can be used to extract responsibilities from a class. The delegation pattern is supported natively by Kotlin, so it prevents the need of calling the delegate. The delegator just needs to specify which instance implements the interface.

In our previous example, we can specify how the animal flies through the constructor, instead of implementing it. For instance, a flying animal that uses wings to fly can be implemented this way:

[43]https://en.wikipedia.org/wiki/Delegation_pattern

```
interface CanFly {
    fun fly()
}

class Bird(f: CanFly) : CanFly by f
```

We can indicate that a bird can fly by using the interface, but the way the bird uses to fly is defined by a delegate that comes through the constructor, so we can have different birds with different flying methods. The way an animal with wings flies is defined in another class:

```
class AnimalWithWings : CanFly {
    val wings: Wings = Wings()
    override fun fly() = wings.move()
}
```

An animal with wings moves its wings to be able to fly. So now we can create a bird that flies using wings:

```
val birdWithWings = Bird(AnimalWithWings())
birdWithWings.fly()
```

But now wings can be used with other animals that are not birds. If we assume that bats always use wings, we could instantiate the object directly where we specify the delegation:

```
class Bat : CanFly by AnimalWithWings()
...
val bat = Bat()
bat.fly()
```

24.3 Implementing an example in our App

Interfaces can be used to extract common code from classes which have some similar behavior. For instance, we can create an interface that deals with the toolbar of the app. Both `MainActivity` and `DetailActivity` share similar code that deals with the toolbar.

But first, some changes need to be done to start using a toolbar included in the layout instead of the standard `ActionBar`. The first thing will be extending a `NoActionBar` theme. That way, the toolbar is not included automatically:

values/styles.xml

```xml
<style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
    <item name="colorPrimary">#ff212121</item>
    <item name="colorPrimaryDark">@android:color/black</item>
</style>
```

We are using a light theme. Next, let's create a toolbar layout that we can include later in some other layouts:

layout/toolbar.xml

```xml
<android.support.v7.widget.Toolbar
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>
```

The toolbar specifies its background, a dark theme for itself and a light theme for the popups it generates (the overflow menu for instance). We get then the same theme we already had: a light theme with dark Action Bar.

Next step will be modifying the `MainActivity` layout to include the toolbar:

layout/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/forecastList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:clipToPadding="false"
        android:paddingTop="?attr/actionBarSize"/>

    <include layout="@layout/toolbar"/>

</FrameLayout>
```

Now that we have added the toolbar to the layout, we can start using it. We are creating an interface that will let us:

- Change the title
- Specify whether it shows the up navigation action or not
- Animate the toolbar when scrolling
- Assign the same menu to all activities and an event for the actions

So let's define the `ToolbarManager`:

ui/activities/ToolbarManager.kt

```
interface ToolbarManager {
    val toolbar: Toolbar
    ...
}
```

It needs a toolbar property. Interfaces are stateless, so the property can be defined, but no value can be assigned. Classes that implement it require overriding the property.

On the other hand, we can implement stateless properties without the need of being overridden. That is, properties that can work without using a backing field. An example would be a property which deals with the toolbar title:

ui/activities/ToolbarManager.kt

```kotlin
var toolbarTitle: String
    get() = toolbar.title.toString()
    set(value) {
        toolbar.title = value
    }
```

As the property just uses the toolbar, it does not need to save additional state.

We are now creating a new function that initializes the toolbar, by inflating a menu and setting a listener:

ui/activities/ToolbarManager.kt

```kotlin
fun initToolbar() {
    toolbar.inflateMenu(R.menu.menu_main)
    toolbar.setOnMenuItemClickListener {
        when (it.itemId) {
            R.id.action_settings -> App.instance.toast("Settings")
            else -> App.instance.toast("Unknown option")
        }
        true
    }
}
```

We can also add a function that enables the navigation icon in the toolbar, sets an arrow icon and a listener that will be fired when the icon is pressed:

ui/activities/ToolbarManager.kt

```kotlin
fun enableHomeAsUp(up: () -> Unit) {
    toolbar.navigationIcon = createUpDrawable()
    toolbar.setNavigationOnClickListener { up() }
}

private fun createUpDrawable() =
     DrawerArrowDrawable(toolbar.ctx).apply { progress = 1f }
```

The function receives the listener, creates the up drawable by using the DrawerArrowDrawable[44] on its final state (when the arrow is already showing) and assigns the listener to the toolbar.

Finally, the interface provides a function that allows the toolbar to be attached to a scroll, and animates the toolbar depending on the direction of the scroll. The toolbar will be hidden while we are scrolling down and displayed again when scrolling up:

ui/activities/ToolbarManager.kt

```
fun attachToScroll(recyclerView: RecyclerView) {
    recyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {
        override fun onScrolled(recyclerView: RecyclerView?, dx: Int, dy: Int) {
            if (dy > 0) toolbar.slideExit() else toolbar.slideEnter()
        }
    })
}
```

Let's create a couple of extension functions that animate the views in and out of the screen. They check if the animation has not been previously performed. That way it prevents the view from being animated every time the scroll varies:

extensions/ViewExtensions.kt

```
fun View.slideExit() {
    if (translationY == 0f) animate().translationY(-height.toFloat())
}

fun View.slideEnter() {
    if (translationY < 0f) animate().translationY(0f)
}
```

After implementing the toolbar manager, it is time to use it in the `MainActivity`, which now must implement the interface:

[44] https://developer.android.com/reference/android/support/v7/graphics/drawable/DrawerArrowDrawable.html

ui/activities/MainActivity.kt

```kotlin
class MainActivity : AppCompatActivity(), ToolbarManager {
    ...
}
```

We first specify the toolbar property. We can implement a lazy find so that the toolbar will be already inflated by the time we use it:

ui/activities/MainActivity.kt

```kotlin
override val toolbar by lazy { find<Toolbar>(R.id.toolbar) }
```

`MainActivity` will initialize the toolbar, attach to the `RecyclerView` scroll and modify the toolbar title:

ui/activities/MainActivity.kt

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

    forecastList.layoutManager = LinearLayoutManager(this)
    attachToScroll(forecastList)

    doAsync {
        val result = RequestForecastCommand(94043).execute()
        uiThread {
            val adapter = ForecastListAdapter(result) {
                startActivity<DetailActivity>(DetailActivity.ID to it.id,
                        DetailActivity.CITY_NAME to result.city)
            }
            forecastList.adapter = adapter
            toolbarTitle = "${result.city} (${result.country})"
        }
    }
}
```

`DetailActivity` also needs some layout modifications:

layout/activity_detail.xml

```xml
<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">

    <include layout="@layout/toolbar"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center_vertical"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        tools:ignore="UseCompoundDrawables">
       ....
    </LinearLayout>

</LinearLayout>
```

The `toolbar` property is specified the same way. `DetailActivity` will initialize the toolbar too, set the title and enable the up navigation icon:

ui/activities/DetailActivity.kt

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

    initToolbar()
    toolbarTitle = intent.getStringExtra(CITY_NAME)
    enableHomeAsUp { onBackPressed() }
    ...
}
```

Interfaces can help us extract common code from classes that share similar behaviors. It can be used as an alternative to composition, which keeps our code better organized and easier to reuse.

Think where interfaces can help you write better code, and check the new one in the branch of this chapter[45].

[45] https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-24

25 Generics

Generic programming consists of writing algorithms without the need of specifying the exact type the code is going to use. That way, we can create functions or types that only differ in the set of types they use, improving code reusability. These units of code are known as generics, and they exist in many languages, including Java and Kotlin.

In Kotlin, generics are even more important, because the high presence of regular and extension functions increments the number of times that generics are of some use for us. Though we have been using them blindly throughout the book, generics are usually one of the trickiest parts of any language, so I am trying to explain it in the simplest possible way so that main ideas are clear enough.

25.1 Basics

For instance, we can create a class which specifies a generic type:

```
class TypedClass<T>(parameter: T) {
    val value: T = parameter
}
```

This class now can be instantiated using any type, and the parameter will use the type in the definition. We could do:

```
val t1 = TypedClass<String>("Hello World!")
val t2 = TypedClass<Int>(25)
```

But Kotlin is all about simplicity and boilerplate reduction, so if the compiler can infer the type of the parameter, we do not even need to specify it:

```
val t1 = TypedClass("Hello World!")
val t2 = TypedClass(25)
val t3 = TypedClass<String?>(null)
```

As the third object is receiving a null reference, the type still needs to be specified because the compiler cannot infer it.

We can go beyond and, as in Java, reduce the types that can be used in a generic class by setting it in the definition. For instance, if we want to restrict the previous class to non-nullable types, we do:

```
class TypedClass<T : Any>(parameter: T) {
    val value: T = parameter
}
```

If you compile previous code, you will see that `t3` now throws an error. Nullable types are not allowed anymore. But restrictions can be obviously more strict. What if we want to accept only classes that extend `Context`? Easy:

```
class TypedClass<T : Context>(parameter: T) {
    val value: T = parameter
}

val t1 = TypedClass(activity)
val t2 = TypedClass(context)
val t3 = TypedClass(service)
```

Now every class which extends `Context` can be used as the type of our class. The rest of types are not allowed.

Of course, types are also allowed in functions. We can build generic functions quite easily:

```
fun <T> typedFunction(item: T): List<T> {
    ...
}
```

25.2 Variance

This is really one of the trickiest parts to understand. In Java, there is a problem when we use generic types. Logic says that `List<String>` should be able to be casted to `List<Object>` because it is less restrictive. But take a look at this example:

```java
List<String> strList = new ArrayList<>();
List<Object> objList = strList;
objList.add(5);
String str = objList.get(0);
```

If the Java compiler allowed us to do this, we could add an `Integer` to an `Object` list, and this would crash at some point. That is why wildcards were added to the language. Wildcards increase flexibility while limiting this problem.

If we add '`? extends Object`' we are using **covariance**, which means that we can deal with any object that uses a type that is more restrictive than `Object`, but we can only do `get` operations safely. If we want to copy a collection of `String`s into a collection of `Object`s, we should be allowed, right?

Then, if we have:

```java
List<String> strList = ...;
List<Object> objList = ...;
objList.addAll(strList);
```

This is possible because the definition of `addAll()` in `Collection` interface is something like:

```java
List<String>
interface Collection<E> ... {
  void addAll(Collection<? extends E> items);
}
```

Otherwise, without the wildcard, we would not be allowed to use a `String` list with this method. The opposite, of course, would fail. We cannot use `addAll()` to add a list of `Objects` to a list of `Strings`. As we are only getting the items from the collection we use in that method, it is a perfect example of covariance.

On the other hand, we can find **contravariance**, which is just the opposite situation. Following with the `Collection` example, if we want to add items to a `Collection` we are passing as a parameter, we could add objects with a more restrictive type into a more generic collection. For instance, we could add `Strings` to an `Object` list:

```java
void copyStrings(Collection<? super String> to, Collection<String> from) {
    to.addAll(from);
}
```

The only restriction we have to add `Strings` to another collection is that the collection accepts `Objects` that are `Strings` or parent classes.

But wildcards have some limitations. Wildcards define **use-site variance**, which means we need to declare them where we use them. This implies adding boilerplate every time we declare a more generic variable.

Let's see an example. Using a class similar to the one we had before:

```java
public class TypedClass<T> {
    public T doSomething(){
        ...
    }
}
```

This code will not compile:

```java
TypedClass<String> t1 = new TypedClass<>();
TypedClass<Object> t2 = t1;
```

Though it really does not make sense, because we could still keep calling all the methods of the class, and nothing would break. We need to specify that the type can have a more flexible definition.

```
TypedClass<String> t1 = new TypedClass<>();
TypedClass<? extends String> t2 = t1;
```

The use of wildcards, in this case, makes things more difficult to understand, and adds some extra boilerplate.

On the other hand, Kotlin deals with it more naturally by using **declaration-site variance**. In Kotlin, we specify that we can deal with less restrictive situations when defining the class or interface, and then we can use it blindly everywhere.

So let's see how it looks. Instead of long wildcards, Kotlin uses ***out*** for covariance and ***in*** for contravariance. In this case, as our class is producing objects that can be saved into less restrictive variables, we are using covariance. We can define this in the class declaration directly:

```
class TypedClass<out T>() {
    fun doSomething(): T {
        ...
    }
}
```

And that is all we need. Now, the same example that would not compile in Java is perfectly possible in Kotlin:

```
val t1 = TypedClass<String>()
val t2: TypedClass<Any> = t1
```

If you were already used to these concepts, I am sure you can use ***in*** and ***out*** in Kotlin without any hassle. Otherwise, it requires a little practice and a good understanding of these concepts.

25.3 Generics examples

After the theory, let's move to some practical functions that will make our lives easier. Instead of reinventing the wheel, I am using three functions that are included in Kotlin standard library. These functions let us do remarkable things with just a generic implementation. They can inspire you to create your own functions.

let

`let` is a simple function that can be called by any object. It receives a function that takes the object as a parameter and returns the value that this function returns. It is handy to deal with nullable objects, for instance. Here it is the definition:

```
inline fun <T, R> T.let(f: (T) -> R): R = f(this)
```

It uses two generic types: `T` and `R`. The first type is defined by the calling object, and it is also the type that the lambda receives as the input argument. The second one is the result of the function.

How can we use it? You may remember that, when we were retrieving data from a data source, the result could be null. We then returned a result mapped to the domain model if it was not null, or just a null reference otherwise:

data/db/ForecastDb.kt

```
if (forecast != null) dataMapper.convertDayToDomain(forecast) else null
```

This can be improved. We should not need to deal with nullable types that way. We can avoid it by using `let`:

data/db/ForecastDb.kt

```
forecast?.let { dataMapper.convertDayToDomain(it) }
```

`let` function is only executed if `forecast` is not null thanks to '`?.`' operator. It returns `null` otherwise. Just what we were trying to achieve.

with

We have talked a lot about this function during the book. `with` receives an object and a lambda that behaves as an extension function. So we can use `this` inside the lambda to refer to the object. It also returns an object defined in the last line of the function.

```
inline fun <T, R> with(receiver: T, f: T.() -> R): R = receiver.f()
```

Generics work the same way here: `T` stands for the receiver type and `R` for the result. As you can see, the lambda is defined as an extension function by using this declaration: `f: T.() -> R`. That is why we can then call `receiver.f()`.

We have several examples throughout the app:

data/db/DbDataMapper.kt

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

apply

It may look very similar to `with` function, but the idea is a bit different. `apply` can be used to avoid the creation of builders, because the object that calls the function can initialize itself the way it needs, and the `apply` function returns the same object:

```
inline fun <T> T.apply(f: T.() -> Unit): T { f(); return this }
```

We only need one generic type here, because the object that calls the function is also the returned value. A nice simple example would be:

```
val textView = TextView(context).apply {
    text = "Hello"
    hint = "Hint"
    textColor = android.R.color.white
}
```

It creates a `TextView`, modifies some properties, and assigns it to a variable. Everything in a simple, readable and compact syntax. Let's use it in our current code. In `ToolbarManager`, we were doing this to create the navigation drawable:

```
private fun createUpDrawable() = with(DrawerArrowDrawable(toolbar.ctx)) {
    progress = 1f
    this
}
```

Using `with` and returning `this` is clearly something that can be done easier by using `apply`:

ui/activities/ToolbarManager.kt

```
private fun createUpDrawable() = DrawerArrowDrawable(toolbar.ctx).apply {
    progress = 1f
}
```

You can review some more little improvements in Kotlin for Android Developers repository[46].

[46]https://github.com/antoniolg/Kotlin-for-Android-Developers/tree/chapter-25

26 Settings Screen

Until now, we have been using a default city to implement the app, but it is time add the ability to select another city. The app needs a settings section where the user can change the city.

We are going to stick to the zip code to identify the city. A real app would probably need more information because a zip code by itself does not identify a city in the whole world. But we at least will show a city around the world that uses the zip code we are defining in settings. This is a good chance to explain a different way to deal with preferences.

26.1 Creating the settings activity

A new activity will be opened when the settings option is selected in the overflow menu in the toolbar. So the first thing we need is a new `SettingsActivity`:

ui/activities/SettingsActivity.kt

```kotlin
class SettingsActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_settings)
        setSupportActionBar(toolbar)
        supportActionBar?.setDisplayHomeAsUpEnabled(true)
    }

    override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
        android.R.id.home -> {
            onBackPressed()
            true
        }
        else -> false
    }
}
```

We will save the preference when the user gets out of the screen activity, so we are going to deal with the `Up` action the same way as the `Back` one, by redirecting the action to `onBackPressed`. Now, let's create the XML layout. A simple `EditText` is enough for the preference:

layout/activity_settings.xml

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

    <include layout="@layout/toolbar"/>

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="?attr/actionBarSize"
        android:padding="@dimen/spacing_xlarge">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/city_zipcode"/>

        <EditText
            android:id="@+id/cityCode"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="@string/city_zipcode"
            android:inputType="number"/>

    </LinearLayout>

</FrameLayout>
```

And then declare the activity in the `AndroidManifest.xml`:

AndroidManifest.xml

```
<activity
    android:name=".ui.activities.SettingsActivity"
    android:label="@string/settings"/>
```

26.2 Accessing Shared Preferences

You probably know what Android Shared Preferences[47] are. They consist of a set of keys and values that can be easily saved and restored by using the Android framework. These preferences are integrated with some parts of the SDK to make some tasks easier. Besides, since Android 6.0 (Marshmallow), shared preferences can be automatically cloud-saved, so when a user restores the app in a new device, these preferences are automatically recovered too.

Thanks to the use of property delegation, we can deal with preferences cleverly. We can create a delegate that queries the preference when `get` is called and saves it when we call to `set`.

As we want to save the zip code, which is a `Long` value, let's create a delegate for `Long` properties. In `DelegatesExtensions.kt`, implement a new `LongPreference` class:

extensions/DelegatesExtensions.kt

```
class LongPreference(
        private val context: Context,
        private val name: String, val default: Long) {

    private val prefs by lazy {
        context.getSharedPreferences("default", Context.MODE_PRIVATE)
    }

    operator fun getValue(thisRef: Any?, property: KProperty<*>): Long =
        prefs.getLong(name, default)

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: Long) {
        prefs.edit().putLong(name, value).apply()
```

[47] http://developer.android.com/training/basics/data-storage/shared-preferences.html

```
    }
}
```

First, we create a lazy access to preferences. That way, if we do not use the property, this delegate never requests the `SharedPreferences` object.

When `get` is called, its implementation uses the preferences instance to retrieve a long property with the name that was specified in the delegate declaration, and defaulting to the `default` value if the property is not found. When a value is set, it requests a preferences editor that saves the value using the name of the property.

We can then define the new delegate in the `DelegatesExt` object, so that it is easier to access when required:

extensions/DelegatesExtensions.kt

```
object DelegatesExt {
    ....
    fun longPreference(context: Context, name: String, default: Long) =
        LongPreference(context, name, default)
}
```

In `SettingsActivity`, we can now define a property to deal with the zip code preference. I am also creating a couple of constants which keep the name and the default value of the property. That way they can be used in other sections of the App.

ui/activities/SettingsActivity.kt

```
companion object {
    const val ZIP_CODE = "zipCode"
    const val DEFAULT_ZIP = 94043L
}

private var zipCode: Long
        by DelegatesExt.longPreference(this, ZIP_CODE, DEFAULT_ZIP)
```

Now it is easy to work with this preference. In `onCreate`, we get the value of the property and assign it to the `EditText`:

ui/activities/SettingsActivity.kt

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    cityCode.setText(zipCode.toString())
}
```

We cannot use the self-generated property `text` because `EditText` returns an `Editable` in `getText`, so the property defaults to that value. If we try to assign a `String`, the compiler will complain. Using `setText()` will be enough.

Now we have everything we need to implement `onBackPressed`. Here, the new value of the property is saved:

ui/activities/SettingsActivity.kt

```
override fun onBackPressed() {
    super.onBackPressed()
    zipCode = cityCode.text.toString().toLong()
}
```

The `MainActivity` requires some little changes. First, it also needs a `zipCode` property.

ui/activities/MainActivity.kt

```
private val zipCode: Long by DelegatesExt.longPreference(this,
        SettingsActivity.ZIP_CODE, SettingsActivity.DEFAULT_ZIP)
```

Moreover, I am moving the forecast load to `onResume` so that every time the activity is resumed, it refreshes the data, just in case the zip code changed. Of course, there are more efficient ways to do this, by checking whether the zip code changed before requesting the forecast again, for instance. But I want to keep this example simple, and the local database already keeps the requested info, so this solution is not that bad:

ui/activities/MainActivity.kt

```kotlin
override fun onResume() {
    super.onResume()
    loadForecast()
}

private fun loadForecast() = doAsync {
    val result = RequestForecastCommand(zipCode).execute()
    uiThread {
        val adapter = ForecastListAdapter(result) {
            startActivity<DetailActivity>(DetailActivity.ID to it.id,
                    DetailActivity.CITY_NAME to result.city)
        }
        forecastList.adapter = adapter
        toolbarTitle = "${result.city} (${result.country})"
    }
}
```

The `RequestForecastCommand` is now using the `zipCode` property instead of the previously fixed value.

There is just one more thing we must do: start the settings activity when the user clicks on the overflow action. In `ToolbarManager`, the `initToolbar` function requires a small change:

ui/activities/ToolbarManager.kt

```kotlin
when (it.itemId) {
    R.id.action_settings -> toolbar.ctx.startActivity<SettingsActivity>()
    else -> App.instance.toast("Unknown option")
}
```

26.3 Generic preference delegate

Now that we are generics experts, why not extending `LongPreference` to be used with any type that Shared Preferences support? Let’s create a new `Preference` delegate:

extensions/DelegatesExtensions.kt

```kotlin
class Preference<T>(private val context: Context, private val name: String,
    private val default: T) {

    private val prefs by lazy {
        context.getSharedPreferences("default", Context.MODE_PRIVATE)
    }

    operator fun getValue(thisRef: Any?, property: KProperty<*>): T =
        findPreference(name, default)

    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        putPreference(name, value)
    }
    ...
  }
```

This preference is very similar to what we had before. We just substituted the `Long` references with a generic type `T`, and called to a couple of functions that do the hard work. These functions are very simple, though a bit repetitive. They check the type and use the specific method from preferences. For instance, the `findPreference` function looks like this:

extensions/DelegatesExtensions.kt

```kotlin
private fun findPreference(name: String, default: T): T = with(prefs) {
    val res: Any = when (default) {
        is Long -> getLong(name, default)
        is String -> getString(name, default)
        is Int -> getInt(name, default)
        is Boolean -> getBoolean(name, default)
        is Float -> getFloat(name, default)
        else -> throw IllegalArgumentException(
            "This type can't be saved into Preferences")
    }

    res as T
}
```

And basically the same for `putPreference` function, but using the preferences editor and saving the result of `when` at the end, by calling `apply()`:

extensions/DelegatesExtensions.kt

```kotlin
private fun putPreference(name: String, value: T) = with(prefs.edit()) {
    when (value) {
        is Long -> putLong(name, value)
        is String -> putString(name, value)
        is Int -> putInt(name, value)
        is Boolean -> putBoolean(name, value)
        is Float -> putFloat(name, value)
        else -> throw IllegalArgumentException("This type can be saved into Preferenc\
es")
    }.apply()
}
```

Now update `DelegatesExt` object and you are done:

extensions/DelegatesExtensions.kt

```kotlin
object DelegatesExt {
    ...
     fun <T> preference(context: Context, name: String, default: T)
         = Preference(context, name, default)
}
```

After this chapter, the user can now access the settings screen and modify the zip code. That way, when they return to the main screen, the forecast is automatically refreshed with the new information. Check the rest of small tweaks in the repository[48].

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

27 A first walk into coroutines

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

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

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

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

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

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

27.1 Understanding how coroutines work

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

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

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

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

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

I am not explaining how to implement the above function. It is a complicated process that is out of the scope of this book, and for most cases, there are solutions already implemented for you. If you are interested in building your own, you can read the specification written in coroutines Github[49]. What you need to know is that the function can have whatever name you want to give it and that it has at least a suspending block as a parameter.

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

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

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

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

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

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

- Raw implementation: it means building your way to use coroutines. This is quite complex and usually not required at all.
- Low-level implementations: Kotlin provides a set of libraries that you can find in kotlinx.coroutines[50] repository, which solves some of the hardest parts and provides a specific implementation for different scenarios. There is one for Android[51], for instance.
- Higher-level implementations: if you want to have a solution that provides everything you need to start using coroutines right away, there are several libraries out there that do all the hard work for you, and the list keeps growing. I am going to stick to Anko, which provides a solution that works well on Android, and you are already familiar with the library.

27.2 Using Anko for coroutines

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

27.3 Using coroutines in our example

Our current example does not have many complex requests, and there is not much value in using coroutines instead of `doAsync`, but I want to show you how to configure your project to start using them. It is pretty straightforward.

First, add the dependency to the `build.gradle`:

app/build.gradle

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

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

app/build.gradle

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

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

You can change this code:

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

with this one:

ui/activities/DetailActivity.kt

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

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

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

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

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

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

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

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

```
private fun loadForecast() {

    val ref = asReference()
    val localZipCode = zipCode

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

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

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

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

ui/activities/MainActivity.kt

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

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

Bye bye, callback hell. Check new changes in chapter-27[52] branch.

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

28 Testing your App

We are reaching the end of this trip. You already learned most Kotlin features throughout this book, but you are probably wondering if you can test your Android Apps using Kotlin exclusively. The answer is: of course!

In Android, we have a couple of different tests: unit and instrumentation tests. This book is not meant to teach you how to write tests; there are entire books dedicated to that matter. My goal in this chapter is to explain how to prepare your environment to be able to write some tests and show you that Kotlin also works fine for testing.

28.1 Unit testing

I will not get into discussions about what unit testing means. There are many definitions out there with some slight differences. A general idea could be that unit tests are the tests that validate an individual unit of source code. What a 'unit' involves is left to the reader. For the sake of simplicity, I am defining a unit test as a test that does not need a device to run. The IDE can run the tests and show a result that identifies which tests succeeded and which ones failed.

Unit testing is usually done using the `JUnit` library. So let's add the dependency to the `build.gradle`. As this dependency is only used when running tests, we can use `testImplementation` instead of `implementation`. This way, the library is left out for regular compilations, reducing the size of the APK:

app/build.gradle

```
dependencies {
    ...
    testImplementation "junit:junit:4.12"
}
```

Now sync Gradle to get the library included in your project. In old versions of Android Studio, you may need to choose which kind of tests you want to run. Go to the 'Build Variants' tab (you probably have it on the left side of the IDE) and click on 'Test Artifact' dropdown. You should choose 'Unit Tests' there. New Android Studio versions can have both kinds of tests enabled at the same time, so this step is not required.

Another thing you need to do is create a new folder. Below `src`, you already probably have `androidTest` and `main`. Create another one called `test`, and a folder called `java` below. So now you should have a `src/test/java` folder colored in green. This is a good indication that the IDE detected that we are in 'Unit Test' mode and that this folder contains test files.

Let's write a simple test to check everything works properly. Create a new Kotlin class called `SimpleTest` using the proper package (*com.antonioleiva.weatherapp* in my case, but you need to use the main package of your app). Once you have created the new file, write this simple test:

test/SimpleTest.kt

```
import org.junit.Test
import kotlin.test.assertTrue

class SimpleTest {
    @Test
    fun `unit testing works`() {
        assertTrue(true)
    }
}
```

Use the `@Test` annotation to identify the function as a test. Be sure to use `org.unit.Test`. Then add a simple assertion. It only checks that `true` is `true`, which should succeed.

Did you see how I wrote the name of the function? If you use backquotes, you can provide a readable name for your tests. When tests run in Android Studio, the summary shows texts that are much easier to read.

You will see that these names show an error. It is because this format is not available for Android code, so it is not available for instrumentation tests. But you can use them without issues in unit tests, so you can disable the inspection just for tests and use it safely.

To execute the tests, right click on the new `java` folder you created below `test`, and choose 'Run All Tests'. When the compilation finishes, it will run the test, and you will see a summary showing the result. You should notice that your test passed.

Now it is time to create some real tests. Everything that deals with the Android framework probably needs an instrumentation test or some extra libraries such as Robolectric[53]. Because of that, in these examples, I am testing things that do not use anything from the framework. For instance, I will test the extension function that creates a date `String` from a `Long`.

Create a new file called `ExtensionTests`, and add this tests:

test/ExtensionsTest.kt

```
class ExtensionsTest {
    @Test
    fun `"longToDateString" returns valid value`() {
        assertEquals("Oct 19, 2015", 1445275635000L.toDateString())
    }

    @Test
    fun `"longToDateString" with full format returns valid value`() {
        assertEquals("Monday, October 19, 2015",
            1445275635000L.toDateString(DateFormat.FULL))
    }
}
```

These tests check that a `Long` instance is properly converted to a `String`. The first one tests the default behavior (which uses `DateFormat.MEDIUM`), while the second one specifies a different format. Run the tests and see that all of them pass. I also recommend you to change something and see how it crashes.

If you are used to testing your apps in Java, you will see there is not much difference here. I have covered a simple example, but from here you can create more complex tests to validate other parts of the app. For instance, we could write some tests for `ForecastProvider`. We can use `Mockito` library to mock some other classes and be able to test the provider independently:

[53] http://robolectric.org/

build.gradle

```
buildscript {
    ...
    ext.mockito_version = '2.16.0'
}
```

app/build.gradle

```
dependencies {
    ...
    testImplementation "junit:junit:4.12"
    testImplementation "org.mockito:mockito-core:$mockito_version"
}
```

Now create a `ForecastProviderTest`. We are going to test that a `ForecastProvider` with a `DataSource` that returns something will get a result that is not null. So first we need to mock a `ForecastDataSource`:

test/domain/datasource/ForecastProviderTest.kt

```
val ds = mock(ForecastDataSource::class.java)
`when`(ds.requestDayForecast(0)).then {
    Forecast(0, 0, "desc", 20, 0, "url")
}
```

As you see, we need backquotes for `when` function, because `when` is a reserved word in Kotlin. So we need to escape it if we find some Java code that uses it.

An alternative would be to rename the import:

```
import org.mockito.Mockito.`when` as whenever

...

whenever(ds.requestDayForecast(0)).then { ... }
```

It is up to you; I will use the backquotes in this case.

Now we create a provider with this data source, and check that the result of the call to that method is not null:

```
val provider = ForecastProvider(listOf(ds))
assertNotNull(provider.requestForecast(0))
```

This is the complete test function:

test/domain/datasource/ForecastProviderTest.kt

```
@Test fun  `data source returns a value`() {
    val ds = mock(ForecastDataSource::class.java)
    `when`(ds.requestDayForecast(0)).then {
        Forecast(0, 0, "desc", 20, 0, "url")
    }

    val provider = ForecastProvider(listOf(ds))
    assertNotNull(provider.requestForecast(0))
}
```

If you run this, you will see that it crashes. Thanks to this test, we are detecting we have something wrong in our code. The test is failing because `ForecastProvider` is initializing `SOURCES` inside its companion object before it is used. We can add some sources to the `ForecastProvider` through the constructor, and this static list would never be used, so it should be lazy loaded:

domain/datasource/ForecastProvider.kt

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

If you now run again, you will see it is now passing all the tests.

We can also test, for instance, that when a source returns null, it will iterate over the next source to get a result:

test/domain/datasource/ForecastProviderTest.kt

```
@Test fun `empty database returns server value`() {
    val db = mock(ForecastDataSource::class.java)

    val server = mock(ForecastDataSource::class.java)
    `when`(server.requestForecastByZipCode(
            any(Long::class.java), any(Long::class.java)))
            .then {
                ForecastList(0, "city", "country", listOf())
            }

    val provider = ForecastProvider(listOf(db, server))

    assertNotNull(provider.requestByZipCode(0, 0))
}
```

As you see, the simple dependency inversion we solved by using default values for arguments is enough to let us implement some simple unit tests. There are many more things we could test about this provider, but this example is enough to show that we can use the basic unit testing tools.

28.2 Mocking closed classes

In Kotlin, everything is closed by default, so you may find yourself in trouble when you want to mock a class. Mockito cannot mock closed (final) classes.

So imagine we want to test `RequestDayForecastCommand`, and check that when its `execute()` method is called, the provider is also called. We could do this:

test/domain/commands/RequestDayForecastCommandTest.kt

```kotlin
class RequestDayForecastCommandTest {

    @Test
    fun `provider is called when command is executed`() {
        val forecastProvider = mock(ForecastProvider::class.java)
        val command = RequestDayForecastCommand(2, forecastProvider)

        command.execute()

        verify(forecastProvider).requestForecast(2)
    }
}
```

If you run this test, you will find that it fails:

```
Mockito cannot mock/spy because :
 - final class
```

You have two options to overcome this situation:

- Use the `open` reserved word, which would allow Mockito to mock your class, but also any other class to extend this one (which was probably closed for a reason).
- Extract an interface, and mock the interface. This one is a cleaner solution, but also slightly redundant if you are not using that interface for anything else.

There is, however, a third alternative: an experimental feature in Mockito 2. This feature allows you to mock final classes and avoid those extra artifacts. It might be that by the time you read these lines, the feature is not experimental anymore. But if it still is, you need to add an extra library to opt in for the feature:

app/build.gradle

```
testImplementation "org.mockito:mockito-inline:$mockito_version"
```

Even if it is experimental, it works pretty well, so feel safe to use it.

Now you can rerun your test, and see how it magically passes.

28.3 Instrumentation tests

Instrumentation tests are a bit different. They are typically used to test UI interactions, where we need an Application instance to be running by the time the tests are executed. To do this, we need to deploy the app and run the tests on a device.

This type of tests reside in the `androidTest` folder, and we must change 'Test Artifact' to 'Android Instrumentation Tests' in 'Build Variants' panel on old Android Studio versions. The official library to implement instrumentation tests is Espresso[54], which helps us easily navigate through our App by writing `Actions`, and filter and check results using `ViewMatchers` and `Matchers`.

The configuration is a bit harder than the previous one. We need a bunch of extra libraries and Gradle configuration. The good thing is that Kotlin does not add any extra overhead, so if you already know how to configure Espresso, it should be easy for you.

First, specify the test runner in `defaultConfig`:

app/build.gradle

```
defaultConfig {
    ...
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
}
```

Once you have configured the runner, it is time to add the rest of the dependencies, this time using `androidTestImplementation`. That way, these libraries are only added when we compile to run the instrumentation tests:

[54] https://google.github.io/android-testing-support-library/

build.gradle

```
buildscript {
    ...
    ext.test_support_version = '1.0.1'
    ext.espresso_version = '3.0.1'
}
```

app/build.gradle

```
dependencies {
    ...
    androidTestImplementation "com.android.support:support-annotations:$support_versi\
on"
    androidTestImplementation "com.android.support.test:runner:$test_support_version"
    androidTestImplementation "com.android.support.test:rules:$test_support_version"
    androidTestImplementation "com.android.support.test.espresso:espresso-core:$espre\
sso_version"
    androidTestImplementation("com.android.support.test.espresso:espresso-contrib:$es\
presso_version") {
        exclude group: 'com.android.support', module: 'appcompat-v7'
        exclude group: 'com.android.support', module: 'support-v4'
        exclude group: 'com.android.support', module: 'design'
        exclude group: 'com.android.support', module: 'recyclerview-v7'
}
```

I do not want to spend much time talking about this, but here it is a brief explanation of why you need these libraries:

- `runner`: It is the test runner, the one we specified in `defaultConfig`.
- `rules`: Includes some rules that help tests inflate and launch the activities. We will use a rule in our examples.
- `espresso-core`: the basic features of Espresso, the library that makes instrument tests easier.
- `espresso-contrib`: it adds some extra features, such as `RecyclerView` testing support. We have to exclude some of its dependencies, because we already have them in the project, and tests crash otherwise.

Let's now create a simple example. The test will click on the first row of the forecast list, and check that it can find a view with the id `R.id.weatherDescription`. This view is in the `DetailActivity`, which implies that we successfully navigated to the detail after clicking on a view inside the `RecyclerView`.

```kotlin
class SimpleInstrumentationTest {

    @get:Rule
    val activityRule = ActivityTestRule(MainActivity::class.java)

    ...
}
```

We need to create an activity rule, which instantiates the activity that the test will use. In Java, you would annotate the field using `@Rule`. However, as you know, fields and properties are not the same, so if you use just that, the execution will fail because the access to the field inside the property is not public. What you need to do is to annotate the getter. Kotlin allows doing that by specifying `get` or `set` before the name of the rule. In this case, write `@get:Rule`.

After that, we are ready to create our first test:

```kotlin
@Test fun itemClickNavigatesToDetail() {
    onView(withId(R.id.forecastList)).perform(
            RecyclerViewActions
                .actionOnItemAtPosition<RecyclerView.ViewHolder>(0, click()))
    onView(withId(R.id.weatherDescription))
           .check(matches(isAssignableFrom(TextView::class.java)))
}
```

As these tests run on a device, we cannot use the same naming format we used for unit tests. Otherwise, it will crash.

The function is annotated with `@Test`, the same way we did with unit tests. We can start using Espresso in the body of the test. It first performs a click over the first position of the recycler. Then, it checks that it can find a view with a specific id and that it is an instance of `TextView`.

To run the test, you can do the same you did for unit tests: right click on the `java` folder below `androidTest`, and choose 'Run All Tests'. Now, in 'Target device', choose the target you prefer. Click 'OK' and then run. You should see how the app starts on your device, and the test clicks on the first position, opens the detail activity and closes the App again.

Now we are going to do a more difficult one. The test will open the overflow from the `Toolbar`, click on `Settings` action, change the city code and check that the `Toolbar` title has changed to the corresponding one.

```
@Test fun modifyZipCodeChangesToolbarTitle() {
    openActionBarOverflowOrOptionsMenu(activityRule.activity)
    onView(withText(R.string.settings)).perform(click())
    onView(withId(R.id.cityCode)).perform(replaceText("94301"))
    pressBack()
    onView(isAssignableFrom(Toolbar::class.java))
            .check(matches(
                withToolbarTitle(`is`("Palo Alto (US)"))))
}
```

What the test exactly does is:

- It first opens the overflow by using `openActionBarOverflowOrOptionsMenu`.
- It then finds a view with the `Settings` text and performs a click on it.
- After that, the settings activity opens, so it looks for the `EditText` and replaces the old city code with a new one.
- It presses the back button, which saves the new value in the preferences, and closes the activity.
- As `onResume` is executed in `MainActivity`, the request is performed again. This retrieves the forecast of the new city.
- The last line checks the `Toolbar` title and sees whether it matches with the proper value.

There is not a default matcher to check `Toolbar` title, but Espresso is easy to extend, so we can create a new matcher which implements the check:

```
private fun withToolbarTitle(textMatcher: Matcher<CharSequence>): Matcher<Any> =
        object : BoundedMatcher<Any, Toolbar>(Toolbar::class.java) {

            override fun matchesSafely(toolbar: Toolbar): Boolean =
                textMatcher.matches(toolbar.title)

            override fun describeTo(description: Description) {
                description.appendText("with toolbar title: ")
                textMatcher.describeTo(description)
            }
        }
```

The `matchesSafely` function is the place where the check happens, while the `describeTo` function adds some information about the matcher.

This chapter has been especially interesting because we have seen how Kotlin is entirely compatible with both unit and integration tests and can interoperate with the testing libraries. Take a look at the code[55] and run the tests by yourself.

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

29 Extra concepts

Throughout this book, we have talked about the most important concepts of the Kotlin language. But we did not use some of them when implementing the app, and I would not want to let them out of these pages. In this chapter, I will review some unrelated features that you could be helpful for you when you develop your next Android app using Kotlin.

29.1 Nested classes

As in Java, we can define classes inside other classes. By default, an inner class will not be able to access the members of the outer class (it would behave as a static class in Java):

```kotlin
class Outer {
  private val bar: Int = 1
  class Nested {
    fun foo() = 2
  }
}

val demo = Outer.Nested().foo() // == 2
```

This is what we used for instance for the `ViewHolder` inside the `ForecastListAdapter`. If we want to access to the members of the outer class, we need to declare it as an **`inner`** class:

```
class Outer {
  private val bar: Int = 1
  inner class Inner {
    fun foo() = bar
  }
}

val demo = Outer().Inner().foo() // == 1
```

29.2 Enum classes

Kotlin also provides a way to implement **enum**:

```
enum class Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Enums can have parameters, which are declared in the constructor:

```
enum class Icon(val res: Int) {
    UP(R.drawable.ic_up),
    SEARCH(R.drawable.ic_search),
    CAST(R.drawable.ic_cast)
}

val searchIconRes = Icon.SEARCH.res
```

Enums can be requested by the `String` that matches the name, and we can also get an `Array` that includes all the values of an Enum, so that we can iterate over them.

```
val search: Icon = Icon.valueOf("SEARCH")
val iconList: Array<Icon> = Icon.values
```

Besides, each Enum constant has functions to obtain its name and the position in the declaration:

```
val searchName: String = Icon.SEARCH.name
val searchPosition: Int = Icon.SEARCH.ordinal
```

Enums implement `Comparable` based on the ordinal, so it is easy to compare them.

29.3 Sealed classes

Sealed classes are used to represent restricted class hierarchies, which means that the number of classes that extend a sealed class is restricted. It is similar to an Enum in the sense that the compiler is aware of the number of subtypes a sealed class has. The difference is that enum instances are unique, while sealed classes can have multiple instances which can contain different states.

We could implement, for instance, something similar to the `Option` class: a type that prevents the use of `null` by returning a `Some` class when the object contains a value or the `None` instance when it is empty:

```
sealed class Option<out T>
object None : Option<Nothing>()
data class Just<out T>(val value: T) : Option<T>()
```

The good thing about sealed classes is that when they are used in a `when` expression, we can check all the options and will not need to add the `else` clause. Also, if at some point we add a new subclass, the compiler will warn us about the places where we should be dealing with it.

```
val result = when (option) {
    None -> "Empty"
    is Just -> "Contains a value"
}
```

29.4 Exceptions

In Kotlin, all exceptions implement `Throwable`, have a message and are unchecked. Consequently, we are not required to use `try/catch` on any of them. That is not the case in Java, where methods that throw `IOException`, for instance, need to be surrounded by a `try/catch` block. Time has shown that checked exceptions were not a good idea. People like Bruce Eckel[56], Rod Waldhoff[57] or Anders Hejlsberg[58] can give you a better perspective on it.

The way to throw an exception is very similar to Java:

```
throw MyException("Exception message")
```

And `try` expression is identical too:

```
try {
  // some code
}
catch (e: SomeException) {
  // handler
}
finally {
  // optional finally block
}
```

Both `throw` and `try` are expressions in Kotlin, which allows to assign them to a variable. This is useful when dealing with edge cases:

```
val s = when(x){
    is Int -> "Int instance"
    is String -> "String instance"
    else -> throw UnsupportedOperationException("Not valid type")
}
```

or

```
val s = try { x as String } catch(e: ClassCastException) { null }
```

[56] http://www.mindview.net/Etc/Discussions/CheckedExceptions
[57] http://radio-weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html
[58] http://www.artima.com/intv/handcuffs.html

30 Java interoperability

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

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

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

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

30.1 Package-level functions

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

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

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

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

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

30.2 Extension functions

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

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

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

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

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

30.3 Function overloads

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

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

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

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

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

And now the class in Java will be named:

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

30.4 Instance and static fields

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

```
class App : Application() {

    val appHelper = AppHelper()

    companion object {
        lateinit var instance: App
    }

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

}
```

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

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

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

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

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

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

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

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

And now you can use it from Java code:

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

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

30.5 Data classes

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

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

We can create instances of this class:

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

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

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

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

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

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

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

30.6 Sealed classes

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

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

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

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

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

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

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

30.7 Inline functions and reified types

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

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

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

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

We can use it without issues like this from Java:

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

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

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

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

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

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

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

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

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

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

31 Conclusion

Thanks for reading this book. Throughout this pages, we have learned Kotlin by implementing an Android app as an example. The weather app was an excellent example to play with the essential features most apps need: a master/detail UI, communication with an API, database storage, shared preferences…

The good thing about this method is that you have learned the most important Kotlin concepts while using them. In my opinion, it is easier to absorb new knowledge when you put it into action. It was my primary goal, because reference books are usually a helpful tool to solve some specific doubts, but they are hard to read from the beginning to the very end. Besides, as the examples are usually out of context, it is difficult to understand which kind of problems those features solve.

That was, in fact, the other goal of the book: to show you real problems we face in Android and how they can be solved using Kotlin. Any Android developer struggles with many questions when dealing with asynchrony, databases, or has to implement verbose listeners or activity navigations. By using a real app as an example, we have answered many of these questions while learning the new language features.

I hope I achieved these goals, and I wish that you not only learned Kotlin but also enjoyed reading this book. Now that Google has adopted Kotlin, I am convinced that learning the language is the way to go: you will enjoy developing apps and get prepared for this new skill that is becoming a need in many Android job positions these days.

This book is finished, but it does not mean that it is dead. I will keep updating it to the latest versions of Kotlin, reviewing and improving it based on your comments and suggestions. Feel free to write me about it at any moment and tell me what you think, the errors you find, concepts that are not clear enough or whatever concern you may have.

It has been a fantastic journey during the months I have been writing this book. I have learned a lot too, so thanks again for helping 'Kotlin for Android Developers' to become a reality.

Best,

Antonio Leiva

- Site: antonioleiva.com[59]
- Email: contact@antonioleiva.com[60]
- Twitter: @lime_cl[61]
- Google+: +AntonioLeivaGordillo[62]

[59] http://antonioleiva.com
[60] mailto:contact@antonioleiva.com
[61] http://twitter.com/lime_cl
[62] http://plus.google.com/+AntonioLeivaGordillo