

KOTLIN

FOR ANDROID DEVELOPERS

Learn Kotlin the easy way while
developing an Android App.

Antonio Leiva

Kotlin for Android Developers

Learn Kotlin the easy way while developing an Android App

© 2015 - 2016 Antonio Leiva

Tweet This Book!

Please help Antonio Leiva by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#kotlinandroiddev](#).

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

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

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

*I also want to do a special mention to **Luis Herrero Jiménez**, who has designed the awesome cover of this book, and to [Gautier Mechling](#) for helping me so much by reviewing this book. It's thanks to him that this 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 I hope it will become a reference. So any claim or suggestion you think it will improve the quality of this book will be welcomed. Feel free to write anytime to contact@antonioleiva.com.

Contents

I. About this book	1
II. Is this book for you?	2
III. About the author	3
1 Introduction	4
1.1 What is Kotlin?	4
1.2 What do we get with Kotlin?	5
2 Getting ready	9
2.1 Android Studio	9
2.2 Install Kotlin plugin	10
3 Creating a new project	11
3.1 Create the project in Android Studio	11
3.2 Configure Gradle	11
3.3 Convert MainActivity to Kotlin code	13
3.4 Test that everything works	13
4 Classes and functions	15
4.1 How to declare a class	15
4.2 Class inheritance	15
4.3 Functions	16
4.4 Constructor and functions parameters	16
5 Writing your first class	19
5.1 Creating the layout	19
5.2 The Recycler Adapter	20
6 Variables and properties	22
6.1 Basic types	22
6.2 Variables	23
6.3 Properties	24
7 Anko and Extension Functions	27

CONTENTS

7.1 What is Anko?	27
7.2 Start using Anko	27
7.3 Extension functions	28
8 Retrieving data from API	29
8.1 Performing a request	29
8.2 Performing the request out of the main thread	30
9 Data Classes	31
9.1 Extra functions	31
9.2 Copying a data class	31
9.3 Mapping an object into variables	32
10 Parsing data	33
10.1 Converting json to data classes	33
10.2 Shaping the domain layer	34
10.3 Drawing the data in the UI	36
11 Operator overloading	38
11.1 Operators tables	38
11.2 The example	39
11.3 Operators in extension functions	40
12 Making the forecast list clickable	41
13 Lambdas	46
13.1 Simplifying setOnClickListener()	46
13.2 Click listener for ForecastListAdapter	47
13.3 Extending the language	48
14 Visibility Modifiers	50
14.1 Modifiers	50
14.2 Constructors	51
14.3 Revising our code	51
15 Kotlin Android Extensions	52
15.1 How to use Kotlin Android Extensions	52
15.2 Refactoring our code	53
16 Application Singleton and Delegated Properties	56
16.1 Application Singleton	56
16.2 Delegated Properties	57
16.3 Standard Delegates	58
16.4 How to create a custom delegate	61
16.5 Reimplementing App Singleton	62

CONTENTS

17 Creating an SQLiteOpenHelper	64
17.1 ManagedSQLiteOpenHelper	64
17.2 Tables definition	65
17.3 Implementing SQLiteOpenHelper	66
17.4 Dependency injection	68
18 Collections and functional operations	70
18.1 Aggregate operations	71
18.2 Filtering operations	73
18.3 Mapping operations	75
18.4 Elements operations	76
18.5 Generation operations	78
18.6 Ordering operations	79
19 Saving and requesting data from database	81
19.1 Creating database model classes	81
19.2 Writing and requesting data	82
20 Null safety in Kotlin	89
20.1 How Null types work	89
20.2 Nullity and Java libraries	91
21 Creating the business logic to data access	93
22. Flow control and ranges	98
22.1 If Expression	98
22.2 When expression	99
22.3 For loops	100
22.4 While and do/while loops	101
22.5 Ranges	101
23 Creating a Detail Activity	103
23.1 Preparing the request	103
23.2 Providing a new activity	105
23.3 Start an activity: reified functions	110
24 Interfaces and Delegation	112
24.1 Interfaces	112
24.2 Delegation	113
24.3 Implementing an example in our App	114
25 Generics	120
25.1 Basics	120
25.2 Variance	121
25.3 Generics examples	123

CONTENTS

26 Settings Screen	126
26.1 Creating the settings activity	126
26.2 Accessing Shared Preferences	127
26.3 Generic preference delegate	130
27 Testing your App	133
27.1 Unit testing	133
27.2 Instrumentation tests	136
28 Extra concepts	140
28.1 Nested classes	140
28.2 Enum classes	140
28.3 Sealed classes	141
28.4 Exceptions	142
29 Conclusion	144

I. About this book

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

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

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

As you could read in previous pages (and probably the site where you downloaded), this is a lean publication. This means that the book has been progressing thanks to the readers comments. Even though it is 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.

Thanks for becoming part of this exciting project.

II. Is this book for you?

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

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

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

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

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

III. About the author

Antonio Leiva is an Android Engineer who spends time learning about new ways to get the most out of Android and then writes about it. He writes a blog at antonioleiva.com¹ about many different topics related to Android development.

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

He now works as an Android Engineer at [Plex](http://plex.tv)², where he also plays an important role in the design and UX of the Android applications.

You can find Antonio on Twitter as [@lime_cl](https://twitter.com/lime_cl)³ or Google+ as [+AntonioLeivaGordillo](http://plus.google.com/+AntonioLeivaGordillo)⁴.

¹<http://antonioleiva.com>

²<http://plex.tv>

³https://twitter.com/lime_cl

⁴<http://plus.google.com/+AntonioLeivaGordillo>

1 Introduction

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

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

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

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

1.1 What is Kotlin?

Kotlin, as described before, is a JVM based language developed by [JetBrains](https://www.jetbrains.com/)⁵, a company known for the creation of IntelliJ IDEA, a powerful IDE for Java development. Android Studio, the official Android IDE, is based on IntelliJ.

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

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

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

- **It's more expressive:** this is one of its most important qualities. You can write more with much less code.

⁵<https://www.jetbrains.com/>

- **It's safer:** Kotlin is null safe, which means that we deal with possible null situations in compile time, to prevent execution time exceptions. We need to explicitly specify if an object can be null, and then check its nullity before using it. You will save a lot of time debugging null pointer exception and fixing nullity bugs.
- **It's functional:** Kotlin is basically an object oriented language, not a pure functional language. However, as many other modern languages, it uses many concepts from functional programming, such as lambda expressions, to resolve some problems in a much easier way. 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 don't have access to the its source code.
- **It's highly interoperable:** You can continue using most libraries and code written in Java, because the interoperability between both languages is excellent. It's even possible to create mixed project, with both Kotlin and Java files coexisting.

1.2 What do we get with Kotlin?

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

Expressiveness

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

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

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

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

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

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

Null Safety

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

We can do things like this:

```
1 // This won't compile. Artist can't be null
2 var notNullArtist: Artist = null
3
4 // Artist can be null
5 var artist: Artist? = null
6
7 // Won't compile, artist could be null and we need to deal with that
8 artist.print()
9
10 // Will print only if artist != null
11 artist?.print()
12
13 // Smart cast. We don't need to use safe call operator if we previously
14 // checked nullity
15 if (artist != null) {
16     artist.print()
17 }
18
19 // Only use it when we are sure it's not null. Will throw an exception otherwise.
20 artist!!.print()
21
22 // Use Elvis operator to give an alternative in case the object is null.
23 val name = artist?.name ?: "empty"
```

Extension functions

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

```
1 fun Fragment.toast(message: CharSequence, duration: Int = Toast.LENGTH_SHORT) {  
2     Toast.makeText(getActivity(), message, duration).show()  
3 }
```

We can now do:

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

Functional support (Lambdas)

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

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

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

2 Getting ready

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

2.1 Android Studio

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

Android Studio is implemented as a plugin over [IntelliJ IDEA](https://www.jetbrains.com/idea/)⁶, a Java IDE created by [Jetbrains](https://www.jetbrains.com/)⁷, the company which is also behind Kotlin. So, as you can see, everything is tightly connected.

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

And second, [Gradle](https://gradle.org/)⁸ became the official build system for Android, which meant a whole bunch of new possibilities related to version building and deploy. Two of the most interesting functions are build systems and flavours, which let you create infinite versions of the app (or even different apps) in an easy way while using the same code base.

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

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

Download [Android Studio from the official page](https://developer.android.com/sdk/index.html)⁹ if you don't have it already.

⁶<https://www.jetbrains.com/idea/>

⁷<https://www.jetbrains.com/>

⁸<https://gradle.org/>

⁹<https://developer.android.com/sdk/index.html>

2.2 Install Kotlin plugin

Though since IntelliJ 15 the plugin comes installed by default, it's possible that your Android Studio doesn't. So you will need to go to the plugins section inside Android Studio Preferences, and install the Kotlin plugin. Use the search tool if you can't 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.

3 Creating a new project

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

Our app is consisting on a simple weather app, such as the one used in [Google's Beginners Course in Udacity](https://www.udacity.com/course/android-development-for-beginners--ud837)¹⁰. We'll be probably paying attention to different things, but the idea of the app will be the same, because it includes many of the things you will find in a typical app. If your Android level is low I recommend this course, it's really easy to follow.

3.1 Create the project in Android Studio

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

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

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

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

3.2 Configure Gradle

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

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

¹⁰<https://www.udacity.com/course/android-development-for-beginners--ud837>

```
1  buildscript {
2      ext.support_version = '23.1.1'
3      ext.kotlin_version = '1.0.0'
4      ext.anko_version = '0.8.2'
5      repositories {
6          jcenter()
7      }
8      dependencies {
9          classpath 'com.android.tools.build:gradle:1.5.0'
10         classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
11     }
12 }
13
14 allprojects {
15     repositories {
16         jcenter()
17     }
18 }
```

As you can see, we are creating a variable which saves current Kotlin version. Check which version is available when you're reading these lines, because there's probably a new version. We need that version number in several places, for instance in the new dependency you need to add for the Kotlin plugin. You'll need it again in the module `build.gradle`, where we'll specify that this module uses the **Kotlin standard library**.

We'll do the same for the support library, as well as Anko library. This way, it's easier to modify all the versions in a row, as well as adding new libraries that use the same version without having to change the version everywhere.

We'll add the dependencies to **Kotlin standard library** and **Anko** library, as well as **Kotlin** and **Kotlin Android Extensions** plugins.

```
1  apply plugin: 'com.android.application'
2  apply plugin: 'kotlin-android'
3  apply plugin: 'kotlin-android-extensions'
4
5  android {
6      ...
7  }
8
9  dependencies {
10     compile "com.android.support:appcompat-v7:$support_version"
11     compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
```

```

12     compile "org.jetbrains.anko:anko-common:$anko_version"
13 }
14
15 buildscript {
16     repositories {
17         jcenter()
18     }
19     dependencies {
20         classpath "org.jetbrains.kotlin:kotlin-android-extensions:$kotlin_version"
21     }
22 }
23 }

```

Anko is a library that uses the power of Kotlin to simplify some tasks with Android. We'll need more Anko parts later on, but for now it's enough if we just add anko-common. This library is split into several smaller ones so that we don't need to include everything if we don't use it.

3.3 Convert MainActivity to Kotlin code

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

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

3.4 Test that everything works

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

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

```

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

```

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

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

At onCreate, you can now get access to that TextView directly:

```
1 override fun onCreate(savedInstanceState: Bundle?) {  
2     super.onCreate(savedInstanceState)  
3     setContentView(R.layout.activity_main)  
4     message.text = "Hello Kotlin!"  
5 }
```

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

Now run the app and see everything it's working fine. Check that the message TextView is showing the new content. If you have any doubts or want to review some code, take a look at [Kotlin for Android Developers repository](https://github.com/antonioolg/Kotlin-for-Android-Developers)¹¹. I'll be adding a new commit for every chapter, when the chapter implies changes in code, so be sure to review it to check all the changes.

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

¹¹<https://github.com/antonioolg/Kotlin-for-Android-Developers>

4 Classes and functions

Classes in Kotlin follow a really simple structure. However, there are some slight differences from Java that you will want to know before we continue. You can use try.kotlinlang.org¹² to test this and some other simple examples without the need of a real project.

4.1 How to declare a class

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

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

Classes have a unique default constructor. We'll see 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. Brackets are not required if the class doesn't have any content:

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

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

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

4.2 Class inheritance

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

¹²<http://try.kotlinlang.org/>

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

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

4.3 Functions

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

```
1 fun onCreate(savedInstanceState: Bundle?) {
2 }
```

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

```
1 fun add(x: Int, y: Int) : Int {
2     return x + y
3 }
```



Tip: Semi-colons are not necessary

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

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

```
1 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.


```
1 fun add(x: Int, y: Int) : Int {  
2     return x + y  
3 }
```

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

```
1 fun toast(message: String, length: Int = Toast.LENGTH_SHORT) {  
2     Toast.makeText(this, message, length).show()  
3 }
```

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

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

This would be equivalent to the next code in Java:

```
1 void toast(String message){  
2     toast(message, Toast.LENGTH_SHORT);  
3 }  
4  
5 void toast(String message, int length){  
6     Toast.makeText(this, message, length).show();  
7 }
```

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

```
1 fun niceToast(message: String,  
2     tag: String = MainActivity::class.java.simpleName,  
3     length: Int = Toast.LENGTH_SHORT) {  
4     Toast.makeText(this, "[$tag] $message", length).show()  
5 }
```

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

```
1 toast("Hello")
2 toast("Hello", "MyTag")
3 toast("Hello", "MyTag", Toast.LENGTH_SHORT)
```

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

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



Tip: String templates

You can use template expressions directly in your strings. This will make it easy to write 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, you need to use the `$` symbol. If the expression is a bit more complex, you'll need to add a couple of brackets: `"Your name is ${user.name}"`.

5 Writing your first class

We already have our `MainActivity.kt` class. This activity will render a list of daily forecast for the next week, 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:

```
1 dependencies {
2     compile fileTree(dir: 'libs', include: ['*.jar'])
3     compile "com.android.support:appcompat-v7:$support_version"
4     compile "com.android.support:recyclerview-v7:$support_version"
5     ...
6 }
```

Now, in `activity_main.xml` :

```
1 <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
2             android:layout_width="match_parent"
3             android:layout_height="match_parent">
4
5     <android.support.v7.widget.RecyclerView
6         android:id="@+id/forecast_list"
7         android:layout_width="match_parent"
8         android:layout_height="match_parent"/>
9
10 </FrameLayout>
```

In `MainActivity.kt`, remove the line we added to test everything worked (it will be showing an error now). We'll continue using the good old `findViewById()` for the time being:

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

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



Object instantiation

Object instantiation presents some differences from Java too. As you can see, we omit the “new” word. The constructor call is still there, but we save four precious characters. `LinearLayoutManager(this)` creates an instance of the object.

5.2 The Recycler Adapter

We need an adapter for the recycler too. I [talked about RecyclerView on my blog¹³](#) 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'll create manually. Add a new Kotlin file called `ForecastListAdapter.kt`, and include this code:

```
1 class ForecastListAdapter(val items: List<String>) :  
2     RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {  
3  
4     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
5         ViewHolder {  
6         return ViewHolder(Textview(parent.context))  
7     }  
8  
9     override fun onBindViewHolder(holder: ViewHolder,  
10        position: Int) {  
11        holder.textview.text = items[position]  
12    }  
13  
14    override fun getItemCount(): Int = items.size  
15  
16    class ViewHolder(val textview: Textview) : RecyclerView.ViewHolder(textview)  
17 }
```

Again, we can access to the context and the text as properties. You can keep doing it as usual (using getters and setter), but you'll get a warning from the compiler. This check can be disabled if you prefer to keep using the Java way. Once you get used to properties you will love them anyway.

¹³<http://antonioleiva.com/recyclerview/>



Default visibility is `public`

Unless some 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.

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

```
1 private val items = listOf(  
2     "Mon 6/23 - Sunny - 31/17",  
3     "Tue 6/24 - Foggy - 21/8",  
4     "Wed 6/25 - Cloudy - 22/17",  
5     "Thurs 6/26 - Rainy - 18/11",  
6     "Fri 6/27 - Foggy - 21/10",  
7     "Sat 6/28 - TRAPPED IN WEATHERSTATION - 23/18",  
8     "Sun 6/29 - Sunny - 20/7"  
9 )  
10  
11 override fun onCreate(savedInstanceState: Bundle?) {  
12     ...  
13     val forecastList = findViewById(R.id.forecast_list) as RecyclerView  
14     forecastList.layoutManager = LinearLayoutManager(this)  
15     forecastList.adapter = ForecastListAdapter(items)  
16 }
```



List creation

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

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

I also moved some classes to new packages, in order to improve organisation.

We reviewed many new ideas in such a small amount of code, so I'll be covering them in the next chapter. We can't continue until we learn some important concepts regarding basic types, variables and properties.

6 Variables and properties

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

6.1 Basic types

Of course, basic types such as integers, floats, characters or booleans still exist, but they all act as an object. 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:

```
1 val i: Int = 7
2 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:

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

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

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

```
1 // Kotlin
2 val bitwiseOr = FLAG1 or FLAG2
3 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](http://kotlinlang.org/docs/reference/basic-types.html#operations)¹⁴ for more information.

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

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

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

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

```
1 // Iterate over String
2 val s = "Example"
3 for (c in s) {
4     print(c)
5 }
```

6.2 Variables

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

¹⁴<http://kotlinlang.org/docs/reference/basic-types.html#operations>

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

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

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

Another thing mentioned before is that we usually don't need to specify object types, they will be inferred from the value, which makes the code cleaner and faster to modify. We already have some examples from the section above.

```
1 val s = "Example" // A String
2 val i = 23 // An Int
3 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:

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

6.3 Properties

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

```
1 public class Person {
2
3     private String name;
4
5     public String getName() {
6         return name;
7     }
8
9     public void setName(String name) {
10        this.name = name;
11    }
```



```
11     }
12 }
13
14 ...
15
16 Person person = new Person();
17 person.setName("name");
18 String name = person.getName();
```

In Kotlin, only a property is required:

```
1 public class Person {
2
3     var name: String = ""
4
5 }
6
7 ...
8
9 val person = Person()
10 person.name = "name"
11 val name = person.name
```

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

```
1 public class Person {
2
3     var name: String = ""
4     get() = field.toUpperCase()
5     set(value) {
6         field = "Name: $value"
7     }
8
9 }
```

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

As mentioned in some previous chapters, when operating with Java code Kotlin will allow to use the property syntax where a getter and a setter are defined in Java. The compiler will just link to the original getters and setters, so there are no performance penalties when using these mapped properties.

7 Anko and Extension Functions

7.1 What is Anko?

[Anko¹⁵](https://github.com/JetBrains/anko) is a powerful library developed by JetBrains. Its main purpose is the generation of UI layouts by using code instead of XML. This is an interesting feature I recommend you to try, but I won't be using it in this project. To me (probably due to years of experience writing UIs) using XML is much easier, but you could like this approach.

However, this is not the only feature we can get from this library. Anko includes a lot of extremely helpful functions and properties that will avoid lots of boilerplate. We will see many examples throughout this book, but you'll quickly see which kind of problems this library solves.

Though Anko is really helpful, I recommend you to understand what it is doing behind the scenes. You can navigate at any moment to Anko source code using `ctrl + click` (Windows) or `cmd + click` (Mac). Anko implementation is really helpful 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 simplify some code. As you will see, anytime you use something from Anko, it will include an import with the name of the property or function to the file. This is because Anko uses **extension functions** to add new features to Android framework. We'll see right below what an extension function is and how to write it.

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

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

We can't use more from the library yet, but Anko can help us to simplify, among others, the instantiation of intents, the navigation between activities, creation of fragments, database access, alerts creation... We'll find lots of interesting examples while we implement the App.

¹⁵<https://github.com/JetBrains/anko>

7.3 Extension functions

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

For instance, we can create a toast function which doesn't ask for the context, which could be used by any `Context` objects, and those whose type extends `Context`, such as `Activity` or `Service`:

```
1 fun Context.toast(message: CharSequence, duration: Int = Toast.LENGTH_SHORT) {  
2     Toast.makeText(this, message, duration).show()  
3 }
```

This function can be used inside an activity, for instance:

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

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

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

Extension functions can also be properties. So you can create extension properties too in a very similar way. The following example is showing a way to generate a property with its own getters and setters. Kotlin already provides this property for us as an interoperability feature, but it's a good exercise to understand the idea behind extension properties:

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

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

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

8 Retrieving data from API

8.1 Performing a request

Our current placeholder texts are good to start feeling the idea of what we want to achieve, but now it's time to request some real data, which will be used to populate the `RecyclerView`. We'll be using [OpenWeatherMap](http://openweathermap.org/)¹⁶ 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](https://github.com/square/retrofit)¹⁷, for server requests. However, as we are just performing a simple API request, we can easily 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 easier. First, we're going to create a new `Request` class:

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

Our request simply receives an url, reads the result and outputs the json in the Logcat. The implementation is really easy when using `readText`, an extension function from the Kotlin standard library. This method is not recommended for huge responses, but it will be good enough in our case.

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

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

¹⁶<http://openweathermap.org/>

¹⁷<https://github.com/square/retrofit>

```
1 <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 to be done in the main thread, it will throw an exception. This is because blocking the UI thread is a really bad practice. The common solution in Android is to use an `AsyncTask`. But these classes are ugly and difficult to implement without any side effects. `AsyncTasks` are dangerous if not used carefully, because by the time it reaches `postExecute` the activity could have been destroyed, and the task will crash.

Anko provides a very easy DSL to deal with asynchrony, which will fit most basic needs. It basically provides an `async` function that will execute 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:

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

A nice thing about `uiThread` is that it's implemented differently depending on the caller object. If it's used by an `Activity`, the `uiThread` code won't be executed if `activity.isFinishing()` returns `true`, and it won't crash if the activity is no longer valid.

You also can use your own executor:

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

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

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

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

9 Data Classes

Data classes are a powerful kind of classes which avoid the boilerplate we need in Java to create POJO: classes which are used to keep state, but are very simple 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:

```
1 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 interesting functions for free, apart from the properties we already talked about (which prevents us from writing getters and setters):

- `equals()`: it compares the properties 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'll see an example later.
- A set of numbered functions that are useful to map an object into variables. It will also be explained soon.

9.2 Copying a data class

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

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

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

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



Be careful with immutability when using Java classes

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

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

9.3 Mapping an object into variables

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

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

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

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

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

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


10 Parsing data

10.1 Converting json to data classes

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

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

```
1 data class ForecastResult(val city: City, val list: List<Forecast>)
2
3 data class City(val id: Long, val name: String, val coord: Coordinates,
4                 val country: String, val population: Int)
5
6 data class Coordinates(val lon: Float, val lat: Float)
7
8 data class Forecast(val dt: Long, val temp: Temperature, val pressure: Float,
9                     val humidity: Int, val weather: List<Weather>,
10                     val speed: Float, val deg: Int, val clouds: Int,
11                     val rain: Float)
12
13 data class Temperature(val day: Float, val min: Float, val max: Float,
14                       val night: Float, val eve: Float, val morn: Float)
15
16 data class Weather(val id: Long, val main: String, val description: String,
17                   val icon: String)
```

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

Now, the `Request` class needs some modifications in order to return the parsed result. It will also receive only the zipcode of the city instead of the complete url, so that it becomes more readable. For now, the static url will belong to a **companion object**. Maybe we need to extract it later if we create more requests against another endpoint of the API.



Companion objects

Kotlin allows to declare objects to define static behaviours. In Kotlin, we can't create static properties or functions, but we need to rely on objects. However, these objects make some well known patterns such as `Singleton` very easy to implement.

If we need some static properties, constants or functions in a class, we can use a **companion object**. This object will be shared among all instances of the class, the same as a static field or method would do in Java.

Check the resulting code:

```

1  class ForecastRequest(val zipCode: String) {
2
3      companion object {
4          private val APP_ID = "15646a06818f61f7b8d7823ca833e1ce"
5          private val URL = "http://api.openweathermap.org/data/2.5/" +
6              "forecast/daily?mode=json&units=metric&cnt=7"
7          private val COMPLETE_URL = "$URL&APPID=$APP_ID&q="
8      }
9
10     fun execute(): ForecastResult {
11         val forecastJsonStr = URL(COMPLETE_URL + zipCode).readText()
12         return Gson().fromJson(forecastJsonStr, ForecastResult::class.java)
13     }
14 }

```

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

```

1  compile "com.google.code.gson:gson:2.4"

```

10.2 Shaping the domain layer

Now we'll create a new package representing the domain layer. It will contain some `Commands` in charge of performing the use cases of the app.

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

```

1 public interface Command<T> {
2     fun execute(): T
3 }

```

These commands will execute an operation and return an object of the class specified in its generic type. It's interesting to know that **every function in Kotlin returns a value**. By default, if nothing is specified, it will return an object of the `Unit` class. So if we want our `Command` to return nothing, we can specify `Unit` as its type.

Interfaces in Kotlin are more powerful than Java (prior to Java 8), because they can contain code. But for now, we don't need more than what we could do in a Java interface. Future chapters will elaborate this topic further.

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:

```

1 data class ForecastList(val city: String, val country: String,
2                         val dailyForecast: List<Forecast>)
3
4 data class Forecast(val date: String, val description: String, val high: Int,
5                   val low: Int)

```

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

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

```

1 public class ForecastDataMapper {
2
3     fun convertFromDataModel(forecast: ForecastResult): ForecastList {
4         return ForecastList(forecast.city.name, forecast.city.country,
5                             convertForecastListToDomain(forecast.list))
6     }
7
8     private fun convertForecastListToDomain(list: List<Forecast>):
9         List<ModelForecast> {
10         return list.map { convertForecastItemToDomain(it) }
11     }
12
13     private fun convertForecastItemToDomain(forecast: Forecast): ModelForecast {
14         return ModelForecast(convertDate(forecast.dt),
15                               forecast.weather[0].description, forecast.temp.max.toInt(),

```

```

16         forecast.temp.min.toInt())
17     }
18
19     private fun convertDate(date: Long): String {
20         val df = DateFormat.getDateInstance(DateFormat.MEDIUM,
21             Locale.getDefault())
22         return df.format(date * 1000)
23     }
24 }

```

As we are using two classes with the same name, we can give a specific name to one of them so that we don't need to write the complete package:

```

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

```

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

```

1 return list.map { convertForecastItemToDomain(it) }

```

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

And now, everything is ready to write the command:

```

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

```

10.3 Drawing the data in the UI

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

```

1  async() {
2      val result = RequestForecastCommand("94043").execute()
3      uiThread {
4          forecastList.adapter = ForecastListAdapter(result)
5      }
6  }

```

The adapter needs some modifications too:

```

1  class ForecastListAdapter(val weekForecast: ForecastList) :
2      RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {
3
4      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
5          ViewHolder? {
6          return ViewHolder(Textview(parent.getContext()))
7      }
8
9      override fun onBindViewHolder(holder: ViewHolder,
10          position: Int) {
11          with(weekForecast.dailyForecast[position]) {
12              holder.textview.text = "$date - $description - $high/$low"
13          }
14      }
15
16      override fun getItemCount(): Int = weekForecast.dailyForecast.size
17
18      class ViewHolder(val textview: Textview) : RecyclerView.ViewHolder(textview)
19  }

```



with function

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

There's a lot of new code in this chapter, so feel free to check it out on the [repository](https://github.com/antonioIlg/Kotlin-for-Android-Developers)¹⁸.

¹⁸<https://github.com/antonioIlg/Kotlin-for-Android-Developers>

11 Operator overloading

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

In order to alert the compiler we want to overload an operator, functions must be annotated 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

<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>
<code>a++</code>	<code>a.inc()</code>
<code>a--</code>	<code>a.dec()</code>

Binary operations

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

Array-like operations

<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

Equals operation

<code>a == b</code>	<code>a?.equals(b) ?: b === null</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: b === null)</code>

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

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

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

Function invocation

<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i, j)</code>	<code>a.invoke(i, j)</code>
<code>a(i_1, ..., i_n)</code>	<code>a.invoke(i_1, ..., i_n)</code>

11.2 The example

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

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

If you remember, we have a data class called `ForecastList`, which basically consists of a list with some extra info. It'd be interesting to access its items directly instead of having to request its internal

list to get an item. On a totally unrelated note, I'm also going to implement a `size()` function, which will simplify the current adapter a little more:

```
1 data class ForecastList(val city: String, val country: String,
2                         val dailyForecast: List<Forecast>) {
3     operator fun get(position: Int): Forecast = dailyForecast[position]
4     fun size(): Int = dailyForecast.size
5 }
```

It makes our `onBindViewHolder` a bit simpler:

```
1 override fun onBindViewHolder(holder: ViewHolder, position: Int) {
2     with(weekForecast[position]) {
3         holder.textView.text = "$date - $description - $high/$low"
4     }
5 }
```

As well as the `getItemCount()` function:

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

11.3 Operators in extension functions

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

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

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

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


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`:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      android:padding="@dimen/spacing_xlarge"
8      android:background="?attr/selectableItemBackground"
9      android:gravity="center_vertical"
10     android:orientation="horizontal">
11
12     <ImageView
13         android:id="@+id/icon"
14         android:layout_width="48dp"
15         android:layout_height="48dp"
16         tools:src="@mipmap/ic_launcher"/>
17
18     <LinearLayout
19         android:layout_width="0dp"
20         android:layout_height="wrap_content"
21         android:layout_weight="1"
22         android:layout_marginLeft="@dimen/spacing_xlarge"
23         android:layout_marginRight="@dimen/spacing_xlarge"
24         android:orientation="vertical">
25
26         <TextView
27             android:id="@+id/date"
28             android:layout_width="match_parent"
29             android:layout_height="wrap_content"
30             android:textAppearance="@style/TextAppearance.AppCompat.Medium"
31             tools:text="May 14, 2015"/>
32
33         <TextView
```

```

34         android:id="@+id/description"
35         android:layout_width="match_parent"
36         android:layout_height="wrap_content"
37         android:textAppearance="@style/TextAppearance.AppCompat.Caption"
38         tools:text="Light Rain"/>
39
40     </LinearLayout>
41
42     <LinearLayout
43         android:layout_width="wrap_content"
44         android:layout_height="wrap_content"
45         android:gravity="center_horizontal"
46         android:orientation="vertical">
47
48         <TextView
49             android:id="@+id/maxTemperature"
50             android:layout_width="wrap_content"
51             android:layout_height="wrap_content"
52             android:textAppearance="@style/TextAppearance.AppCompat.Medium"
53             tools:text="30"/>
54
55         <TextView
56             android:id="@+id/minTemperature"
57             android:layout_width="wrap_content"
58             android:layout_height="wrap_content"
59             android:textAppearance="@style/TextAppearance.AppCompat.Caption"
60             tools:text="15"/>
61
62     </LinearLayout>
63
64 </LinearLayout>

```

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

```

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

```

In ForecastDataMapper:

```

1 private fun convertForecastItemToDomain(forecast: Forecast): ModelForecast {
2     return ModelForecast(convertDate(forecast.dt),
3         forecast.weather[0].description, forecast.temp.max.toInt(),
4         forecast.temp.min.toInt(), generateIconUrl(forecast.weather[0].icon))
5 }
6
7 private fun generateIconUrl(iconCode: String): String
8     = "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. [Picasso¹⁹](https://square.github.io/picasso/) is a really good option. It must be added to build.gradle dependencies:

```

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

```

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

```

1 interface OnItemClickListener {
2     operator fun invoke(forecast: Forecast)
3 }

```

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

```

1 itemClick.invoke(forecast)
2 itemClick(forecast)

```

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

```

1 class ViewHolder(view: View, val itemClick: OnItemClickListener) :
2     RecyclerView.ViewHolder(view) {
3
4     private val iconView: ImageView
5     private val dateView: TextView
6     private val descriptionView: TextView
7     private val maxTemperatureView: TextView
8     private val minTemperatureView: TextView
9
10    init {

```

¹⁹[http://square.github.io/picasso/](https://square.github.io/picasso/)

```

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

```

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

```

1 public class ForecastListAdapter(val weekForecast: ForecastList,
2     val itemClick: ForecastListAdapter.OnItemClickListener) :
3     RecyclerView.Adapter<ForecastListAdapter.ViewHolder>() {
4
5     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
6         ViewHolder {
7         val view = LayoutInflater.from(parent.ctx)
8             .inflate(R.layout.item_forecast, parent, false)
9
10        return ViewHolder(view, itemClick)
11    }
12
13    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
14        holder.bindForecast(weekForecast[position])
15    }
16    ...
17 }

```

If you use this code, `parent.ctx` won't compile. Anko provides a lot of extension functions to make Android coding simpler. It, for instance, includes a `ctx` property for activities and fragments, among

others, which returns the context, but it lacks of the same property for views. So we are going to create a new file called `ViewExtensions.kt` inside `ui.utils`, and add this extension property:

```
1 val View.ctx: Context
2     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's a good example of how to use extension properties.

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

```
1 forecastList.adapter = ForecastListAdapter(result,
2     object : ForecastListAdapter.OnItemClickListener{
3         override fun invoke(forecast: Forecast) {
4             toast(forecast.date)
5         }
6     })
```

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

Try the new changes from the repository. The UI starts looking much nicer.

13 Lambdas

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

13.1 Simplifying `setOnClickListener()`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

More than five times smaller than the original code in Java, and much easier to understand what is doing. Really impressive.

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's time to put what you learnt into practice. We are removing the listener interface from the `ForecastListAdapter` and using a lambda instead:

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

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

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

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

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

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

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

13.3 Extending the language

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

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

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

```
1 with(forecast) {
2     Picasso.with(itemView.ctx).load(iconUrl).into(iconView)
3     dateView.text = date
4     descriptionView.text = description
5     maxTemperatureView.text = "$high"
6     minTemperatureView.text = "$low"
7     itemView.setOnClickListener { itemClick(this) }
8 }
```



Inline functions

Inline functions are a bit different from regular functions. An inline function will be substituted by its code during compilation, instead of really calling to a function. It will reduce memory allocations and runtime overhead in some situations. For instance, if we have a function as an argument, a regular function will internally create an object that contains that function. On the other hand, inline functions will substitute the code of the function in the place where it is called, so it won't require an internal object for that.

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


```
1 inline fun supportsLollipop(code: () -> Unit) {  
2     if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {  
3         code()  
4     }  
5 }
```

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

```
1 supportsLollipop {  
2     window.setStatusBarColor(Color.BLACK)  
3 }
```

For instance, Anko is also based on this idea to implement the DSL for Android layouts. You can also check an example from Kotlin reference, where a [DSL to write HTML](http://kotlinlang.org/docs/reference/type-safe-builders.html)²⁰ from code is created.

²⁰<http://kotlinlang.org/docs/reference/type-safe-builders.html>

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 save a lot of time and characters. But here it is the long explanation. How do the different visibility modifiers work in Kotlin?

14.1 Modifiers

private

The `private` modifier is the most restrictive we can use. It implies it will only be visible inside its own file. So if we declare a class as `private`, we won't be 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 won't be able to 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 will only be visible by that class or interface.

protected

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

internal

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

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

public

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

14.2 Constructors

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

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

14.3 Revising our code

We've already been making use of the `public` default refactor, but there are many other details we could change. For instance, in `RequestForecastCommand`, the property we create from the `zipCode` constructor parameter could be private.

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

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

Besides, in Kotlin we don't need to specify the return type of a function if it can be computed by the compiler. An example of how we can get rid of the returning types:

```
1 data class ForecastList(...) {  
2     fun get(position: Int) = dailyForecast[position]  
3     fun size() = dailyForecast.size()  
4 }
```

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

The rest of the modifications are quite straightforward, so you can check them in the repository.

15 Kotlin Android Extensions

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

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

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

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

15.1 How to use Kotlin Android Extensions

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

```
1  apply plugin: 'kotlin-android-extensions'
2
3  ...
4
5  buildscript {
6      repositories {
7          jcenter()
8      }
9      dependencies {
10         classpath "org.jetbrains.kotlin:kotlin-android-extensions:$kotlin_version"
11     }
12 }
13 }
```

The only thing required by the plugin is the addition of a special “synthetic” import to the class that makes use of this feature. We have a couple of ways to use it:

Android Extensions for Activities or Fragments

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

The import we need to use will start with `kotlinx.android.synthetic` plus the name of the XML we want to bind to the activity. We also have to specify the build variant:

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

From that moment, we can access the views after `setContentView` is called. New Android Studio versions are adding nested layouts to default activity templates, by using `include` tag. It's important to know that we'll need to add a synthetic import for any XMLs we use:

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

Android Extensions for Views

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

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

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

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

15.2 Refactoring our code

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

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

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

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

```
1 <FrameLayout
2     xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent">
5
6     <android.support.v7.widget.RecyclerView
7         android:id="@+id/forecastList"
8         android:layout_width="match_parent"
9         android:layout_height="match_parent"/>
10
11 </FrameLayout>
```

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

```
1 override fun onCreate(savedInstanceState: Bundle?) {
2     super.onCreate(savedInstanceState)
3     setContentView(R.layout.activity_main)
4
5     forecastList.layoutManager = LinearLayoutManager(this)
6     ...
7 }
```

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

First, add the synthetic import for item_forecast:

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

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

We don't need to declare properties for the views anymore, we now can just use them:

```
1 class ViewHolder(view: View, val itemClick: (Forecast) -> Unit) :  
2     RecyclerView.ViewHolder(view) {  
3  
4     fun bindForecast(forecast: Forecast) {  
5         with(forecast) {  
6             Picasso.with(itemView.ctx).load(iconUrl).into(itemView.icon)  
7             itemView.date.text = date  
8             itemView.description.text = description  
9             itemView.maxTemperature.text = "$high"  
10            itemView.minTemperature.text = "$low"  
11            itemView.setOnItemClickListener { itemClick(this) }  
12        }  
13    }  
14 }
```

Kotlin Android Extensions plugin helps us reduce some more boilerplate and minimise the code required to access our views. Check the latest changes at the repository.

16 Application Singleton and Delegated Properties

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

16.1 Application Singleton

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

```
1 class App : Application() {
2
3     companion object {
4         private var instance: Application? = null
5         fun instance() = instance!!
6     }
7
8     override fun onCreate() {
9         super.onCreate()
10        instance = this
11    }
12 }
```

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

```
1 <application
2     android:allowBackup="true"
3     android:icon="@mipmap/ic_launcher"
4     android:label="@string/app_name"
5     android:theme="@style/AppTheme"
6     android:name=".ui.App">
7     ...
8 </application>
```


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

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

16.2 Delegated Properties

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

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

The structure of a property delegate is:

```
1 class Delegate<T> : ReadWriteProperty<Any?, T> {
2     fun getValue(thisRef: Any?, property: KProperty<*>): T {
3         return ...
4     }
5
6     fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
7         ...
8     }
9 }
```

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

This is how the property delegate is assigned:

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

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

16.3 Standard Delegates

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

Lazy

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

```
1 class App : Application() {  
2     val database: SQLiteOpenHelper by lazy {  
3         MyDatabaseHelper(applicationContext)  
4     }  
5  
6     override fun onCreate() {  
7         super.onCreate()  
8         val db = database.writableDatabase  
9     }  
10 }
```

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

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

Observable

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

```
1 class ViewModel(val db: MyDatabase) {
2
3     var myProperty by Delegates.observable("") {
4         d, old, new ->
5             db.saveChanges(this, new)
6     }
7
8 }
```

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

Vetoable

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

```
1 var positiveNumber = Delegates.vetoable(0) {
2     d, old, new ->
3     new >= 0
4 }
```

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

lateinit

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

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

The second option is to use `lateinit`, which identifies that the property should have a non-nullable value, but its assignment will be delayed. If the value is requested before it is assigned, it will throw an exception that clearly identifies the property being accessed.

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

This could be helpful in the App singleton example:

```
1 class App : Application() {
2
3     companion object {
4         lateinit var instance: App
5     }
6
7     override fun onCreate() {
8         super.onCreate()
9         instance = this
10    }
11 }
```

lateinit is also indispensable when working with a dependency injector, 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 let us do really powerful things, because we can easily 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 will require a `MutableMap` as constructor parameter instead.

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

```
1 class Configuration(map: Map<String, Any?>) {
2     val width: Int by map
3     val height: Int by map
4     val dp: Int by map
5     val deviceName: String by map
6 }
```

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

```
1 val conf = Configuration(mapOf(  
2     "width" to 1080,  
3     "height" to 720,  
4     "dp" to 240,  
5     "deviceName" to "mydevice"  
6 ))
```

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. Second time it's assigned, it will throw an exception.

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

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

```
1 private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any?, T> {  
2  
3     override fun getValue(thisRef: Any?, property: KProperty<*>): T {  
4         throw UnsupportedOperationException()  
5     }  
6  
7     override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {  
8  
9     }  
10 }
```

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

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

```

1 private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any?, T> {
2
3     private var value: T? = null
4
5     override fun getValue(thisRef: Any?, property: KProperty<*>): T {
6         return value ?: throw IllegalStateException("${desc.name} " +
7             "not initialized")
8     }
9
10    override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
11        this.value = if (this.value == null) value
12        else throw IllegalStateException("${desc.name} already initialized")
13    }
14 }

```

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

```

1 object DelegatesExt {
2     fun notNullSingleValue<T>():
3         ReadWriteProperty<Any?, T> = NotNullSingleValueVar()
4 }

```

16.5 Reimplementing App Singleton

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

```

1 class App : Application() {
2
3     companion object {
4         lateinit var instance: App
5     }
6
7     override fun onCreate() {
8         super.onCreate()
9         instance = this
10    }
11 }

```

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's easy to solve by using a private `set`:

```
1 companion object {  
2     lateinit var instance: App  
3     private set  
4 }
```

But we'll be making use of our custom delegate instead:

```
1 companion object {  
2     var instance: App by DelegatesExt.notNullSingleValue()  
3 }
```

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

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's really lightweight. That's why it is a good option for mobile Apps.

However, the API to work with databases in Android is quite raw. You'll see you need to write many SQL sentences and 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. But it's possible you don't need them for simple databases, 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 shouldn't forget to call `close()`. With a `ManagedSQLiteOpenHelper` we just do:

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

Inside the lambda we can use `SQLiteDatabase` functions directly. How does it work? It's really interesting to read the implementation of Anko functions, you can learn a lot of Kotlin from it:

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

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


```
1 val result = forecastDbHelper.use {  
2     val queriedObject = ...  
3     queriedObject  
4 }
```

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

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

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

17.2 Tables definition

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

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

```
1 object CityForecastTable {  
2     val NAME = "CityForecast"  
3     val ID = "_id"  
4     val CITY = "city"  
5     val COUNTRY = "country"  
6 }
```

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

```
1 object DayForecastTable {  
2     val NAME = "DayForecast"  
3     val ID = "_id"  
4     val DATE = "date"  
5     val DESCRIPTION = "description"  
6     val HIGH = "high"  
7     val LOW = "low"  
8     val ICON_URL = "iconUrl"  
9     val CITY_ID = "cityId"  
10 }
```

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:

```
1 dependencies {
2     ...
3     compile "org.jetbrains.anko:anko-sqlite:$anko_version"
4 }
```

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:

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

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

```
1 companion object {
2     val DB_NAME = "forecast.db"
3     val DB_VERSION = 1
4     val instance by lazy { ForecastDbHelper() }
5 }
```

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

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

```

1 db.createTable(CityForecastTable.NAME, true,
2     Pair(CityForecastTable.ID, INTEGER + PRIMARY_KEY),
3     Pair(CityForecastTable.CITY, TEXT),
4     Pair(CityForecastTable.COUNTRY, TEXT))

```

- The first parameter is the name of the table.
- The second parameter, when set to true, will check if the table doesn't exist before trying to create it.
- The third parameter is a vararg of Pairs. The vararg type also exists in Java, and it's a way to pass a variable number of arguments of the same type to a function. The function will receive an array with the objects.

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

```

1 fun SqlType.plus(m: SqlTypeModifier) : SqlType {
2     return SqlTypeImpl(name, if (modifier == null) m.toString()
3         else "$modifier $m")
4 }

```

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

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

```

1 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:

```

1 val pair = object1 to object2

```

And this, applied to the creation of our tables:

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

This is how the whole method looks:

```
1 override fun onCreate(db: SQLiteDatabase) {
2     db.createTable(CityForecastTable.NAME, true,
3         CityForecastTable.ID to INTEGER + PRIMARY_KEY,
4         CityForecastTable.CITY to TEXT,
5         CityForecastTable.COUNTRY to TEXT)
6
7     db.createTable(DayForecastTable.NAME, true,
8         DayForecastTable.ID to INTEGER + PRIMARY_KEY + AUTOINCREMENT,
9         DayForecastTable.DATE to INTEGER,
10        DayForecastTable.DESCRPTION to TEXT,
11        DayForecastTable.HIGH to INTEGER,
12        DayForecastTable.LOW to INTEGER,
13        DayForecastTable.ICON_URL to TEXT,
14        DayForecastTable.CITY_ID to INTEGER)
15 }
```

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

```
1 override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
2     db.dropTable(CityForecastTable.NAME, true)
3     db.dropTable(DayForecastTable.NAME, true)
4     onCreate(db)
5 }
```

17.4 Dependency injection

Although I try not to add much complexity to the code regarding architectures, clean testable code or good practices, I thought it'd be a good idea to show another way to simplify our code using Kotlin. If you want to know a little more about topics like dependency inversion or injection, you

can check my set of articles about [dependency injection in Android using Dagger](#)²¹. The first article covers a simple explanation about these terms.

In a simple way, if we want to have classes that are independent of other classes, way more testable, and write code 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.

But now those dependencies must be provided from somewhere, so the dependency injection consists of providing the collaborators required by the classes. This is usually done by using a dependency injector. [Dagger](#)²² is probably the most popular dependency injector in Android. It is, of course, a very good alternative when we need some complexity to provide those dependencies.

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

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

Now we have two ways to create this class:

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

I'll be using this mechanism here and there, so I didn't want to continue without explaining the reason. We already have the tables, so it's time to start adding and requesting data from them. But before that, I want to talk about collections and functional operations. Don't forget checking the repository to find latest changes.

²¹<http://antonioleiva.com/dependency-injection-android-dagger-part-1/>

²²<http://square.github.io/dagger/>

18 Collections and functional operations

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

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

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

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

18.1 Aggregate operations

any

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

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

all

Returns true if all the elements match the given predicate.

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

count

Returns the number of elements matching the given predicate.

```
1 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.

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

foldRight

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

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

forEach

Performs the given operation to each element.

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

forEachIndexed

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

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

max

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

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

maxBy

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

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

min

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

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

minBy

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

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

none

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


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

reduce

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

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

reduceRight

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

```
1 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.

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

18.2 Filtering operations

drop

Returns a list containing all elements except first n elements.

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

dropWhile

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

```
1 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.

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

filter

Returns a list containing all elements matching the given predicate.

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

filterNot

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

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

filterNotNull

Returns a list containing all elements that are not null.

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

slice

Returns a list containing elements at specified indices.

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

take

Returns a list containing first n elements.

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

takeLast

Returns a list containing last n elements.

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

takeWhile

Returns a list containing first elements satisfying the given predicate.

```
1 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.

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

groupBy

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

```
1 assertEquals(mapOf("odd" to listOf(1, 3, 5), "even" to listOf(2, 4, 6)),  
2 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.

```
1 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.

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

mapNotNull

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

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

18.4 Elements operations

contains

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

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

elementAt

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

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

elementAtOrElse

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

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

elementOrNull

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

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

first

Returns the first element matching the given predicate

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

firstOrNull

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

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

indexOf

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

```
1 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.

```
1 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.

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

last

Returns the last element matching the given predicate.

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

lastIndexOf

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

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

lastOrNull

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

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

single

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

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

singleOrNull

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

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

18.5 Generation operations

merge

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

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

partition

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

```
1 assertEquals(Pair(listOf(2, 4, 6), listOf(1, 3, 5)),
2     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.

```
1 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.

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

unzip

Generates a Pair of Lists from a List of Pairs

```
1 assertEquals(Pair(listOf(5, 6), listOf(7, 8)), listOf(Pair(5, 7), Pair(6, 8)).un\
2 zip())
```

18.6 Ordering operations

reverse

Returns a list with elements in reversed order.

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

sort

Returns a sorted list of all elements.

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

sortBy

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

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

sortDescending

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

```
1 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.

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


19 Saving and requesting data from database

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

19.1 Creating database model classes

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

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

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

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

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

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

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

```
1 class DayForecast(var map: MutableMap<String, Any?>) {
2     var _id: Long by map
3     var date: Long by map
4     var description: String by map
5     var high: Int by map
6     var low: Int by map
7     var iconUrl: String by map
8     var cityId: Long by map
9
10    constructor(date: Long, description: String, high: Int, low: Int,
11                iconUrl: String, cityId: Long) : this(HashMap()) {
12        this.date = date
13        this.description = description
14        this.high = high
15        this.low = low
16        this.iconUrl = iconUrl
17        this.cityId = cityId
18    }
19 }
```

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

19.2 Writing and requesting data

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

```
1 class ForecastDb(  
2     val forecastDbHelper: ForecastDbHelper = ForecastDbHelper.instance,  
3     val dataMapper: DbDataMapper = DbDataMapper()) {  
4     ...  
5 }
```

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

```
1 fun requestForecastByZipCode(zipCode: Long, date: Long) = forecastDbHelper.use {  
2     ....  
3 }
```

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:

```
1 val dailyRequest = "${DayForecastTable.CITY_ID} = ? " +  
2     "AND ${DayForecastTable.DATE} >= ?"  
3  
4 val dailyForecast = select(DayForecastTable.NAME)  
5     .whereSimple(dailyRequest, zipCode.toString(), date.toString())  
6     .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's very similar to what we'd do in a regular use of the helper. There is another function called `simple where`, which takes some tags and values and match them. I don't 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. This is how it would look with it:

```

1 val dailyRequest = "${DayForecastTable.CITY_ID} = {id}" +
2   "AND ${DayForecastTable.DATE} >= {date}"
3
4 val dailyForecast = select(DayForecastTable.NAME)
5   .where(dailyRequest, "id" to zipCode, "date" to date)
6   .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 magic happens. In this case we are using the function `parseList`, which assumes we are requesting a list of items. It uses a `RowParser` or `MapRowParser` to convert the cursor into a list of object. The difference between both is that the `RowParser` relies on the order of the columns, while the `MapRowParser` uses the name of the column as the key of the map.

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

```

1 fun <T : Any> SelectQueryBuilder.parseList(
2   parser: (Map<String, Any>) -> T): List<T> =
3   parseList(object : MapRowParser<T> {
4     override fun parseRow(columns: Map<String, Any>): T = parser(columns)
5   })

```

This function helps simplify the `parseList` request to:

```

1 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.

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

```

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

```

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

```

1 public fun <T : Any> SelectQueryBuilder.parseOpt(
2     parser: (Map<String, Any>) -> T): T? =
3     parseOpt(object : MapRowParser<T> {
4         override fun parseRow(columns: Map<String, Any>): T = parser(columns)
5     })

```

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?`:

```

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

```

`DataMapper` function is easy:

```

1 fun convertToDomain(forecast: CityForecast) = with(forecast) {
2     val daily = dailyForecast.map { convertDayToDomain(it) }
3     ForecastList(_id, city, country, daily)
4 }
5
6 private fun convertDayToDomain(dayForecast: DayForecast) = with(dayForecast) {
7     Forecast(date, description, high, low, iconUrl)
8 }

```

So this is how the complete function looks like:

```

1 fun requestForecastByZipCode(zipCode: Long, date: Long) = forecastDbHelper.use {
2
3     val dailyRequest = "${DayForecastTable.CITY_ID} = ? AND " +
4         "${DayForecastTable.DATE} >= ?"
5     val dailyForecast = select(DayForecastTable.NAME)
6         .whereSimple(dailyRequest, zipCode.toString(), date.toString())
7         .parseList { DayForecast(HashMap(it)) }
8
9     val city = select(CityForecastTable.NAME)
10        .whereSimple("${CityForecastTable.ID} = ?", zipCode.toString())
11        .parseOpt { CityForecast(HashMap(it), dailyForecast) }
12
13     if (city != null) dataMapper.convertToDomain(city) else null
14 }

```

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

Saving a forecast

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

```

1 fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {
2     ...
3 }

```

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

```

1 fun SQLiteDatabase.clear(tableName: String) {
2     execSQL("delete from $tableName")
3 }

```

The function is applied to both tables:

```

1 clear(CityForecastTable.NAME)
2 clear(DayForecastTable.NAME)

```

Now it's time to convert the data, and use the result to execute the insert queries. At this point you probably know I'm a fan of the with function:

```

1 with(dataMapper.convertFromDomain(forecast)) {
2     ...
3 }

```

The conversion from the domain model is straightforward too:

```

1 fun convertFromDomain(forecast: ForecastList) = with(forecast) {
2     val daily = dailyForecast.map { convertDayFromDomain(id, it) }
3     CityForecast(id, city, country, daily)
4 }
5
6 private fun convertDayFromDomain(cityId: Long, forecast: Forecast) =
7     with(forecast) {
8         DayForecast(date, description, high, low, iconUrl, cityId)
9     }

```

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

```

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

```

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

So, with this new function we can do:

```

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

```

It inserts a new row in the `CityForecast` table. The '*' used before the result of `toVarargArray` indicates that the array will be decomposed 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:

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

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

The complete code of this function:

```
1 fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {  
2  
3     clear(CityForecastTable.NAME)  
4     clear(DayForecastTable.NAME)  
5  
6     with(dataMapper.convertFromDomain(forecast)) {  
7         insert(CityForecastTable.NAME, *map.toVarargArray())  
8         dailyForecast.forEach {  
9             insert(DayForecastTable.NAME, *it.map.toVarargArray())  
10        }  
11    }  
12 }
```

A lot of new code was involved in this chapter, so you can take a look at the repository to review it.

20 Null safety in Kotlin

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

Being `null` considered the [billion-dollar mistake by its own creator](#)²³, it's true that we sometimes need to define whether a variable contains a value or not. In Java, though annotations and IDEs are helping a lot these days, we can still do something like:

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

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

20.1 How Null types work

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

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

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

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

²³https://en.wikipedia.org/wiki/Tony_Hoare

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

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

```
1 val a: Int? = null
2 ...
3 if (a != null) {
4     a.toString()
5 }
```

Inside the `if`, `a` becomes `Int` instead of `Int?`, so we can use it without checking nullity anymore. The code outside the `if` context, of course, will have to deal with it. This only works if a variable can't be concurrently modified, because otherwise the value could've been changed from another thread and the previous check would be false at that moment. It is supported on `val` properties or local (`val` or `var`) variables

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

```
1 val a: Int? = null
2 ...
3 a?.toString()
```

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

```
1 val a: Int? = null
2 ...
3 val myString = a?.toString() ?: ""
```

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

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

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

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

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

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

20.2 Nullity and Java libraries

Ok, so the previous explanation works perfectly well with Kotlin code. But what happens with Java libraries in general, and Android SDK in particular? In Java, every object can be null by definition. So we would have to deal with a lot potentially null variables which in real life are never null. 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'll probably see that many parameters are marked with a single **!** when any Java methods are used. For instance, something that gets an `Object` in Java will be represented as `Any!` in Kotlin. This means that it's up to the developer to decide whether that variable should be null or not.

Luckily, latest versions of Android are starting using the `Nullable` and `NonNull` annotations to identify the parameters that can be null or the functions that can return null, and the Kotlin compiler is able to 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:

```
1 override fun onCreate(savedInstanceState: Bundle?) {
2 }
```

Otherwise it will show an error. We can't use this implementation:

```
1  override fun onCreate(savedInstanceState: Bundle) {  
2  }
```

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

21 Creating the business logic to data access

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

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

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

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

```
1 interface ForecastDataSource {  
2     fun requestForecastByZipCode(zipCode: Long, date: Long): ForecastList?  
3 }
```

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

```
1 class ForecastProvider(val sources: List<ForecastDataSource> =  
2     ForecastProvider.SOURCES) {  
3  
4     companion object {  
5         val DAY_IN_MILLIS = 1000 * 60 * 60 * 24  
6         val SOURCES = listOf(ForecastDb(), ForecastServer())  
7     }  
8     ...  
9 }
```

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

So the main method looks like this:

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

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

```
1 inline fun <T, R : Any> Iterable<T>.firstResult(predicate: (T) -> R?): R {
2     for (element in this) {
3         val result = predicate(element)
4         if (result != null) return result
5     }
6     throw NoSuchElementException("No element matching predicate was found.")
7 }
```

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

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

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

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

```

1 fun requestSource(source: ForecastDataSource, days: Int, zipCode: Long):
2     ForecastList? {
3     val res = source.requestForecastByZipCode(zipCode, todayTimeSpan())
4     return if (res != null && res.size() >= days) res else null
5 }

```

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

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

```

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

```

The complete code of this class would be:

```

1 class ForecastProvider(val sources: List<ForecastDataSource> =
2     ForecastProvider.SOURCES) {
3
4     companion object {
5         val DAY_IN_MILLIS = 1000 * 60 * 60 * 24;
6         val SOURCES = listOf(ForecastDb(), ForecastServer())
7     }
8
9     fun requestByZipCode(zipCode: Long, days: Int): ForecastList
10         = sources.firstResult { requestSource(it, days, zipCode) }
11
12     private fun requestSource(source: RepositorySource, days: Int,
13         zipCode: Long): ForecastList? {
14         val res = source.requestForecastByZipCode(zipCode, todayTimeSpan())
15         return if (res != null && res.size() >= days) res else null
16     }
17
18     private fun todayTimeSpan() = System.currentTimeMillis() /
19         DAY_IN_MILLIS * DAY_IN_MILLIS
20 }

```

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

```

1 class ForecastDb(val forecastDbHelper: ForecastDbHelper =
2     ForecastDbHelper.instance, val dataMapper: DbDataMapper = DbDataMapper())
3     : ForecastDataSource {
4
5     override fun requestForecastByZipCode(zipCode: Long, date: Long) =
6         forecastDbHelper.use {
7             ...
8         }
9         ...
10 }

```

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

```

1 class ForecastServer(val dataMapper: ServerDataMapper = ServerDataMapper(),
2     val forecastDb: ForecastDb = ForecastDb()) : ForecastDataSource {
3
4     override fun requestForecastByZipCode(zipCode: Long, date: Long):
5         ForecastList? {
6         val result = ForecastByZipCodeRequest(zipCode).execute()
7         val converted = dataMapper.convertToDomain(zipCode, result)
8         forecastDb.saveForecast(converted)
9         return forecastDb.requestForecastByZipCode(zipCode, date)
10    }
11
12 }

```

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 auto-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.


```
1 class RequestForecastCommand(val zipCode: Long,
2     val forecastProvider: ForecastProvider = ForecastProvider()) :
3     Command<ForecastList> {
4
5     companion object {
6         val DAYS = 7
7     }
8
9     override fun execute(): ForecastList {
10         return forecastProvider.requestByZipCode(zipCode, DAYS)
11     }
12 }
```

The rest of code modifications consist of some renames and package organisation here and there. Take a look at the corresponding commit at [Kotlin for Android Developers repository](https://github.com/antonioIlg/Kotlin-for-Android-Developers)²⁴.

²⁴<https://github.com/antonioIlg/Kotlin-for-Android-Developers>

22. Flow control and ranges

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

22.1 If Expression

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

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

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

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

This also implies we don't need a ternary operation similar to the Java one, because we can solve it easily with:

```
1  val z = if (condition) x else y
```

So the `if` expression always returns a value. If one of the branches returns `Unit`, the whole expression will return `Unit`, which can be ignored, and it will work as a regular Java `if` condition in that case.

22.2 When expression

When expressions are similar to `switch/case` in Java, but far more powerful. This expression will try to match its argument against all possible branches in order until it finds one that is satisfied. It will then apply the right side of the expression. The difference with a `switch/case` in Java is that the argument can be literally anything, and the conditions for the branches too.

For the default option, we can add an `else` branch that will be executed if none of the previous conditions is satisfied. The code executed when a condition is satisfied can be a block too:

```
1 when (x) {
2     1 -> print("x == 1")
3     2 -> print("x == 2")
4     else -> {
5         print("I'm a block")
6         print("x is neither 1 nor 2")
7     }
8 }
```

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

```
1 val result = when (x) {
2     0, 1 -> "binary"
3     else -> "error"
4 }
```

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

```
1 when(view) {
2     is TextView -> view.setText("I'm a TextView")
3     is EditText -> toast("EditText value: ${view.getText()}")
4     is ViewGroup -> toast("Number of children: ${view.getChildCount()} ")
5     else -> view.visibility = View.GONE
6 }
```

The argument is automatically casted in the right part of the condition, so you don't need to do any explicit casting.

It's possible to check whether the argument is inside a range (I'll explain ranges later in this chapter), or even inside a collection:

```
1 val cost = when(x) {  
2     in 1..10 -> "cheap"  
3     in 10..100 -> "regular"  
4     in 100..1000 -> "expensive"  
5     in specialValues -> "special value!"  
6     else -> "not rated"  
7 }
```

Or you could even get rid of the argument and do any crazy check you may need. It could easily substitute an if / else chain:

```
1 val res = when {  
2     x in 1..10 -> "cheap"  
3     s.contains("hello") -> "it's a welcome!"  
4     v is ViewGroup -> "child count: ${v.getChildCount()}"  
5     else -> ""  
6 }
```

22.3 For loops

Though you won't probably use them too much if you make use of functional operators in collections, for loops can be useful in some situations, so they are still available. It works with anything that provides an iterator:

```
1 for (item in collection) {  
2     print(item)  
3 }
```

If we want to get a more typical iteration over indices, we can also do it using ranges (there are usually smarter solutions anyway):

```
1 for (index in 0..viewGroup.getChildCount() - 1) {  
2     val view = viewGroup.getChildAt(index)  
3     view.visibility = View.VISIBLE  
4 }
```

When iterating over an array or a list, a set of indices can be requested to the object, so the previous artifact is not necessary:

```
1 for (i in array.indices)
2     print(array[i])
```

22.4 While and do/while loops

You can keep using while loops too, though it won't be very common either. There are usually simpler and more visual ways to resolve a problem. A couple of examples:

```
1 while (x > 0) {
2     x--
3 }
4
5 do {
6     val y = retrieveData()
7 } while (y != null) // y is visible here!
```

22.5 Ranges

It's difficult to explain flow control without talking about ranges. But their scope is much wider. Range expressions make use of an operator in the form of “..₁” that is defined implementing a RangeTo function.

Ranges help simplify our code in many creative ways. For instance we can convert this:

```
1 if (i >= 0 && i <= 10)
2     println(i)
```

Into this:

```
1 if (i in 0..10)
2     println(i)
```

Range is defined by any type that can be compared, but for numerical types the compiler will optimise it by converting it to simpler analogue code in Java, to avoid the extra overhead. The numerical ranges can also be iterated, and the loops are optimised too by converting them to the same bytecode a for with indices would use in Java:

```
1 for (i in 0..10)
2     println(i)
```

Ranges are incremental by default, so something like:

```
1 for (i in 10..0)
2     println(i)
```

Would do nothing. You can, however, use the function `downTo`:

```
1 for (i in 10 downTo 0)
2     println(i)
```

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

```
1 for (i in 1..4 step 2) println(i)
2
3 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`:

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

This previous line will print from 0 to 3, but will skip 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:

```
1 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. Everything is done in a single line.

If you want to know more about how ranges are implemented and a lot more examples and useful information, you can go to [Kotlin reference](https://kotlinlang.org/docs/reference/ranges.html)²⁵.

²⁵<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 when an item is clicked, but it's time to change that.

23.1 Preparing the request

As we need to know which item we are going to show in the detail activity, logic tells we'll need to send the *id* of the forecast to the detail. So the domain model needs a new *id* property:

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

The `ForecastProvider` also needs a new function, which returns the requested forecast by id. The `DetailActivity` will need it to recover the forecast based on the id it will receive. As all the requests always iterate over the sources and return the first non-null result, we can extract that behaviour to another function:

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

The function is generified using a non-nullable type. It will receive a function which uses a `ForecastDataSource` to return an nullable object of the generic type, and will finally return a non-nullable object. We can rewrite the previous request and write the new one this way:

```
1 fun requestByZipCode(zipCode: Long, days: Int): ForecastList = requestToSources {  
2     val res = it.requestForecastByZipCode(zipCode, todayTimeSpan())  
3     if (res != null && res.size() >= days) res else null  
4 }  
5  
6 fun requestForecast(id: Long): Forecast = requestToSources {  
7     it.requestDayForecast(id)  
8 }
```

Now the data sources need to implement the new function:

```
1 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:

```
1 override fun requestDayForecast(id: Long): Forecast? = forecastDbHelper.use {  
2     val forecast = select(DayForecastTable.NAME).byId(id).  
3         parseOpt { DayForecast(HashMap(it)) }  
4  
5     if (forecast != null) dataMapper.convertDayToDomain(forecast) else null  
6 }
```

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:

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

It just 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` will never be used, because the info will be always cached in the database. We could implement it to defend our code from uncommon edge cases, but we're not doing it in this case, so it will just throw an exception if it's called:

```
1 override fun requestDayForecast(id: Long)  
2     = 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:

```
1 val x = when(y) {  
2     in 0..10 -> 1  
3     in 11..20 -> 2  
4     else -> throw Exception("Invalid")  
5 }
```

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

```
1 val x = try { doSomething() } catch { null }
```

The last thing we need to be able to perform the request from the new activity is to create a command. The code is really simple:

```
1 class RequestDayForecastCommand(  
2     val id: Long,  
3     val forecastProvider: ForecastProvider = ForecastProvider()) :  
4     Command<Forecast> {  
5  
6     override fun execute() = forecastProvider.requestForecast(id)  
7 }
```

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

23.2 Providing a new activity

We are now prepared to create the `DetailActivity`. Our detail activity will receive a couple of parameters from the main one: the forecast id and the name of the city. The first one will be used to request the data from the database, and the name of the city will fill the toolbar. So we first need a couple of names to identify the parameters in the bundle:

```

1 public class DetailActivity : AppCompatActivity() {
2
3     companion object {
4         val ID = "DetailActivity:id"
5         val CITY_NAME = "DetailActivity:cityName"
6     }
7     ...
8 }

```

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:

```

1 <LinearLayout
2     xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent"
6     android:orientation="vertical"
7     android:paddingBottom="@dimen/activity_vertical_margin"
8     android:paddingLeft="@dimen/activity_horizontal_margin"
9     android:paddingRight="@dimen/activity_horizontal_margin"
10    android:paddingTop="@dimen/activity_vertical_margin">
11
12    <LinearLayout
13        android:layout_width="match_parent"
14        android:layout_height="wrap_content"
15        android:orientation="horizontal"
16        android:gravity="center_vertical"
17        tools:ignore="UseCompoundDrawables">
18
19        <ImageView
20            android:id="@+id/icon"
21            android:layout_width="64dp"
22            android:layout_height="64dp"
23            tools:src="@mipmap/ic_launcher"
24            tools:ignore="ContentDescription"/>
25
26        <TextView
27            android:id="@+id/weatherDescription"
28            android:layout_width="wrap_content"
29            android:layout_height="wrap_content"
30            android:layout_margin="@dimen/spacing_xlarge"

```

```

31         android:textAppearance="@style/TextAppearance.AppCompat.Display1"
32         tools:text="Few clouds"/>
33
34     </LinearLayout>
35
36     <LinearLayout
37         android:layout_width="match_parent"
38         android:layout_height="wrap_content">
39
40         <TextView
41             android:id="@+id/maxTemperature"
42             android:layout_width="0dp"
43             android:layout_height="wrap_content"
44             android:layout_weight="1"
45             android:layout_margin="@dimen/spacing_xlarge"
46             android:gravity="center_horizontal"
47             android:textAppearance="@style/TextAppearance.AppCompat.Display3"
48             tools:text="30"/>
49
50         <TextView
51             android:id="@+id/minTemperature"
52             android:layout_width="0dp"
53             android:layout_height="wrap_content"
54             android:layout_weight="1"
55             android:layout_margin="@dimen/spacing_xlarge"
56             android:gravity="center_horizontal"
57             android:textAppearance="@style/TextAppearance.AppCompat.Display3"
58             tools:text="10"/>
59
60     </LinearLayout>
61
62 </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:

```

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

```

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

```
1  async() {  
2      val result = RequestDayForecastCommand(intent.getLongExtra(ID, -1)).execute()  
3      uiThread { bindForecast(result) }  
4  }
```

When the result is recovered from the database, the `bindForecast` function is called in the UI thread. I'm using Kotlin Android Extensions plugin again in this activity, to get the properties from the XML without using `findViewById`:

```
1  import kotlinx.android.synthetic.main.activity_detail.*  
2  
3  ...  
4  
5  private fun bindForecast(forecast: Forecast) = with(forecast) {  
6      Picasso.with(ctx).load(iconUrl).into(icon)  
7      supportActionBar?.subtitle = date.toString(DateFormat.FULL)  
8      weatherDescription.text = description  
9      bindWeather(high to maxTemperature, low to minTemperature)  
10 }
```

There are some interesting things here. For instance, I'm creating another extension function able to convert a `Long` object into a visual date string. Remember we were using it in the adapter too, so it's a good moment to extract it into a function:

```
1  fun Long.toString(dateFormat: Int = DateFormat.MEDIUM): String {  
2      val df = DateFormat.getDateInstance(dateFormat, Locale.getDefault())  
3      return df.format(this)  
4  }
```

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

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

```

1 private fun bindWeather(vararg views: Pair<Int, TextView>) = views.forEach {
2     it.second.text = "${it.first.toString()}"
3     it.second.textColor = color(when (it.first) {
4         in -50..0 -> android.R.color.holo_red_dark
5         in 0..15 -> android.R.color.holo_orange_dark
6         else -> android.R.color.holo_green_dark
7     })
8 }

```

For each pair, it assigns the text that will show the temperature and a color based on the value of the temperature: red for low temperatures, orange for mild ones and green for the rest. The values are taken quite randomly, but it's a good representation of what we can do with a when expression, how clean and short the code becomes.

color is an extension function I miss from Anko, which simplifies the way to get a color from resources, similar to the dimen one we've used in some other places. 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:

```

1 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's not automatically parsed. A definition could be this one:

```

1 var TextView.textColor: Int
2     get() = currentTextColor
3     set(v) = setTextColor(v)

```

There is an implementation in another Anko package (it returns an exception in get, it could be an alternative), but it's 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 only to use one or two functions. That part of the library is huge and you will waste a good part of method count if you don't use Proguard.

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

```
1 <activity
2     android:name=".ui.activities.DetailActivity"
3     android:parentActivityName=".ui.activities.MainActivity" >
4     <meta-data
5         android:name="android.support.PARENT_ACTIVITY"
6         android:value="com.antonioleiva.weatherapp.ui.activities.MainActivity" />
7 </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:

```
1 val adapter = ForecastListAdapter(result) {
2     val intent = Intent(MainActivity@this, javaClass<DetailActivity>())
3     intent.putExtra(DetailActivity.ID, it.id)
4     intent.putExtra(DetailActivity.CITY_NAME, result.city)
5     startActivity(intent)
6 }
```

But this is too verbose. As usual, Anko provides a much simpler way to start an activity by using a **reified function**:

```
1 val adapter = ForecastListAdapter(result) {
2     startActivity<DetailActivity>(DetailActivity.ID to it.id,
3         DetailActivity.CITY_NAME to result.city)
4 }
```

What's the magic behind reified functions? As you may know, when we create a generic method in Java, there is no way to get the class from the generic type. 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 really does would be the next (I'm only using `String` extras in this example):

```
1 public inline fun <reified T: Activity> Context.startActivity(  
2     vararg params: Pair<String, String>) {  
3  
4     val intent = Intent(this, T::class.javaClass)  
5     params.forEach { intent.putExtra(it.first, it.second) }  
6     startActivity(intent)  
7 }
```

The real implementation is a bit more complex because it uses a long and boring when expression to add the extras depending on the type, but it doesn't add much useful knowledge to the concept.

Reified functions are, once more, a syntactic sugar that simplifies the code and improves its comprehension. In this case, it creates an intent by getting the `javaClass` from the generic type, iterates over `params` and adds them to the intent, and 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. We now have a very simple (but complete) master-detail App implemented in Kotlin without using a single line of Java.

24 Interfaces and Delegation

24.1 Interfaces

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

```
1 interface FlyingAnimal {  
2     fun fly()  
3 }
```

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

```
1 class Bird : FlyingAnimal {  
2     val wings: Wings = Wings()  
3     override fun fly() = wings.move()  
4 }  
5  
6 class Bat : FlyingAnimal {  
7     val wings: Wings = Wings()  
8     override fun fly() = wings.move()  
9 }
```

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

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

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

```
1 interface FlyingAnimal {  
2     val wings: Wings  
3     fun fly() = wings.move()  
4 }
```

As mentioned, classes need to override the property:


```
1 class Bird : FlyingAnimal {
2     override val wings: Wings = Wings()
3 }
4
5 class Bat : FlyingAnimal {
6     override val wings: Wings = Wings()
7 }
```

And now both birds and bats can fly:

```
1 val bird = Bird()
2 val bat = Bat()
3
4 bird.fly()
5 bat.fly()
```

24.2 Delegation

The [delegation pattern](https://en.wikipedia.org/wiki/Delegation_pattern)²⁶ is a really useful pattern that can be used to extract responsibilities from a class. The delegation pattern is supported natively by Kotlin, so it prevents from 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 specified this way:

```
1 interface CanFly {
2     fun fly()
3 }
4
5 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 through a delegate that is defined in the constructor, so we can have different birds with different flying methods. The way an animal with wings flies is defined in another class:

²⁶https://en.wikipedia.org/wiki/Delegation_pattern

```
1 class AnimalWithWings : CanFly {  
2     val wings: Wings = Wings()  
3     override fun fly() = wings.move()  
4 }
```

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

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

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

```
1 class Bat : CanFly by AnimalWithWings()  
2 ...  
3 val bat = Bat()  
4 bat.fly()
```

24.3 Implementing an example in our App

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

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

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

```

1 <android.support.v7.widget.Toolbar
2     xmlns:app="http://schemas.android.com/apk/res-auto"
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     android:id="@+id/toolbar"
5     android:layout_width="match_parent"
6     android:layout_height="?attr/actionBarSize"
7     android:background="?attr/colorPrimary"
8     android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
9     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: light theme with dark Action Bar.

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

```

1 <FrameLayout
2     xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent">
5
6     <android.support.v7.widget.RecyclerView
7         android:id="@+id/forecastList"
8         android:layout_width="match_parent"
9         android:layout_height="match_parent"
10        android:clipToPadding="false"
11        android:paddingTop="?attr/actionBarSize"/>
12
13     <include layout="@layout/toolbar"/>
14
15 </FrameLayout>

```

Now that the toolbar has been added 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:

```
1 interface ToolbarManager {  
2     val toolbar: Toolbar  
3     ...  
4 }
```

It will need a toolbar property. Interfaces are stateless, so the property can be defined but no value can be assigned. Classes that implement it will need to override the property.

On the other hand, we can implement stateless properties without the need of being overridden. That is, properties that don't need a backing field. An example would be a property which deals with the toolbar title:

```
1 var toolbarTitle: String  
2     get() = toolbar.title.toString()  
3     set(value) {  
4         toolbar.title = value  
5     }
```

As the property just uses the toolbar, it doesn't need to save any new state.

We're now creating a new function that initialises the toolbar, by inflating a menu and setting a listener:

```
1 fun initToolbar() {  
2     toolbar.inflateMenu(R.menu.menu_main)  
3     toolbar.setOnMenuItemClickListener {  
4         when (it.itemId) {  
5             R.id.action_settings -> App.instance.toast("Settings")  
6             else -> App.instance.toast("Unknown option")  
7         }  
8         true  
9     }  
10 }
```

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:

```
1 fun enableHomeAsUp(up: () -> Unit) {
2     toolbar.navigationIcon = createUpDrawable()
3     toolbar.setNavigationOnClickListener { up() }
4 }
5
6 private fun createUpDrawable() = with (DrawerArrowDrawable(toolbar.ctx)) {
7     progress = 1f
8     this
9 }
```

The function receives the listener, creates the up drawable by using the [DrawerArrowDrawable](#)²⁷ on its final state (when the arrow is already showing) and assigns the listener to the toolbar.

Finally, the interface will provide 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:

```
1 fun attachToScroll(recyclerView: RecyclerView) {
2     recyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {
3         override fun onScrolled(recyclerView: RecyclerView?, dx: Int, dy: Int) {
4             if (dy > 0) toolbar.slideExit() else toolbar.slideEnter()
5         }
6     })
7 }
```

We'll be creating a couple of extension functions which animate the views in and out of the screen. They check if the animation hasn't been previously performed. That way it prevents the view from being animated every time the scroll varies:

```
1 fun View.slideExit() {
2     if (translationY == 0f) animate().translationY(-height.toFloat())
3 }
4
5 fun View.slideEnter() {
6     if (translationY < 0f) animate().translationY(0f)
7 }
```

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

²⁷<https://developer.android.com/reference/android/support/v7/graphics/drawable/DrawerArrowDrawable.html>

```
1 class MainActivity : AppCompatActivity(), ToolbarManager {
2     ...
3 }
```

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

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

MainActivity will just initialise the toolbar, attach to the RecyclerView scroll and modify the toolbar title:

```
1 override fun onCreate(savedInstanceState: Bundle?) {
2     super.onCreate(savedInstanceState)
3     setContentView(R.layout.activity_main)
4     initToolbar()
5
6     forecastList.layoutManager = LinearLayoutManager(this)
7     attachToScroll(forecastList)
8
9     async {
10         val result = RequestForecastCommand(94043).execute()
11         uiThread {
12             val adapter = ForecastListAdapter(result) {
13                 startActivity<DetailActivity>(DetailActivity.ID to it.id,
14                     DetailActivity.CITY_NAME to result.city)
15             }
16             forecastList.adapter = adapter
17             toolbarTitle = "${result.city} (${result.country})"
18         }
19     }
20 }
```

DetailActivity also needs some layout modifications:

```

1  <LinearLayout
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:tools="http://schemas.android.com/tools"
4      android:layout_width="match_parent"
5      android:layout_height="match_parent"
6      android:orientation="vertical">
7
8      <include layout="@layout/toolbar"/>
9
10     <LinearLayout
11         android:layout_width="match_parent"
12         android:layout_height="wrap_content"
13         android:orientation="horizontal"
14         android:gravity="center_vertical"
15         android:paddingTop="@dimen/activity_vertical_margin"
16         android:paddingLeft="@dimen/activity_horizontal_margin"
17         android:paddingRight="@dimen/activity_horizontal_margin"
18         tools:ignore="UseCompoundDrawables">
19         ....
20     </LinearLayout>
21
22 </LinearLayout>

```

The toolbar property is specified the same way. DetailActivity will initialise the toolbar too, set the title and enable the up navigation icon:

```

1  override fun onCreate(savedInstanceState: Bundle?) {
2      super.onCreate(savedInstanceState);
3      setContentView(R.layout.activity_detail)
4
5      initToolbar()
6      toolbarTitle = intent.getStringExtra(CITY_NAME)
7      enableHomeAsUp { onBackPressed() }
8      ...
9  }

```

Interfaces can help us extract common code from classes that share similar behaviours. It can be used as an alternative way of composition that will keep our code better organised, and it will be easier to reuse. Think where interfaces can help you write better code.

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 will increment the amount of times that generics are of some use for us. Though we've been using them blindly throughout the book, generics are usually one of the trickiest parts of any language, so I'm 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:

```
1 class TypedClass<T>(parameter: T) {  
2     val value: T = parameter  
3 }
```

This class now can be instantiated using any type, and the parameter will use the type in the definition. We could do:

```
1 val t1 = TypedClass<String>("Hello World!")  
2 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 don't even need to specify it:

```
1 val t1 = TypedClass("Hello World!")  
2 val t2 = TypedClass(25)  
3 val t3 = TypedClass<String?>(null)
```

As the third object is receiving a null reference, the type still needs to be specified because it can't be inferred.

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 previous class to non-nullable types, we just need to do:


```

1 class TypedClass<T : Any>(parameter: T) {
2     val value: T = parameter
3 }

```

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 only classes that extend `Context`? Easy:

```

1 class TypedClass<T : Context>(parameter: T) {
2     val value: T = parameter
3 }
4
5 val t1 = TypedClass(activity)
6 val t2 = TypedClass(context)
7 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 construct generic functions quite easily:

```

1 fun <T> typedFunction(item: T): List<T> {
2     ...
3 }

```

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's less restrictive. But take a look at this example:

```

1 List<String> strList = new ArrayList<>();
2 List<Object> objList = strList;
3 objList.add(5);
4 String str = objList.get(0);

```

If the Java compiler let us do this, we could add an `Integer` to an `Object` list, and this would obviously crash at some point. That's why wildcards were added to the language. Wildcards will 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 `Strings` into a collection of `Objects`, we should be allowed, right?

Then, if we have:

```
1 List<String> strList = ...;
2 List<Object> objList = ...;
3 objList.addAll(strList);
```

This is possible because the definition of `addAll()` in `Collection` interface is something like:

```
1 List<String>
2 interface Collection<E> ... {
3     void addAll(Collection<? extends E> items);
4 }
```

Otherwise, without the wildcard, we wouldn't be allowed to use a `String` list with this method. The opposite, of course, would fail. We can't 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's 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 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:

```
1 void copyStrings(Collection<? super String> to, Collection<String> from) {
2     to.addAll(from);
3 }
```

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 their own limitations. Wildcards define **use-site variance**, which means we need to declare it where we use it. 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:

```
1 class TypedClass<T> {
2     public T doSomething(){
3         ...
4     }
5 }
```

This code will not compile:

```

1 TypedClass<String> t1 = new TypedClass<>();
2 TypedClass<Object> t2 = t1;

```

Though it really doesn't 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.

```

1 TypedClass<String> t1 = new TypedClass<>();
2 TypedClass<? extends String> t2 = t1;

```

This makes things more difficult to understand, and adds some extra boilerplate.

On the other hand, Kotlin deals with it in an easier manner by using **declaration-site variance**. This means that 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 in Kotlin. Instead of long wildcards, Kotlin just uses **out** for covariance and **in** for contravariance. In this case, as our class is producing objects that could be saved into less restrictive variables, we'll be using covariance. We can define this in the class declaration directly:

```

1 class TypedClass<out T>() {
2     fun doSomething(): T {
3         ...
4     }
5 }

```

And that's all we need. Now, the same example that wouldn't compile in Java is perfectly possible in Kotlin:

```

1 val t1 = TypedClass<String>()
2 val t2: TypedClass<Any> = t1

```

If you were already used to these concepts, I'm sure you will easily be able to use **in** and **out** in Kotlin. Otherwise, it just requires a little practice and some concepts understanding.

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'm using three functions that are included in Kotlin standard library. These functions let us do awesome things with just a generic implementation. They can inspire you to create your own functions.

let

`let` is a really simple function that can be called by any object. It receives a function that will receive the object as a parameter, and returns the value that this function returns. It is really useful to deal with nullable objects for instance. This is the definition:

```
1 inline fun <T, R> T.let(f: (T) -> R): R = f(this)
```

It uses two generic types: `T` and `R`. The first one is defined by the calling object, and it's the type that the function receives. 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 wasn't null, or just a null reference otherwise:

```
1 if (forecast != null) dataMapper.convertDayToDomain(forecast) else null
```

This is really ugly, we shouldn't need to deal with nullable types that way. And in fact we don't need to if we use `let`:

```
1 forecast?.let { dataMapper.convertDayToDomain(it) }
```

`let` function will only be executed if `forecast` is not null thanks to `'?.'` operator. It will return null otherwise. Just what we were trying to achieve.

with

We've talked a lot about this function during the book. `with` receives an object, and a function that will be executed as an extension function. This means we can use `this` inside the function to refer to the object. It will also return an object defined in the last line of the function.

```
1 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 function is defined as an extension function by using this declaration: `f: T.() -> R`. That's why we can then call `receiver.f()`.

We have several examples throughout the app:

```
1 fun convertFromDomain(forecast: ForecastList) = with(forecast) {
2     val daily = dailyForecast.map { convertDayFromDomain(id, it) }
3     CityForecast(id, city, country, daily)
4 }
```

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 initialise itself the way it needs, and the `apply` function will return the same object:

```
1 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 one it's returned. A nice simple example would be:

```
1 val textView = TextView(context).apply {  
2     text = "Hello"  
3     hint = "Hint"  
4     textColor = android.R.color.white  
5 }
```

It creates a `TextView`, modifies some properties, and assigns it to a variable. Everything in a simple, readable and compact syntax. But let's use it in our current code. In `ToolbarManager`, we were doing this to create the navigation drawable:

```
1 private fun createUpDrawable() = with(DrawerArrowDrawable(toolbar.ctx)) {  
2     progress = 1f  
3     this  
4 }
```

Using `with` and returning `this` is clearly something that can be done easier by using `apply`:

```
1 private fun createUpDrawable() = DrawerArrowDrawable(toolbar.ctx).apply {  
2     progress = 1f  
3 }
```

You can review some more little improvements in [Kotlin for Android Developers repository](#).

26 Settings Screen

Until now, we've been using a default city to implement the App, but it's time add the ability to select a city. Our 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 doesn't 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 will be a good example to explain an interesting way to deal with preferences.

26.1 Creating the settings activity

A new activity will we opened when the settings option is selected in the overflow menu in the toolbar. So first thing we need is a new `SettingsActivity`:

```
1 class SettingsActivity : AppCompatActivity() {
2
3     override fun onCreate(savedInstanceState: Bundle?) {
4         super.onCreate(savedInstanceState)
5         setContentView(R.layout.activity_settings)
6         setSupportActionBar(toolbar)
7         supportActionBar?.setDisplayHomeAsUpEnabled(true)
8     }
9
10    override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
11        android.R.id.home -> { onBackPressed(); true }
12        else -> false
13    }
14 }
```

We'll save the preference when the user gets out of the screen activity, so we're going to deal with the Up action the same way as the Back one, by redirecting the action to `onBackPressed`. By now, let's create the XML layout. A simple `EditText` will be enough for the preference:

```

1  <FrameLayout
2      xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent">
5
6      <include layout="@layout/toolbar"/>
7
8      <LinearLayout
9          android:orientation="vertical"
10         android:layout_width="match_parent"
11         android:layout_height="match_parent"
12         android:layout_marginTop="?attr/actionBarSize"
13         android:padding="@dimen/spacing_xlarge">
14
15         <TextView
16             android:layout_width="wrap_content"
17             android:layout_height="wrap_content"
18             android:text="@string/city_zipcode"/>
19
20         <EditText
21             android:id="@+id/cityCode"
22             android:layout_width="match_parent"
23             android:layout_height="wrap_content"
24             android:hint="@string/city_zipcode"
25             android:inputType="number"/>
26
27     </LinearLayout>
28
29 </FrameLayout>

```

And then just declare the activity in the `AndroidManifest.xml`:

```

1  <activity
2      android:name=".ui.activities.SettingsActivity"
3      android:label="@string/settings"/>

```

26.2 Accessing Shared Preferences

You probably know what Android [Shared Preferences](http://developer.android.com/training/basics/data-storage/shared-preferences.html)²⁸ are. They consist of a set of keys and values that can be easily saved and restored by using the Android framework. These preferences

²⁸<http://developer.android.com/training/basics/data-storage/shared-preferences.html>

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 restored too.

Thanks to the use of property delegation, we can deal with preferences in a very easy way. 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:

```

1 class LongPreference(val context: Context, val name: String, val default: Long)
2     : ReadWriteProperty<Any?, Long> {
3
4     val prefs by lazy {
5         context.getSharedPreferences("default", Context.MODE_PRIVATE)
6     }
7
8     override fun getValue(thisRef: Any?, property: KProperty<*>): Long {
9         return prefs.getLong(name, default)
10    }
11
12    override fun setValue(thisRef: Any?, property: KProperty<*>, value: Long) {
13        prefs.edit().putLong(name, value).apply()
14    }
15 }
```

First, we create a lazy access to preferences. That way, if we don't use the property, this delegate will never request the `SharedPreferences` object.

When `get` is called, its implementation will use 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, a preferences editor is requested and the value will be saved using the name of the property.

We can then define the new delegate in the `DelegatesExt` object, so that it's easier to access when required:

```

1 object DelegatesExt {
2     ....
3     fun longPreference(context: Context, name: String, default: Long) =
4         LongPreference(context, name, default)
5 }
```

In `SettingsActivity`, a property to deal with zip code preference can now be defined. I'm also creating a couple of constants which keep the name and default value of the property. That way they can be used in other sections of the App.


```
1 companion object {  
2     val ZIP_CODE = "zipCode"  
3     val DEFAULT_ZIP = 94043L  
4 }  
5  
6 var zipCode: Long by DelegatesExt.longPreference(this, ZIP_CODE, DEFAULT_ZIP)
```

Now it's really easy to work with this preference. In `onCreate`, we get the value of the property and assign it to the `EditText`:

```
1 override fun onCreate(savedInstanceState: Bundle?) {  
2     ...  
3     cityCode.setText(zipCode.toString())  
4 }
```

We can't use the auto-generated property text because `EditText` returns an `Editable` in `getText`, so the property is defaulted 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:

```
1 override fun onBackPressed() {  
2     super.onBackPressed()  
3     zipCode = cityCode.text.toString().toLong()  
4 }
```

The `MainActivity` requires some little changes. First, it also needs a `zipCode` property.

```
1 val zipCode: Long by DelegatesExt.longPreference(this, SettingsActivity.ZIP_CODE,  
2     SettingsActivity.DEFAULT_ZIP)
```

And then, I'm 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 really changed before requesting the forecast again, for instance. But I want to keep this example simple, and the requested info is already saved in a local database, so this solution is not that bad:

```

1  override fun onResume() {
2      super.onResume()
3      loadForecast()
4  }
5
6  private fun loadForecast() = async {
7      val result = RequestForecastCommand(zipCode).execute()
8      uiThread {
9          val adapter = ForecastListAdapter(result) {
10             startActivity<DetailActivity>(DetailActivity.ID to it.id,
11                 DetailActivity.CITY_NAME to result.city)
12         }
13         forecastList.adapter = adapter
14         toolbarTitle = "${result.city} (${result.country})"
15     }
16 }

```

The `RequestForecastCommand` is now using the `zipCode` property instead of the previous fixed value.

There's 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:

```

1  when (it.itemId) {
2      R.id.action_settings -> toolbar.ctx.startActivity<SettingsActivity>()
3      else -> App.instance.toast("Unknown option")
4  }

```

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:

```

1  class Preference<T>(val context: Context, val name: String, val default: T)
2      : ReadWriteProperty<Any?, T> {
3
4      val prefs by lazy {
5          context.getSharedPreferences("default", Context.MODE_PRIVATE)
6      }
7
8      override fun getValue(thisRef: Any?, property: KProperty<*>): T {
9          return findPreference(name, default)
10     }

```

```

11
12     override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
13         putPreference(name, value)
14     }
15     ...
16 }

```

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 will do the hard work. These functions are very simple, though a bit repetitive. They will check the type and use the specific method from preferences. For instance, the findPreference function looks like this:

```

1 private fun <T> findPreference(name: String, default: T): T = with(prefs) {
2     val res: Any = when (default) {
3         is Long -> getLong(name, default)
4         is String -> getString(name, default)
5         is Int -> getInt(name, default)
6         is Boolean -> getBoolean(name, default)
7         is Float -> getFloat(name, default)
8         else -> throw IllegalArgumentException(
9             "This type can be saved into Preferences")
10    }
11
12    res as T
13 }

```

And basically the same for putPreference function, but using the preferences editor and saving the result of when at the end, by calling apply():

```

1 private fun <U> putPreference(name: String, value: U) = with(prefs.edit()) {
2     when (value) {
3         is Long -> putLong(name, value)
4         is String -> putString(name, value)
5         is Int -> putInt(name, value)
6         is Boolean -> putBoolean(name, value)
7         is Float -> putFloat(name, value)
8         else -> throw IllegalArgumentException("This type can be saved into Pref\
9 erences")
10    }.apply()
11 }

```

Now update DelegatesExt object and you're done:

```
1 object DelegatesExt {  
2     ...  
3     fun preference<T : Any>(context: Context, name: String, default: T)  
4         = Preference(context, name, default)  
5 }
```

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 will automatically be refreshed with the new information. Check the rest of small tweaks at the repository.

27 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 well differentiated tests: unit tests and instrumentation tests. This book is obviously not meant to teach you how tests should be done, there are whole 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.

27.1 Unit testing

I'm not entering into discussions about what unit testing is. 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' includes is left to the reader. In our case, I'm just going to define a unit test as a test that doesn't need a device to be run. The IDE will be able to run the tests and show a result identifying which tests succeeded and which ones failed.

Unit testing is usually done using JUnit library. So let's add the dependency to the `build.gradle`. As this dependency is only used when running tests, we can use `testCompile` instead of `compile`. This way, the library is left out of regular compilations, reducing the size of the APK:

```
1 dependencies {  
2     ...  
3     testCompile 'junit:junit:4.12'  
4 }
```

Now sync gradle to get the library included into your project. In some 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 in the left side of the IDE) and click on 'Test Artifact' dropdown. You should choose 'Unit Tests' there. New Android Studio versions will be able to run enable both kind of tests at the same time, so this step won't be required.

Another thing you need is to 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 coloured in green. This is a good indication that the IDE detected that we are in 'Unit Test' mode and that this folder will contain test files.

Let's write a very simple test to see everything works properly. Create a new Kotlin class called `SimpleTest` using the proper package (in my case `com.antoniroleiva.weatherapp`, but you need to use the main package of your app). Once you've created the new file, write this simple test:

```
1 import org.junit.Test
2 import kotlin.test.assertTrue
3
4 class SimpleTest {
5
6     @Test fun unitTestingWorks() {
7         assertTrue(true)
8     }
9 }
```

Use the `@Test` annotation to identify the function as a test. Be sure to use `org.junit.Test`. Then add a simple assertion. It will just check that `true` is `true`, which should obviously succeed. This test will just check that everything is configured properly.

To execute the tests, just right click over the new `java` folder you created below `test`, and choose ‘Run All Tests’. When compilation finishes, it will run the test and you’ll see a summary showing the result. You should see that your test passed.

Now it’s time to create some real tests. Everything that deals with Android framework will probably need an instrumentation test or more complex libraries such as [Robolectric](http://robolectric.org/)²⁹. Because of that, in these examples I’ll be testing things that don’t use anything from the framework. For instance, I’ll test the extension function that creates a date `String` from a `Long`.

Create a new file called `ExtensionTests`, and add this tests:

```
1 class ExtensionsTest {
2
3     @Test fun testLongToDateString() {
4         assertEquals("Oct 19, 2015", 1445275635000L.toString())
5     }
6
7     @Test fun testDateStringFullFormat() {
8         assertEquals("Monday, October 19, 2015",
9             1445275635000L.toString(DateFormat.FULL))
10    }
11 }
```

These tests check that a `Long` instance is properly converted to a `String`. The first one tests the default behaviour (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.

²⁹<http://robolectric.org/>

If you're used to test your apps in Java, you'll see there's not much difference here. I've covered a very simple example, but from here you can create more complex tests to validate other parts of the App. For instance, we could do some tests about `ForecastProvider`. We can use `Mockito` library to mock some other classes and be able to test the provider independently:

```

1 dependencies {
2     ...
3     testCompile "junit:junit:4.12"
4     testCompile "org.mockito:mockito-core:1.10.19"
5 }

```

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`:

```

1 val ds = mock(ForecastDataSource::class.java)
2 `when`(ds.requestDayForecast(0)).then {
3     Forecast(0, 0, "desc", 20, 0, "url")
4 }

```

As you see, we need backquotes for when function. This is because when is a reserved word in Kotlin, so we need to escape it if we find some Java code that uses it. Now we create a provider with this data source, and check that the result of the call to that method is not null:

```

1 val provider = ForecastProvider(listOf(ds))
2 assertNotNull(provider.requestForecast(0))

```

This is the complete test function:

```

1 @Test fun testDataSourceReturnsValue() {
2     val ds = mock(ForecastDataSource::class.java)
3     `when`(ds.requestDayForecast(0)).then {
4         Forecast(0, 0, "desc", 20, 0, "url")
5     }
6
7     val provider = ForecastProvider(listOf(ds))
8     assertNotNull(provider.requestForecast(0))
9 }

```

If you run this, you'll see that it crashes. Thanks to this test, we detect we have something wrong in our code. The test is failing because `ForecastProvider` is initialising `SOURCES` inside its companion object before it's 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:

```

1 companion object {
2     val DAY_IN_MILLIS = 1000 * 60 * 60 * 24
3     val SOURCES by lazy { listOf(ForecastDb(), ForecastServer()) }
4 }

```

If you now run again, you'll see it's 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:

```

1 @Test fun emptyDatabaseReturnsServerValue() {
2     val db = mock(ForecastDataSource::class.java)
3
4     val server = mock(ForecastDataSource::class.java)
5     `when`(server.requestForecastByZipCode(
6         any(Long::class.java), any(Long::class.java)))
7         .then {
8             ForecastList(0, "city", "country", listOf())
9         }
10
11     val provider = ForecastProvider(listOf(db, server))
12
13     assertNotNull(provider.requestByZipCode(0, 0))
14 }

```

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 are able to use the basic unit testing tools.

27.2 Instrumentation tests

Instrumentation tests are a bit different. They are normally 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'll need to deploy the App and run the tests in a device.

This type of tests must be included in the `androidTest` folder, and we must change 'Test Artifact' to 'Android Instrumentation Tests' in 'Build Variants' panel. The official library to implement instrumentation tests is [Espresso](https://google.github.io/android-testing-support-library/)³⁰, which will help us easily navigate through our App by writing Actions, and filter and check results using ViewMatchers and Matchers.

³⁰<https://google.github.io/android-testing-support-library/>

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 doesn't add any extra overhead, so if you already know how to configure Espresso, it will be easy for you.

First, specify the test runner in `defaultConfig`:

```
1 defaultConfig {
2     ...
3     testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
4 }
```

Once you've dealt with the runner, it's time to add the rest of the dependencies, this time using `androidTestCompile`. That way, these libraries only will be added when we compile to run the instrumentation tests:

```
1 dependencies {
2     ...
3     androidTestCompile "com.android.support:support-annotations:$support_version"
4     androidTestCompile "com.android.support.test:runner:0.4.1"
5     androidTestCompile "com.android.support.test:rules:0.4.1"
6     androidTestCompile "com.android.support.test.espresso:espresso-core:2.2.1"
7     androidTestCompile ("com.android.support.test.espresso:espresso-contrib:2.2.\
8 1"){
9         exclude group: 'com.android.support', module: 'appcompat'
10        exclude group: 'com.android.support', module: 'support-v4'
11        exclude module: 'recyclerview-v7'
12    }
13 }
```

I don't want to spend much time talking about this, but here it is a brief reasoning about why you need these libraries:

- `support-annotations`: it's required by some of the other libraries.
- `runner`: It's the test runner, the one we specified in `defaultConfig`.
- `rules`: Includes some rules that help tests inflate and launch the activities. We'll use a rule in our examples.
- `espresso-core`: the basic implementation 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 will 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 means we are testing that we successfully navigated to the detail after clicking on a view inside the `RecyclerView`.

```

1 class SimpleInstrumentationTest {
2
3     @get:Rule
4     val activityRule = ActivityTestRule(MainActivity::class.java)
5
6     ...
7 }

```

First we need to specify that it's being run using `AndroidJUnit4`. Then, create an activity rule, that will instantiate the activity the test will use. In Java, you would annotate the field using `@Rule`. But as you now, 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 to do that by specifying `get` or `set` before the name of the rule. In this case, just write `@get:Rule`.

After that, we are ready to create our first test:

```

1 @Test fun itemClick_navigatesToDetail() {
2     onView(withId(R.id.forecastList)).perform(
3         RecyclerViewActions
4             .actionOnItemAtPosition<RecyclerView.ViewHolder>(0, click()))
5     onView(withId(R.id.weatherDescription))
6         .check(matches(isAssignableFrom(TextView::class.java)))
7 }

```

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 an specific id and that it is an instance of `TextView`.

To run the test, click on the top 'Run configurations' dropdown and choose 'Edit Configurations...'. Press the '+' icon, select 'Android Tests', and choose the app module. Now, in target device, choose the target you prefer. Click 'OK' and then run. You should see how the App is started in 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.

```

1  @Test fun modifyZipCode_changesToolbarTitle() {
2      openActionBarOverflowOrOptionsMenu(activityRule.activity)
3      onView(withText(R.string.settings)).perform(click())
4      onView(withId(R.id.cityCode)).perform(replaceText("28830"))
5      pressBack()
6      onView(isAssignableFrom(Toolbar::class.java))
7          .check(matches(
8              withToolbarTitle(`is`("San Fernando de Henares (ES)"))))
9  }

```

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 is opened, so it will look for the `EditText` and replace the old city code with a new one.
- It presses the back button. This will save the new value inside the preferences, and close the activity.
- As `onResume` is executed in `MainActivity`, the request is performed again. This will retrieve the forecast of the new city.
- Last line will check the `Toolbar` title and see 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:

```

1  private fun withToolbarTitle(textMatcher: Matcher<CharSequence>): Matcher<Any> =
2      object : BoundedMatcher<Any, Toolbar>(Toolbar::class.java) {
3
4          override fun matchesSafely(toolbar: Toolbar): Boolean {
5              return textMatcher.matches(toolbar.title)
6          }
7
8          override fun describeTo(description: Description) {
9              description.appendText("with toolbar title: ")
10             textMatcher.describeTo(description)
11         }
12     }

```

The `matchesSafely` function is the place where the check is done, while the `describeTo` function adds some information about the matcher.

This chapter has been specially interesting, because we've seen how Kotlin is perfectly compatible with both unit and integration tests and is able to interoperate with the testing libraries. Take a look at the code and run the tests by yourself.

28 Extra concepts

Through this book, we've talked about most important concepts of Kotlin language. But we didn't use some of them when implementing the App, and I wouldn't want to let them out of these pages. In this chapter, I will review some unrelated features that you could need when you develop your next Android App using Kotlin.

28.1 Nested classes

As in Java, we can define classes inside other classes. If it's a regular class, it won't be able to access the members of the outer class (it would work as an static class in Java):

```
1 class Outer {
2     private val bar: Int = 1
3     class Nested {
4         fun foo() = 2
5     }
6 }
7
8 val demo = Outer.Nested().foo() // == 2
```

If we want to access to the members of the outer class, we need to declare it as an **inner** class:

```
1 class Outer {
2     private val bar: Int = 1
3     inner class Inner {
4         fun foo() = bar
5     }
6 }
7
8 val demo = Outer().Inner().foo() // == 1
```

28.2 Enum classes

Kotlin also provides a way to implement **enums**:

```
1 enum class Day {  
2     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
3     THURSDAY, FRIDAY, SATURDAY  
4 }
```

Enums can have parameters:

```
1 enum class Icon(val res: Int) {  
2     UP(R.drawable.ic_up),  
3     SEARCH(R.drawable.ic_search),  
4     CAST(R.drawable.ic_cast)  
5 }  
6  
7 val searchIconRes = Icon.SEARCH.res
```

Enums can be requested by the `String` that matches a name, and we can also get an `Array` which includes all the values of an enum, so that we can iterate over them.

```
1 val search: Icon = Icon.valueOf("SEARCH")  
2 val iconList: Array<Icon> = Icon.values
```

Besides, every Enum constant has functions to obtain its name and the position in the declaration:

```
1 val searchName: String = Icon.SEARCH.name  
2 val searchPosition: Int = Icon.SEARCH.ordinal
```

Enums implement `Comparable` based on the constants ordinal, so it's easy to compare them.

28.3 Sealed classes

Sealed classes are used to represent restricted class hierarchies, which means that the number of classes that extend a sealed class is fixed. It's similar to an Enum in the sense that you can know beforehand the options you have when looking at the specific type of an object of the parent sealed class. 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 in Scala: 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's empty:

```

1 sealed class Option<out T> {
2     class Some<out T> : Option<T>()
3     object None : Option<Nothing>()
4 }

```

The good thing about sealed classes is that when they are used in a `when` expression, we can check all the options and won't need to add the `else` clause.

```

1 val result = when (option) {
2     is Option.Some<*> -> "Contains a value"
3     is Option.None -> "Empty"
4 }

```

28.4 Exceptions

In Kotlin, all exceptions implement `Throwable`, have a message and are unchecked. This means we are not obliged to use `try/catch` on any of them. That's not the case in Java, where methods that throw `IOException`, for instance, need to be surrounded by a `try/catch` block. Years of dealing with them have shown that checked exceptions were not a good idea. People like [Bruce Eckel](#)³¹, [Rod Waldhoff](#)³² or [Anders Hejlsberg](#)³³ can give you a better perspective about it.

The way to throw an exception is very similar to Java:

```

1 throw MyException("Exception message")

```

And `try` expression is identical too:

```

1 try {
2     // some code
3 }
4 catch (e: SomeException) {
5     // handler
6 }
7 finally {
8     // optional finally block
9 }

```

Both `throw` and `try` are expressions in Kotlin, which means they can be assigned to a variable. This is really useful when dealing with edge cases:

³¹<http://www.mindview.net/Etc/Discussions/CheckedExceptions>

³²<http://radio-weblogs.com/0122027/stories/2003/04/01/JavaCheckedExceptionsWereAMistake.html>

³³<http://www.artima.com/intv/handcuffs.html>

```
1 val s = when(x){  
2     is Int -> "Int instance"  
3     is String -> "String instance"  
4     else -> throw UnsupportedOperationException("Not valid type")  
5 }
```

or

```
1 val s = try { x as String } catch(e: ClassCastException) { null }
```

29 Conclusion

Thanks for reading this book. Throughout this pages, we've learned Kotlin by implementing an Android App as an example. The weather App was a good example to implement the basic 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, new knowledge is more easily absorbed when it is put into action. It was my main goal, because reference books are usually a nice tool to solve some punctual doubts, but they are hard to read from the beginning to the very end. Besides, as the examples are usually out of a bigger context, it's difficult to understand which kind of problems those features solve.

And 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 finds a lot of questions when dealing with asynchrony, databases, or has to deal with really verbose listeners or activity navigations. By using a real App as an example, we have answered many of these questions while learning new language or libraries features.

I hope these goals were achieved, and I really hope that you not only learned Kotlin but also enjoyed reading this book. I'm convinced that Kotlin is right now the best alternative to Java for Android developers, and that we'll see a boost during next months. You will be one of the first in the boat when that happens, and will be perfectly positioned to be a reference in your circle.

This book is finished, but it doesn't mean it's dead. I'll 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's been an amazing journey during the months I've 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: antoniroleiva.com³⁴
- Email: contact@antoniroleiva.com³⁵
- Twitter: [@lime_cl](http://twitter.com/lime_cl)³⁶
- Google+: [+AntonioLeivaGordillo](http://plus.google.com/+AntonioLeivaGordillo)³⁷

³⁴<http://antoniroleiva.com>

³⁵<mailto:contact@antoniroleiva.com>

³⁶http://twitter.com/lime_cl

³⁷<http://plus.google.com/+AntonioLeivaGordillo>