Table of Contents

《Kotlin for android developers》English version of this repo

> Wrong words, sentence, translation errors and other issues can mention issues. Please explain the cause of the error.

1. Read or download online from GitBook
2. Github

I hope you buy genuine, it is recommended to read the original English：https://leanpub.com/kotlin-for-android-developers

future plans:

- Compare every page of this electronic book to real one
- Translate to Persian too

Contributes will be accepted

- Github: https://github.com/sinadarvi/kotlin-for-android-developers

Write in front

Learn through the Kotlin language to simply develop android applications.

About this book

In this book, I will use Kotlin as the main language to develop an android application. The way is through the development of an application to learn this language, rather than according to the traditional structure to learn. I will stop at the point of interest through the way with Java1.7 Kotlin talk about some of the concepts and characteristics. In this way you will be able to know their differences, and know which part of the language features can let you improve your work efficiency.

This book is not a language reference book, but it is an Android developer to learn Kotlin and use a tool in its own project. I will use a number of language features and interesting tools and libraries to solve many of our daily life will encounter typical problems.

This book is very practical, so I suggest you follow my example and code practice in front of the computer. Whenever you can have some ideas in the depth of practice to go.

As you know, it's a lean publication. That is to say that this book is written with you. I will be based on your reply and suggestions to write new content and check the previous content. Although this book has been completed, but I will be updated in accordance with the new Kotlin version. So despite the preparation of advice to tell me your views on this book, or need to improve the place. I hope this book will be a perfect tool for Android developers, because of this, welcome everyone's ideas and help.

Thank you will be part of this exciting project.

Is this book suitable for you?

Write this book is to help those interested in using Kotlin language to develop Android developers.

If you meet the following situations, then this book is for you:

- You have the basics of Android development and Android SDK.
- You want to follow Kotlin using an example written in Kotlin.
- you need a guide on how to use a more concise and vivid language to solve typical problems encountered in everyday life.

On the other hand, this book may not be right for you because:

- This book is not the Kotlin Bible. I will explain all the basic grammar of Kotlin, and even include some of the more complex ideas I need in the process. So you are going through an example to learn, not the other way.
- I will not explain how to develop an Android application. You do not need to develop knowledge very well, but at least understand the basics, such as Android Studio, Gradle, Java and Android SDK. You may learn some new things about Android development.
- This book is not a functional programming language guide. Of course, because Java 7 is not a functional style, I will explain what you need to know, but not very in-depth to explain the topic of functional programming.

About the author

Antonio Leiva is an Android engineer who focuses on researching new potential Android development possibilities and then writing notes. He maintains a blog about many different Android development topics antonioleiva.com.

Antonio started with a CRM technical consultant, but after a while he was looking for new passion, and he found the Android world. In the excellent platform to get the relevant experience, then he joined a major Spanish mobile phone company to lead a number of projects as a new adventure.

Now he is an Android engineer at Plex and plays an important role in Android design and UX.

You can follow him on Twitter @lime_cl.

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.

What is Kotlin?

Kotlin, as mentioned earlier, is a JVM-based language developed by JetBrains. 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 learn 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.
- **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.

What do we get through 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:

Expresiveness

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:

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

    public long getId() {
        return id;
    }


    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public String getUrl() {
        return url;
    }

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

    public String getMbid() {
        return mbid;
    }

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

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

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

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

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

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:

```
// can not be compiled here. Artist can not be null
Var notNullArtist: Artist = null

// Artist can be null
Var artist: Artist ?= Null

// can not compile, artist may be null, we need to deal with
Artist.print ()

// print as soon as artist != Null
Artist?.print ()

// Smart conversion. If we had an empty check before, you would not need to use the secure call operator call
If (artist != Null) {
   Artist.print ()
}

// it is only possible to make sure that the artist is not null, otherwise it will throw an exception
Artist!!.Print ()

// use the Elvis operator to give a substitute value in the case of null
Val name = artist?.name?: "null"
```

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:

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

We can do it now:

```
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:

```
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.

Ready to work

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.

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, a Java IDE created by JetBrains, 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 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 f lavours, 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.

If you have not AndroidStudio, click here from the official website to download。

Install the Kotlin plugin

IDE it does not understand Kotlin itself. As mentioned in the previous section, it is designed for Java development. But the Kotlin team created a series of powerful plug-ins to make it easier for us to achieve. Go to the `Plugin` column in `Preferences` in Android Studio and install the following two plugins:

- Kotlin: This is a basic plugin. It allows Android Studio to understand Kotlin code. It will publish a new plugin version every time the new Kotlin language version is released, so that we can discover new version features and discarded warnings through it. This is the only plugin you want to use Kotlin to write Android apps. But we still need another one now.
- Kotlin Android Extensions: Kotlin team for Android development also released another interesting plug-in. This Android Extensions allows you to automatically inject all View from Activity in XML. For example, you do not need to use `findViewById()`. You will immediately get a view of the conversion from the property. You will need to install this plugin to use this feature. We will explain this in depth in the next chapter.

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.

Create 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. 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.

Create a 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: `Weather App` 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.

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:

```
buildscript {
    ext.support_version = '23.1.1'
    ext.kotlin_version = '1.0.0'
    ext.anko_version = '0.8.2'
    repositories {
        jcenter()
        dependencies {
            classpath 'com.android.tools.build:gradle:1.5.0'
            classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        }
    }
}
allprojects {
    repositories {
        jcenter()
    }
}
```

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

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

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

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

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.

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.

Test whether everything is ready

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:

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

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

In `onCreate`, you can now get and access this TextView directly.

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

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. Each section as long as the modified code, I will be submitted, so be sure to check all the code changes.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.

Classes and functions

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

How to define a class

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

```
class MainActivity{

}
```

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:

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

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

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

Class inheritance

By default, a class always extends from `Any` (similar to `Object` in java), 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`:

```
open class Animal(name: String)
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.

Function

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

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

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

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

> Tip: semicolon is not required
>
> > 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:

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

Construction methods and function parameters

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

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

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

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

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

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

This would be equivalent to the next code in Java:

```
void toast(String message){
}

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

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

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

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

```
toast("Hello")
toast("Hello", "MyTag")
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:

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

> Tip: String template
>
> > 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}"`

Write your first class

We already have the `MainActivity.kt` class. The Activity will show a series of weather forecasts next week, and all its layout needs to be changed.

Create a 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:

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

Now, in `activity_main.xml`:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
             android:layout_width="match_parent"
             android:layout_height="match_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecast_list"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</FrameLayout>
```

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

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

As you can see, we define the variable and cast it to `RecyclerView`. It'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.

The Recycler Adapter

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

The views used for `RecyclerView` adapter will be just `TextView`, 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:

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

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        return ViewHolder(TextView(parent.context))
    }
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.textView.text = items.[position]
    }

    override fun getItemCount(): Int = items.size

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

Again, we can access 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.

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

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

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

> List created
>
> > 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`, `arrayListOf` 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.

Variables and attributes

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.

basic type

Of course, basic types such as integers, f loats, 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 `Double` to a `Int` variable. An explicit conversion must be done, using one of the many functions available:

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

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

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

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

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

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

> There are many other bitwise operations, such as `sh1` , `shs` , `ushr` , `xor` or `inv` .You can take a look at the Kotlin official website 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:

```
val i = 12 // An Int
val iHex = 0x0f // —Hex type of hex
val l = 3L // A Long
val d = 3.5 // A Double
val f = 3.5F // A Float
```

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

```
val s = "Example"
val c = s[2] //This is a character 'a'
// Iteration String
val s = "Example"
for(c in s){
    print(c)
}
```

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 **immutable** is a very important concept in Kotlin (and many other modern languages).

An immutable object is an object whose state cannot change after instantiation. If you need a modified version 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.

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

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

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

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:

```
public class Person {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
...
Person person = new Person();
person.setName("name");
String name = person.getName();
```

In Kotlin, only a property is required:

```
public class Person {
    var name: String = ""
}

...

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

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

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

If the property needs access to its own value in custom getter and setter (as in this case), it requires the creation of a `backing field`. It can be accessed by using `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.

What is Anko?

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.

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 function** 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`:

```
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.

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`:

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

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

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

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

```
toast("Hello world!")
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:

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

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

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

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 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, 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:

```
public class Request(val url: String) {
    public fun run() {
        val forecastJsonStr = URL(url).readText()
        Log.d(javaClass.simpleName, forecastJsonStr)
    }
}
```

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`, `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`:

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

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:

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

A nice thing about `UIThread` is that it's implemented differently depending on the caller object. If it's used by an `Activity`, the 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:

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

`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

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:

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

Extra function

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.

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:

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

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

> Be careful with immutability when using Java classes
>
> > If you decide to work with immutability, be aware that Java classes 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.

Mapping an object into a variable

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:

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

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

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

The logic behind this feature is 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:

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

Convert json to data class

We now know how to create a data class, then we began to prepare to resolve the data. In the `date` package, create a new file named `ResponseClasses.kt`. If you open the url in Chapter 8, you can see the entire structure of the json file. Its basic composition includes a city, a series of weather forecasts, the city has id, name, where the coordinates. Every weather forecast has a lot of information, such as date, different temperatures, and an id by description and icon.

In our current UI, we will not use all of these data. We will parse all inside the class, because it may be used in some cases later. The following is the class we need to use:

```
data class ForecastResult(val city: City, val list: List<Forecast>)
data class City(val id: Long, val name: String, val coord: Coordinates,
                val country: String, val population: Int)
data class Coordinates(val lon: Float, val lat: Float)
data class Forecast(val dt: Long, val temp: Temperature, val pressure: Float,
                    val humidity: Int, val weather: List<Weather>,
                    val speed: Float, val deg: Int, val clouds: Int,
                    val rain: Float)
data class Temperature(val day: Float, val min: Float, val max: Float,
                       val night: Float, val eve: Float, val morn: Float)
data class Weather(val id: Long, val main: String, val description: String,
                   val icon: String)
```

When we use Gson to parse json into our class, the names of these attributes must be the same as those in json, or you can specify a `serialized name` (serialized name). A good practice is that most of the software structure will be based on our app layout to decouple into different models. So I like to use declarations to simplify these classes because I will parse these classes before using it in other parts of app. The name of the attribute is exactly the same as the name in the json result.

Now, in order to return the parsed results, our `Request` class needs to be modified. It will still only receive a city's `zipcode` as a parameter instead of a full url, so it becomes more readable. Now, I will put this static url in a `companion object` (with the object). If we have to add more requests to the API, we need to extract it.

> **Companion objects**
>
> > Kotlin allows us to define something that behaves like a static object. Although these objects can be implemented in well-known patterns, such as easy-to-implement singleton patterns.
> >
> > We need a class which has some static properties, constants or functions, we can use the `companion object`. This object is shared by all the objects of this class, like static properties or methods in Java.

The following is the last code:

```
public class ForecastRequest(val zipCode: String) {
    companion object {
        private val APP_ID = "15646a06818f61f7b8d7823ca833e1ce"
        private val URL = "http://api.openweathermap.org/data/2.5/" +
                "forecast/daily?mode=json&units=metric&cnt=7"
        private val COMPLETE_URL = "$URL&APPID=$APP_ID&q="
    }

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

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

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

Build the domain layer

We now create a new package as the `domain` layer. This layer will contain some `Commands` implementation to perform tasks for the app.

First, you must define a `Command` :

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

The command will perform an operation and return a certain type of object, which can be specified by the pattern. You need to know an interesting concept, **everything kotlin function will return a value**. If it is not specified, it will return a `Unit` class by default. So if we want Command not to return the data, we can specify it for Unit.

Kotlin interfaces are much more powerful than Java (before Java 8) because they can contain code. But we do not need more code now, in the next chapter will be carefully talk about this topic.

The first command needs to request the weather forecast structure and then convert the result to the domain class. The following classes are defined in the domain class:

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

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

When more features are added, these classes may need to be reviewed later. But now these classes are enough for us.

These classes must map from the data to our domain class, so I need to create a `DataMapper` :

```
public class ForecastDataMapper {
    fun convertFromDataModel(forecast: ForecastResult): ForecastList {
        return ForecastList(forecast.city.name, forecast.city.country,
                convertForecastListToDomain(forecast.list))
    private fun convertForecastListToDomain(list: List<Forecast>):
            List<ModelForecast> {
        return list.map { convertForecastItemToDomain(it) }
    }
    private fun convertForecastItemToDomain(forecast: Forecast): ModelForecast {
        return ModelForecast(convertDate(forecast.dt),
                forecast.weather[0].description, forecast.temp.max.toInt(),
                forecast.temp.min.toInt())
}
    private fun convertDate(date: Long): String {
        val df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.getDefault())
        return df.format(date * 1000)
    }
}
```

When we use two classes with the same name, we can assign an alias to one of them, so we do not need to write the full package name:

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

Another interesting thing about this code is how we convert from a forecast list to a domain model:

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

With this statement, we can loop through the collection and return a new list of conversions. Kotlin provides a lot of nice function operators in the List that can be applied to each item in this List and convert them in any way. Compare Java 7, which is one of Kotlin's powerful features. We will soon see all the different operators. Knowing their presence is important because they are much more convenient and can save a lot of time and templates.

Now, ready before writing the command:

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

Draw the data in the UI

`MainActivity` in the code some minor changes, because now there is real data needs to be filled into the adapter. Asynchronous calls need to be rewritten as:

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

Adapter also needs to be modified:

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

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

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

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

> **with function**
>
> > With is a very useful function that is included in Kotlin's standard library. It takes an object and an extension function as its argument, and then causes the object to extend the function. This means that all of the code we write in parentheses is an extension function of the object (the first argument), and we can use all of its public methods and properties just like this. This is very useful for simplifying the code when we do a lot of work on the same object.

In this chapter there are many new code to join, so check out the library code bar.

Operator overload

Kotin has some fixed number of symbolic operators, and we can use them easily in any class. The method is to create a method that calls the reserved operator keyword so that the behavior of the operator can be mapped to this method. Overloading these operators can increase code readability and simplicity.

Operator table

Here you can see a series of tables that include the `operator` and `corresponding methods`. The corresponding method must be implemented in the specified class by various possibilities.

unary operator

Operators	Functions
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()
a++	a.inc()
a--	a.dec()

binary operator

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

array operator

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

is equal to operator

Operators	Functions
a == b	a?.equals(b) ?: b === null
a != b	!(a?.equals(b) ?: b === null)

Equal operators are a bit different, in order to achieve the correct and appropriate checks to do a more complex conversion, because to get an exact function structure comparison, not just the specified name. The method must be implemented as follows:

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

The operators `===` and `!==` are used for identity checking (they are `==` and `!=` In Java, respectively), and they can not be overloaded.

function call

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

example

You can imagine that the Kotlin List implements the array operator, so we can access each item of the List like an array in Java. In addition: In the modified List, each item can also be set directly in a simple way:

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

If you remember, we have a data class called ForecastList, which is made up of a lot of other extra information. Interestingly it is possible to access each item directly instead of asking the inside of the list to get an item. Do a completely irrelevant thing, I'm going to implement a `size()` method that can slightly simplify the current adapter:

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

It will make our `onBindViewHolder` much simpler:

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

Of course there are `getItemCount()` methods:

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

The operator in the extended function

We do not need to extend our own classes, but I need to use the extension function to extend the classes we already exist to allow third-party libraries to provide more operations. A few examples, we can go to visit the List as a way to visit the `viewGroup` view:

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

Now really can be very simple from a `ViewGroup` through the position to get a view:

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

Do not forget to go to the Kotlin for Android developers repository to view the code.

Make Forecast list available

As a real app, the current list of each item layout should do some work. The first thing is to create a suitable XML, to meet our needs on the line. We want to display an icon, date, description and maximum and minimum temperatures. So let's create a layout called `item_forecast.xml`.

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

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

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

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

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

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

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

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

    </LinearLayout>
</LinearLayout>
```

Domain model and data mapping must generate a complete icon uil, so we can do this to load it:

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

In `ForecastDataMapper`:

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

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

We get the icon from the first request code, used to form the completion of the icon url. The easiest way to load an image is to use the image load library. `Picasso` is a good choice. It needs to be added to the `build.gradle` dependency:

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

So, Adapter also need a big change. Also need a click listener, we have to define it:

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

If you still remember the previous lesson, the `invoke` method can be omitted when called. So let's use it to simplify it. The listener can be called in two ways:

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

`ViewHolder` will be responsible for binding data to the new View:

```
class ViewHolder(view: View, val itemClick: OnItemClickListener) :
                RecyclerView.ViewHolder(view) {
    private val iconView: ImageView
    private val dateView: TextView
    private val descriptionView: TextView
    private val maxTemperatureView: TextView
    private val minTemperatureView: TextView

    init {
        iconView = view.find(R.id.icon)
        dateView = view.find(R.id.date)
        descriptionView = view.find(R.id.description)
        maxTemperatureView = view.find(R.id.maxTemperature)
        minTemperatureView = view.find(R.id.minTemperature)
    }

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

Now the constructor of the Adapter receives a `itemClick`. Creating and binding data is also simpler:

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

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

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

If you use the above code, `parent.ctx` will not be compiled successfully. Anko offers a number of extended functions to make Android programming easier. For example, the activitys, fragments, and others contain the `ctx` attribute, which returns the context through the `ctx` attribute, but is missing this property in the View. So we want to create a new name called `ViewExtensions.kt` file instead of `ui.utils`, and then add this extended attribute:

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

From now on, any View can use this property. This is not necessary because you can use the extended context attribute, but I think that if we use `ctx`, it will be more coherent in other classes. And that's a good example of how to use extended attributes.

Finally, the MainActivity calls the setAdapter, and the result is this:

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

As you can see, to create an anonymous inner class, we went to create an implementation of the interface just created the object. Looks not very good, right? This is because we have not started experimenting with another powerful functional programming feature, but you will learn how to convert the code to the simpler in the next chapter.

Go to the codebase to update the new code. UI starts to look better.

Lambdas

Lambda expression is a very simple way to define an anonymous function. Lambda is very useful because they avoid getting some of the abstract classes or interfaces that contain some of the functions and then implementing them in the class. In Kotlin, we take a function as a function of another function.

The click listener for the ForecastListAdapter

In the previous chapter, I was so hard to write the click listener's purpose is to better develop in this chapter. But it is time to use what you have learned to go to practice. We removed the listener interface from the ForecastListAdapter and replaced it with lambda:

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

The `itemClick` function takes a `forecast` parameter and does not return anything. `ViewHolder` can also be modified as follows:

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

The other code remains the same. Just change `MainActivity` :

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

We can simplify the last sentence. If the function only receives a parameter, then we can use `it` reference, rather than to specify the parameters on the left. So we can do that:

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

Extended language

Thanks to these changes, we can go to create your own `builder` and code blocks. We are already using some interesting functions, such as `with`. The following simple implementation:

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

This function takes a `T` type object and a function that is used as an extension function. Its implementation just let this object execute this function. Because the second argument is a function, so we can put it outside the parentheses, so we can create a block of code in which we can use `this` and direct access to all public methods and properties :

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

> Inline function
>
> > Inline functions are a bit different from ordinary functions. An inline function is replaced at compile time, not a real method call. This in some cases can reduce memory allocation and run-time overhead. For example, if we have a function, we only accept a function as its argument. If it is an ordinary function, the internal will create an object that contains the function. On the other hand, the inline function will replace the place where we call this function, so it does not need to generate an internal object for it.

Another example: we can create code blocks that only provide `Lollipop` or later to execute:

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

It just checks the version, and then if the conditions are met. Now we can do this:

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

For example, Anko is based on this idea to achieve `DSL` of the `Android Layout`. You can also see an example of `Kotlin reference use of DSL to write HTML`.

Visibility modifier

These modifiers in Kotlin are somewhat different from those used in our Java. The default modifier in this language is `public`, which saves a lot of time and characters. But here's a detailed explanation of how the different visibility modifiers work in Kotlin.

Modifier

private

The `private` modifier is the most restrictive modifier we use. It means that it can only be seen by the file where it is. So if we declare a class as `private`, we can not use it in a file that defines this class.

On the other hand, if we use the `private` modifier in a class, that access is restricted to this class. Even subclasses that inherit this class can not use it.

So first-class citizens, classes, objects, interfaces ... (ie, package members) are defined as `private`, so they will only be visible to the file where the definition is located. If they are defined in a class or interface, they are only visible to this class or interface.

protected

This modifier can only be used in class or interface members. A package member can not be defined as `protected`. Defined in a member, the same way as in Java: it can be seen by its own members and members that inherit it (for example, the class and its subclass).

internal

If it is a definition of `internal` package members, then the entire `module` visible. If it is a member of another area, it will depend on the visibility of that area. For example, if we write a `private` class, then the visibility of its `internal` modifier function will limit the visibility of the class with which it is located.

We can access the same `module` in the `internal` modified class, but can not access the other `module`.

> What is `module`
>
> > According to Jetbrains definition, a `module` should be a separate functional unit, it should be able to be compiled separately, run, test, debug. Depending on the modules of our project, you can create different `module`s in Android Studio. In Eclipse, these `module`s can be considered in a `workspace`'s different `project`.

public

You should probably think that this is the least restrictive modifier. This is the default **modifier**, members are modified to `public` at any place, it is clear that it is only limited to its fields. A member defined as `public` is included in a `private` decorated class, and this member is not visible outside of this class.

Constructor

All constructors are `public` by default. They are visible and can be used elsewhere. We can also use this syntax to modify the constructor to `private`:

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

Polish our code

We are already ready to use `public` for refactoring, but we have a lot of other details that need to be modified. For example, in `RequestForecastCommand`, we create the property `zipCode` in the constructor that can be defined as `private`:

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

What we have done is that we create an unmodifiable attribute zipCode whose value we can only get and can not modify it. So this little change makes the code look clearer. If we are in the preparation of the class, you think some attributes because of what reason can not be visible to others, then it is defined as `private`.

And, in Kotlin, we do not need to specify the return type of a function, which allows the compiler to infer it. Give an example of the return value type:

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

We can omit the return value type of the typical scenario is when we want to give a function or an attribute assignment time. Without having to write code blocks to achieve.

The rest of the changes are fairly simple, and you can synchronize them in the code base.

Kotlin Android Extensions

Another Kotlin team developed a new plugin that could make it easier to develop `Kotlin Android Extensions`. Currently only the binding of the view is included. This plugin automatically creates a lot of properties to let us directly access the view in XML. This way does not require you to explicitly find these views from the layout before you start using it.

The name of these attributes is from the corresponding view of the id, so we take id when the time to be very careful, because they will be a very important part of our class. The type of these attributes is also from the XML, so we do not need to carry out additional types of conversion.

One of the advantages of Kotlin Android `Extensions` is that it does not need to rely on other additional libraries in our code. It is only by the plug-in layer, when needed to generate the code needed to work, only need to rely on Kotlin's standard library.

How does it work behind it? The plugin will replace the function call function, such as access to the view and has a cache function, so that each time the property is called to get the view again. It should be noted that this cache will only be effective in `Activity` or `Fragment`. If it is added in an extension function, the cache will be skipped because it can be used in the `Activity` but the plugin can not be modified, so there is no need to add a cache function.

How to use Kotlin Android Extensions

If you still remember, now the project is ready to use Kotlin Android Extensions. When we created this project, we have already added this dependency to `build.gradle`：

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

The only thing that needs this plugin to do is add a specific "manual" `import` in the class to use this feature. We have two ways to use it：

`Activities` or `Fragments` 's Android Extensions

This is the most typical way to use it. They can be accessed as properties of `activity` or `fragment`. The name of the attribute is the id of the view in XML.

We need to use the `import` statement to start with `kotlin.android.synthetic` and then add the name of the XML we want to bind to the activity XML:

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

After that, we can access these views after `setContentView` is called. The new Android Studio version can add an embedded layout to the Activity default layout by using the `include` tag. It is important to note that for these layouts, we also need to add manual import:

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

`Views` of `Android Extensions`

The use of the front is still limited, because there may be a lot of code need to visit the view in XML. For example, a custom view or an adapter. For example, bind a view in xml to another view. The only difference is the need for `import`：

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

If we need an adapter, for example, we now want to access properties from the inflater's View:

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

Rebuild our code

Now it's time to use `Kotlin Android Extensions` to modify our code. The modification is fairly simple.

We start with `MainActivity`. We are currently using only the `forecastList` of RecyclerView. But we can simplify the code. First, add manual for `activity_main` XML import:

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

As mentioned before, we use id to access views. So I want to modify the `RecyclerView` id, do not use the underscore, making it more suitable for the Kotlin variable name. XML last as follows:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecastList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</FrameLayout>
```

And now, we can not need `find` this line:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    forecastList.layoutManager = LinearLayoutManager(this)
    ...
}
```

This is already the smallest simplification, because this layout is very simple. But `ForecastListAdapter` can also benefit from this plugin. Here you can use a device to bind these properties to the view, it can help us remove all `ViewHolder` of `find` code.

First, add a manual import for `item_forecast`:

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

Then we can now use the properties contained in `itemView` in `ViewHolder`. In fact, you can use these properties in any view, but it is clear that if the view does not contain the sub view to get will crash.

Now we can directly access the view of the property:

```
class ViewHolder(view: View, val itemClick: (Forecast) -> Unit) :
        RecyclerView.ViewHolder(view) {
    fun bindForecast(forecast: Forecast) {
        with(forecast){
            Picasso.with(itemView.ctx).load(iconUrl).into(itemView.icon)
            itemView.date.text = date
            itemView.description.text = description
            itemView.maxTemperature.text = "${high.toString()}"
            itemView.minTemperature.text = "${low.toString()}"
            itemView.onClick { itemClick(forecast) }
        }
    }
}
```

The Kotlin Android Extensions plugin helps us to reduce a lot of template code and simplify the way we visit view. Check out the latest code from the library.

Application singleton and delegates of attributes

We are going to implement a database soon, and if we want to keep the simplicity and level of our code (rather than adding all the code to the Activity), we need to have a simpler way to access the application context.

Delegate attribute

We may need a property with some of the same behavior, using `lazy` or `observable` can be very interesting to achieve reuse. Instead of declaring the same code again and again, Kotlin provides a way to delegate attributes to a class. This is what we know about the `delegate attribute`

When we use the `get` or `set` attribute, the `getValue` and `setValue` 'of the delegate are called.

The structure of the attribute delegate is as follows:

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

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

This is the type of delegate attribute. The `getValue` function takes a reference to a class and a property's metadata. The `setValue` function in turn receives a set value. If this property is not modifiable (val), there will be only one `getValue` function.

The following shows how the property is set up:

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

It uses the `by` keyword to specify a delegate object.

Standard commission

There is a series of standard delegates in Kotlin's standard library. They include most useful delegates, but we can also create our own delegates.

Lazy

It contains a lambda, when the first implementation of `getValue` when the lambda will be called, so this property can be delayed initialization. Subsequent calls will only return the same value. This is a very interesting feature when we do not have to need them before they are actually called for the first time. We can save memory and do not initialize before these properties really need it.

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

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

In this example, the database is not actually initialized until the first call to the `onCreate` . After that, we only ensure that the applicationContext exists and is ready to be used. The `lazy` operator is thread safe.

If you are not worried about multithreading or want to improve more performance, you can also use `lazy(LazyThreadSafeMode.NONE){ ... }`

Observable

This commission will help us monitor the changes we want to observe. When the `set` method of the observed property is called, it automatically executes the lambda expression we specified. So once the attribute is assigned a new value, we will receive the delegate attribute, the old value and the new value.

```
class ViewModel(val db: MyDatabase) {
    var myProperty by Delegates.observable("") {
        d, old, new ->
        db.saveChanges(this, new)
    }
}
```

This example shows that some of the ViewMode we need to care about, each time the value is modified, they will save them to the database.

Vetoable

This is a special `observable` , which lets you decide whether this value needs to be saved. It can be used to make some conditional judgments before it is actually saved.

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

The above delegate is only allowed to be saved when the new value is positive. In lambda, the last line represents the return value. You do not need to use the return keyword (essentially can not be compiled).

Not Null

Sometimes we need to initialize this property somewhere, but we can not be determined in the constructor, or we can not do anything in the constructor. The second case is common in Android: in the Activity, fragment, service, receivers ... ... In any case, a non-abstract attribute in the constructor before the implementation of the need to be assigned. In order to assign these attributes, we can not let it wait until we want to assign it to the time. We have at least two options.

The first is to use a nullable type and assign it to null until we have a value that really wants to be assigned. But we need to check in every place whether or not it is null. If we are sure that this property will not be null when we use it anymore, this may cause us to write some of the necessary code.

The second option is to use the `notNull` delegate. It will contain a nullable variable and will allocate a true value when we set this property. If the value is not allocated before it is fetched, it throws an exception.

This is useful in this example of a single case App:

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

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

Map values from the Map

Another way to attribute the delegate is that the value of the attribute gets the value from a map, and the name of the attribute corresponds to the key in the map. This delegate allows us to do something very powerful because we can easily create an instance of an object from a dynamic map. If we import `kotlin.properties.getValue` , we can map from the constructor to the `val` property to get an unmodifiable map. If we want to modify the map and properties, we can also import `kotlin.properties.setValue` . The class requires a `MutableMap` as a constructor argument.

Imagine that we loaded a configuration class from a Json and then assigned their keys and values to a map. We can just by passing in a map constructor to create an instance:

```
import kotlin.properties.getValue

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

As a reference, here I show under this class how to create a necessary map:

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

How to create a custom delegate

Let's say what we want to achieve, for example, we create a `notNull` delegate, it can only be assigned once, if the second assignment, it will throw exception.

The Kotlin library provides several interfaces, and our own delegates must implement: `ReadOnlyProperty` and `ReadWriteProperty` . Depending on whether the object we are entrusted is `val` or `var` .

The first thing we have to do is create a class and then inherit `ReadWriteProperty` :

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

        override fun getValue(thisRef: Any?, property: KProperty<*>): T {
            throw UnsupportedOperationException()
        }

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

This delegate can be applied to any non-null type. It receives any type of reference, and then uses the same as getter and setter. Now we need to implement these functions.

- If the Getter function has been initialized, it will return a value, otherwise it will throw an exception.
- Setter function if it is still null, then assignment, otherwise it will throw an exception.

```
private class NotNullSingleValueVar<T>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null
    override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return value ?: throw IllegalStateException("${desc.name} " +
                "not initialized")
    }

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

Now you can create an object and then add the function using your delegate:

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

Reimplementing Application instantiation

In this scenario, the commission can help us. We will not be null until our singleton is null, but we can not use the constructor to initialize the property. So we can use `notNull` delegate:

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

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

In this case there is a problem, we can anywhere in the app to modify this value, because if we use `Delegates.notNull()` , the property must be var. But we can use the delegate just created, so that you can protect a little more. We can only modify this value once:

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

Although, in this case, using a single case may be the easiest way, but I would like to show you how to create a custom delegate in the form of code.

Create a SQLiteOpenHelper

As you know, Android uses SQLite as its database management system. SQLite is a database embedded in the app, it is really very light. That's why this is a good choice for mobile app app.

Nevertheless, its API for operating the database is very native in Android. You will need to write a lot of SQL statements and your object with `ContentValues` or `Cursors` between the parsing process. Very grateful to the joint use of Kotlin and Anko, we can greatly simplify these.

Of course, there are a lot of Android can be used on the database library, thanks to Kotlin interoperability, all of these libraries can be used normally. But for a simple database can not use any of them, after a minute you can see.

ManagedSqliteOpenHelper

Anko offers a lot of powerful SqliteOpenHelper to greatly simplify code. When we use a generic `SqliteOpenHelper`, we need to call `getReadableDatabase()` or `getWritableDatabase()`, and then we can perform our search and get the result. After that, we can not forget to call `close()`. Use `ManagedSqliteOpenHelper` we only need:

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

In lambda inside, we can directly use the `SqliteDatabase` function. How does it work It's really fun to read the way the Anko function is implemented. You can learn a lot of Kotlin's knowledge from here:

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

First, `use` receives an extension function of `SQLiteDatabase`. This means that we can use `this` in curly braces and are in the `SQLiteDatabase` object. This function extension can return a value, so we can do this:

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

Keep in mind that in a function, the last line represents the return value. Because T does not have any restrictions, so we can return any object. Even if we do not want to return any value on the use of `Unit`.

With `try-finall`, the `use` method ensures that the database will be shut down regardless of whether the database operation succeeds or fails.

Also, there are a lot of useful extension functions in `sqliteDatabase`, which we will use later. But now let's first define our table and implement `SqliteOopenHelper`.

Define tables

Create a few `objects` s so that we can avoid table name spelling errors, repeat and so on. We need two tables: one to save the city's information and the other to keep the weather forecast for the day. The second table will have a field associated with the first table.

`CityForecastTable` provides the name of the table and there is a need for an id (the city's zipCode), the name of the city and the country of the country.

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

`DayForecast` has more information, just as you see a lot of columns below. The last column `cityId` , used to keep belongs to the city id.

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

Implement SqliteOpenHelper

The basic composition of our `SqliteOpenHelper` is the creation and updating of the database and the provision of a `SqliteDatebase` so that we can use it to work. The query can be extracted and placed in other classes:

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

We used in the previous section we used to create the `App.instance`, this time we also include the database name and version. These values we will be defined with `SqliteOpenHelper` in the `companion object`:

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

`Instance` This property uses the `lazy` delegate, which indicates that it will not be created until it is actually called. In this way, if the database has never been used, we do not need to create this object. The generic `lazy` delegate code block can prevent multiple objects from being created in multiple different threads. This will only occur in the two threads in the colleague time to visit the `instance` object, it is difficult to happen but there are specific to see the implementation of the app. No one how, `lazy` commission is thread safe.

In order to create these defined tables, we need to provide an implementation of the `onCreate` function. When there is no library to use, the creation of the table will be through our preparation of the original contains our definition of the column and type of `CREATE TABLE` statement to achieve. However, Anko provides a simple extension function that takes a table name and a series of `Pair` objects built by column names and types:

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

- The first argument is the name of the table
- The second parameter, when it is true, will check whether the table exists before it is created.
- The third argument is a `vararg` parameter of the `Pair` type. `Vararg` also exists in Java, which is a function that passes in a function that links many of the same types of parameters. This function also receives an array of objects.

Anko has a special type called `SqlType`, which can be mixed with `SqlTypeModifiers`, such as `PRIMARY_KEY`. `Operator` is rewritten as before. The `plus` function will combine the two in the right way and then return a new `SqlType`:

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

As you can see, she will combine multiple modifiers.

But back to our code, we can modify it better. The Kotlin standard library contains a function called `to`, and once again, let's show the power of Kotlin. It takes the extended function of the first argument, receives another object as a parameter, assembles the two and returns a `Pair`.

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

Because a function with a function argument can be used inline, so the result is very clear:

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

Then, apply them to the creation of the table:

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

That's what the whole function looks like:

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

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

We have a similar function used to delete the table. `OnUpgrade` will just delete the tables and then rebuild them. We just put our database as a cache, so this is a simple and safe way to ensure that our tables will be rebuilt as we would expect. If I have important data to keep, we need to optimize the code for `onUpgrade` to make the corresponding data transfer based on the database version.

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

Dependent injection

I tried not to add very complicated structural code, keep the code of the testability and good practice, and I think I should use Kotlin to simplify the code from other aspects. If you want to learn about some of the reverse or dependency injection topics, you can check out a series of articles about using Dagger in Android. The first article has a brief description of their team.

A simple way, if we want to have some classes that are independent of other classes, make it easier to test and write code that is easy to extend and maintain when we need to use dependency injection. We do not instantiate in the class, we provide them in other places (usually through the constructor) or instantiate them. In this way, we can use other objects to replace them. For example, you can achieve the same interface or in the use of mocks tests.

But now that these dependencies must be provided in some places, dependency injection consists of classes that provide partners. These are usually done using a dependency injector. Dagger is probably the most popular dependency injector on Android. Of course, when we provide a certain degree of complexity is a good alternative.

But the smallest alternative is to use the default value in this constructor. We can provide a dependency to the constructor's parameters by assigning default values, and then provide different instances in different situations. For example, in our `ForecastDbHelper` we can use a more intelligent way to provide a context:

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

Now we have two ways to create this class:

```
val dbHelper1 = ForecastDbHelper() // 它会使用 App.instance
val dbHelper2 = ForecastDbHelper(mockedContext) // 比如，提供给测试tests
```

I will use this feature everywhere, so I will continue to explain after explaining it clearly. We already have tables, so it's time to start adding and asking for them. But before that, I would like to talk about the combination and function operators. Do not forget to check the codebase to find the latest code.

Set and function operators

We have used the collection in our project, but it is time to show how powerful they are after combining the function operators. On the functional programming is very good that we do not have to explain how we do, but directly that I want to do. For example, if I want to filter a list, do not have to create a list, traverse each of the list, and then if you meet certain conditions into a new collection, but directly eat the filer function and indicate that I want to use Of the filter. In this way, we can save a lot of code.

Although we can directly use the collection in Java, but Kotlin also provides some of the local interface you want to use:

- **Iterable**: parent class. All we can traverse a series of are to achieve this interface.
- **MutableIterable**: A support for traversal can also be performed while deleting the Iterables.
- **Collection**: This class is a collection of norm. We can access through the function can return the size of the collection, whether it is empty, whether it contains one or some items. All the methods of this collection provide queries because connections are unmodifiable.
- **MutableCollection**: A collection that supports adding and deleting items. It provides additional functions, such as `add` , `remove` , `clear` and so on.
- **List**: probably the most popular collection type. It is a normative and orderly collection. Because of its order, we can use the `get` function to access through the position.
- **MutableList**: A List that supports adding and deleting items.
- **Set**: A disordered collection that does not support duplicate items.
- **MutableSet**: A set that supports adding and deleting items.
- **Map**: a collection of key-value pairs. Key is unique in the map, that is to say there can not be two pairs of keys is the same key value exists in a map.
- **MutableMap**: A map that supports adding and deleting items.

There are many different sets of available function operators. I want to show you through an example. It is useful to know which optional operators are used, because it makes it easier to tell when they are used.

Total operator

any

Returns true if at least one element matches the given judgment condition.

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

all

Returns true if all elements match the given judgment condition.

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

count

Returns the total number of elements that match the criteria given.

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

fold

Accumulates all the elements from the first item to the last item on the basis of an initial value.

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

foldRight

Same as `fold`, but the order is from the last item to the first item.

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

forEach

Traverse all the elements and perform the given operation.

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

forEachIndexed

And `forEach`, but we can get the index of the element at the same time.

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

max

Returns the largest item, or null if no.

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

maxBy

Returns the largest item according to the given function, or null if it does not.

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

min

Returns the smallest item, or null if none.

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

minBy

Returns the smallest item based on the given function, or null if it does not.

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

none

Returns true if no element matches a given function.

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

reduce

Same as `fold`, but not an initial value. Through a function from the first item to the last one to accumulate.

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

reduceRight

Same as `reduce`, but the order is from the last item to the first item.

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

sumBy

Returns the sum of all the data after each function conversion.

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

Filter operator

drop

Returns a list containing all the elements that removed the first n elements.

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

dropWhile

Returns a list of the specified elements removed from the first item according to the given function.

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

dropLastWhile

Returns a list of the specified elements removed from the last item based on the given function.

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

filter

Filter all elements that meet the given function's condition.

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

filterNot

Filter all elements that do not meet the given function's condition.

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

filterNotNull

Filter all elements that are not null in all elements.

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

slice

Filter the elements of the specified index in the list.

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

take

Returns the n elements starting from the first.

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

takeLast

Returns the n elements starting from the last one

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

takeWhile

Returns the element that matches the given function condition from the first.

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

Mapping operator

flatMap

Traverse all the elements, create a collection for each, and finally put all the collections in a collection.

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

groupBy

Returns a map that is grouped according to a given function.

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

map

Returns a List of each element that is converted according to the given function.

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

mapIndexed

Returns a List of each element consisting of a given function that contains the element index.

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

mapNotNull

Returns a List of each non-null element that is converted according to the given function.

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

Element operator

contains

Returns true if the specified element can be found in the collection.

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

elementAt

Returns the element corresponding to the given index, which throws `IndexOutOfBoundsException` if the index array is out of bounds.

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

elementAtOrElse

Returns the element corresponding to the given index. If the index array is out of bounds, the default value is returned according to the given function.

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

elementAtOrNull

Returns the element corresponding to the given index, or null if the index array is out of bounds.

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

first

Returns the first element that matches the given function's condition.

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

firstOrNull

Returns the first element that matches the given function's condition, or null if there is no match.

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

indexOf

Returns the first index of the specified element, or `-1` if it does not exist.

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

indexOfFirst

Returns the index of the first element that matches the condition of the given function, or `-1` if it does not match.

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

indexOfLast

Returns the index of the last element that matches the condition of the given function, and returns `-1` if it does not match.

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

last

Returns the last element that matches the given function condition.

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

lastIndexOf

Returns the last index of the specified element, or `-1` if it does not exist.

lastOrNull

Returns the last element that matches the given function condition, or null if there is no match.

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

single

Returns a single element that matches the given function, and throws an exception if it does not match or exceeds one.

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

singleOrNull

Returns a single element that matches the given function, or null if none or more than one.

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

Production operator

merge

The two sets are merged into a new, identical index element by a given function to merge into a new element as an element of the new collection, returning the new collection. The size of the new collection is determined by the size of the smallest set.

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

partition

Divide a given set into two, the first set is composed of elements of the original set of elements that match the given function condition to return `true`, and the second set is matched by the original set of each element Set the function condition to return the `false` 's element.

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

plus

Returns a collection containing all the elements in the original collection and a given set. Because of the name of the function, we can use the `+` operator.

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

zip

Returns a List consisting of `pair`, each `pair` consists of the same index elements in both collections. The size of the returned List is determined by the smallest set.

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

unzip

Generates a Pair containing a List from the List containing the pair.

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

Order operator

reverse

Returns a list in the reverse order of the specified list.

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

sort

Returns a list of naturally sorted ones.

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

sortBy

Returns a list sorted by the specified function.

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

sortDescending

Returns a descending sorted list.

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

sortDescendingBy

Returns a list that is sorted by descending sorted by the specified function.

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

Save or query data from the database

In the previous section we talked about the creation of `SQLiteOpenHelper`, but we need to have the means to save our data to the database, or to query the data from our database if necessary. Another one called `ForecastDb` class will do this.

Create the database model class

But first of all, we have to create a model class for the database. Do you remember the way we saw the map delegate before We have to map these properties directly to the database, and vice versa.

We first look under the `CityForecast class` :

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

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

The default constructor gets a map containing the attributes and the corresponding values, and a dailyForecast. Thanks to the delegate, these values are mapped to the corresponding attributes according to the name of the key. If we want the mapping process to run perfectly, then the name of the property must be exactly the same as the name in the database. We will talk about the reasons behind.

However, the second constructor is also necessary. This is because we need to map from the domain to the database class, it can not use the map, set the value from the property is also convenient. We pass in an empty map, but again, thanks to the delegate, when we set the value to the property, it will automatically add all the values to the map. In this way, we are ready to map to the database. Using these useful code, I will see it running as magical as magic.

Now we need the second class, DayForecast, it will be the second table. It contains every column in the table as its property, and it also has a second constructor. The only difference is that you do not need to set id because it will grow from SQLite.

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

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

These classes will help us to map each SQLite table with each other.

Write and query the database

`SqliteOpenHelper` is just a tool that is a channel between SQL World and OOP. We want to create several new classes to request data that has been saved in the database and save the new data. The defined class will use `ForecastDbHelper` and `DataMapper` to convert the data in the database to `domain models`. I still use the default value to achieve a simple dependency injection:

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

All the functions use the `use()` function mentioned in the previous section. The value returned by lambda is also used as the return value for this function. So let's define a function that uses `zip code` and `date` to query a `forecast` :

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

So there is no explanation: we use the `use` function to return the results as a result of this function.

Query a forecast

The first query to do is the daily weather forecast, because we need this list to create a `city` object. Anko provided a simple request builder, so let's take advantage of this favorable condition:

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

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

The first line, `dailyRequest`, is part of the `where` 's in the query. It is the first parameter that the `whereSimple` function needs, which is very similar to what we do with the usual helper. There is another simplified `where` function, which requires some tags and values to match. I do not like this way, because I think this adds to the code of the template, although this is the analysis of the value of String is very beneficial to us. At last it looks like this:

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

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

You can choose one of the ways you like. `Select` function is very simple, it just needs to be a lookup table name. `Parse` function when there will be some magic in it. In this example we assume that the result of the request is a list, using the `parseList` function. It uses the `RowParser` or `RapRowParser` function to convert the cursor into a collection of objects. The two differences are that `RowParser` is the order of the columns, and `MapRowParser` is the key name from the map.

There are two overloaded conflicts between them, so we can not directly create the needed objects in a simplified way. But nothing can not be solved by an extended function. I created a function that takes a lambda function to return a `MapRowParser`. The parser will call this lambda to create this object:

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

This function can help us simply go to the `parseList` query results:

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

The `immortable map` received by the parser is transformed into a `mutable map` (which we can modify in `database model`) by using the corresponding `HashMap` constructor. The `HashMap` will be used in the constructor in `DayForecast`.

So, this query returns a `Cursor`, to understand what is happening behind this scene. `ParseList` will iterate over it, then get every row of `Cursor` until the last one. For each row, it creates a map that contains the key for this column and the assignment to the corresponding key. And then return this map to the parser.

If the query does not have any results, `parseList` will return an empty list.

The next step to query the city is the same way:

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

The difference is that we are using `parseOpt`. This function returns a nullable object. The result can be a null or a single object, depending on whether the request can query the data in the database. There is another function called `parseSingle`, which is essentially the same, but it returns a non-nullable object. So if it does not find this data in the database, it will throw an exception. In our example, the first time you query a city, it certainly does not exist, so using `parseOpt` will be safer. I also created a nice function to prevent the creation of the objects we need:

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

Finally, if the returned city is not null, we use `dataMapper` to convert it to `domain object` and return it. Otherwise, we return null directly. You should remember that the last line of lambda represents the return value. So this will return a `CityForecast?` Type of object:

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

The `DataMapper` function is simple:

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

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

The last complete function is as follows:

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

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

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

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

Another funny function of Anko is here to show that you can use `classParser()` instead of `MapRowParser`, which is based on the column name to generate objects by reflection. I like another way because I do not need to use reflection and have control over the conversion, but sometimes it's useful for you.

Save a forecast

The `saveForecast` function simply clears the data from the database, then converts the `domain` model to the database model, and then inserts the `forecast` and `city forecast` 'for each day. This structure is simpler than before: it returns the data from `database helper` through the `use` function. In this example we do not need to return the value, so it will return `Unit`.

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

First, we empty the two tables. Anko did not offer a pretty way to do this, but that did not mean we could not. So we created an extension function of `SQLiteDatabase` so that we can execute it like a SQL query:

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

Empty the two tables:

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

Now, it is time to convert the implementation of `insert` after the return of the data. At this point you may be up to the fans of my `with` function:

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

The way to convert from `domain model` is also straightforward:

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

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

In the code block, we can use `dailyForecast` and `map` without using references and variables, just as we do in this class. For the insert we use another Anko function, it takes a table name and a `vararg` modified `Pair<String, Any>` as a parameter. This function converts `vararg` to the `ContentValues` object needed by the Android SDK. So our task is to convert `map` into a `vararg` array. We created an extension function for `MutableMap`:

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

It supports the null values (this is the condition `map delegate`), convert it to a non-null value `Pairs` (`select` function needed) `Array` thereof. Do not worry even if you do not fully understand this function, I will soon talk about the nullability.

So, this new function we can use:

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

It inserted a new line of data in the `CityForecast`. Expressed using this array is broken down into a `vararg` front `toVarargArray` parameter function results. This is handled automatically in Java, but we need to specify in Kotlin.

The weather forecast is the same every day:

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

So, by using the `map`, we can convert the class to a data table in a very simple way, and vice versa. Because we have created a new extension function, we can use in other projects, this is really valuable place.

The complete code for this function is as follows:

```
fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {
    clear(CityForecastTable.NAME)
    clear(DayForecastTable.NAME)

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

There are a lot of code needed in this chapter, so you can check out the checkout in the code base.

Null in Kotlin

If you are working with Java 7, null security is one of the most interesting features of Kotlin. But as you see in this book, it does not seem to exist, until the last chapter we almost do not need to worry about it.

We have sometimes need to define a variable package that does not contain a value by thinking about null by the 100 million dollars error we created ourselves. In Java, although annotations and the IDE have helped us a lot in this area, we can still do this:

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

This code can be compiled perfectly (you may get a warning from the IDE) and then execute it normally, but obviously it will throw a `NullPointerException` '. This is quite safe. And according to our idea, we should go to control everything, as the code grows, we will slowly control some of the null. So eventually get a lot of `NullPointerException` or lose a lot of null checks (maybe both).

How the nullable type works

Most of the modern languages use certain methods to solve this problem, Kotlin's method is similar to other languages is quite different and different. But the gold rule is the same: if the variable can be null, the compiler forces us to do it in some way.

Specifying a variable is nullable by **adding a question mark** at the end of the type. Because everything in Kotlin is an object (even the original data type in Java), everything is nullable. So, of course we can have a nullable integer:

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

—A nul type, you can not use it directly before you check it out. This code can not be compiled:

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

The previous line of code is marked as null, and the compiler will know it, so you can not use it before you check it. There is also a feature that when we check the nullability of an object, the object is automatically transformed into a non-nullable type, which is the smart conversion of the Kotlin compiler:

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

In `if` , `a` changes from `Int?` To `Int` , so we can use it without having to check it again. `If` product code, of course, we have to check processing. This is only valid if the variable can not be changed at this time, because otherwise the value may be modified by another thread, and the previous check will return false. `val` attribute or local (`val` or `var`) variable.

It sounds like it will make things more. Do we have to write a lot of code to carry out the checkability can be made? Of course not, first of all, because most of the time you do not need to use null type. Null references are not useful in our imagination, and you will find this when you want to figure out whether a variable can be null. But Kotlin also has its own solution to make it more concise. For example, we simplified the code as follows:

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

Here we use the secure access operator (`?`). Only when this variable is not null will go to the implementation of the previous line of code. Otherwise it will not do anything. And we can even use **Elvis operator** (`?:`):

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

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

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

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

Then we may encounter this scenario, we make sure we are using a non-null variable, but this type is nullable. We can use the `!!` Operator to force the compiler may skip execution restriction checking null type:

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

The above code will be compiled, but it will obviously crash. So we want to make sure we can only use it in certain circumstances. Usually we can choose ourselves as a solution. If a piece of code that is so full of `!!`, it smelled of code that is badly handled the.

Can be null and Java libraries

Well, the previous section explains the use of Kotlin code to work perfectly. But what happens with the regular Java libraries and the Android SDK? In Java, all objects can be defined as null. So we have to deal with a lot of potential null variables that can not be null in reality. This means that our code may eventually have hundreds of `!!` operators, which is definitely not a good idea.

When we go to the Android SDK, you may see that all Java method parameters are marked as a single `!`. For example, Java in some way to get the object in the Kotlin display returns `Any!`. This means that the developer himself decides whether the variable is null or not.

Fortunately, the new version of Android began using the `@Nullable` and `@NonNull` annotations to see if the argument could be null or whether a function could return null. When we suspect that we can enter the source code to check whether we will receive a null object. My guess is that in the future, the compiler can read these annotations and then force (or at least suggest) a better way.

Now, when a Jetbrains `@Nullable` annotation (which differs from an Android comment) is annotated in a non-null variable, a warning is obtained. The relative does not happen on the `@NotNull` annotation.

So let's give an example if we create a Java test class:

```
import org.jetbrains.annotations.Nullable;
public class NullTest {

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

And then used in Kotlin：

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

We will find that a warning is displayed on the `getObject` function. But this is only a compiler check from now on, and it does not yet know the annotations of Android, so we may have to spend more time waiting for a smarter way. In any case, using source code annotations and some Androd SDK knowledge, we are also very difficult to make mistakes.

Such as rewriting the `onCraete` function of `Activity`, we can decide whether to make `savedInstanceState` possible null:

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

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

These two methods will be compiled, but the second is wrong, because an Activity is likely to receive a null bundle. Just be careful a little bit enough. When you have questions, you can then use the nullable object and then deal with the possible null. Remember, if you are using `!!`, it may be because you are sure that the object can not be null, and if so, please define it as non-null.

This flexibility is really necessary in the Java library, and as the compiler evolves, we may see a better interaction (which may be annotated), but now that the mechanism is flexible enough.

Create business logic to access data

After implementing the access to the server and interacting with the local database, it is time to integrate things together. The logical steps are as follows:

- Get data from the database
- Check if there is data for the corresponding week
- if yes, return to UI and render
- If not, request the server to get the data
- The result is saved in the database and returned to the UI render

But our `commands` should not handle all of these logic. Data source should be a specific implementation so that it can be easily modified, so add some extra code, then `command` abstracted from data access sounds like a good way. In our implementation, it traverses the entire list until the result is found.

So we first come to the interface to define some of our implementation `provider` need to use the data source:

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

`Provider` needs one to receive `zip code` and a `date`, then it should be based on that day to return the weather forecast for the week.

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

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

`Forecast provider` receives a list of data sources, passed in the constructor (for example, for testing), but I set the source default to `SOURCES` List defined in the `companion object`. I will use the database's data source and server-side data source. The order is important because it traverses the sources according to the order, and then stops the query once a valid return value is obtained. The logical order is first in the local query (local database), and then through the API query.

So the main function of the code is as follows：

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

It will get the first result that is not null and then return. When I searched for a large number of function operators in Chapter 18, I did not find exactly what I wanted. So when i look at the source code of Kotlin, I copied the `first` function and then modify them to achieve the purpose I want:

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

The function receives an assert function that takes a `T` type object and then returns a value of type `R?`. This means that `predicate` can return a null type, but our `firstResult` can not return null. This is why the reason for returning `R`.

How does it work? It will traverse each element in the collection and then execute the assert function. When the result of this assertion function is not null, the result is returned.

If we can allow sources to return null, then we can use the `firstOrNull` function instead. The difference is that the last line returns null and throws exception. But I am not in the code inside to deal with these details.

In our example, `T = ForecastDataSource`, `R = ForecastList`. But remember that the function specified in `ForecastDataSource` returns a SearchList, that is, `R?`, So everything is so perfectly matched. `requestSource` makes the preceding function look more readable:

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

If the result is not null and the number is also matched, the query is executed and only one data is returned. Otherwise, the data source does not have enough data to return a successful result.

The function `todayTimeSpan()` calculates the time of the millisecond today and excludes the "time difference". Some of the data sources (the database in our example) may need it. Because if we do not specify more information, the server default is today, so we do not need to set it.

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

The complete code for this class is as follows:

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

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

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

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

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

We have defined a `ForecastDb`. Now we need to implement `ForcastDataSource`:

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

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

`ForecastServer` has not yet been implemented, but this is very simple. It will receive data from the server after the use of `ForecastDb` to save to the database. In this way, we can cache the data to the database, provided to the future of the query.

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

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

It also uses the `data mapper` we created before. Finally, we changed the names of some of the functions to make it more similar to the mapper we used before in `database model`. You can view the `provider` to see the details.

The rewritten method is used to request the server to convert the results to `domain objects` and save them to the database. It finally queries the database to return the data, because we need to use the word id to insert into the database.

This is the last step in the implementation of the `provider`. Now we need to start using it. `ForecastCommand` will no longer interact directly with the server, nor will it convert the data to `domain model`.

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

    companion object {
        val DAYS = 7
    }

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

Other modifications include the renaming and the structural adjustment of the package. View the corresponding submission in Kotlin for Android Developers repository.

Flow control and ranges

I used some conditional expressions in our code, but it is time to go deeper to explain them. We usually use procedural programming language when we rarely use code flow control mechanism to write (some procedural programming language has almost disappeared), but they are still very useful. It is also a new powerful idea to make it easier to solve some of the problems in a particular situation.

If expression

In Kotlin everything is an expression, that is to say everything returns a value. If the `if` condition does not contain an exception, then we can use it as we usually do:

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

We can also assign the result to a variable. We used it many times in our code:

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

This also means that I do not need to have a ternary operator like Java, because we can use it for simple implementation:

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

So the `if` expression always returns a value. If a branch returns a Unit, the entire expression will also return to Unit, which can be ignored. In this case, its usage is the same as the .

when expression

`when` expression is similar to `switch/case` in Java, but much more powerful. This expression will try to match all possible branches until a satisfactory one is found. Then it will run the expression on the right. Unlike Java's `switch/case` ', the argument can be of any type, and the branch can also be a condition.

For the default option, we can add a `else` branch, which will be executed before any condition matches. The code that executes after a successful match can also be a block of code:

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

Because it is an expression, it can also return a value. We need to consider when to use as an expression, it must cover the possibility of all branches or implement the `else` branch. Otherwise it will not be compiled successfully:

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

As you can see, the condition can be a series of comma-separated values. But it can be more matching. For example, we can detect the parameter type and make judgments:

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

In the code on the right side, the parameters are automatically converted, so you do not need to explicitly do the type conversion.

It also makes it possible to detect the parameter no in an array range or even a collection range (I will speak this later in this chapter):

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

Or you can even get rid of the almost crazy check of the parameters needed. It can be replaced with a simple `if/else` chain:

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

For loop

Although you will not use the for loop too much after you have used the collection's function operator, the for loop is still useful in some cases. Provide an iterator which can act on anything above:

```
for (item in collection) {
    print(item)
}
```

If you need more typical iterations using index, we can also use `ranges` (anyway it's usually a more intelligent solution):

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

In a iteration of an array or list, a series of indexes can be used to get to the specified object, so the above method is not necessary:

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

While and do / while loops

You can also use the `while` loop, although both of them are not particularly useful. They can usually be easier, more visually easier to understand the way to solve a problem, two examples:

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

do{
    val y = retrieveData()
} while (y != null) // y在这里是可见的!
```

Ranges

It is difficult to explain `control flow`, if not talk about skipping. But their scope is much wider. `Range` expression uses a `..` operator, which is defined to implement a `RangTo` method.

`Ranges` helps us to use a lot of creative ways to simplify our code. For example, we can put it:

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

转化成：

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

`Range` is defined as any type that can be compared, but for numeric types, the comparator optimizes it by converting it to a simple Java code to avoid extra overhead. Numeric `scope`s can also be iterated, and the compiler will convert them to the same bytecode as the for loop of index in Java:

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

`Ranges` will grow by default, so if the following code is used:

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

It will not do anything. But you can use the `downTo` function:

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

We can use `step` in `range` to define a different gap from 1 to a value:

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

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

If you want to create an open range (you do not include the last term, you can use the `until` function:

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

This line will print from 0 to 3, but will skip the last value. That is to say `0 until 4 == 0..3`. When iterating over a list, it is easier to understand `(i in 0 until list.size)` than `(i in 0..list.size - 1)`.

As mentioned before, the use of `ranges` does have a creative way. For example, a simple way to get a Views list from a `ViewGroup` can do this:

```
val views = (0..viewGroup.childCount - 1).map { viewGroup.getChildAt(it) }
```

`ranges` mix function and operator we can avoid the use of loop iterations to a set of clear, there is definitely going to add that we create a list views are. Everything is done in a line of code.

If you want to know more about the implementation of `ranges` and more examples and swimming information, you can go to Kotlin reference

Create a detail interface

When we clicked on the main screen, we would like to jump to a detail interface and see some additional information about the weather forecast for that day. We just clicked on one after just showing a toast, but now is the time to modify it.

Prepare for a request

Because we need to know which item we want to show in the details of the interface, so the logic tells us need to send a weather forecast `id` to the details interface. So `domain model` needs a new `id` attribute:

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

`ForecastProvider` also requires a new function that returns the result after the request with `id` . `DetailActivity` will need to receive the request via the `id` request to get the weather forecast data. Since all requests will iterate over all the data sources and return the first non-null result, we can extract and define a new function:

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

This function uses a non-null type as a paradigm. It will receive a function and return a nullable object. Where the received function receives a `ForecastDataSource` and returns an nullable object. We can rewrite the last request and write a new one as follows:

```
fun requestByZipCode(zipCode: Long, days: Int): ForecastList = requestToSources {
    val res = it.requestForecastByZipCode(zipCode, todayTimeSpan())
    if (res != null && res.size() >= days) res else null
}

fun requestForecast(id: Long): Forecast = requestToSources {
    it.requestDayForecast(id)
}
```

Now the data source needs to implement a new function:

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

`ForcastDb` will always get the desired value in the last request to be cached, so we can get it in this way:

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

`Select` from the query is very similar to the previous one. I created another tool function called `byId` because requests from `id` are very generic, and using a function like this simplifies the process and is more readable. The realization of the function is also quite simple:

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

It just uses the `whereSimple` function to use the `_id` field to query the data. This function is quite common, but as you can see, you can create the required extension functions based on the needs of your database structure, which can greatly simplify the readability of your code. `DataMapper` has some minor changes that are not worth mentioning. You can view them through the code base.

On the other hand, `ForecastServer` will no longer be used because the information will always be cached in the database. We can in some strange scenes to achieve some code protection, but we do not do any treatment in this example, so if it happens will only throw an exception:

```
override fun requestDayForecast(id: Long): Forecast?
        = throw UnsupportedOperationException()
```

> `Try` and `throw` are expressions
>
> > In Kotlin, almost everything is an expression, that is to say everything will return a value. This is very important in functional programming, when you use `try-catch` to deal with boundary problems or when throwing an exception. For example, in the previous example, we can assign an exception to the result, even if they are not the same type, rather than having to create a complete code block. It is also useful when we need to throw an exception in a `when` branch:
> >
> > ```
> >     val x = when(y){
> >                 in 0..10 -> 1
> >                 in 11..20 -> 2
> >                 else -> throw Exception("Invalid")
> >     }
> > ```
> >
> > `Try-catch` is the same, we can according to the results of try to assign a value:
> >
> > ```
> >     val x = try{ doSomething() }catch{ null }
> > ```

The last thing we need to do is create a command in the new activity to execute the request:

```
class RequestDayForecastCommand(
        val id: Long,
        val forecastProvider: ForecastProvider = ForecastProvider()) :
        Command<Forecast> {
    override fun execute() = forecastProvider.requestForecast(id)
}
```

The request returns a result of the `Forecast` 'that will be used for the activity drawing UI.

Provide a new activity

Now we are going to create a `DetailActivity`. Our details activity will receive a set of parameters passed from the main activity: `forecast id` and `city name`. The first argument will be used to request data from the database, and the city name is used to display on the toolbar. So we first need to define the name of a set of parameters:

```
public class DetailActivity : AppCompatActivity() {
    companion object {
        val ID = "DetailActivity:id"
        val CITY_NAME = "DetailActivity:cityName"
    }
    ...
}
```

In the `onCreate` function, the first step is to set the content view. UI is very simple, but for the app is enough:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center_vertical"
        tools:ignore="UseCompoundDrawables">

        <ImageView
            android:id="@+id/icon"
            android:layout_width="64dp"
            android:layout_height="64dp"
            tools:src="@mipmap/ic_launcher"
            tools:ignore="ContentDescription"/>

        <TextView
            android:id="@+id/weatherDescription"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="@dimen/spacing_xlarge"
            android:textAppearance="@style/TextAppearance.AppCompat.Display1"
            tools:text="Few clouds"/>
    </LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView
            android:id="@+id/maxTemperature"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_margin="@dimen/spacing_xlarge"
            android:gravity="center_horizontal"
            android:textAppearance="@style/TextAppearance.AppCompat.Display3"
            tools:text="30"/>
        <TextView
            android:id="@+id/minTemperature"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_margin="@dimen/spacing_xlarge"
            android:gravity="center_horizontal"
            android:textAppearance="@style/TextAppearance.AppCompat.Display3"
            tools:text="10"/>
    </LinearLayout>
</LinearLayout>
```

And then set it in the `onCreate` code. Use the name of the city to set the title of the toolbar. `Intent` and `title` are automatically mapped to attributes by the following method:

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

`Another part of the implementation of` onCreate` is to call command. This is very similar to what we did before:

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

When the result is fetched from the database, the `bindForecast` function is called in the UI thread. We have used the Kotlin Android Extensions plugin again in this activity to do not use `findViewById` to get the property from XML:

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

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

Here are some interesting places. For example, I created another extension function to convert a `Long` object to a date string for display. Remember that we are also used in the adapter, so clearly defined it as a function is a good practice:

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

I'll get a `date format` (or use the default DateFormat.MEDIUM) and converted into a `Long` user can understand `String` .

Another interesting place is the `bindWeather` function. It receives a `Int`` pairs` and `TextView`` vararg` a composition, and to set a different `TextView`` text` and `text color` accordance with the temperature.

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

Each pair, it will set a `text` to show the temperature and a different color according to the temperature match: low temperature with red, medium temperature with orange, the other with green. The temperature value is relatively random, but this is a good representation of using the `when` expression to make the code thinner.

`Color` is an extension function I miss Anko in, it can be very simple way to get a color from resources, similar to our use elsewhere `dimen` . When we write this line, the current `support library` relies on `ContextCompat` to get a color from a different Android version:

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

`AndroidManifest` also needs to know the existence of new activity:

```
<activity
    android:name=".ui.activities.DetailActivity"
    android:parentActivityName=".ui.activities.MainActivity" >
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value="com.antonioleiva.weatherapp.ui.activities.MainActivity" />
</activity>
```

Start an activity: reified function

The last step is to start a `detail activity` from `main activity`. We can rewrite the adapter instance as follows:

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

But it is very lengthy. As always, Anko provided a much simpler way to start an activity with 'reified function`'

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

`Reified function` behind what magic in the end it? As you might know, when we create a paradigm in Java, we have no way to get the type of class. A popular workaround is passed as a parameter to a Class. In Kotlin, an inline (`inline`) function can be embodied (`reified`), which means that we can get and use the type of class in the function. Anko really uses a simple example of it to speak next (in this case I only used `String`):

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

The real realization is more complicated because it uses a very long annoying `when` expression to increase the extra information determined by the type, but conceptually it does not add other more useful knowledge.

`Reified` function is a grammar that can simplify code and improve comprehension. In this example, it creates an intent, iterates all the parameters and adds to the intent, and then uses Intent to start the activity by getting the `javaClass` of the schema type. `Reified` is restricted to the subclass of activity.

The remaining details are already described in the code base. We now have a very simple (but complete) master-view (`master-detail`) App, which uses Kotlin implementation, does not use a line of Java code.

interface

Kotlin interface is much more powerful than Java 7. If you use Java 8, they are very similar. In Kotlin, we can use interfaces like Java. Imagine that we have some animals, some of them can fly. This is our interface to the flying animal:

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

Birds and bats can fly by flapping wings. So we create two classes for them:

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

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

When two classes inherit from an interface, it is very typical that both of them share the same implementation. However, the interface in Java 7 can only define behavior, but it can not be implemented.

The Kotlin interface in one aspect it can implement the function. The only difference between them and the class is that they are stateless, so attributes require subclasses to be overridden. Classes need to be responsible for saving the status of interface properties.

We can let the interface implement `fly` function:

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

As mentioned, classes need to rewrite attributes:

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

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

Now birds and bats can fly:

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

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

Commissioned

Delegate mode is a useful model that can be used to extract the main part of the class from the class. The delegate pattern is native to Kotlin, so it avoids the need to call delegates. The delegate only needs to specify an instance of the implemented interface.

In our previous example, we can use the constructor to specify how the animal is flying, rather than implementing it. For example, an animal flying with wings can be specified in this way:

```
interface CanFly {
    fun fly()
}

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

We can use the interface to indicate that the bird can fly, but the bird's flight mode is defined in a delegate, which is defined in the constructor, so we can use different modes of flight for different birds. Animals use wings to fly in the other class:

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

Animals flap wings to fly. So we can create a bird that uses wings to fly:

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

But now the wings can be used by other animals that are not birds. If we assume that bats use wings, we can specify the delegate directly to instantiate the object:

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

Implement an example in our App

An interface can be used to extract generic code of similar behavior from a class. For example, we can create an interface for handling the app's toolbar. `MainActivity` and `DetailActivity` in processing these `toolbar` share similar code.

But limited, we need to make some changes, use is defined in the layout of the `toolbar`, not the standard `ActionBar`. The first thing is to inherit the theme of `NoActionBar`. Such `toolbar` not automatically include it:

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

We use the `light` theme. And then we create a `toolbar` layout, we will later use it in other layouts:

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

`Toolbar` specify its own background, a (`overflow menu` instance) for their `dark` theme and a theme for `light` generate pop-up box. We now have the same theme: `light` theme and `dark` theme `Action Bar`

The next step we will modify the layout of `MainActivity`, add a toolbar:

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

    <android.support.v7.widget.RecyclerView
        android:id="@+id/forecastList"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:clipToPadding="false"
        android:paddingTop="?attr/actionBarSize"/>

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

Now that the toolbar is added to the layout, we can start using it. We created an interface that allows us to:

- change title
- Specifies whether to display the previous navigation action
- scroll when the toolbar animation
- set the same menu for all the activities, and even behavior

Then let's define `ToolbarManager`:

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

It will need a toolbar property. The interface is stateless, so the attribute can be defined, but can not be assigned. Subclasses will implement this interface and override this property.

On the other hand, we can not use rewrite to achieve stateless properties. That is, attributes do not need to maintain a `backup field`. An example of handling the toolbar title attribute:

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

Because the property just uses the toolbar, it does not need to save any new state.

We now create a new function to initialize the toolbar, inflate a menu and set a listener:

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

We can add a function to open the toolbar above the navigation icon, set an arrow icon and set an icon when the trigger is triggered by the event:

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

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

This function takes a listener, uses DrawerArrowDrawable to create a last hop (when the arrow has been shown), and then sets the listener to the toolbar.

Finally, the interface will provide a function that allows the toolbar to be attached to a scroll above, and the animation is executed according to the direction of the scroll. When the scroll down the toolbar will disappear, scroll up the toolbar will show:

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

We will create two extension functions for viewing the view from the screen or disappearing animation. We will check whether the animation has not been implemented before. This way you can avoid each time a different scroll view will be animated:

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

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

After `toobar manager` implementation, it is time to use it in `MainActivity` '. We first specify the toolbar attribute. We can use `lazy` commission to achieve, this will be the first time we use it will be inflate:

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

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

```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main) initToolbar()
    forecastList.layoutManager = LinearLayoutManager(this)
    attachToScroll(forecastList)
    async {
        val result = RequestForecastCommand(94043).execute()
        uiThread {
            val adapter = ForecastListAdapter(result) {
            startActivity<DetailActivity>(DetailActivity.ID to it.id,
                        DetailActivity.CITY_NAME to result.city)
            }
            forecastList.adapter = adapter
            toolbarTitle = "${result.city} (${result.country})"
        }
    }
}
```

`DetailActivity` also needs some changes on the layout:

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

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

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:gravity="center_vertical"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        tools:ignore="UseCompoundDrawables">
       ....
    </LinearLayout>

</LinearLayout>
```

Use the same way to specify the toolbar attribute. `DetailActivity` will also initialize the toolbar, set the title and turn on the navigation Return icon:

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

    initToolbar()
    toolbarTitle = intent.getStringExtra(CITY_NAME)
    enableHomeAsUp { onBackPressed() }
    ...
}
```

The interface can help us to extract common code from the class to share similar behavior. Can be used as a refinement alternative to our code refinement. Think about which interface can help you write better code.

Generic

Generic programming involves writing algorithms without specifying the exact type used in the code. In this way, we can create functions or types, the only difference is that they use different types, improve the reusability of the code. This code unit is what we know the generics, they exist in many languages, including Java and Kotlin.

In Kotlin, generics are even more important because frequent use of the extended function will multiply the frequency of our generic use. Although we have used generics blindly in this book, generics are usually more difficult in any language, so I try to use it as simple as possible to explain it, so that the main idea will be enough Clear.

basis

For example, we can create a specified generic class:

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

This class can now be initialized using any type, and the parameter will also use the defined type, we can do this:

```
val t1 = TypedClass<String>("Hello World!")
val t2 = TypedClass<Int>(25)
```

But Kotlin is simple and downsets the template code, so if the compiler can infer the type of the argument, we do not even need to specify it:

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

If the third object receives a null reference, it still needs to specify its type because it can not be inferred.

We can add the type restriction in the way that is specified in Java as defined in Java. For example, if we want to restrict the non-null type in the previous class, we only need to do this:

```
class TypedClass<T : Any>(parameter: T) {
    val value: T = parameter
}
```

If you go to compile the previous code, you will see that `t3` will now throw an error. The nullable type is no longer allowed. But the restrictions can be more severe. What if we only want the subclasses of `Context` to do? Very simple:

```
class TypedClass<T : Context>(parameter: T) {
    val value: T = parameter
}
```

Now all classes that inherit `Context` can be used in our class. Other types are not allowed.

Of course, you can use the function. We can build generic functions fairly simply:

```
fun <T> typedFunction(item: T): List<T> {
    ...
}
```

Variants

This is really one of the most difficult to understand parts. In Java, when we use the generic when there will be problems. The logic tells us that `List<String>` should be able to transition to `List<Object>` because it has weaker restrictions. But let's take a look at this example:

```
List<String> strList = new ArrayList<>();
List<Object> objList = strList;
objList.add(5);
String str = objList.get(0);
```

If the Java compiler allows us to do so, we can add an `Integer` to `Object` List, but it will obviously crash at some point. This is why the language adds wildcards. Wildcards can increase flexibility in limiting this problem.

If we add `? Extends Object` , we use covariance, which means that we can handle any object that uses a type that is more restrictive than Object, but we are safe if we use `get` The If we want to copy a `Strings` collection into the `Objects` collection, we should be allowed, right? Then, if we do：

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

This is possible because the `addAll()` defined in the `Collection` interface is like this:

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

Otherwise, there is no wildcard, we will not allow the use of `String` List in this method. On the contrary, of course it will fail. We can not use `addAll ()` to add a `Objects` List to `Strings` List. Because we just use that method to get elements from `collection` , which is an example of a perfect covariant (`covariance`).

On the other hand, we can find the inverter (`contravariance`) on the opposite side. According to the example of the collection, if we want to pass the parameters to the collection, we can add more restrictive types to the generic set. For example, we can add `Strings` to `Object` List:

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

The only limitation that adds `Strings` to another collection is that the collection receives `Strings` or parent.

But the wildcard has its limitations. Wildcards define the use of scene variants (`use-site variance`), which means that we need to declare it when we use it. This means that each time we declare a generic variable will increase the template code.

Let's look at an example. Use our previous similar classes:

```
class TypedClass<T> {
    public T doSomething(){
        ...
    }
}
```

The code will not be compiled:

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

Although it does not make sense because we still keep all the methods in the class and do not have any damage. We need to specify a type that can have a more flexible definition.

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

This makes the code more difficult to understand, and adds some extra template code.

On the other hand, Kotlin can be handled in a much easier way by using the declaration-site variance. This means that when we define a class or interface, we can handle weakly restricted scenes, and we can use it directly elsewhere.

So let's see how it works in Kotlin. Compared to lengthy wildcards, Kotlin uses `out` only for covariance (`covariance`) and uses `in` for the invert (`contravariance`)). In this example, when the objects generated by our class can be saved to weakly constrained variables, we use covariates. We can define the declaration directly in the class {

```
class TypedClass<out T>() {
    fun doSomething(): T {
        ...
    }
}
```

That's all we need. Now, in Java can not compile the code in Kotlin can run perfectly:

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

If you have used these concepts, I'm sure you can simply use `in` and `out` in Kotlin. Otherwise, you only need some contact and conceptual understanding.

Generic example

After the theory, we moved to some of the actual functions above, which would make it easier for us to master it. In order not to repeat the invention of the wheel, I used three functions in the three Kotlin standard libraries. These functions let us only use the generic implementation can do some great things. It can inspire you to create your own function.

let

`Let` is really a simple function that can be called by any object. It takes a function (receives an object, returns the result of the function) as a parameter, and returns the result as a function of the return value of the entire function. It is very useful in dealing with null objects, the following is its definition:

```
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 caller, and its type is received by the function. The second is the return type of the function.

How do we use it? You may remember that when we get data from the data source, the result may be null. If it is not null, the result is mapped to `domain model` and returns the result, otherwise it returns null:

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

This code is very ugly, and we do not need to use this way to handle nullable objects. In fact, if we use `let` , do not need `if` :

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

Thanks to the `?.` operator, the `let` function will only be executed if `forecast` is not null. Otherwise it will return null. That is, we want to achieve the effect.

with

In this book we have a lot of this function. `With` receives an object and a function that functions as an extension of the object. This means that we can use `this` in the function according to the inference.

```
inline fun <T, R> with(receiver: T, f: T.() -> R): R = receiver.f()
```

Generics are also run in the same way here: `T` represents the receive type, `R` represents the result. As you can see, the function is defined as an extension function with the `f: T.() -> R` declaration. That's why we can call `receiver.f()` .

Through this app, we have a few examples:

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

apply

It looks like `with` very similar, but it is a bit different. `Apply` can be used to avoid creating a builder because the function called by the object can initialize itself according to its own needs, and then the `apply` function will return it to the same object:

```
inline fun <T> T.apply(f: T.() -> Unit): T { f(); return this }
```

Here we only need a generic type, because the object that calls this function is the object that this function returns. A nice example:

```
val textView = TextView(context).apply {
    text = "Hello"
    hint = "Hint"
    textColor = android.R.color.white
}
```

It creates a `TextView`, modifies some properties, and then assigns it to a variable. Everything is simple, readable and rugged. Let's use it in the current code. In `ToolbarManager`, we use this way to create a navigation drawable:

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

Using `with` and returning `this` is very clear, but using `apply` can be easier:

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

You can view these small optimizations in the `Kotlin for Android Developer` codebase.

Set the interface

Until now, we are using the default city to implement this app, but now it is time to add a function to select the city. Our App needs a setting bar to let the user modify the city.

We use `zip code` (zip code) to distinguish between cities. A real App may need more information, because only the zip code in the whole world may not be used as a basis for identification. But we will at least show the world's cities defined in the settings using `zip code`. This would be an example of how to use the funny way to handle preferences.

Create a setup activity

When the `settings` menu option for the `overflow menu` is clicked, you need to open a new Activity. So the first thing to do when you need a new `SettingActivity` :

```
class SettingsActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_settings)
        setSupportActionBar(toolbar)
        supportActionBar.setDisplayHomeAsUpEnabled(true)
    }

    override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
        android.R.id.home -> { onBackPressed(); true }
        else -> false
    }
}
```

When the user leaves the interface, we need to save the user 'preferences', so we need to handle the `Up` action like `Back` to redirect the action to `onBackPressed` . Now let's create an XML layout. For this `preference` a simple` EditText 'is enough:

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

    <include layout="@layout/toolbar"/>
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="?attr/actionBarSize"
        android:padding="@dimen/spacing_xlarge">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/city_zipcode"/>

        <EditText
            android:id="@+id/cityCode"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="@string/city_zipcode"
            android:inputType="number"/>

    </LinearLayout>
</FrameLayout>
```

And then only need to declare this activity in `AndroidManifest.xml` '

```
<activity
android:name=".ui.activities.SettingsActivity"
android:label="@string/settings"/>
```

Access Shared Preferences

You may know what is Android Shared Preferences. A simple set of key and value pairs that can be easily stored through the Android framework. These `preferences` are part of the SDK and make the task easier. And from Android 6.0 (Marshmallow), `shared preferences` can be automatically stored by the cloud, so when a user in a new device to restore the App, their `preferences` will be restored.

Thanks to the use of attribute delegation, we can use very simple way to deal with `preferences`. We can create a delegate, when `get` is called to query, when `set` is called to perform the save operation.

Because we want to save `zip code`, it is a long type, so let's create a Long attribute of the commission it. In `DelegatesExtensions.kt`, implement a new `LongPreference` class:

```
class LongPreference(val context: Context, val name: String, val default: Long)
    :  ReadWriteProperty<Any?, Long> {

    val prefs by lazy {
        context.getSharedPreferences("default", Context.MODE_PRIVATE)
    }

    override fun getValue(thisRef: Any?, property: KProperty<*>): Long {
        return prefs.getLong(name, default)
    }

    override fun setValue(thisRef: Any?, property: KProperty<*>, value: Long) {
        prefs.edit().putLong(name, value).apply()
    }
}
```

First, we use the `lazy` delegate to create a preferences. In this case, if we do not use this attribute, the delegate will not request the `SharedPreferences` object.

When `get` is called, its implementation is to use the preferences instance to get a long attribute for the specified name in the delegate declaration. If this attribute is not found, default is used. When a value is `set`, get the `preferences editor` and save it with the attribute name.

We can define a new delegate in `DelegatesExt`, so that when we visit a lot easier:

```
object DelegatesExt {
    ....
    fun longPreference(context: Context, name: String, default: Long) =
        LongPreference(context, name, default)
}
```

In `SettingActivity`, you can now define a property to handle `zip code` preferences. I created two constants for default values for names and attributes. This way can be used elsewhere in App:

```
companion object {
    val ZIP_CODE = "zipCode"
    val DEFAULT_ZIP = 94043L
}

var zipCode: Long by DelegatesExt.longPreference(this, ZIP_CODE, DEFAULT_ZIP)
```

It is very simple to work now, we can get from the attribute and assigned to the `EditText`:

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    cityCode.setText(zipCode.toString())
}
```

We can not use the auto-generated property `text` because `EditText` returns `Editable` in `getText`, so the attribute defaults to that value. If I try to allocate a `String`, the compiler will error, using `setText()` is enough.

Now have all the things to achieve `onBackPressed`. Here, the new value of a property is stored:

```
override fun onBackPressed() {
    super.onBackPressed()
    zipCode = cityCode.text.toString().toLong()
}
```

`MainActivity` needs some minor changes. First, it also requires a `zip code` attribute.

```
val zipCode: Long by DelegatesExt.longPreference(this, SettingsActivity.ZIP_CODE,
            SettingsActivity.DEFAULT_ZIP)
```

Then I moved the `forecast load` code to `onResume`, so that every time the activities resumed, it refreshes the data to prevent `code zip` from being modified. Of course, there are more complex ways to do this, such as by asking the forecast before the test whether the `zip code` really changed. But I like to keep the simplicity of this example, and because the requested data has been stored in the local database, so this solution is not too bad:

```
override fun onResume() {
    super.onResume()
    loadForecast()
}

private fun loadForecast() = async {
    val result = RequestForecastCommand(zipCode).execute()
    uiThread {
        val adapter = ForecastListAdapter(result) {
            startActivity<DetailActivity>(DetailActivity.ID to it.id,
                    DetailActivity.CITY_NAME to result.city)
        }
        forecastList.adapter = adapter
        toolbarTitle = "${result.city} (${result.country})"
    }
}
```

`RequestForecastCommand` now uses `zipCode` instead of the previous one is a fixed value.

There is also the opinion that we have to do things: when the overflow menu of the `settings` click to start the `setting activity`. The `initToolbar` function in `ToolbarManager` needs to have some minor changes:

```
when (it.itemId) {
    R.id.action_settings -> toolbar.ctx.startActivity<SettingsActivity>()
    else -> App.instance.toast("Unknown option")
}
```

Generic preference

Now that we are already a generic expert, why not extend `LongPreference` to support all `Shared Preferences` support types? We come to create a `Preference` delegate:

```
class Preference<T>(val context: Context, val name: String, val default: T)
    : ReadWriteProperty<Any?, T> {

    val prefs by lazy {
        context.getSharedPreferences("default", Context.MODE_PRIVATE)
    }

    override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return findPreference(name, default)
    }

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

This preference is very similar to what we used before. We only replaced the `long` for the generic type `T` , and then called the two functions to do the specific important work. These functions are very simple, although some repetitions. They will check the type and then use the specified way to operate. For example, the `findPrefernce` function is as follows:

```
private fun <T> findPreference(name: String, default: T): T = with(prefs) {
        val res: Any = when (default) {
        is Long -> getLong(name, default)
        is String -> getString(name, default)
        is Int -> getInt(name, default)
        is Boolean -> getBoolean(name, default)
        is Float -> getFloat(name, default)
        else -> throw IllegalArgumentException(
            "This type can be saved into Preferences")
    }

    res as T
}
```

`PutPreference` function is the same, but in `when` Finally through `apply` , use the `preferences editor` to save the results:

```
private fun <U> putPreference(name: String, value: U) = with(prefs.edit()) {
    when (value) {
        is Long -> putLong(name, value)
        is String -> putString(name, value)
        is Int -> putInt(name, value)
        is Boolean -> putBoolean(name, value)
        is Float -> putFloat(name, value)
        else -> throw IllegalArgumentException("This type can be saved into Pref\
erences")
    }.apply()
}
```

Now modify the `DelegateExt` :

```
object DelegatesExt {
    ...
    fun preference<T : Any>(context: Context, name: String, default: T)
        = Preference(context, name, default)
}
```

After this chapter, the user can access the setup interface and modify the `zip code` . Then when we return to the main interface, the forecast will automatically re-refresh and display the new information. View the remaining xi wei in the code base

Test your app

We are about to reach the end of the trip. Through this book you have learned most of the knowledge of Kotlin, but you may wonder if you can test your only use Kotlin prepared Android App? The answer is: of course!

In Android we have two different tests: `unit test` and `instrumentation test`. It is clear that this book will not teach you how to test, there are many specifically written for this book. My goal in this chapter is how to build your test environment, show you to see Kotlin in the test can also work very well.

Unit testing

I will not discuss the topic of `unit testing` (unit test). There are many definitions, but there are some subtle differences. A common view may be `unit testing` to verify the test of a unit (`unit`) source code. A unit (`unit`) contains nothing left to the reader. In our example, I just defined a `unit test` as a test that does not require a device to run. The IDE will run these tests and then show the final result to resolve which tests were successful and which tests failed.

`Unit testing` usually uses the `JUnit` library. So let's add this dependency to `build.gradle` . Because this dependency will only be used when running the test, so we can use `testCompile` instead of `compile` . In this way, the library will be ignored in the official compiler, you can reduce the size of APK:

```
dependencies {
    ...
    testCompile 'junit:junit:4.12'
}
```

Now synchronize gradle to get the library and add it to your project. To open `unit testing` , open the `Build Variants` tab (you may find it on the left side of the IDE) and click `Test Artifact` . You should select `Unit Tests` .

Another thing you need to do is create a new folder. Under src, you probably already have `androidTest` and `main` 's. Create another folder called `test` , and then create a `java` folder below it. So now you should have a folder named `src / test / java` green. This is the IDE that we are using the `Unit Test` model good signs that this folder will include some test files.

Let's write a very simple test to see if everything is running properly. Use the appropriate package name (my `com.antonioleiva.weatherapp` , but you need to use the main package name in your app) to create a new Kotlin class called `SimpleTest` . When you create it, write the following simple tests:

```
import org.junit.Test
import kotlin.test.assertTrue
class SimpleTest {
    @Test fun unitTestingWorks() {
        assertTrue(true)
    }
}
```

Use the `@ Test` annotation to identify the function as a test. Confirm that it is `org.unit.Test` . Then add a simple assertion. It just judges whether true is true, it will obviously succeed. This test is only used to confirm that all configurations are correct.

Perform a test, just right-click on a folder in the new `java` file you created in the `test` , then select `Run All Tests` . When the compilation is complete, it runs the test and sees the display of the result profile. You should be able to see our test passed.

Now it's time to create a real test. All tests that are handled using the Android framework may require a `instrumentation test` or a more complex library like Robolectric. So in these examples I will not use the framework of anything. For example, I will test the extension function from `Long` to `String` .

Create a new file named `ExtensionTests` and add the following tests:

```
class ExtensionsTest {

    @Test fun testLongToDateString() {
        assertEquals("Oct 19, 2015", 1445275635000L.toDateString())
    }

    @Test fun testDateStringFullFormat() {
        assertEquals("Monday, October 19, 2015",
            1445275635000L.toDateString(DateFormat.FULL))
    }
}
```

These tests detect whether the `Long` instance can be converted to a `String`. The first test default behavior (using DateFormat.MEDIUM)), and the second one specifies a different format. Run these tests and then you will see them all through it. I suggest you modify them and see how they fail.

If you have used a test in Java, you will find that this is not much different. I will demonstrate a simple example where we can make some tests on `ForecastProvider`. We can use the `Mockito` library to simulate other classes and then test the provider independently:

```
dependencies {
    ...
    testCompile "junit:junit:4.12"
    testCompile "org.mockito:mockito-core:1.10.19"
}
```

Now create a `ForecastProviderTest`. We're going to test `ForecastProvider` and use `DataSource` to return the result to see if it is null. So first we need to simulate a `ForecastDataSource`:

```
val ds = mock(ForecastDataSource::class.java)
`when`(ds.requestDayForecast(0)).then {
    Forecast(0, 0, "desc", 20, 0, "url")
}
```

As you can see, we need to put a counter-quotation on `when`. Because `when` is a reserved keyword in Kotlin, so if we use it in some Java code we need to avoid it. Now we use this data source to create a provider, and then detect whether the result of calling the method is null:

```
val provider = ForecastProvider(listOf(ds))
assertNotNull(provider.requestForecast(0))
```

This is the complete test function:

```
@Test fun testDataSourceReturnsValue() {
    val ds = mock(ForecastDataSource::class.java)
    `when`(ds.requestDayForecast(0)).then {
           Forecast(0, 0, "desc", 20, 0, "url")
    }

    val provider = ForecastProvider(listOf(ds))
    assertNotNull(provider.requestForecast(0))
}
```

If you run it, you will see that it will go wrong. Thanks to this test, we found some errors in our code. The test failed because the `ForecastProvider` was initialized in its `companion object` before it was used. We can add some data source to `ForecastProvider` by constructor, and this static List will never be used, so it should be loaded with `lazy`:

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

If you are going to run again now, you will find that all tests will now pass. We can also test some, for example, when the data source returns null, it will facilitate the next data source to get the results:

```
@Test fun emptyDatabaseReturnsServerValue() {
    val db = mock(ForecastDataSource::class.java)
    val server = mock(ForecastDataSource::class.java)
    `when`(server.requestForecastByZipCode(
            any(Long::class.java), any(Long::class.java)))
            .then {
                ForecastList(0, "city", "country", listOf())
    val provider = ForecastProvider(listOf(db, server))
    assertNotNull(provider.requestByZipCode(0, 0))
}
```

As you can see, by using the default value of the parameter, this simple dependency is enough to allow us to implement some simple `unit tests`. There are a lot of things we can test for this provider, but this example is enough for us to learn to use the `unit testing` tool.

Instrumentation tests

`Instrumentation tests` is a little different. They are usually used in UI interaction, and we need an application instance to run while performing the tests. To achieve this, we need to deploy and run on the device.

This type of test must be placed in the `androidTest` folder, and we have to modify the `Test Variants S Test Artifact` for `Android Instrumentation Tests` . The official realized instrumentation library is Espresso, which by `Actions` , `filter` and `ViewMatchers` and `Matchers` test results can help us more easily use.

Configuration is more difficult than before. We need to download additional libraries and `Gradle` 's configuration. The good thing is that Kotlin's test does not need to add extra stuff, so if you already know how to configure `Espresso` , it will be very simple for you.

First, specify `test runner` in `defaultConfig` '

```
defaultConfig {
    ...
    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
}
```

When you have finished dealing with the runner, then add the other dependencies, this time with `androidTestCompile` . In this way, these libraries will only be compiled and run when the `instrumentation tests` is added:

```
dependencies {
    ...
    androidTestCompile "com.android.support:support-annotations:$support_version"
    androidTestCompile "com.android.support.test:runner:0.4.1"
    androidTestCompile "com.android.support.test:rules:0.4.1"
    androidTestCompile "com.android.support.test.espresso:espresso-core:2.2.1"
    androidTestCompile ("com.android.support.test.espresso:espresso-contrib:2.2.1"){
        exclude group: 'com.android.support', module: 'appcompat'
        exclude group: 'com.android.support', module: 'support-v4'
        exclude module: 'recyclerview-v7'
}
```

I do not want to spend a lot of time to talk about these, but here is why the need for a brief reason for these libraries:

- `support-annotations` : need to be used in other libraries.
- `runner` : This is `test runner` , that is what we specified in `defaultConfig` 's.
- `rules` : Include some rules that test inflate to start activity. We will use these rules in our example.
- `espresso-core` : `Espresso` 's basic implementation, it makes `instrument tests` easier.
- `espresso-contrib` : it adds other extra features, such as support for the `RecyclerView` test. We have to exclude some of its dependence, because we have been used in this project, otherwise the test will go wrong.

Let's now create a simple example. The test will click on the first row of the forecast list and it will determine if a view with the id is named `R.id.weatherDescription` . This view is in `DetailActivity` , which means that we can successfully navigate to the details page after testing in the RecyclerView.

```
class SimpleInstrumentationTest {

    @get:Rule
    val activityRule = ActivityTestRule(MainActivity::class.java)

    ...
}
```

First we need to specify it to run using `AndroidJUnit4` . Then, create an activity rule that instantiates the activity that a test requires. In Java, you can use `@ Rule` . But as you know, the fields and attributes are not the same, so if you use it like that, the execution will fail because the fields in the access property are not public. You need to add annotations to getter above. Kotlin allows you to specify `get`

or `set` in front of the name of the rule. In this example, just `@get: Rule`.

After that we are ready to create the first test:

```
@Test fun itemClick_navigatesToDetail() {
    onView(withId(R.id.forecastList)).perform(
            RecyclerViewActions
                .actionOnItemAtPosition<RecyclerView.ViewHolder>(0, click()))
    onView(withId(R.id.weatherDescription))
           .check(matches(isAssignableFrom(TextView::class.java)))
}
```

Function with the `@ Test` annotation, this root we use the `unit test` the same way. We can start using `Espresso` in the test body. It first executed a click in the first position of `RecyclerView`. Then it detects whether you can find a view of the specified id and the view is a `TextView`.

To run this test, click on the top of the `Runencies drop-down select` Edit Configurations ... press the `+` icon, select `Android Tests`, then select the `app` module. Now, in the `target device` choose your favorite `target`. Click `OK` and run. You should be able to see how App is starting in your device, it will test the first position, open the details page and then close the app again.

Now we have to do something more complicated. Test will open from the banner of a overflow menu, click the `settings` column, change the city's `code`, and then check the toolbar of the title is changed to the corresponding title.

```
@Test fun modifyZipCode_changesToolbarTitle() {
    openActionBarOverflowOrOptionsMenu(activityRule.activity)
    onView(withText(R.string.settings)).perform(click())
    onView(withId(R.id.cityCode)).perform(replaceText("28830"))
    pressBack()
    onView(isAssignableFrom(Toolbar::class.java))
            .check(matches(
                withToolbarTitle(`is`("San Fernando de Henares (ES)"))))
}
```

This test actually does things:

- it first uses `openActionBarOverflowOrOptionsMenu` to open the overflow menu.
- then it looks for a view based on the `Settings` text and then clicks on it.
- After that, the setup interface will be opened, so it will look for a `EditText` and replace it with a new `code`.
- it will click the Back button. It will save the new value in the preferences, and then close the Activity.
- because the `mainActivity`, `onResume` will be called, the request will be called again. At this point it will get the forecast for the new city.
- the last line will detect the Toolbar we see whether the title is the title of the new city.

This is not the default matcher for a toolbar title, but `Espresso` is very easy to expand, so we can create a new matcher to implement the detection:

```
private fun withToolbarTitle(textMatcher: Matcher<CharSequence>): Matcher<Any> =
        object : BoundedMatcher<Any, Toolbar>(Toolbar::class.java) {

    override fun matchesSafely(toolbar: Toolbar): Boolean {
        return textMatcher.matches(toolbar.title)
    }

    override fun describeTo(description: Description) {
        description.appendText("with toolbar title: ")
        textMatcher.describeTo(description)
    }
}
```

`MatchSafely` function is where we detect, and `describeTo` function adds some new information to matcher.

This chapter is particularly interesting because we see how to integrate tests in Kotlin perfectly, and they can integrate the test without any problems. Look at the code and run your own.

Other concepts

Through this book, we explain most of the concepts in the Koltin language. But some of them in this App did not use, I will put them into this chapter. In this chapter, we review some irrelevant content so that you can use them in your own Kotlin project.

Internal class

In Java, we can redefine classes in the class. If it is a normal class, it can not access members of an external class (like static in Java):

```
class Outer {
  private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

If you need to access the members of the external class, we need to declare this class with `inner`:

```
class Outer {
    private val bar: Int = 1
    inner class Inner{
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

enumerate

Kotlin also provides an enumeration (`enums`) implementation:

```
enum class Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Enumerations can have parameters:

```
enum class Icon(val res: Int) {
    UP(R.drawable.ic_up),
    SEARCH(R.drawable.ic_search),
    CAST(R.drawable.ic_cast)
}

val searchIconRes = Icon.SEARCH.res
```

Enumeration can be obtained through `String` matching name, we can get that contains all enumerated `Array` , so we can traverse it.

```
val search: Icon = Icon.valueOf("SEARCH")
val iconList: Array<Icon> = Icon.values()
```

And each enumeration has some function to get its name, declared the location:

```
val searchName: String = Icon.SEARCH.name()
val searchPosition: Int = Icon.SEARCH.ordinal()
```

Enumerate the `Comparable` interface according to its order, so it's easy to sort them.

Sealed class

The sealed class is used to restrict the inheritance of classes, which means that the number of subclasses of the sealed class is fixed. It looks like an enumeration, and when you want to find a specified class in a subclass of a sealed class, you can know all the subclasses in advance. The difference is that the enumerated instance is unique, and the sealed class can have many instances, and they can have different states.

We can achieve, for example, similar to Scala in the `Option` class: this type can prevent the use of null, when the object contains a value to return `Some` class, when the object is empty,

```
sealed class Option<out T> {
    class Some<out T> : Option<T>()
    object None : Option<Nothing>()
}
```

There is a very good thing about the sealed class is that when we use the `when` expression, we can match all the options without using the `else` branch:

```
val result = when (option) {
    is Option.Some<*> -> "Contains a value"
    is Option.None -> "Empty"
}
```

Exceptions

In Kotlin, all `Exception`s are implemented with `throwable`, containing a `message` and unchecked. This means that we will not force us to use `try / catch` anywhere. This is not the same as in Java, such as throwing the `IOException` method, we need to use `try-catch` to enclose the code block. It is not a good idea to handle the display by checking the exception. Like Bruce Eckel, Rod Waldhoff or Anders Hejlsberg and others can give you a better view of this.

The exception is thrown in a similar way to Java:

```
throw MyException("Exception message")
```

`Try` expression is the same:

```
try{
    // Some code
}
catch (e: SomeException) {
    // deal with
}
finally {
    // Optional finally block
}
```

In Kotlin, `throw` and `try` are expressions, which means that they can be assigned to a variable. This is really useful when dealing with some border issues:

```
val s = when(x){
    is Int -> "Int instance"
    is String -> "String instance"
    else -> throw UnsupportedOperationException("Not valid type")
}
```

or

```
val s = try { x as String } catch(e: ClassCastException) { null }
```

END

Thank you for reading this book. Through this book, we learned through the example of an Android App to learn Kotlin. The weather forecast App is a good example, it implements most of the App need some of the basic features: a master / slave UI, through the API communication, database storage, shared preferences ... ...

The good place to use this way is that you use their use to learn most of the important concepts in Kotlin. I think the new language is more easily grasped in real practice. This is my main goal, the reference book is indeed a good tool to solve some of the standard problems, but we read from start to finish is very difficult. And as some examples are also out of a large context, it is difficult to understand what these characteristics can solve the problem.

And in fact the other goals of this book: show you the real problems you will encounter in Android, and how to use Kotlin to solve them. Some Android developers find a lot of problems when dealing with asynchronous, database, or handling very lengthy listener in Activity. Through the real App as an example, we encountered a lot of problems and learned the characteristics of new languages and libraries.

I hope that these goals have been met, and I really hope that you are not only learning Kotlin, but also in the book reading to enjoy. I was persuaded, Kotlin for Android developers is the best alternative to Java, we will see its progress in the next time. When it happens, you will be the first person to board the boat, and in your circle you will be in a perfect reference person's position.

The book is over, but it does not mean that it is dead. I will always keep the update (at least 1.0) based on the latest version, according to your comments and suggestions to check and optimize it. Have any idea to be able to contact me at any time, tell me your thoughts, the mistakes you find, the notions that are clear enough, or anything that you are worried about.

The process of writing this book for a few months has gone through an incredible trip. I have learned a lot, so thank you for your help to make the book `Kotlin for Android Developers` this book a reality.

best wishes to you,

Antonio Leiva

- website：antonioleiva
- mailbox：contact@antonioleiva.com
- Twitter：@lime_cl
- Google+：+AntonioLeivaGordillo