BY GREG LIM

BEGINNING ANDROID KOTLIN DEVELOPMENT

Beginning Android Development With Kotlin

Greg Lim

First Edition: March 2020
Second Edition: February 2022

Table of Contents

Preface

About this book

In this book, we take you on a fun, hands-on and pragmatic journey to learning Android application development using Kotlin. You'll start building your first Android app from scratch within minutes. Every section is written in a bite-sized manner and straight to the point as I don’t want to waste your time (and most certainly mine) on the content you don't need. In the end, you will have the skills to create an app and submit it to the app store.

In the course of this book, we will cover:

- Chapter 1: Introduction & BMI Calculator App
- Chapter 2: Quotes App Using TableView
- Chapter 3: To Do List App Using RecyclerView & Shared Preferences
- Chapter 4: To Do List with Realm
- Chapter 5: Connecting to an API: Cryptocurrency Price Tracker
- Chapter 6: Reading From GitHub API
- Chapter 7: Face Detection, Text Recognition with ML Kit
- Chapter 8: Publishing Our App on Google Play Store

The goal of this book is to teach you Android development in a manageable way without overwhelming you. We focus only on the essentials and cover the material in a hands-on practice manner for you to code along.

Requirements

No previous knowledge of Android development or Kotlin required, but you should have basic programming knowledge. We will learn how to make Android apps while at the same time learning the Kotlin programming language.

Getting Book Updates

To receive updated versions of the book, subscribe to our mailing list by sending a mail to support@i-ducate.com. I try to update my books to use the latest

version of software, libraries and will update the codes/content in this book. So do subscribe to my list to receive updated copies!

Contact and Code Examples

Contact me at support@i-ducate.com to obtain the source codes used in this book. Comments or questions concerning this book can also be directed to the same.

Chapter 1: Introduction & BMI Calculator App

Welcome to Beginning Android Development with Kotlin! I'm Greg and I'm so excited that you decided to come along for this. With this book, you will go from absolute beginner to having your app submitted to the Google Play Store and along the way, equip yourself with valuable Android app development skills.

Working Through This Book

This book is purposely broken down into chapters where the development process of each chapter will center it on different essential Android topics. The book takes a practical hands-on approach to learning through practice. You learn best when you code along with the examples in the book. Along the way, if you encounter any problems, do drop me a mail at support@i-ducate.com where I will try to answer your query.

Downloading Android Studio

Next, there is an essential piece of software you need to have on your computer before we can move forward. It's called Android Studio and is the official integrated development environment (IDE) provided by Google to write Kotlin and Java code to make Android apps. It includes the code editor, graphical user interface (GUI) editor, debugging tools, a device emulator (to test our apps without real devices) and much more. All examples are created within Android Studio. Let ’ s go ahead to get it downloaded before proceeding.

Download the latest stable version of Android Studio – Chipmunk | 2021.2.1. (at time of writing) from *https://developer.android.com/studio* (fig. 1.1).

Figure 1.1

The download file is quite big, about 1017 MB. The good thing is that Android Studio is available on all platforms. In this book, I will be using the Mac version, but the Windows and Mac versions are very similar. So don ’ t worry if you are using Windows to follow along.

After you have downloaded the installation file, run it. Installation should be pretty straightforward.
When running Android Studio for the first time, it will prompt you if you have Android Studio previously installed, you can import your settings. But I am assuming that you have not before. So, choose “ do not import settings ” .

You will be taken through the Android Studio Setup Wizard (fig. 1.2).

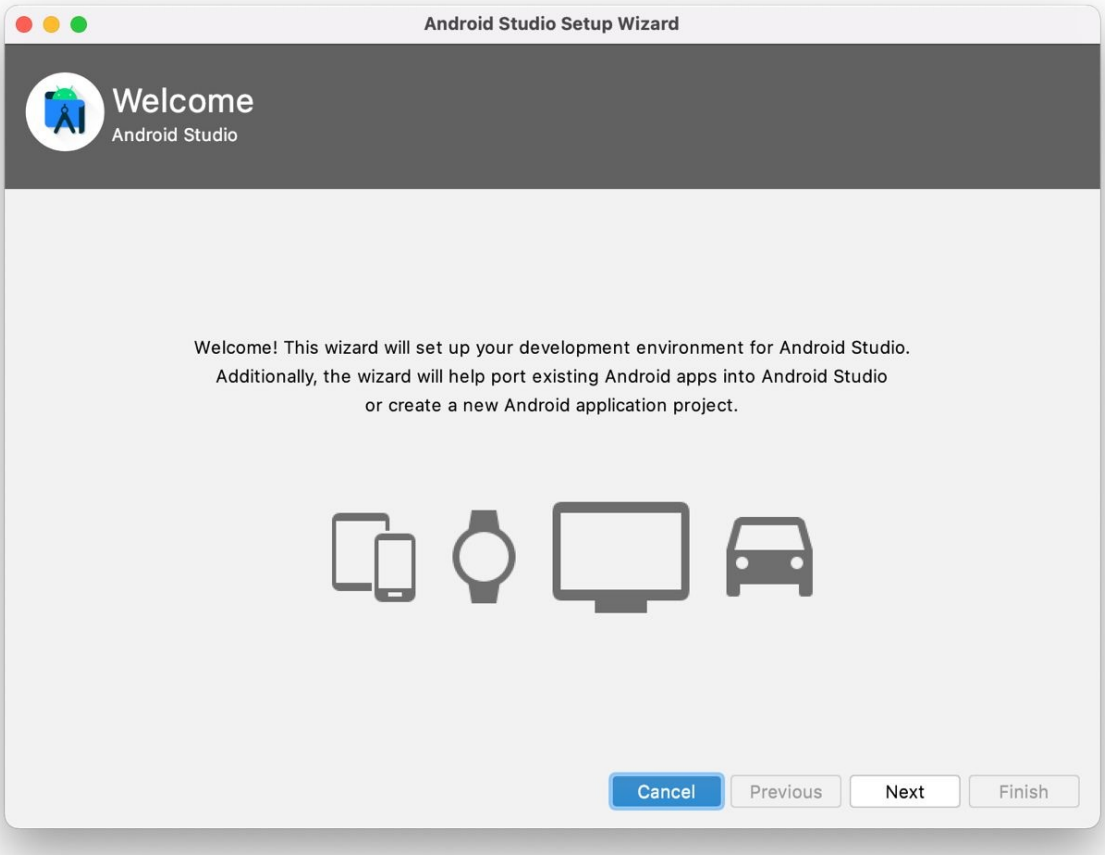


Figure 1.2

Click *Next*, then *Standard*, and select the UI Theme you prefer. I will be using *IntelliJ Light* in this book. You can go back and change to the *Darcula* (i.e. dark) version later if you want. Next, click *Finish*. Android Studio will then proceed to download components.

Once the installation is complete, you should see a welcome screen (fig. 1.3):

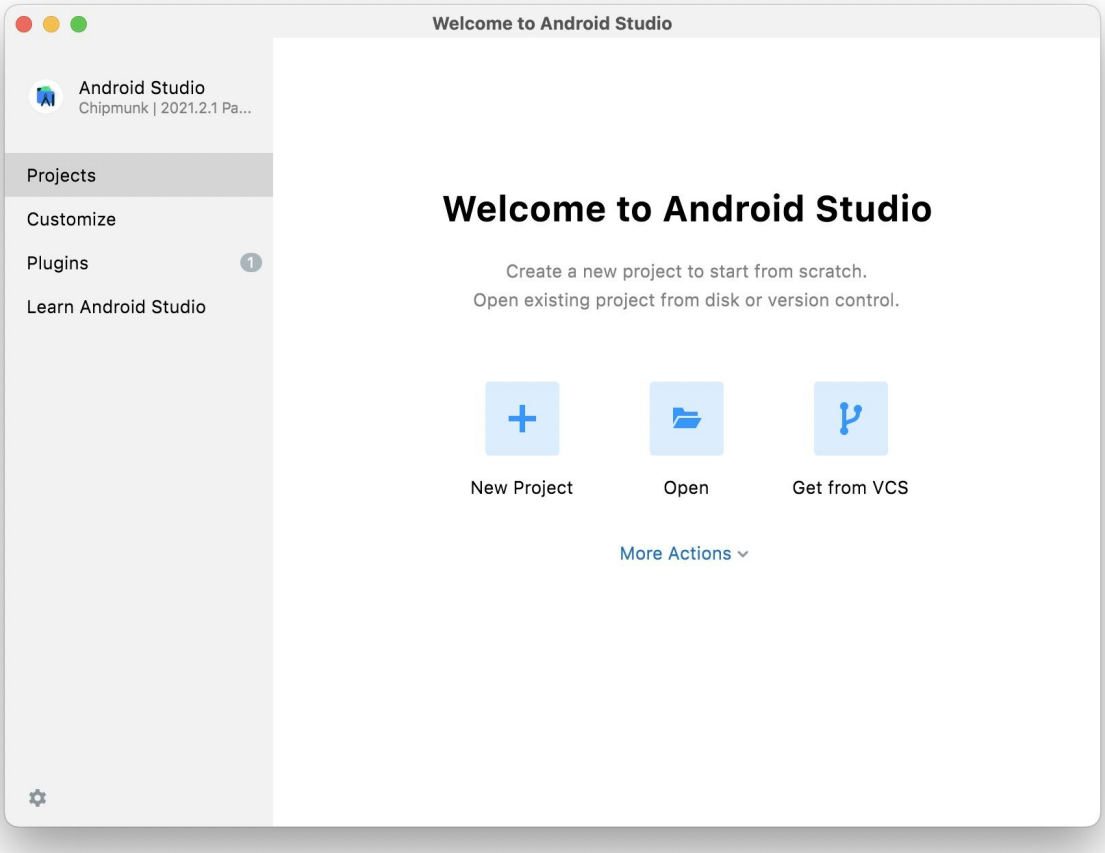


Figure 1.3

Each time you open up Android Studio, this is the welcome page where you can start a new project or open an existing one.

Introducing Kotlin and Android Studio

The two terms you're going to encounter throughout this book is Kotlin and Android Studio. First, why the name Kotlin? Kotlin is actually an island near St Petersburg, Russia. I am not too sure what is the significance of the island, but Kotlin is the new programming language we use to make Android apps. Prior to that, Java was the standard programming language used to build Android apps. But for beginners, Kotlin is easier and cleaner to learn. iOS developers will realize that it is very similar to Swift. So, iOS developers on one hand will find it easy to pick up Kotlin. On the other, learning Kotlin will open up doors in iPhone/iPad development.

Even experienced programmers think of Kotlin as a really clean and modern language. Because it is succinct, it is less prone to errors and makes development faster with less code. Google considers Kotlin an official Android language. Examples of apps coded in Kotlin are Kindle, Evernote, Twitter, Netflix and more.

Android Studio is the program that allows us to make Android apps. We're going to type Kotlin code into Android Studio and also use Android Studio for designing the visual side of our app like, where do we want a button, what color do we want it to be, where do we want to place our list, etc. Android Studio will provide everything we need to develop, setup and publish Android apps. So throughout this book, these are the two skills that we will be improving upon step by step.

In the background, Android Studio merges our Kotlin code with the Java from the Android Software Development Kit (SDK). It merges them in an intermediate form before being converted into the Dalvik Executable or DEX code. This conversion process is called compiling. The Android device then converts this DEX code to a running app in what is called an APK (Android application package). Whether one has programmed an app in Kotlin or Java, the resulting DEX code is the same. As developers, it is not necessary to remember the details of these, but it is good knowing what goes on behind the scenes.

Android Studio Walkthrough

In this section, we will be better acquainted with Android Studio, so go ahead and open it.

At the time of writing, this book uses Android Studio – Chipmunk | 2021.2.1. But make sure you're using the latest official version. Android Studio is constantly updating all the time and there are new versions of Kotlin or new SDKs (Android Studio itself prompts you to get these latest updates). In general, use the latest version of whatever there is. Now, when you get an update, it might break some part of your code which makes it a pain. But things should generally work fine with slight code changes here and there. And this is what a successful developer should learn to do so as well. So in general, just keep

everything updated.

In the ‘ Welcome to Android Studio ’ screen, go ahead and create a new project.

When you do so, it's going to bring up a page (fig. 1.5) that asks what kind of project do you want to make, whether Phone and Tablet, Wear OS, TV, Android Auto or Android Things.

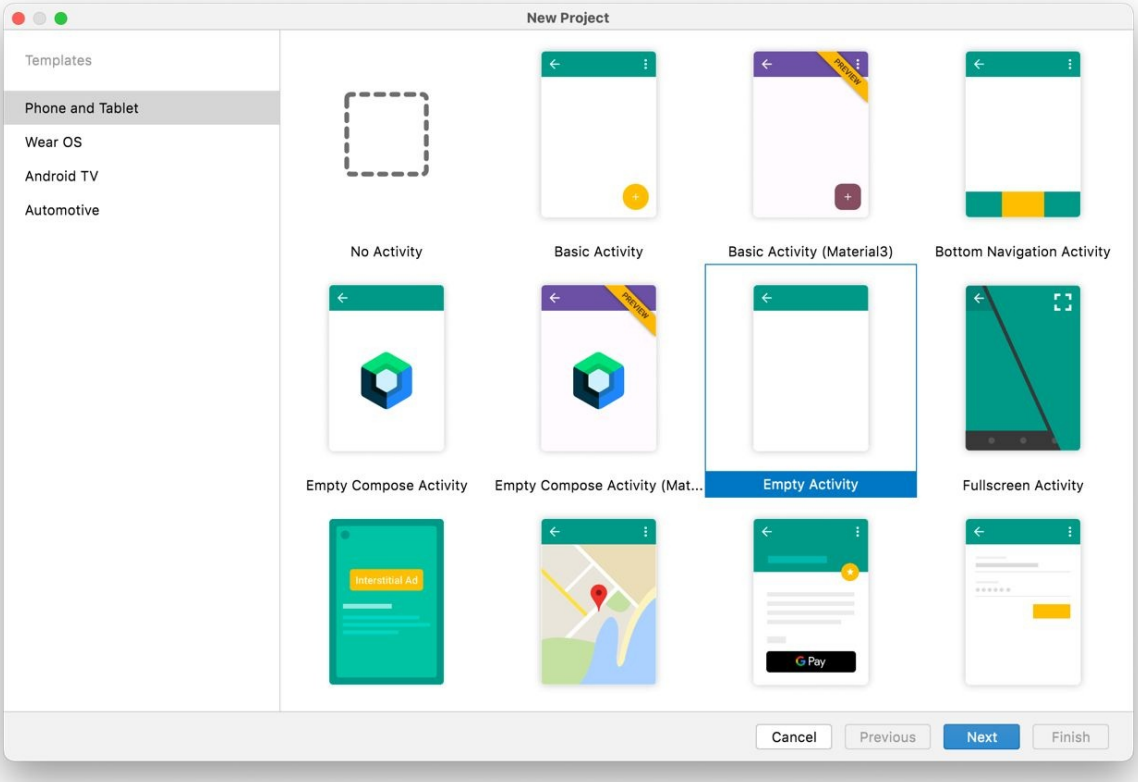


Figure 1.5

For us, we will be focusing on apps for the phone. Thus, select ‘ Phone and Tablet ’ .

For an Android app, there are lots of different templates that you can start with. The templates help you get started with some boilerplate code. Let ’ s select *Empty Activity*. This is essentially the blank starting point for almost every app that we're going to make. You can think of an Activity as a screen. So select that and click *Next*.

You will then have to input the below fields for your project (fig. 1.6):

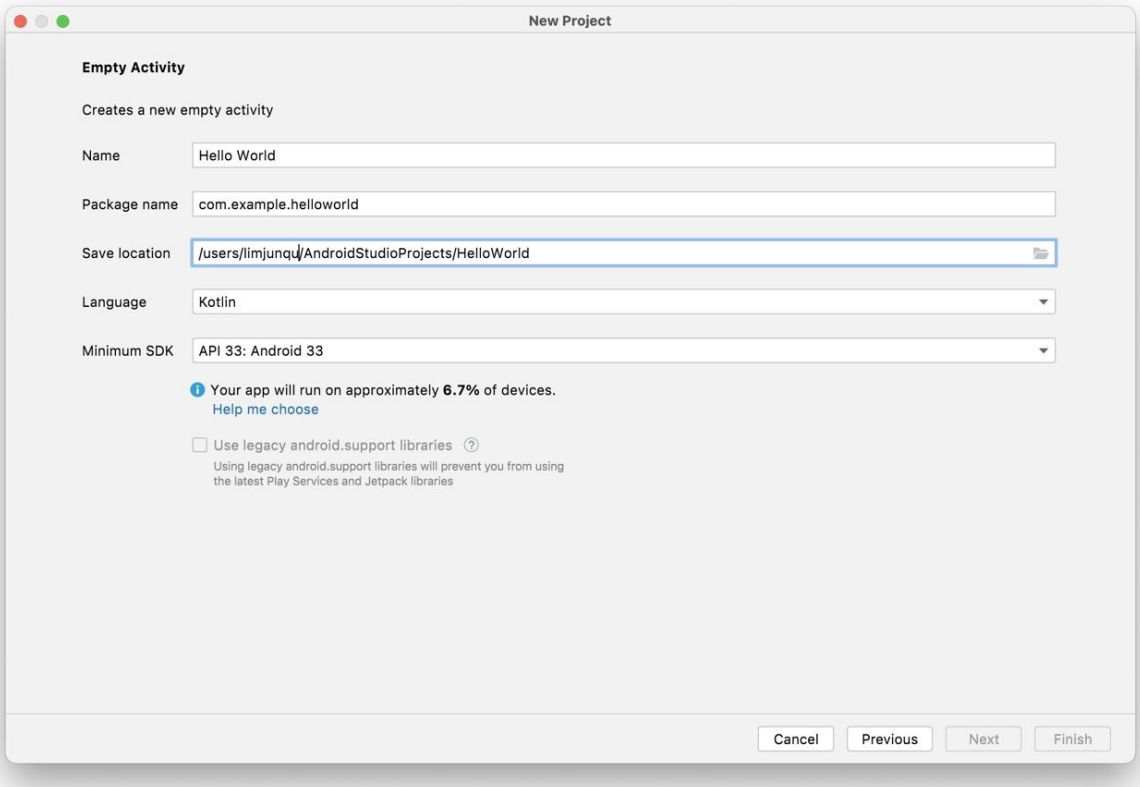


Figure 1.6

Name: as this is our first project, we will name it *Hello World*

Package name: usually the reverse of your website e.g. com.iducate.calculator. If you do not have a website, use the default provided package name will do. The purpose of the package name is to distinguish it from other projects, especially when we want to publish our app on the Play Store.

Save Location: You need to pick a place to save your project. For now, I recommend that you stick with what Android Studio suggests. They will make *AndroidStudioProjects* the folder where you contain all your Android projects in one place.

Language: select *Kotlin.* You can write Android apps in a few different languages like C++ and Java. But Kotlin is the newest and best official language

of Android.

Minimum API level: Select the latest API level. At time of writing this book, it is Android 33. What are all these different versions of Android? If you click on the ‘ Help me choose ’ link, it will bring up a useful chart (fig. 1.7) to help you decide which version of Android you should be choosing.

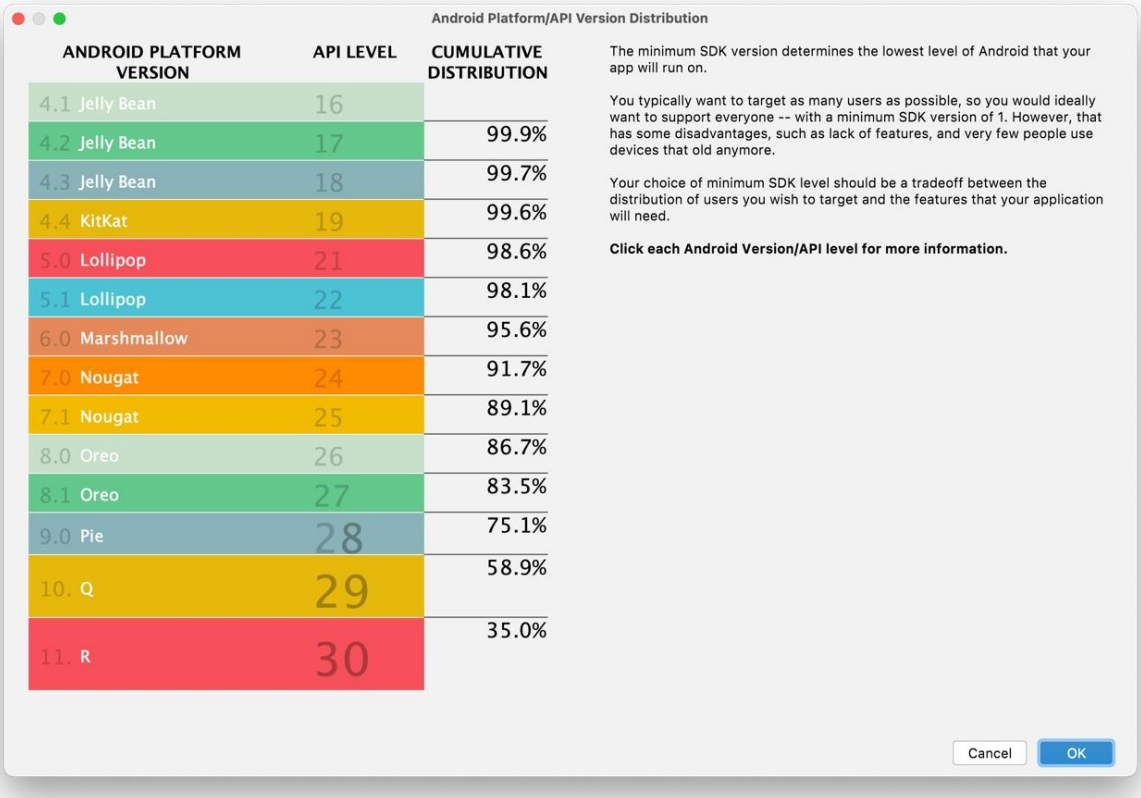


Figure 1.7

The chart essentially tells you that if you choose API 28 for example, it is only going to be able to run on 75.1% of Android devices. Whereas if you choose API 17, you will get to 99.9% of Android devices out there. The chart, unfortunately, does not display versions later than API 30 such as API 31, API 32 and API 33, but hopefully you get the point.

As the description beside the chart says, “ you typically want to target as many users as possible [...] with a minimum SDK version of 1. However, that has some disadvantages, such as lack of features and very few people use devices

that old anymore.

Your choice of minimum SDK level should be a tradeoff between the distribution of users you wish to target and the features that your application will need. ”

When we start making Android apps, we will talk more about how we can use the latest API but also go almost all the way back to every device. But for now, choose the API (33).

Go ahead and click ‘ Finish ’ after you have filled in the fields. You can change the field values later in your project, so don ’ t worry if you have inputted a wrong value.

After that, Android Studio might prompt you that it is downloading certain new components or updates. It will also do some syncing and building to set up for us a blank app for us to begin with. In the case that you get prompted with some error, Android Studio will usually prompt you to update something. Like any good SDK, Android Studio often has something for you to update. Each time the Android SDK gets a significant update, the version number is increased. Each new version comes with some new features. But go ahead to get whatever is needed to get the project working. It is going to take a little bit of time to get this all set up.

Once our project is up and running, on the left side of Android Studio, you can see the folder-file structure of the project (fig. 1.8). This is the *Project* window. It enables us to explore the folders, code and resources of the project.

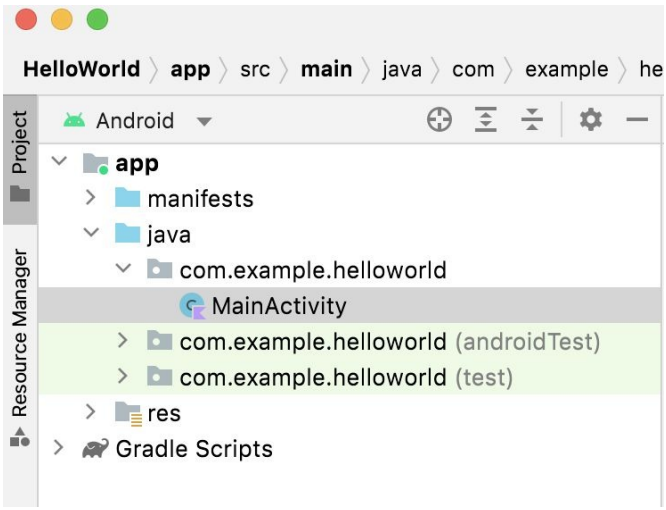


Figure 1.8

We have our *app* folder at the root and another four folders (*manifests*, *java*, *java(generated)*, *res*) inside of that. The *res* folder stores resources for our app such as images, sound and user interface layouts. As we move along this book, you will be more familiar with the purpose of each folder.

The structure of the files and folders here closely resembles the structure that will eventually end up in the finished APK file.

An important folder is the *java* folder. This folder holds the code to make things happen. In *java*, you have the folder with our package + project name (*com.example.helloworld*). In it, we have the file *MainActivity.kt*. ‘ .kt ’ is the extension to say that this file contains Kotlin code. Although it consists of just one file now, as our project grows, we will add more code files.

Note that since the release of Android Studio version 3.3, there is also a folder named *generatedjava*. But we don ’ t need to explore it. Also, you will see two other folders with the same name i.e. *com.example.helloworld(androidTest)*, *com.example.helloworld(test)*. The reason for these folders is due to automated testing, which is beyond the scope of this book. Thus, you can safely ignore these folders that end with *(androidTest)* and *(test)*. The only folder we are interested in is *com.example.helloworld*.

If you select *MainActivity.kt*, its source code (shown below) will be displayed in the code editor on the right:

```
package com.example.helloworld

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

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

This is essentially the code behind the main screen of our app (fig 1.9).

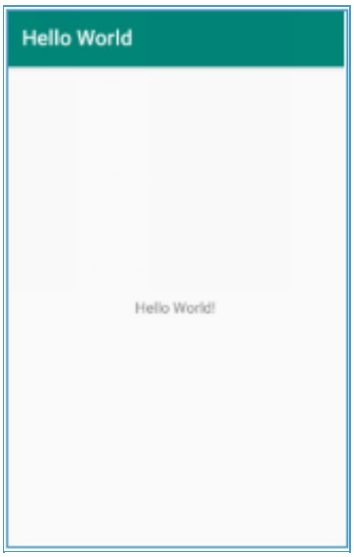


Figure 1.9

We will walk through the steps needed to run your app, but for now, understand that the code in *MainActivity.kt* is behind this activity. In Android apps, every screen we have is called an activity. For example, if we move from one screen to another, we are transiting from one activity to another. We will revisit *MainActivity.kt* later, but for now, let ' s see how to get this app up and running.

Creating an Android Virtual Device

We will explore the other folders later, but for now, we want to get our simple app up and running. We do so by clicking on the ' Play ' button 

located at the top. If you try to run the app now, you will get an error saying " No target device found " . If you have an Android device plugged via USB,

you can test it on the device. We will later show you how to run your app on a real Android device. But for now, we will illustrate running an emulator which is a simple way to test your app especially if you don ’ t have an Android device.

(Note that if you are using a slow computer, it will probably not handle the Android emulator well (i.e. your machine slows down and you constantly hear the ‘ whirring ’ sound of the computer fan). In this case, it will be better if you run your apps on an actual Android device.)

In the drop-down list, select ‘ Device Manager ’ (fig. 1.10a)

Select ‘ Virtual Device ’ . It will then prompt you to select a device definition to create (fig. 1.10b).

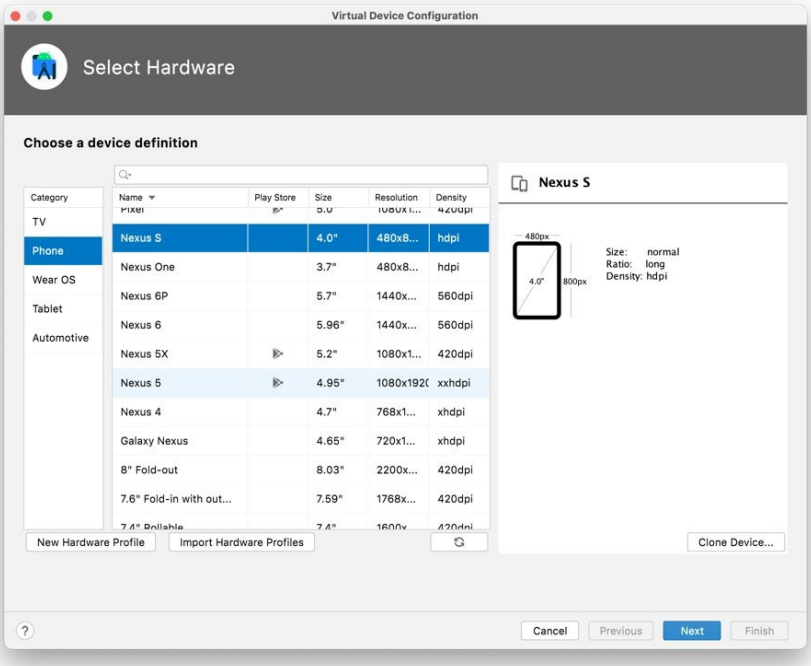


Figure 1.10b

There are all sorts of different options here that you can choose. The Android

emulator allows us to simulate our apps on the many thousands of Android devices that we don ’ t own. For now, it doesn ’ t matter which one you choose. You can select your preferred one. I will be selecting the ‘ Nexus S ’ . There ’ s nothing special about this but I choose it mainly because it has a moderate resolution as compared to the others with higher resolution. A virtual device with higher resolution will be more demanding on your computer.

Click ‘ Next ’ and you have to select a *system image* (fig. 1.11). Just download the latest recommended one and it will install the requested components.

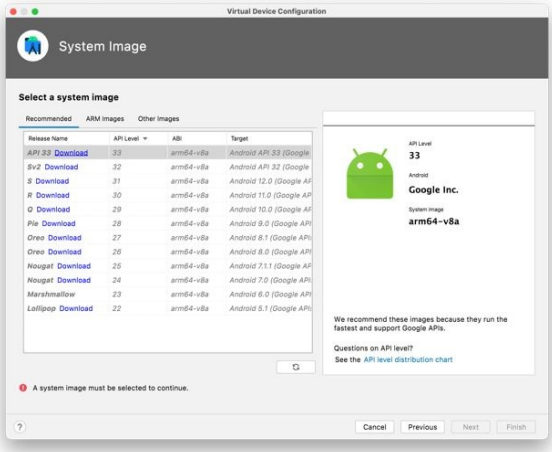


Figure 1.11

The downloading and installation process gets the selected image (API 33) on our computer to make the virtual device (fig 1.12). It will take some time.

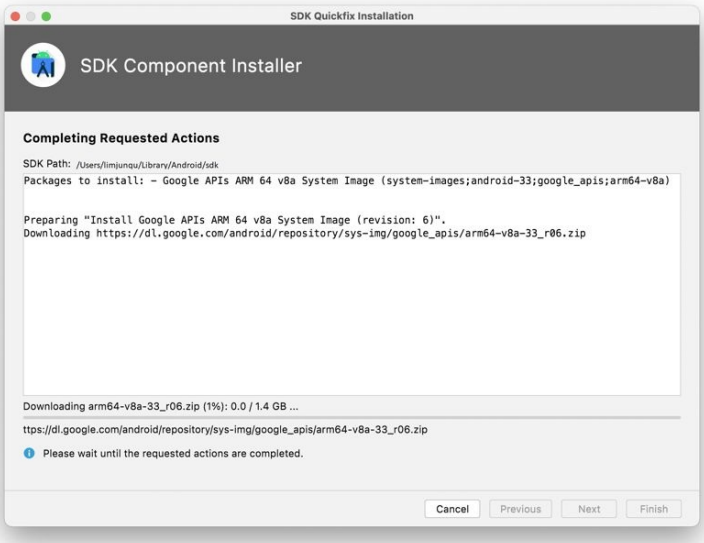


Figure 1.12

When the download and installation finishes, it will ask you to give a name for the AVD (fig. 1.13).

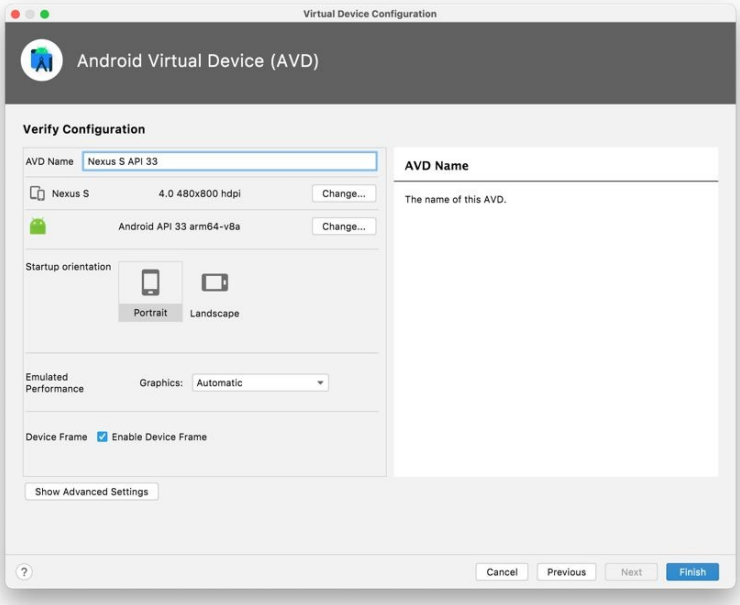


Figure 1.13

We can keep the default name i.e. ‘ Nexus S API 33 ’ . With the name, we know which device it is emulating (Nexus S) and its API level (API 33). Click

‘ Finish ’ .

The new virtual device will then be created and you can select and run it.

So go ahead, launch the virtual device by clicking on the ‘ Run ’ button. It will take some time to get the virtual device set up. Eventually, you should see your virtual device showing (fig. 1.14).

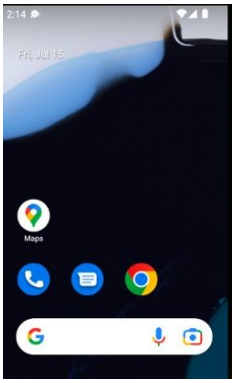


Figure 1.14

It will then run our Hello World app (fig. 1.15).

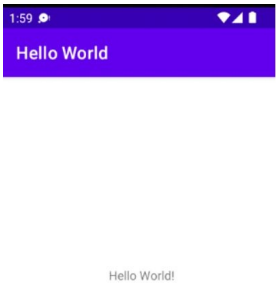


Figure 1.15

Running the App on a Real Device

To run an app on a real device, we first need to set up the Android device for debugging. To enable debugging on the phone, go to ‘ Settings ’ , go to ‘ About device ’ or ‘ About Phone ’ . Find the ‘ Build Number ’ option and tap it repeatedly until you get a message that ‘ You are now a developer! ’ (fig. 1.15b)

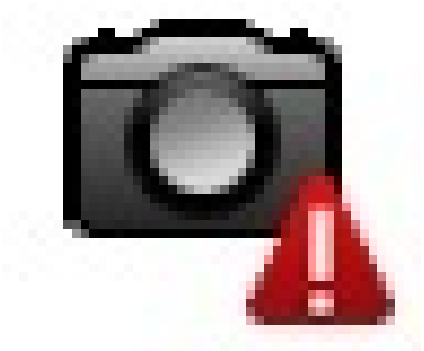


Figure 1.15b

Different device manufacturers might structure the above in slightly different menu options, but by and large, the sequence should be quite close.

Next, go back to ‘ Settings ’ . Tap on ‘ Developer options ’ . Toggle the ‘ USB Debugging ’ option (fig. 1.15c).

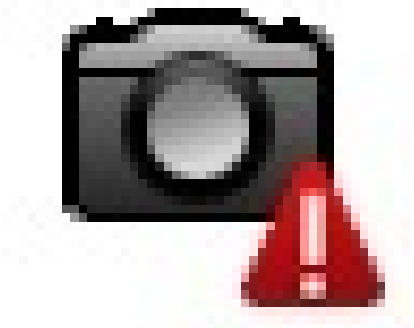


Figure 1.15c

You can now connect your Android device to the computer via USB. Click the play icon on the top of the Android Studio toolbar and when prompted, click ‘ OK ’ to run the app on your specified device.

You should now be able to run your app on debug mode on your actual device.

Running our App on Older Versions of Android

In this section, we wish to touch on how to configure our app so that it can run on older devices with previous versions of Android.

On the left pane of Android Studio, open ‘ Grade Scripts ’ and select *build.gradle(Module: Hello_World.app)* (fig. 1.16)

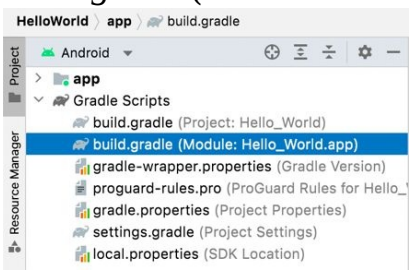


Figure 1.16

In the configuration code, change *minSdk* from 33 to 17:

```
…
android {
    compileSdk 33

    defaultConfig {
        applicationId "com.example.helloworld"
        minSdk 17
        targetSdk 33
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
    …
```

At time of writing this book, version 17 is the one that will support 99.9% (i.e. nearly 100%) percent of devices.

When you make the change, Android Studio will prompt you to do a project sync (fig. 1.17).

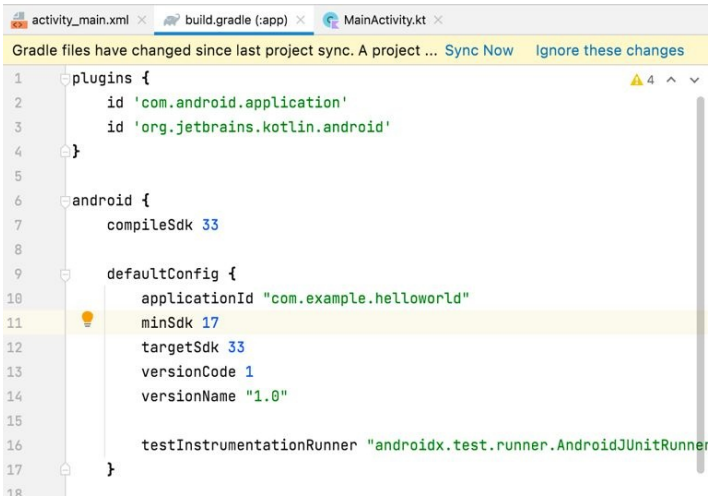


Figure 1.17

So click on ‘ Sync Now ’ .

What we have done in this file is to target version 33, but this should be able to run back all the way to 17. Android Studio will keep this in mind and if you ever write a line of code that uses a feature that can ’ t be used on Android 16, 18, and so on, it will notify you to increase the *minSdk*to API 19 or whatever it is. You can come back to this file and change this back to 18 or 20 or whichever version you want so that you can take advantage of those features in that API.

And if you ever want to test out your app on an earlier API version, you can always create a new virtual device and install a previous version of Android. So this is how we enable our app to be inclusive to as many versions of Android as possible yet utilizing the features offered by the latest versions.

Introduction to MainActivity

Next, to provide a high-level understanding of how an Android app works, we will give a brief explanation of *MainActivity.kt* where we have the code:

```
package com.example.helloworld

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
```

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

Code Explanation

```
package com.example.helloworld
```

We first have the package declaration. This is the package name we chose when we created the project. Every Kotlin file will have a package declaration at the top.

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
```

Next, we import the *AppCompatActivity* class from the *androidx.appcompat.app* package and sub packages. In doing so, we will have access to these classes. We won ’ t discuss all of these classes in this chapter, but what we want is to give the idea that we can import classes to give us access to more functionality. Later, we will illustrate importing more classes to improve our app.

```
class MainActivity : AppCompatActivity()
```

In the above, we have our class declaration of class *MainActivity*. The colon ‘ : ’ means that our class *MainActivity* will be of type *AppCompatActivity*. This means that we have access to code in *AppCompatActivity*.

onCreate

```
class MainActivity : AppCompatActivity() {

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

In class *MainActivity*, we have the function *onCreate*. The *fun* keyword makes it clear that this is the start of a function. *onCreate* is called just before the app

starts. So if we want certain code to run when an activity first shows up (e.g. *setContentView*), we have to write it in this function.

setContentView is a function provided by the Android API and is the function that prepares and displays the UI to the user. We will later explain what is *R.layout.activity_main*.

Now, let ' s illustrate how code changes to *MainActivity* work by changing the background color when the activity first shows up. Add the following code in **bold**:

```
…
class MainActivity : AppCompatActivity() {

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

        window.decorView.setBackgroundColor(Color.GREEN)
    }
}
```

Now when you hit the run button and the app first opens up, it should now appear with a green background (fig. 1.18).

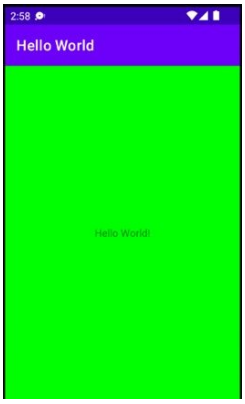


Figure. 1.18

This is a simple example of how we can write code to change things in our app.

Let ' s now talk more about Activities. As mentioned, Activities are like screens in our app. If we move between multiple screens in our apps, each one of those

would be an Activity. The code version of the Activity is in a Kotlin code file like *MainActivity.kt*. You also have the layout file which determines how the app will look for that activity. To see the layout file, go to the left side menu and under *app/res/layout* (fig. 1.19), you have *activity_main.xml*.

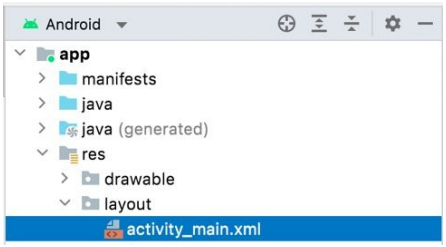


Figure 1.19

activity_main.xml is where we define the visual part of our app. For example, whenever we want to add a button or label to our app, it's all going to happen inside of *activity_main.xml*. A layout file is automatically created for us whenever we create a new Activity.

Remember our *onCreate* method in *MainActivity*?

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

In *setContentView*, the argument passed in (*R.layout.activity_main*) actually refers to the *activity_main.xml* located in the *res/layout* folder, where *R* refers to ‘ res ’ . This is the connection between our Kotlin code and our XML layout/design.

If we look inside *activity_main.xml*, you have a little ‘ Hello World ’ label showing up in the middle of the screen. If it appears too small for you, you can zoom in. It is actually called a *TextView* and it ’ s a way to display some text. We can go ahead and change this text by opening the ‘ Attributes ’ pane on the right. And under ‘ Declared Attributes ’ , change text to ‘ Hello Greg! ’ or whatever text you like (fig. 1.21).

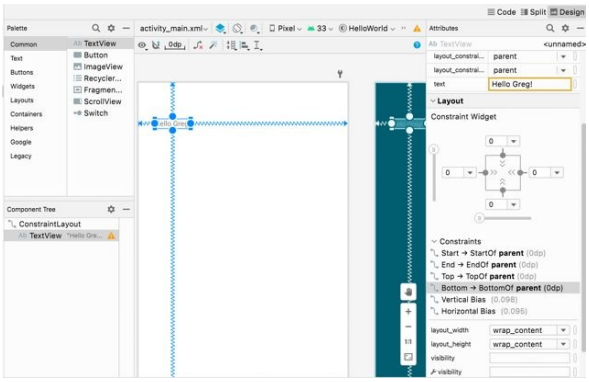


Figure 1.21

In an older version of Android Studio, because I felt the text size was too small, I could go to the *textSize* attribute under ‘ All Attributes ’ and change the font size to 36sp (fig. 1.22). However, with the latest version of Android Studio at the time of writing this is unfortunately no longer apparent and the text size can only really be done in XML code which will be shown below.

(*sp* stands for scalable pixel. This means that when the user changes the font size settings on their Android device, the font will dynamically resize itself.)

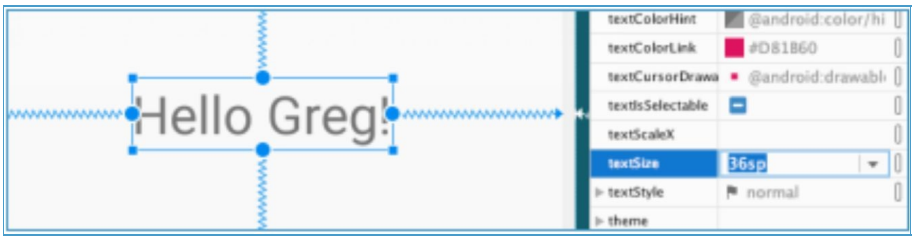
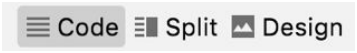


Figure 1.22

You can go ahead and run the app on your virtual device to see how it looks like (fig. 1.23).

Figure 1.23

So now you understand that the *Design* view is a graphical representation of the XML code contained in *activity_main.xml.*

Next, if we click on the ‘ Code ’ tab, 

,we can actually see the XML code that defines our layout file. That is, whatever attributes we have added or edited, it actually configures this XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:textSize="36sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.43" />
```

</androidx.constraintlayout.widget.ConstraintLayout>

We will talk more about layouts later, but to recap for now, whenever we make a new ‘ screen ’ in Android, the correct technical term for that is an Activity. There is going to be a layout file (e.g. *activity_main.xml)* and a code file for that activity (e.g. *MainActivity.kt*). The layout file is actually an XML file beneath. In the next section, we will go ahead to implement the layout for our Body Mass Index Calculator app and show how to connect the layout file to the Kotlin code file.

Body Mass Index (BMI) Calculator app

Now, let ’ s begin developing our first functional app! We will be developing a Body Mass Index (BMI) Calculator app. If you have not heard of BMI before, the BMI helps to see if you are at a healthy weight. To work out your BMI:

- divide your weight in kilograms (kg) by your height in meters (m)
- then divide the answer by your height again to get your BMI

For example: If you weigh 70kg and you're 1.75m tall, divide 70 by 1.75 – the answer is 40
then divide 40 by 1.75 – Your BMI is 22.9.

We then have the following classifications:
Underweight: Your BMI is less than 18.5.
Healthy weight: Your BMI is 18.5 to 24.9.
Overweight: Your BMI is 25 to 29.9.
Obese: Your BMI is 30 or higher.

Our app will look something like (fig. 1.23b):

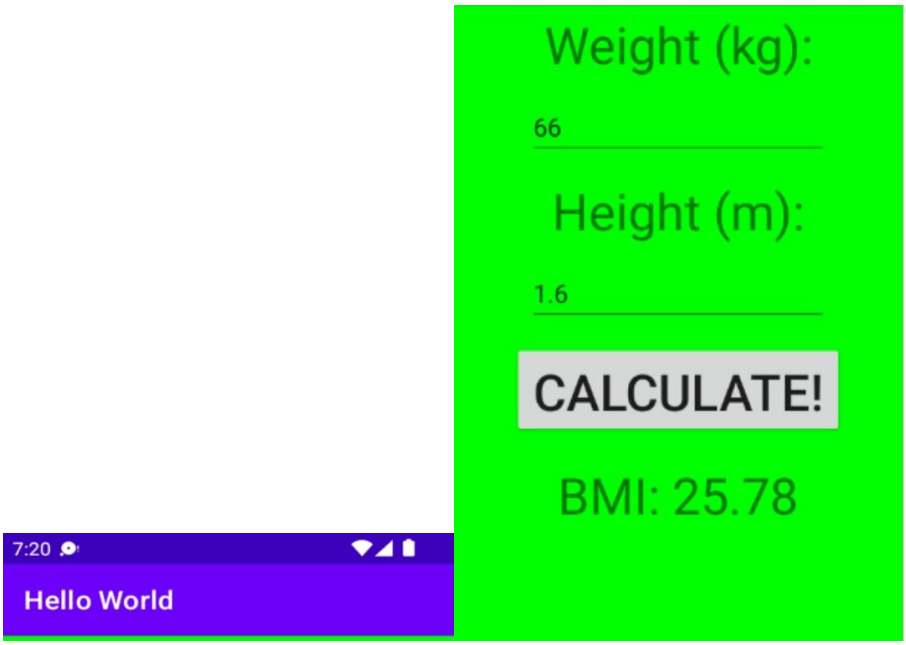


Figure 1.23b

Now, let ’ s design its layout. In our current layout, we have a single TextView. We will not be using the TextView as I want you to experience what it ’ s like to start from scratch. So, select the text view and hit the delete key.

It is easy in Android to add visual objects to our app. We do so by selecting them from the palette (fig. 1.24).

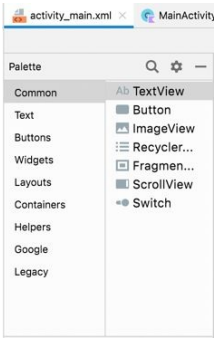


Figure 1.24

The palette contains a list of visual objects that you can add to your app. Under ‘ Common ’ , there are things like a TextView, Button, ImageView etc. And if

you explore the other categories e.g. Text, Buttons, it shows you other different things you can add.

You can also search for the visual that you want by typing it into the search field (fig. 1.25).

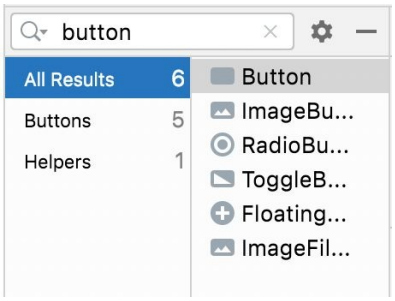


Figure 1.25

For our app, drag a textview into our app and place it somewhere near the top. Let ’ s increase the text size to 36sp (fig. 1.26).

Figure 1.26

We increase the text size by referring to the *Attributes* window to the right of the editing window (fig. 1.26b).

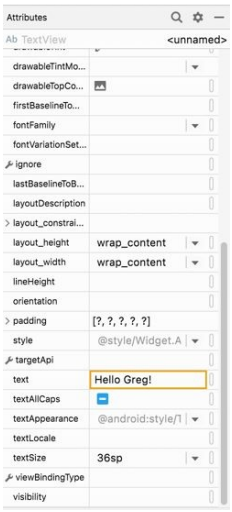


Figure 1.26b

You will see a substantial number of attributes. But you do not have to be concerned with the majority of them for now as we will focus on a few core attributes in this book. Just be aware of the vast number of attributes that shows the versatility Android Studio provides for UI design.

Now, because Android devices come in various screen sizes, how do we ensure that our text view is always the same distance from the top and centered?

To do so, we will be using constraints. To center the textview, hold ‘ Control ’ and select the text view, choose ‘ Center ’ , ‘ Horizontally ’ (fig. 1.27).

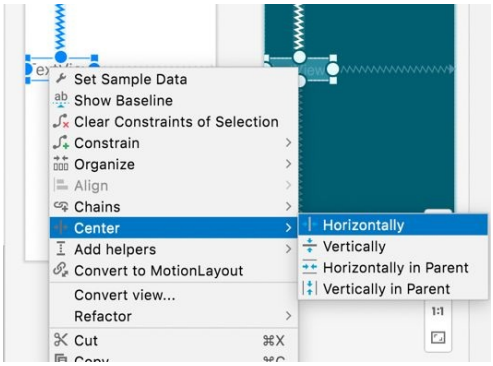


Figure 1.27

You will then see a left and right constraint to center the text view (fig. 1.28).

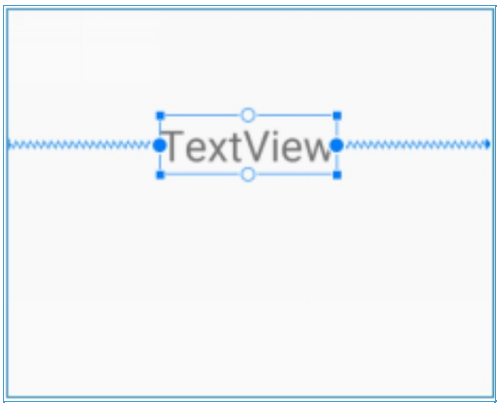


Figure 1.28

To add a constraint saying that it should be a certain distance from the top, under ‘ Layout ’ , ‘ Constraint Widget ’ , hit the ‘ + ’ button at the top (fig. 1.29).

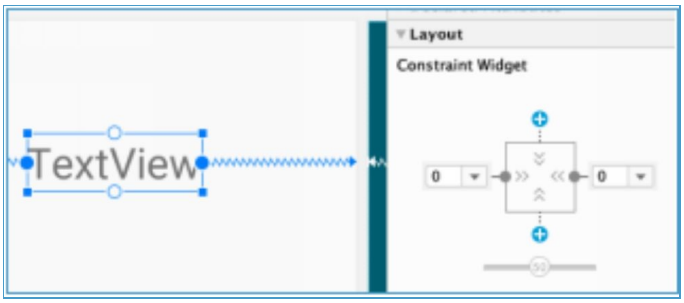


Figure 1.29

And then specify 20, meaning that it should be 20 points from the top (fig. 1.30).

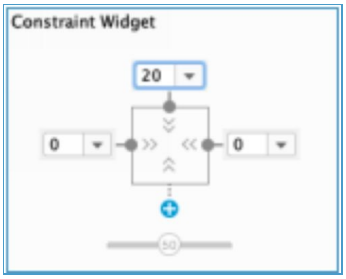


Figure 1.30

And now, we have added our two constraints saying that the textview should be 20 from the top and centered in the middle regardless of the screen size (fig. 1.30b).

Figure 1.30b

If ever we want to change the constraints, we can just select the constraint and edit the values.

You can test to see how your app looks on different devices (e.g. Pixel, Nexus 7) by selecting from the drop-down (fig. 1.31)

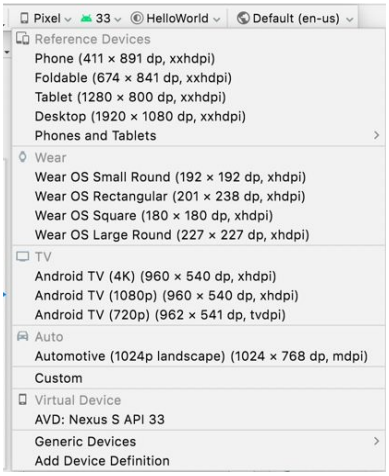


Figure 1.31

And because we have added our constraints, the textview will always be 20 from the top and centered in the middle regardless of the device.

Adding an EditText

We have added a text view to display a text. Now how do we let the user input some text?

Back in the Palette, under ‘ Text ’ , are the various input text boxes we can use (fig. 1.32). They are called EditText.

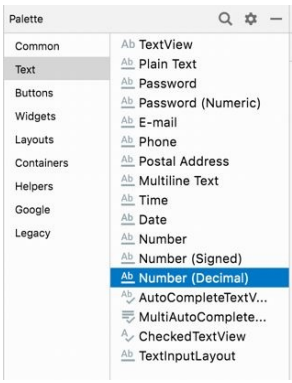


Figure 1.32

There ’ s an EditText for password, for emails, phone numbers, date etc. Because we are working with numbers, we want to drag a *Number (Decimal)* edit text into our app below the text view. At the same time, change the text of the text view to ‘ Weight (kg): ’ (fig. 1.33). Our *Number (Decimal)* edit text will be used to input the weight of a user which could hold values like 64.8, 50.5.

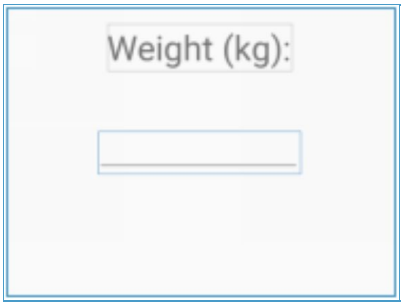


Figure 1.33

Again, let ’ s center the *Number (Decimal)* edit text by hitting on ‘ Control ’ and selecting it, click ‘ Center ’ and select ‘ Horizontally ’ (fig. 1.34).

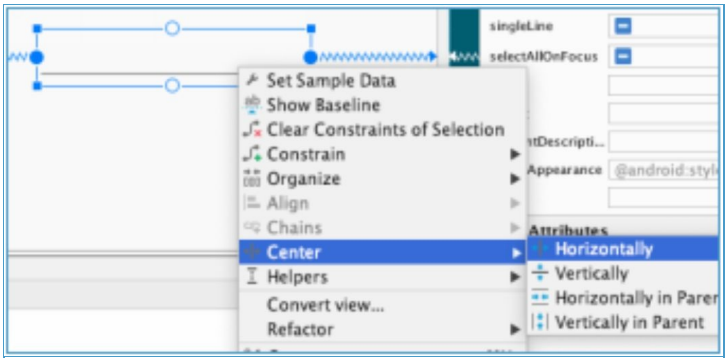


Figure 1.34

Next, we want the distance from the *Number (Decimal)* edit text to the text view to be fixed across different devices. To do so, click on the top constraint of the edit text and then click-drag to join it to the bottom constraint of the Weight text view (fig. 1.35).

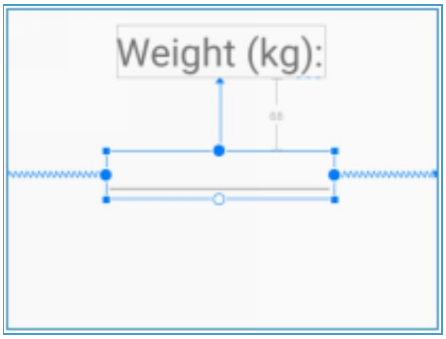


Figure 1.35

Now, run your app and see what it looks like (fig. 1.36).

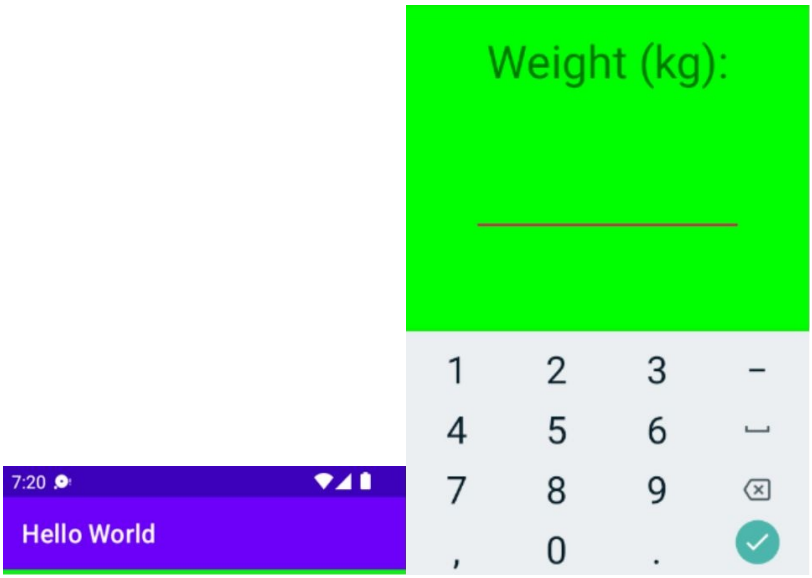


Figure 1.36

And when you click on the EditText, the number keyboard is shown and you can punch in some numbers.

Try It Out

Can you try adding a “ Height (m): ” TextView and an EditText to capture its

value from the user? Add them below the “ Weight ” TextView and EditText. Remember to center them and also add the vertical constraints. In the end, you should have something like (fig. 1.37):

Figure 1.37

Notice that I have a hint text in my edit text boxes to help the user input values i.e. “ enter weight in kg ” (fig. 1.38).

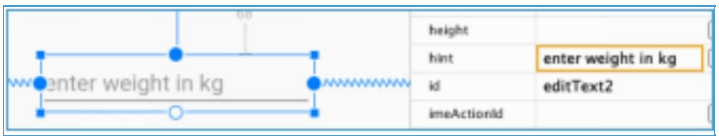


Figure 1.38

You can specify the hint text in the *hint* attribute. And when you run your app, it should look like (fig. 1.39):

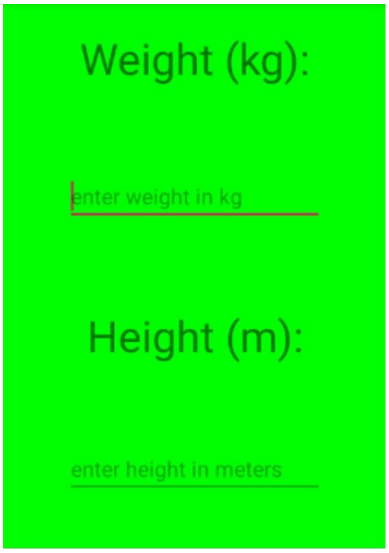


Figure 1.39

Hopefully, you are feeling a bit more confident on how to design your layout via dragging and dropping visual controls. As compared to designing user interfaces for Android in the past, things have indeed been so much easier and neater!

Next, let ’ s add a button that calculates the BMI index.

Try It Yourself

Can you locate the button from the palette to drag into your app (fig. 1.40)?

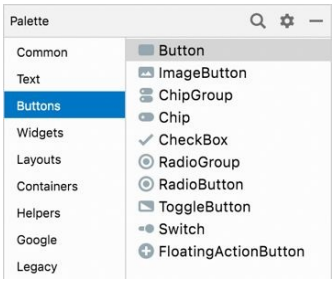


Figure 1.40

When you have added your button, let ’ s increase the size of the text to 36sp just as what we have done with the TextViews and EditTexts. We will also change the text of the button to ‘ Calculate! ’ .

Remember to center your button horizontally and add the vertical constraint

from the button ‘ s top constraint to the above edit text view ’ s bottom constraint. It should look something like (fig. 1.41):

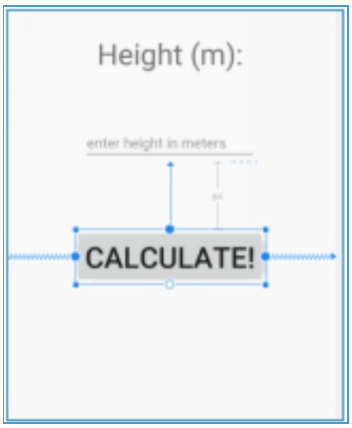


Figure 1.41

Linking Layout to Code

Now, we are going to get the items in our layout to work with our programming code. To do so, we first have to identify them. For example, we want to know what has been entered into our Weight and Height edit texts. To identify our Weight and Height edit texts, we give them an *id* (fig. 1.42).

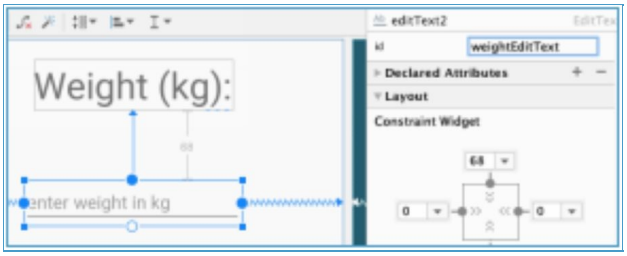


Figure 1.42

Select the weight EditText, and under *id*, name it *weightEditText*. You can of course name it any name you want, but when naming things in the Android world, I first give the name of what it is, in our case *weight*, and then append it with what kind of a visual control it is i.e. *editText*, thus *weightEditText*.

We will do the same for height edit text to give it an *id heightEditText* (fig. 1.43).

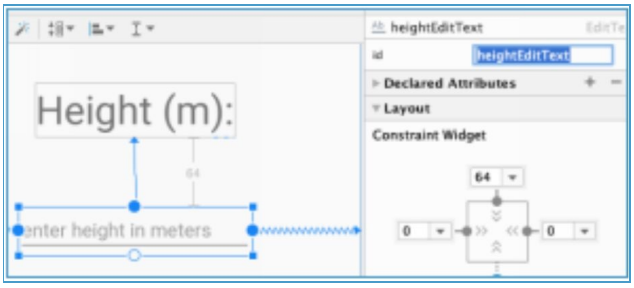


Figure 1.43

We do the same for our calculate button to give it an *id calculateButton* (fig. 1.44).

Figure 1.44

Till now, we have not provided a place for the BMI result to be outputted. Let ’ s drag a TextView into our app below the button for that purpose. Remember to add the center, top constraints and enlarge the text size as what we have done earlier (fig. 1.45).

Figure 1.45

We then give the TextView an *id*, *resultsTextView*. With this TextView, we show back to the user the BMI classification results.

Calculate Button and Output

Next, we will make it so that when we hit the calculate button, we retrieve the information from the weight, height edit texts, calculate the results and output it to the results text view.

Go back to *MainActivity.kt*.

To access our weight edit text from our Kotlin code file, in the *onCreate* function, we simply type out the *id*, *weightEditText* and you can start working with it:

```
class MainActivity : AppCompatActivity() {

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

        window.decorView.setBackgroundColor(Color.GREEN)

        weightEditText
    }
}
```

Android Studio has now made it so much easier to work with visual controls in the layout file. Before, with Java, you had to type a much longer line of code to access the visual control. Now, we simply specify the *id* and then start working with it.

For example, we want our *resultsTextView* to be visible only when *Calculate!* button is pressed. In the meantime, we want it to be invisible. To do this, we add the below line of code:

```
class MainActivity : AppCompatActivity() {

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

        window.decorView.setBackgroundColor(Color.GREEN)

        resultsTextView.visibility = View.INVISIBLE
    }
```

}

We access the *visibility* attribute of *resultsTextView* to make it invisible.

Let ’ s run our app to see if this works. And yes, it does! The text view is invisible (fig. 1.46).

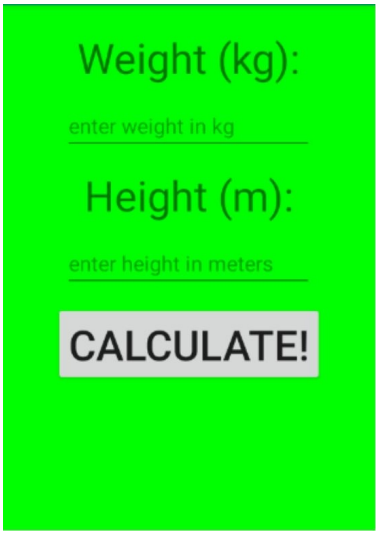


Figure 1.46

*Note: if your calculate button seems to be cut off from the emulator screen, reduce the vertical space between the text view and edit texts to fit them.

Calculate Button

Next, we want to write some code to say that whenever ‘ Calculate! ’ button is hit, we run the code to calculate BMI index. To do so, we set the *setOnClickListener* method of *calculateButton* as shown below:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    window.decorView.setBackgroundColor(Color.GREEN)

    resultsTextView.visibility = View.INVISIBLE
```

```
    calculateButton.setOnClickListener{
        // code here will be ran when calculateButton is clicked.
    }
}
```

By now, I hope that you see that the link between the layout and Kotlin code. The Kotlin code is able to access the different visual controls using their *id*.

Calculating BMI Index

Let ' s go ahead and implement the code in *calculateButton.setOnClickListener* to calculate the BMI index. Add in the following codes:

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

    calculateButton.setOnClickListener{
        var weight = weightEditText.text.toString().toDouble()

        var height = heightEditText.text.toString().toDouble()
        var bmi = weight / (height * height)

        resultsTextView.visibility = View.VISIBLE
        resultsTextView.text = "BMI: " + bmi
    }
}
```

Code Explanation

```
var weight = weightEditText.text.toString().toDouble()
var height = heightEditText.text.toString().toDouble()
```

In the above code, we assign *weight* and *height* with the user inputted value from their EditTexts. We access their inputted value with the *text* property and convert it to a *String* with *toString()*. We then convert the *String* to *Double* using *toDouble(…)* because we will be executing the mathematical operation:

```
var bmi = weight / (height * height)
```

We then display the calculated BMI in the *resultsTextView*:

```
resultsTextView.visibility = View.VISIBLE
resultsTextView.text = "BMI: " + bmi
```

We first make *resultsTextView* visible. We then append the text ‘ BMI: ’ to the BMI value and assign it to *resultsTextView*.

Running Your App

When you run your app in the simulator, fill in your weight in kilograms and height in meters (e.g. weight: 80, height: 1.7) and when you click ‘ Calculate! ’ , you have your BMI index displayed (fig. 1.47)! How did you fare? My BMI isn ’ t too good. I had better exercise more!

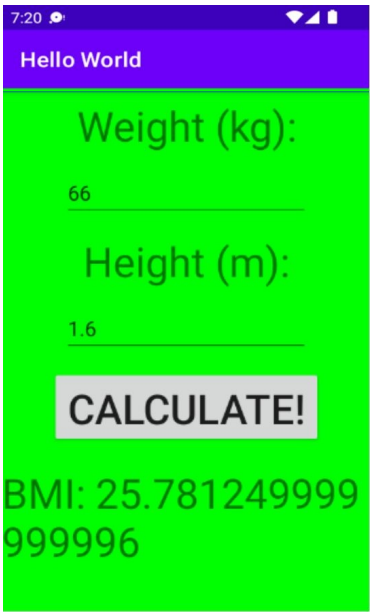


Figure 1.47

Now, you might encounter a problem where your number keyboard is blocking the results (fig. 1.48).

Figure 1.48

In which case, you have to hit the ‘ back ’ button  to hide the keyboard and show the results. This however is not a good user experience and we will improve this in the next section by displaying the result in a separate activity.

Another issue is that we should only display the result to two decimal places (d.p.). To format our result to two d.p., we do the following:

```
resultsTextView.text = "BMI: " + String.format("%.2f",bmi)
```

and our BMI should now display a nicely formatted to 2 d.p. result (fig. 1.49).

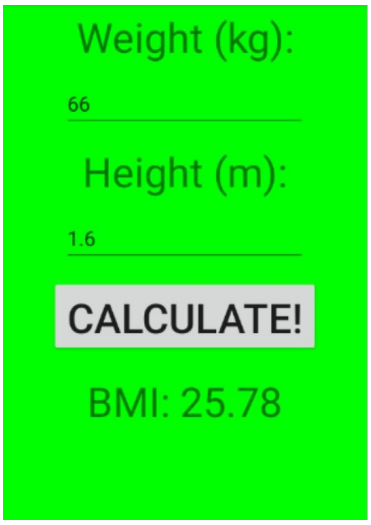


Figure 1.49

Showing BMI Results in a New Activity

In this section, we will improve our app by displaying the result in a separate activity. That is when someone hits ‘ Calculate! ’ , the result is shown in a new screen. To do so, we first need an Activity for the new screen.

To add a new Activity, in the project pane, under *java*, right-click and select ‘ New ’ , ‘ Activity ’ , ‘ Empty Activity ’ (fig. 1.50).

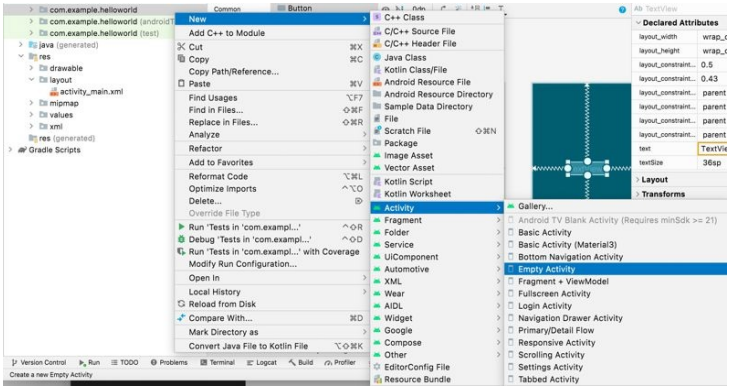


Figure 1.50

Name the activity *BMIResultsActivity* and ensure that the ‘ Generate Layout File ’ checkbox is checked (fig. 1.51).

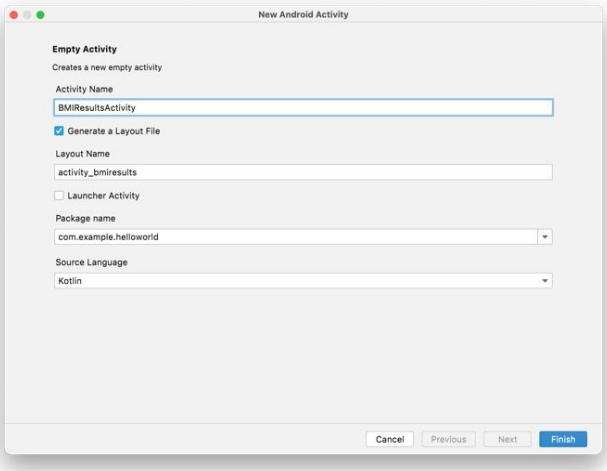


Figure 1.51

When you click ‘ Finish ’ , *BMIResultsActivity.kt* will be created for you. In *res/layout*, *activity_bmiresults.xml* (the layout file for the activity) will also be created for you.

Drag a text view into the activity and name its *id resultTextView*. Remember to increase the text size and center it (fig. 1.52).

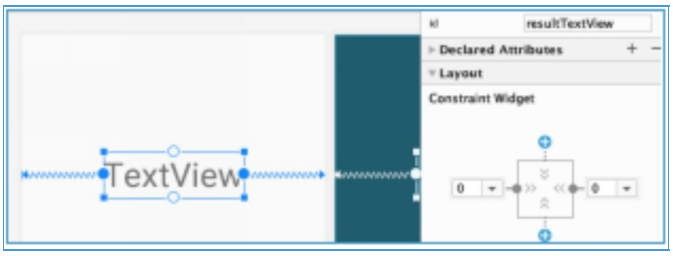


Figure 1.52

Next in *MainActivity.kt*, comment out the codes as shown and add the codes in **bold**:

```
calculateButton.setOnClickListener{
  var weight = weightEditText.text.toString().toDouble()

  var height = heightEditText.text.toString().toDouble()
  var bmi = weight / (height * height)
```

```
    //resultsTextView.visibility = View.VISIBLE
    //resultsTextView.text = "BMI: " + String.format("%.2f",bmi)

    val resultIntent = Intent(this,BMIResultsActivity:: class.java)

            resultIntent.putExtra("result","BMI: " +
                                    String.format("%.2f",bmi))
    startActivity(resultIntent)
}
```

Code Explanation:

```
    //resultsTextView.visibility = View.VISIBLE
    //resultsTextView.text = "BMI: " + String.format("%.2f",bmi)
```

We comment out the codes to display BMI results to the text view since we are displaying them in the new Activity.

```
val resultIntent = Intent(this,BMIResultsActivity:: class.java)
```

In the above line of code, we are creating an *intent*. An *intent* lets us move from one *Activity* instance to another. In our case, we specify that we move from the current activity (*this*) to the *BMIResultsActivity*. You might be wondering, what is that weird “ :: class.java ” for?

:: class is to retrieve the full name of the *BMIResultsActivity* instance as declared in *AndroidManifest.xml.*

.java on the end is because all the Kotlin code is turned into Java byte code. Thus, the entire statement *BMIResultsActivity:: class.java* provides the fully qualified name for referencing the destination activity.

Note also that Android Studio might prompt you to import *android.content.Intent* in order to use Intent. In general, when you hover over code that needs to be imported (as shown by the red squiggly underline), hold the ‘ Alt ’ key and tap *Enter*. Android Studio will add the required *import* statement at the top of the code with the other imports.

```
resultIntent.putExtra("result","BMI: " + String.format("%.2f",bmi))
```

Because we need to pass the BMI results to the new Activity, we use the

putExtra method which allows us to store values that we want to pass over. The values are stored in something like a hashtable. i.e. we provide a key to store a specific value. In our case, we specify the ‘ result ’ key which stores the formatted BMI result. This will be retrieved by the destination activity using *extras.getString* as we shall see later.

```
startActivity(resultIntent)
```

We then start the Activity. Android Studio should prompt you to *import androidx.core.content.ContextCompat.startActivity*.

Here, we are using an *Intent* to switch between Activity instances. But *Intent* also allows us to interact with other apps too. For example, to open the relevant activity for a user to send an email, make a phone call, open a web page in a browser etc. In a later chapter, we will illustrate opening a web page in a browser.

BMIResultsActivity.kt

Now, in the destination activity, we need to retrieve the results passed in. To do so, in *BMIResultsActivity.kt*, add in the *onStart* method in **bold**:

```
class BMIResultsActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_bmiresults)
    }

    override fun onStart() {
        super.onStart()
        val result = intent.extras.getString("result")
        resultTextView.text = result
    }
}
```

In the above, you might ask, why use *onStart*? Won ’ t *onCreate* be sufficient? Now that is because *onCreate* is executed when the Activity is being created, that is, we get things ready for the app, e.g. UI, graphics, sound etc. Whereas *onStart* is called (after *onCreate*) when the app is in the starting phase. In our case, to get the value from the *intent*, it would be better suited to place it in

onStart after our app is readied.

```
val result = intent.extras.getString("result")
```

In *onStart*, we retrieve the value passed in by providing the key ‘ result ’ and assign the retrieved value in the *result* variable. We then display it to *resultTextView* with *resultTextView.text = result*.

When you run your app now, enter your weight, height and hit ‘ Calculate! ’ , your results will now be shown in a separate screen (fig. 1.53)!

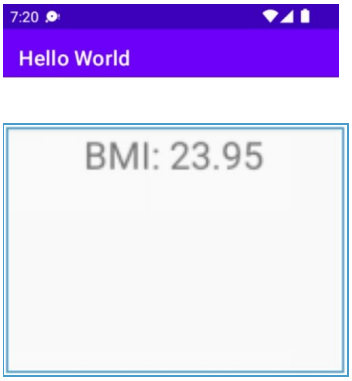


Figure 1.53

You would have noticed that we use two different keywords to declare variables, *val* and *var*. *val* is for storing values that cannot be changed again during execution. It is only readable and what we term technically as immutable.

var corresponding is for values that can be changed. It is readable, writable and is technically termed as mutable.

If you write code that attempts to change a *val* type, Android Studio will show an error. So what is the reason we use *val*? It is to let the compiler help protect us from making mistakes. For example, in the above code, we define result as a *val* because it is a computed result being passed from a previous Activity and should not be changed. Other examples when we use *val* are constants like:

```
val pi: Float = 3.14f
```

Summary

In this chapter, we began the journey on **Kotlin and Android** App Development to go from absolute beginner to having our app submitted to the App Store.

Chapter 2: Quotes App Using RecyclerView

In this chapter, we will be building a quotes app where you have a list of quotes in a *RecyclerView* (fig. 2.0).

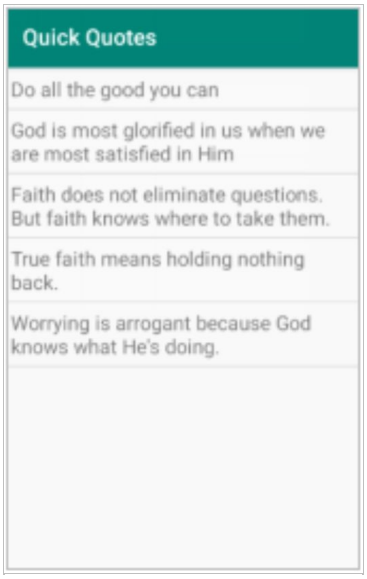


Figure 2.0

The concept we will be learning here is the *RecyclerView* which is an important class to support the display of a collection of data in a list form. It is a modernized version of the *ListView*. Most apps require listing out information. An example is Whatsapp where we can scroll through different chat groups (fig. 2.1).

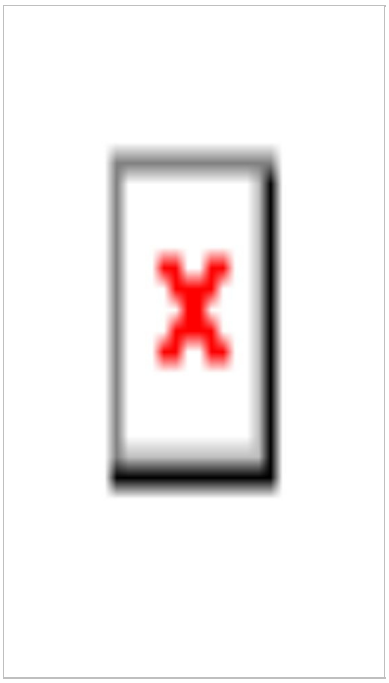


Figure 2.1

Other examples include ‘ Settings ’ (fig. 2.2) and ‘ Contacts ’ .

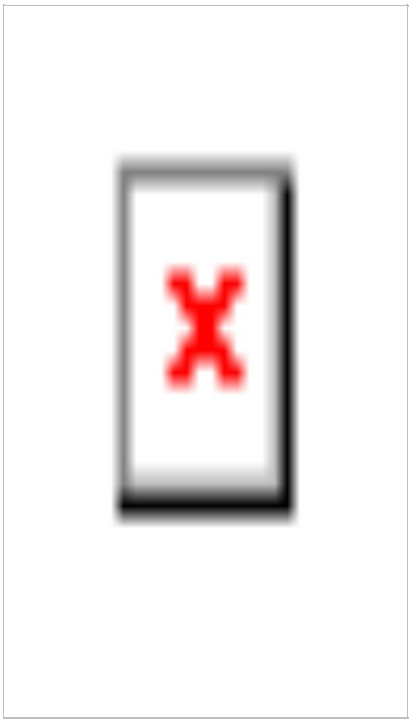


Figure 2.2

So, let ’ s begin developing our RecyclerView app!

First, begin a new Empty Activity project in Android Studio, call it *QuickQuotes* (fig. 2.3). Under ‘ Language ’ choose ‘ Kotlin ’ . Specify minimum API level to be the latest API (33 as of writing). Click ‘ Finish ’ .

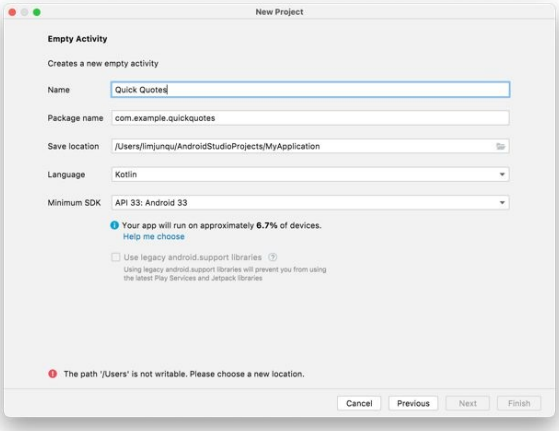


Figure 2.3

Once again, in Gradle Scripts *build.gradle(Module: app)* we specify *minSdkVersion* to be 17.

```
android {
    compileSdk 33

    defaultConfig {
        applicationId "com.example.quickquotes"
        minSdk 17
        targetSdk 33
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
```

We will learn how to list information with a *RecyclerView*. Previously Android provided the ListView to list information. The *RecyclerView* is the updated version of this that has been encouraged by the Android team because it is specifically designed to display potentially long lists of data.

Back in *activity_main.xml*, delete the default TextView and instead, drag a

RecyclerView into it (fig. 2.4).

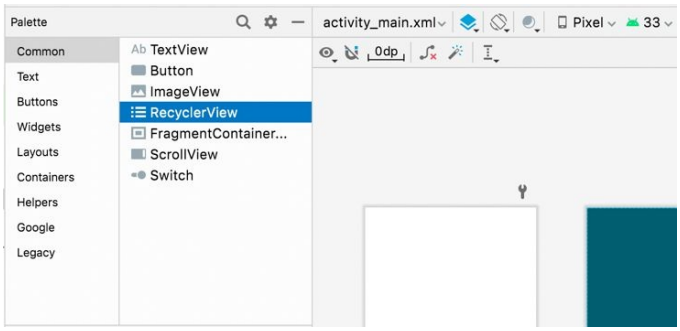


Figure 2.4

Because we want our RecyclerView to take up the entire screen, we apply left/right/top/bottom constraints by making space constraints of 8 all around it (fig. 2.5).

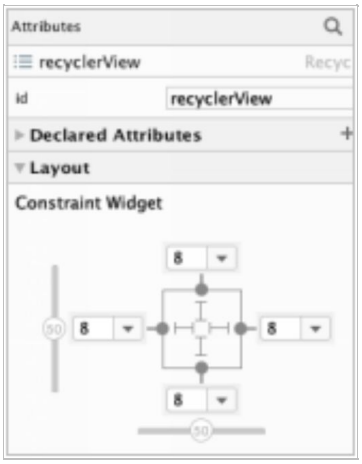


Figure 2.5

Next, make sure that its *layout_width* and *layout_height* attributes are set to ‘match_constraint’. Else, the item will not be displayed correctly. Give the recycler view the id *recyclerView*.

RecyclerView Adapter

We interact with the *RecyclerView* via a special type of class called an *adapter* that understands how the *RecyclerView* works. The job of an adapter is to connect to and show the data that goes inside each row. Let ’ s go ahead to create one.

To create an adapter, we need to create a new Kotlin class in our project. So under, *app/java/<project>* folder, hit ‘ Control ’ and select ‘ New ’ , ‘ Kotlin File/Class ’ (fig. 2.6).

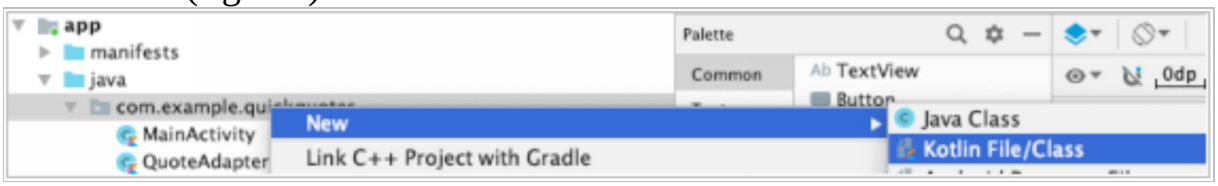


Figure 2.6

You will have a blank file with the package name on the top. In it, fill in the codes in **bold**:

```
package com.example.quickquotes

class QuoteAdapter{


}
```

QuoteAdapter is not just a new class that we are creating. We are going to inherit from another class, which means that we are going to use existing methods/properties from that class and then in our *QuoteAdapter* add on top of it with our own methods/properties. We inherit the Recycler with the ‘ : ’ as shown below,

```
package com.example.quickquotes

import androidx.recyclerview.widget.RecyclerView

class QuoteAdapter : RecyclerView.Adapter(){

}
```

When you inherit *RecyclerView.Adapter()*, Android Studio should automatically add the *import androidx.recyclerview.widget.RecyclerView* statement for you.

Android Studio will also prompt you that it is expecting a *RecyclerView.ViewHolder* argument (fig. 2.7).

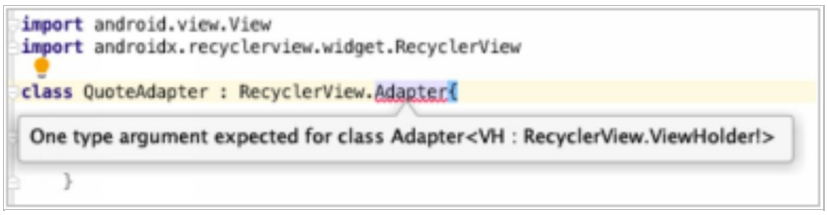


Figure 2.7

RecyclerView.ViewHolder essentially is the view for each row.

To add *ViewHolder*, inside of *QuoteAdapter* class, we need to create a custom *ViewHolder* inner class that inherits from *RecyclerView.ViewHolder*. Add in the below codes into *QuoteAdapter.kt*:

```
import android.view.View
import androidx.recyclerview.widget.RecyclerView

class QuoteAdapter : RecyclerView.Adapter
<QuoteAdapter.QuoteViewHolder>() {

    class QuoteViewHolder(v: View): RecyclerView.ViewHolder(v) {


    }
}
```

We have created an inner class *QuoteViewHolder* in *QuoteAdapter*. An inner class is a class declared inside of a regular class. This gives the outer class the advantage of using the properties and functions of the inner class by declaring an instance of it. In our case, *QuoteAdapter* has access to properties/functions of *QuoteViewHolder*. This ability to define a new type within a class, create instances and share data with it is useful in a RecyclerView implementation.

With this, we have connected our *QuoteAdapter* to our *QuoteViewHolder*. However, Android still highlights an error with *QuoteAdapter* (fig. 2.8).

Figure 2.8

That is because *QuoteAdapter* inherits from *RecyclerView.Adapter* which has certain methods that it has to override and implement. If you click on the red squiggly underline and then on the red little light bulb (fig. 2.9), it prompts you to ‘ Implement members ’ . Hit that.

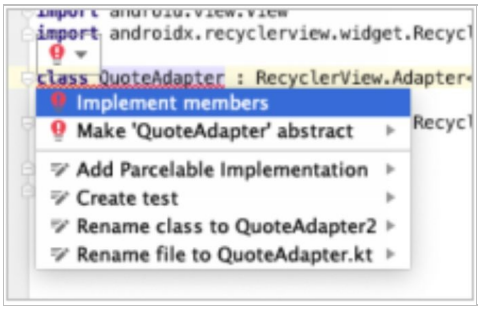


Figure 2.9

In the next screen, press ‘ Shift ’ and highlight all the members to implement and hit ‘ Finish ’ (fig. 2.10).

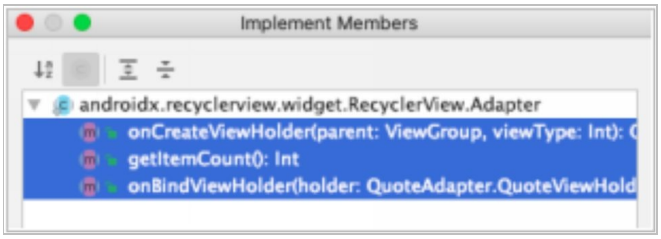


Figure 2.10

Android Studio will then provide some boilerplate code for you to implement the methods:

```
import android.view.View
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView

class QuoteAdapter : RecyclerView.Adapter<QuoteAdapter.QuoteViewHolder>(){
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): QuoteViewHolder {
      …
   }

   override fun getItemCount(): Int {
```

```
        …
    }

        override fun onBindViewHolder(holder: QuoteViewHolder, position: Int) {
        …
    }

    class QuoteViewHolder(v: View): RecyclerView.ViewHolder(v){

    }
}
```

These are the essential functions that help get the recycler adapter to work. As we progress along, the purpose of each function will be clearer. For now, understand that the function *getItemCount* lets the recycler view know how many things it should display. It hence returns an *Int*. Let ' s return 10 for now.

```
    override fun getItemCount(): Int {
        return 10
    }
```

onCreateViewHolder is called when the layout for a list item is required.

```
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): QuoteViewHolder {
        …
    }
```

We will later illustrate the usage of the other methods. For now, let ' s create the view for each row item.

Designing how each row item looks like

To do so, in the side menu, under *res/layout*, control-click ' New ' , ' Layout resource file ' (fig. 2.11). This will be the view holding the quote for each row.

Figure 2.11

Under *File name*, we will just name it *recyclerview_item_row*. Let ' Root element ' be *LinearLayout* and leave the rest of the fields to their default values (fig. 2.12). Hit ' Ok ' .

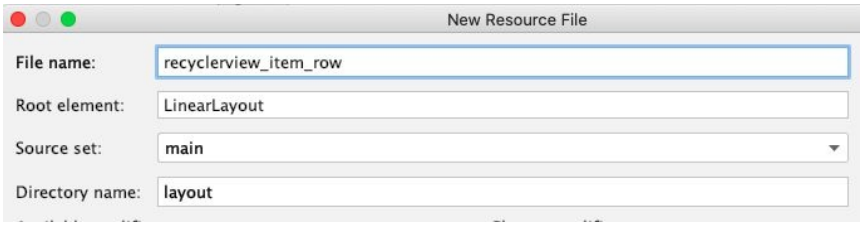


Figure 2.12

recycler_item_row.xml will then be generated. If you look at the design, it might look like a full-screen size, but it is not.

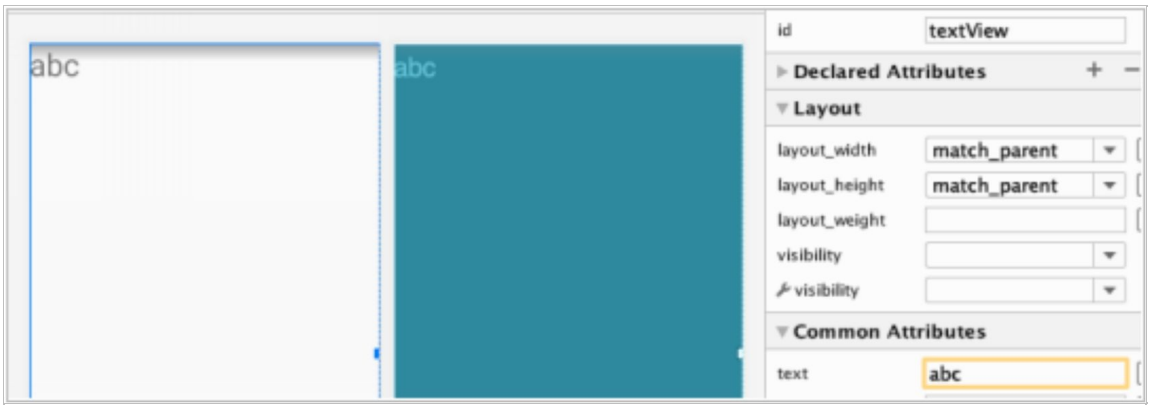


Figure 2.13

Drag a text view into it (fig. 2.13). Give it any text you like for e.g. abc and increase its text size so that it is easier to see. I have left the id as the default *textView*.

Adapter Setup

Now, going back to *QuoteAdapter*, add the following codes in class *QuoteViewHolder*:

```
class QuoteAdapter : RecyclerView.Adapter<QuoteAdapter.QuoteViewHolder>(){
   …


   class QuoteViewHolder(v: View): RecyclerView.ViewHolder(v){
      var view: View = v
      var quote : String = ""

      fun bindQuote(quote: String){
         this.quote = quote
```

```
            view.textView.text = quote
        }
    }
}
```

In *QuoteViewHolder*, we have a reference to its view with the variable *view*. We store the quote in the variable *quote*. We then have the function *bindQuote* which will be called by the *QuoteAdapter* to bind the quote to the text view.

And in *QuoteAdapter*, implement *onBindViewHolder* as:

```
    override fun onBindViewHolder(holder: QuoteViewHolder, position: Int) {
        holder.bindQuote("Sample Quote")
    }
```

onBindViewHolder is called when the *RecyclerAdapter* instance binds to the *RecyclerView* instance in the layout. For now, we provide a hardcoded sample quote.

In *onCreateViewHolder*, implement it as:

```
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): QuoteViewHolder {
        return QuoteViewHolder(LayoutInflater.from(parent.context)
                .inflate(R.layout.recycler_item_row,parent,false))
    }
```

onCreateViewHolder is called when the layout for a list item is required. In our implementation, *LayoutInflater* takes the XML provided, i.e. *recycler_item_row.xml*, and adds (inflates) it to the parent view to get the view ready to be shown on the screen.

Initializing our Layout Manager and Adapter

Now back to *MainActivity.kt*, fill in the below codes:

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.recyclerview.widget.LinearLayoutManager
import Kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    lateinit var layoutManager : LinearLayoutManager
    lateinit var adapter: QuoteAdapter
```

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

        layoutManager = LinearLayoutManager(this)
        recyclerView.layoutManager = layoutManager

        adapter = QuoteAdapter()
        recyclerView.adapter = adapter
    }
}
```

Code *Explanation*

```
    layoutManager = LinearLayoutManager(this)
    recyclerView.layoutManager = layoutManager


    adapter = QuoteAdapter()
    recyclerView.adapter = adapter
```

We connect our recycler view widget to the layout manager and attach an adapter for the data to be displayed.

```
    lateinit var layoutManager : LinearLayoutManager
    lateinit var adapter: QuoteAdapter
```

You will realize that layout manager and adapter are declared with the *lateinit* keyword. The *lateinit* stands for late initialization. Because we don ' t want to initialize layout manager and adapter at first, but instead want to initialize it in *onCreate* and we can guarantee that. We thus declare these two variables with the *lateinit* keyword.

Running our *RecyclerView*

Before we run our app on the emulator, in *recyclerview_item_row.xml*, select the *LinearLayout* and set its *layout_height* to ' wrap_content ' (fig. 2.14). ' wrap_content ' as her name suggests ensures that it is just enough to contain the content within it. Else, if *layout_height* is left as ' match_parent ' there will be a

huge spacing issue since the item row stretches across the full height of the screen.

*the *parent* in this case is the recycler view which is matched to the entire screen

Figure 2.14

When you run your app now, you should see ‘ Sample Quote ’ displayed ten times (fig. 2.15).

Figure 2.15

Now if you make the following changes to change from Linear Layout to Grid Layout with the following changes:

```
..
import androidx.recyclerview.widget.GridLayoutManager
..

class MainActivity : AppCompatActivity() {

    lateinit var layoutManager : GridLayoutManager
    lateinit var adapter: QuoteAdapter

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

```
        layoutManager = GridLayoutManager(this,3)
        recyclerView.layoutManager = layoutManager

                        ..
    }
}
```

You get instead (fig. 2.16) where the quotes are displayed in a grid layout.

Figure 2.16

This is to show you how flexible these recycler views are, especially in switching between different forms of layouts i.e. Linear, Grid. Before we move on, change back to the Linear layout.

Populating our *RecyclerView* Rows from an ArrayList

Currently, we are displaying the same text for each row. We now show how to display distinct values for each row by reading from an ArrayList.

First, we have our *QuoteAdapter* receive an *Arraylist* of different strings with:

```
class QuoteAdapter (val quotes: ArrayList<String>) :
RecyclerView.Adapter<QuoteAdapter.QuoteViewHolder>(){
```

Next in *getItemCount*, return *quotes.count*:

You can see that we no longer return a hardcoded value of 10. We should rather be returning the size of the *ArrayList* to display the corresponding number of rows.

```
override fun getItemCount(): Int {
    return quotes.count()
     }
```

And then in *onBindViewHolder*, we retrieve the quote for that row with *quotes[position]* and then call *bindQuote* with it.

```
override fun onBindViewHolder(holder: QuoteViewHolder, position: Int)
{
  var quote = quotes[position]
  holder.bindQuote(quote)
}
```

position contains the index for that current row. For example, for the first row, *position* will return 0. For the second row, *position* will return 1. Thus, if you add *println(position)* to *onBindViewHolder*, you can see the console printing:

```
0
1
2
3
```

We thus can use *position* to refer to each corresponding element in the *ArrayList*.

And in *MainActivity.kt*, provide your *Arraylist* of strings when you instantiate *QuoteAdapter*:

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

    layoutManager = LinearLayoutManager(this)
    recyclerView.layoutManager = layoutManager

    adapter = QuoteAdapter(arrayListOf(
      "Do all the good you can",
      "God is most glorified in us when we are most satisfied in Him",
      "Faith does not eliminate questions. But faith knows where to take them.",
      "True faith means holding nothing back. ",
      "Worrying is arrogant because God knows what He's doing."))
    recyclerView.adapter = adapter
```

```
}
```

*Note that the above are just my favorite quotes. You can of course use your own.

Running your App

When you run your app now, you should see your list of quotes displayed (fig. 2.17).

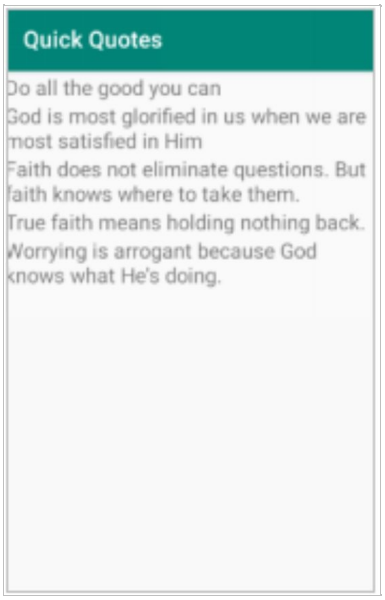


Figure 2.17

For space ’ s sake, I have decreased the text size of the text view. But of course, you can specify your own text size. For now, our text looks lumped together. We can ’ t really differentiate each quote. Let ’ s add a margin padding to each quote. We can do so by specifying a layout margin of 5 dp. Select the *LinearLayout* and in its *layout_margin* property, specify 5 dp all around (fig. 2.18).

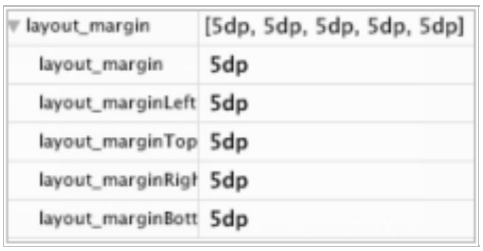


Figure 2.18

(because there are thousands of different Android devices, Android uses density-independent pixels or *dp* as a unit of measurement across different devices. It calculates the density of the pixels of the device the app is running on)

Next, we will add a divider line that divides each row item by adding the below:

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

    layoutManager = LinearLayoutManager(this)
    recyclerView.layoutManager = layoutManager
    recyclerView.addItemDecoration(
        DividerItemDecoration(
            recyclerView.context,
            DividerItemDecoration.VERTICAL
        )
    )
        …
}
```

And when you run your app now, we see a nice divider that divides each row item (fig. 2.19).

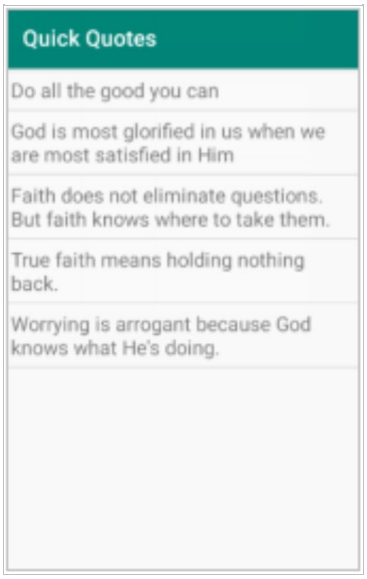


Figure 2.19

We have gone through quite a bit to illustrate the use of a recycler view. Essentially, a recycler has to have a layout manager and an adapter. We instantiated a linear layout manager and then created a *QuoteAdapter* which inherits from *RecyclerView.Adapter* which in turn works with a *ViewHolder* (i.e.

QuoteViewHolder) to display the quote string.

Chapter 3: To Do List App Using RecyclerView & Shared Preferences

In this chapter, we will be building the classic ‘ To Do List ’ app (fig. 3.1). Whether it ’ s a shopping list, errands, lists and so on, the classic ‘ To Do List ’ app always serves as a foundational example for mobile app development. In the process of building the app, we will also revisit and cement the concept of recycler views.

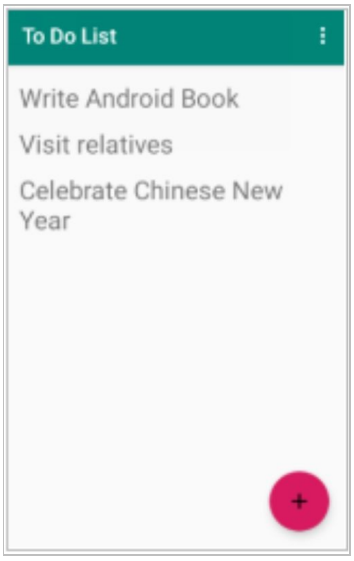


Figure 3.1

So go ahead and create a new Android project. This time however, choose ‘Basic Activity’. Ensure that you have chosen ‘Kotlin’ as Language and name the project ‘To Do List’ (fig. 3.2, 3.3).

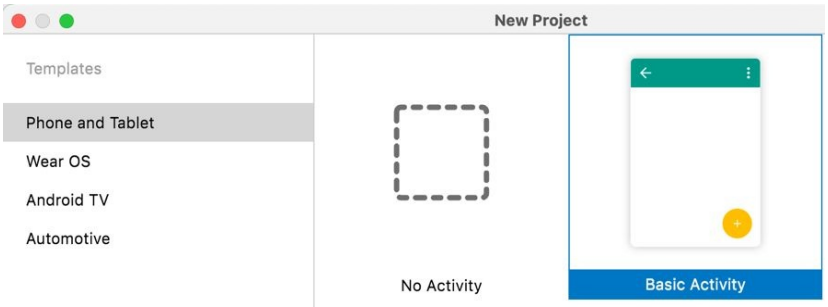


Figure 3.2

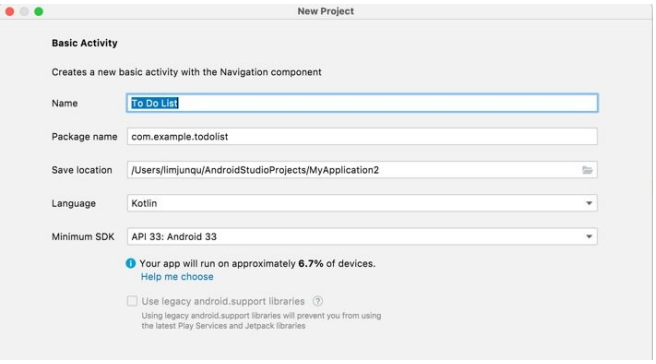


Figure 3.3

Click *Finish*. Once again, set *minSdkVersion* to 17.

For a Basic Activity project, there are two files in *res/layout*: *activity_main.xml* and *content_main.xml*. Note two things. First, *activity_main.xml* has a ‘FAB’ or floating action button (fig. 3.4).


Figure 3.4

Second, to change the content in *activity_main*, for e.g. to edit the text of the TextView, you have to edit *content_main.xml*.

So, go to *content_main.xml*, delete the existing TextView and bring in a RecyclerView. Give it the id *recyclerView* and also make space constraints of 8 all around it (fig. 3.5). Also, make sure that its *layout_width* and *layout_height* are set to ‘match_constraint’. Else, later on the item will not be displayed properly.

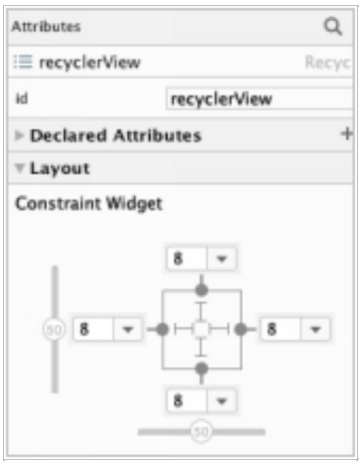


Figure 3.5

The FAB is currently showing an ‘email’ icon. Because we want the button to add to-do items, we will be changing it to a ‘+’. To change the icon of the FAB, select it in *main_activity.xml* and under *srcCompat*, this is where we specify that we want a ‘+’ button (fig. 3.6). But you will realize that there is no ‘+’ to choose from.

Figure 3.6

To have a ‘+’ to choose from, we have to go to *res* folder, Control-click ‘New’, ‘Vector Asset’ (fig. 3.7).

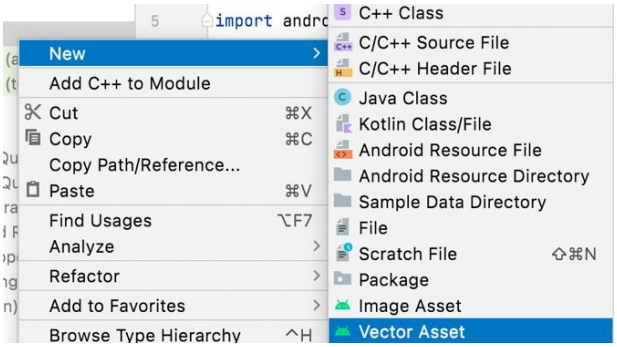


Figure 3.7

and the ‘Configure Vector Asset’ window will appear (fig. 3.8).

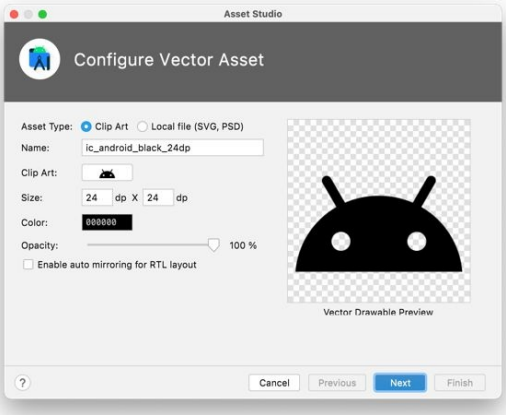


Figure 3.8

Hit on ‘Clip Art’ and you will find many clip art to choose from. Search for ‘add’ and select the ‘add’ icon (fig. 3.9).

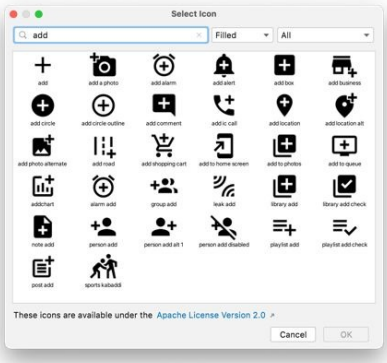


Figure 3.9

Save it to the ‘drawable’ folder which should be selected for you by default (fig. 3.10). Hit ‘Finish’.

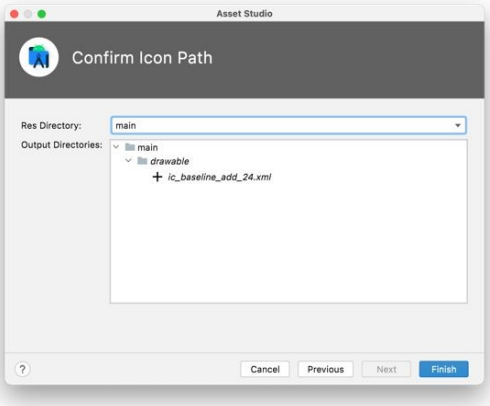


Figure 3.10

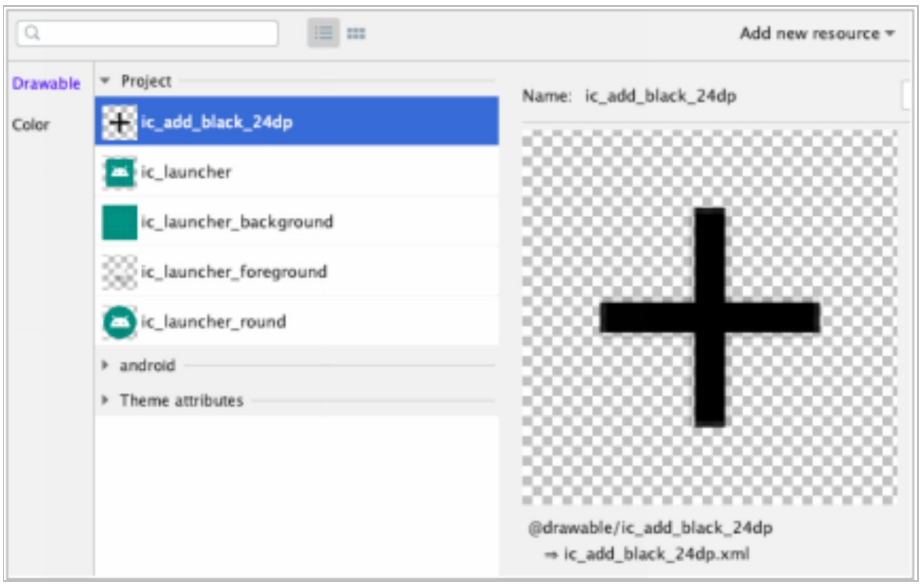


Figure 3.11

Now when you go back to the FAB's *srcCompact* attribute, you should be able to find and select the *add* icon.

Creating the Add To-Do Item Layout

Next, we are going to implement going to the Create To-Do item activity when a user hits on '+'. Control-click on the *java/com…todolist/* folder, select 'New', 'Activity', 'Empty Activity' (fig. 3.12).

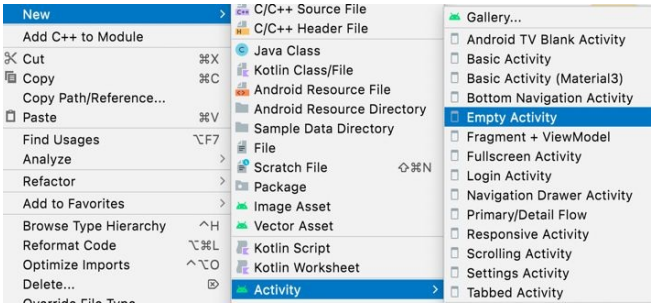


Figure 3.12

Name the activity ‘CreateToDo’ (fig. 3.13).

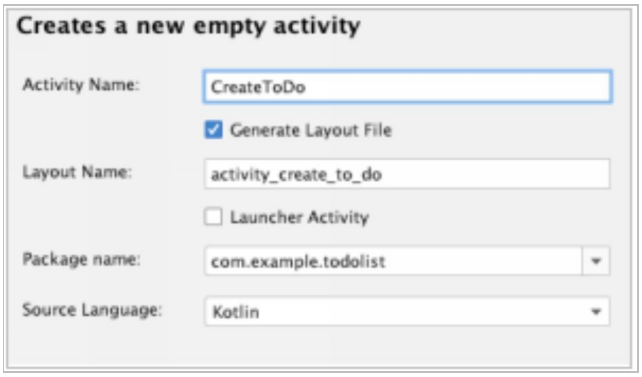


Figure 3.13

res/layout will now have a new *activity_create_to_do.xml*. Let’s now go back to *MainActivity* to create the intent that will link our ‘+’ button to this new activity.

Back in *MainActivity.kt*, there is some default code for us, and there should be a prompt that tells you to import the package “com.google.android.material.snackbar.Snackbar”. Make sure you do for the following to work:

```
…
class MainActivity : AppCompatActivity() {

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

        fab.setOnClickListener { view ->
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show()
        }
    }
```

…

The *setOnClickListener* of *fab* is the code that runs when it is clicked. Let's remove the existing code and replace it with the code to create an intent:

```
fab.setOnClickListener { view ->
    val intent = Intent(this, CreateToDo:: class.java)
    startActivity(intent)
}
```

If you try running your app now and hit the *fab*, it should bring you to a new empty create to-do screen.

Create To-Do Screen

Now let's design our create to-do screen. In *activity_create_to_do.xml*, bring in an *EditText*. Make sure that its *layout_height* is *wrap_content*. Set its up/left/right constraints to be 16 (fig. 3.14).
Set *hint* to be 'To-Do Description' and increase text size to 24sp.

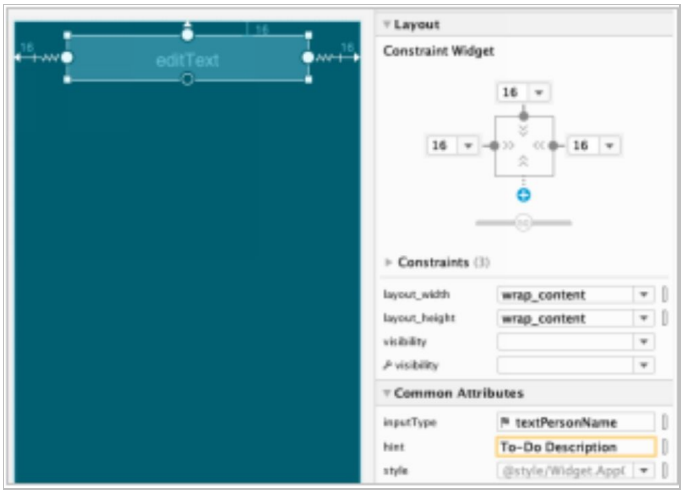


Figure 3.14

Next, drag in a check box, edit its text to 'Important' and change its text size to 24sp. Then link the check box's left constraint to the above edit text's left constraint (fig. 3.15). In that way, whenever you move the EditText, the check box will also follow. Also, link the check box's top constraint to the bottom of the EditText.

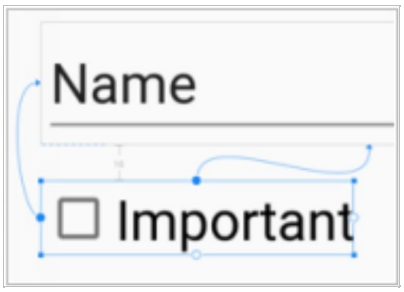


Figure 3.15

Next, drag in a button, edit its text to ‘Add’ and change its text size to 24sp as well. As what we have done with the check box, link the button’s left constraint to the above check box’s left constraint. Now, whenever you move the edit text, both the check box and the button will also follow. Lastly, link the button’s top constraint to the bottom of the check box (fig. 3.15b).

Figure 3.15b

Saving Data Persistently with Shared Preferences

We will now implement saving the To-Do item persistently in our app. That is to say when we close the app or switch off the phone, and when we re-open the app, our to-do data is kept *persistent*.

In *CreateToDo.kt*, add in the following code:

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

    button.setOnClickListener{
        var todoDesc = ""
        if(checkBox.isChecked){
            todoDesc = "!" + editText.text.toString()
        }
        else{
```

```
            todoDesc = editText.text.toString()
        }
    }
}
```

Code Explanation

button.setOnClickListener{…}

In the code, we implement the button's *setOnClickListener* method which gets called when someone taps the *add* button.

```
if(checkBox.isChecked){
    todoDesc = "!" + editText.text.toString()
}
else{
    todoDesc = editText.text.toString()
}
```

If the 'important' check box is checked, we append a '!' to the text of the *editText* and then assign it to the variable *todoDesc*. Else, we straight away assign the *editText*'s text.

In Mac OS Mojave and above, instead of just adding the '!' on the keyboard, you can add a nicer looking emoji. Press 'Control', 'Command' and 'Space' together to bring up the emoji keyboard. Search for 'exclamation' and you should find the exclamation emoji (fig. 3.16).

Figure 3.16

So an important item (e.g. "Get Married") will look like (fig. 3.17):

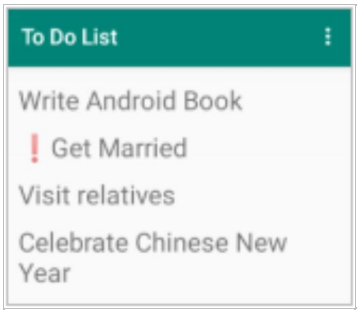


Figure 3.17

SharedPreferences

Next, we apply Shared Preferences to our code. Add the below codes in **bold**:

```
button.setOnClickListener{
    var todoDesc = ""
    if(checkBox.isChecked){
        todoDesc = "!" + editText.text.toString()
    }
    else{
        todoDesc = editText.text.toString()
    }

    var prefs =
            getSharedPreferences("com.example.todolist.sharedprefs",
            Context.MODE_PRIVATE)
    var todos = prefs.getStringSet("todos", setOf()).toMutableSet()
    todos.add(todoDesc)
    prefs.edit().putStringSet("todos",todos).apply()
    finish()
}
```

Code Explanation

Android provides Shared Preferences which are meant to save and access small pieces of data.

```
var prefs =
        getSharedPreferences("com.example.todolist.sharedprefs",
        Context.MODE_PRIVATE)
```

We get a shared preference with *getSharedPreferences*. We supply an *id* "com.example.todolist.sharedprefs" and specify *Context.MODE_PRIVATE* to ensure that these shared preferences won't be accessible by other apps and are private only to the current app.

```
var todos = prefs.getStringSet("todos", setOf()).toMutableSet()
```

Similar to *Intent.getExtra*, data in Shared Preferences are stored in a key-value pairs. We will be storing our *todos* arrayList with the key ‘ todos ’ . You can of course use your own custom key. We retrieve the existing *todos* list (if any) by supplying the key ‘ todos ’ .

If there is nothing there, i.e. we have not added any to-dos, we return an empty set by specifying *setOf()* in the second argument. The second argument is the value to return if the preference does not exist. *toMutableSet* specifies that the returned set can be changed.

```
todos.add(todo3Desc)
```

Having retrieved *todos* from Shared Preferences, we then add the new *to-do* item to the list.

```
prefs.edit().putStringSet("todos",todos).apply()
```

WE then put the edited *todos* back into the Shared Preferences with the same key ‘ todos ’ . Once .*apply*() is executed, the data is stored. We can quit the app, even turn off the device and the data will persist.

```
finish()
```

We then move back to the previous activity with *finish()*.

Displaying Items in our RecyclerView

Now that we have implemented adding to-dos, how do we display them? We will use what we have learned in the previous chapter to display the to-dos.

Ensure that the recycler view that you have dragged into *content_main.xml* has an id *recyclerView*. Then in *MainActivity.kt*, add the following two declarations:

```
…
class MainActivity : AppCompatActivity() {

    lateinit var layoutManager: LinearLayoutManager
    lateinit var adapter: ToDoAdapter
```

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

As mentioned in the previous chapter, to implement a recycler view, it needs two important components. The layout manager and the adapter. We will be implementing the *ToDoAdapter* later on.

Next, implement the *onResume* method as shown:

```
override fun onResume() {
    super.onResume()
    var prefs = getSharedPreferences("com.example.todolist.sharedprefs",
                                     Context.MODE_PRIVATE)
    var todos = prefs.getStringSet("todos", setOf()).toMutableSet()

    layoutManager = LinearLayoutManager(this)
    adapter = ToDoAdapter(todos.toList())


    recyclerView.layoutManager = layoutManager
    recyclerView.adapter = adapter
}
```

Notice that we are not using *onCreate* which gets called only once. When we add new *to-dos* (and removing *to-dos*) and go back to the main screen, we will need to refresh our retrieval of the *todos* array. Hence, we use *onResume* which gets called each time our Activity resumes after being previously paused.
onResume is useful in other instances; for example, we might reload the data from when the app had been interrupted, perhaps by a phone call or the user running another app.

Similar to the add *to-do* method, we retrieve the *todos* from Shared Preferences using:

```
var prefs = getSharedPreferences("com.example.todolist.sharedprefs",
                                 Context.MODE_PRIVATE)
var todos = prefs.getStringSet("todos", setOf()).toMutableSet()
```

We then instantiate the layout manager, adapter and assign them to the recycler view:

```
layoutManager = LinearLayoutManager(this)
adapter = ToDoAdapter(todos.toList())
```

```
    recyclerView.layoutManager = layoutManager
    recyclerView.adapter = adapter
```

Note that Android Studio should prompt you to import:

```
import androidx.recyclerview.widget.LinearLayoutManager
```

Implementing ToDoAdapter

We will now implement *ToDoAdapter*. In your project, create a new Kotlin class and name it *ToDoAdapter*, you should be able to do this on your own by now. The steps to create an adapter should be familiar to you since we have done this in the earlier chapter. But as a challenge, try implementing the adapter on your own, and if you are stuck, just follow along the steps below.

First, we define the class *ToDoAdapter* to receive an argument of *List<String>*. *ToDoAdapter* should inherit from *RecyclerView.Adapter*.

```
class ToDoAdapter(val todos: List<String>):
 RecyclerView.Adapter<ToDoAdapter.ToDoHolder>() {

   class ToDoHolder(v: View) : RecyclerView.ViewHolder(v){

      var view: View = v
      var todoDesc: String = ""

      fun bindToDo(todoDesc: String){
         this.todoDesc = todoDesc
         view.textView.text = todoDesc
      }
   }

      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
          ToDoAdapter.ToDoHolder {
      return ToDoHolder(LayoutInflater.from(parent.context).
                  inflate(R.layout.to_do_row,parent,false))
   }

   override fun getItemCount(): Int {
      return todos.count()
   }

   override fun onBindViewHolder(holder: ToDoAdapter.ToDoHolder, position: Int) {
      val todoDesc = todos[position]
      holder.bindToDo(todoDesc)
   }
```

```
}
```

Code Explanation

```
class ToDoAdapter(val todos: List<String>):
 RecyclerView.Adapter<ToDoAdapter.ToDoHolder>() {
```

First, we define the class *ToDoAdapter* to receive an argument of *List<String>*. *ToDoAdapter* should also inherit from *RecyclerView.Adapter*.

```
   class ToDoHolder(v: View) : RecyclerView.ViewHolder(v){
      var view: View = v
      var todoDesc: String = ""

      fun bindToDo(todoDesc: String){
         this.todoDesc = todoDesc
         view.textView.text = todoDesc
      }
   }
```

Next, we implement the inner class *ToDoHolder* which inherits from *RecyclerView.ViewHolder*. Remember that an inner class gives the outer class the advantage of using its properties and functions, i.e. *ToDoAdapter* has access to properties/functions of *ToDoHolder*.

In *ToDoHolder*, we implement *bindToDo* function which takes in a *String*. We will later display this *String* in the text view of each row.

```
   override fun getItemCount(): Int {
      return todos.count()
   }

   override fun onBindViewHolder(holder: ToDoAdapter.ToDoHolder, position: Int) {
      val todoDesc = todos[position]
      holder.bindToDo(todoDesc)
      }

      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
       ToDoAdapter.ToDoHolder {
      return ToDoHolder(LayoutInflater.from(parent.context).
                  inflate(R.layout.to_do_row,parent,false))
   }
```

We next implement the member methods of *RecyclerView.Adapter*. If you have forgotten the purpose of each function, refer to the previous chapter.

```
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
     ToDoAdapter.ToDoHolder {
    return ToDoHolder(LayoutInflater.from(parent.context).
                inflate(R.layout.to_do_row,parent,false))
}
```

Note particularly in *onCreateViewHolder* where we inflate *R.layout.**to_do_row*** . This means that we will need to create a layout resource file in *res/layout* folder called *to_do_row.xml*. Go ahead and do so now.

Next in *to_do_row.xml*, drag in a text view. Ensure that you select *LinearLayout* and set its *layout_height* to *wrap_content* to avoid the huge height spacing issue in between individual rows.

As what we have done in the previous chapter to avoid our text looking too lumped together, we can differentiate each row better by adding a margin padding to each row. Specify a layout margin of 5 dp. Select the LinearLayout and in its *layout_margin* property, specify 5 dp all around.

Now run your app. Hit the *fab* and you should see your add to-do item screen (fig. 3.18).

Figure 3.18

Try creating a new to-do item and after you hit ‘ Add ’ , you should be able to see your to-do items listed out. If your text appears too small, try changing them to a larger size e.g. 24sp (fig. 3.19)

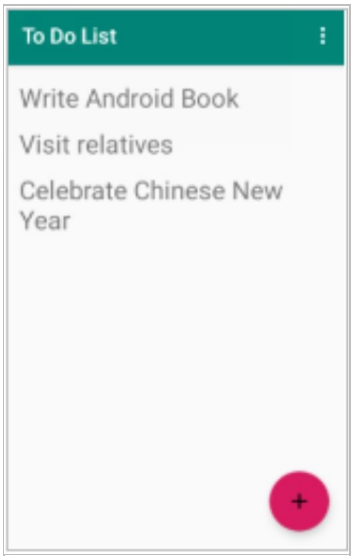


Figure 3.19

Debugging

Now, if for some reason you don ' t see anything, you can try to debug your application by adding log statements. For example, suppose I suspect that my *todos* array is not populated. I can try printing out the array by printing *todos*:

```
override fun onResume() {
   super.onResume()
   var prefs = getSharedPreferences("com.example.todolist.sharedprefs",
         Context.MODE_PRIVATE)
    var todos = prefs.getStringSet("todos", setOf()).toMutableSet()

    Log.d("greg",todos.toString())
         …
}
```

Use *Log.d* to log in debug mode. You have to supply a string to label the log i.e. ' greg ' .

Run the app then by clicking on the debug icon (fig. 3.20):

Figure 3.20

To see the logs, go to ‘ Logcat ’ and search for the string you have supplied. In my case, it is ‘ greg ’ (fig 3.21).

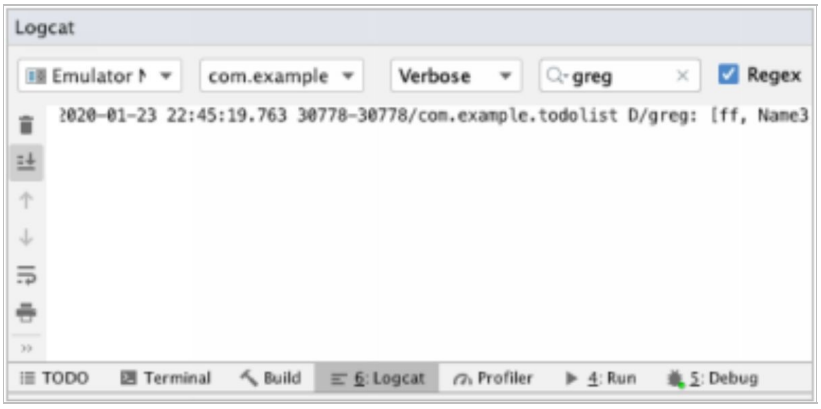


Figure 3.21

If you don ’ t search for the string you supplied, you will realize that there are too many other scrolling logs that clutter out the specific logs you want to see (fig. 3.22).

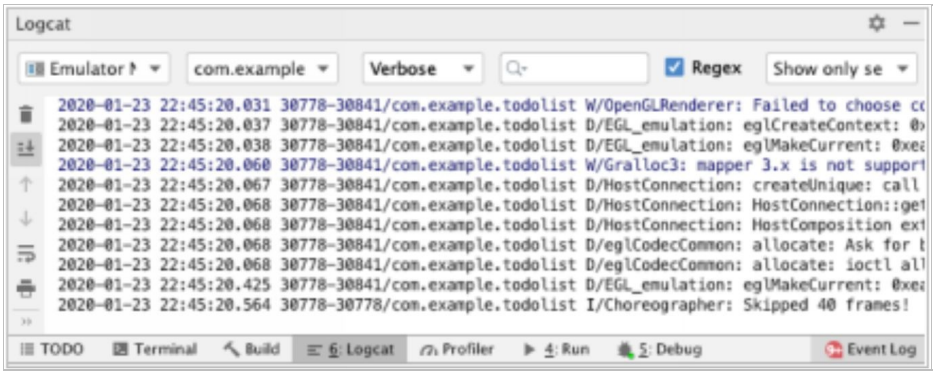


Figure 3.22

The logcat (or sometimes referred to as *console*) allows us to know what goes on with our app. If the app crashes or has errors, the reason or clues will appear here. And if you can ’ t figure why your app is crashing, copying and pasting a bit of the text from logcat into Google will often give you the solution to resolving it.

We would typically be switching between the *Build* window to see that our project has compiled/launched without errors and the Logcat window to view any debugging output or errors from our apps.

In the next section, we will implement marking to-do items completed and removing them from the list.

Marking To-Dos As Complete

We will be re-using the *CreateToDo* activity to serve as our *CompleteToDo* activity. This is because a *CreateToDo* and *CompleteToDo* are very similar; they display the same fields (*title*, *important*) and the only difference is the text in the button, “ Add ” and “ Complete ” . In this way, we avoid creating a whole new Activity that mostly resembles another.

When a user taps on an existing to-do item, we will show it in the *CreateToDo* activity. But this time around, we will change the button text from ‘ Add ’ to ‘ Complete ’ .

Before we do that, we have to add the *onClickListener* to each row in the recycler view to make it clickable. Currently, our recycler view items are view only; they cannot be clicked. To let them be clickable, in the *ToDoHolder* inner class, we have to implement *View.OnClickListener* as shown in **bold**:

```
class ToDoAdapter(val todos: List<String>): RecyclerView.Adapter<ToDoAdapter.ToDoHolder>() {

      class ToDoHolder(v: View) : RecyclerView.ViewHolder(v), View.OnClickListener{

    var view: View = v
    var todoDesc: String = ""

    fun bindToDo(todoDesc: String){
        this.todoDesc = todoDesc
        view.textView.text = todoDesc // add view below
    }

    init {
        v.setOnClickListener(this)
    }

    override fun onClick(p0: View?) {
        val intent = Intent(view.context,CreateToDo:: class.java)
        intent.putExtra("toDoDesc",todoDesc)
        startActivity(view.context,intent,null)
    }
}
…
```

Code Explanation

```
val intent = Intent(view.context,CreateToDo:: class.java)
```

We have to implement the *onClick* method which gets called for each row that is clicked. In it, we create an intent which let's us move from the current activity to the *CreateToDo* activity. We have introduced this in chapter one.

```
intent.putExtra("toDoDesc",todoDesc)
startActivity(view.context,intent,null)
```

We then pass in the to-do description value to be retrieved by the destination activity, i.e. CreateToDo.

At this point, you can run your app to test and see if it goes to *CreateToDo* whenever you tap on an to-do item in the recycler view.

CompleteToDo Activity

In the *CreateToDo* activity, we first check if *intent* has any *Extras*. This is to check if *CreateToDo* has been called by the 'Add' floating action button or called by tapping on an existing to-do item. We do this check by adding the following lines:

```
class CreateToDo : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_create_to_do)
      if(intent.hasExtra("todoDesc")) {
          …
      }
      else {
         button.setOnClickListener {
            var todoDesc = ""
            if (checkBox.isChecked) {
               todoDesc = "!" + editText.text.toString()
            } else {
               todoDesc = editText.text.toString()
            }

            var prefs =
getSharedPreferences("com.example.todolist.sharedprefs", Context.MODE_PRIVATE)
            var todos = prefs.getStringSet("todos", setOf()).toMutableSet()
            todos.add(todoDesc)
```

```
            prefs.edit().putStringSet("todos", todos).apply()
            finish()
        }
    }
  }
}
```

First, we check if the intent *hasExtras* of key “todoDesc”. If it has, we know that it is called from tapping on an existing to-do. Else, we know that it is called from the *CreateToDo*’s *fab* button. Thus, we move the existing code to add a to-do item into the *else* clause.

Next, implement the *if(intent.hasExtra("todoDesc"))* clause as follows:

```
    if(intent.hasExtra("todoDesc")) {
        val todo = intent.extras.getString("todoDesc")
        editText.setText(todo)
        editText.isEnabled = false
        checkBox.visibility = View.INVISIBLE
        button.setText("Complete")
        button.setOnClickListener{
            var prefs =
getSharedPreferences("com.example.todolist.sharedprefs", Context.MODE_PRIVATE)
            var todos = prefs.getStringSet("todos", setOf())
                                                .toMutableSet()
            todos.remove(todo)
            prefs.edit().putStringSet("todos", todos).apply()
            finish()
        }
    }
```

Code Explanation

```
    if(intent.hasExtra("todoDesc")) {
        val todo = intent.extras.getString("todoDesc")
        editText.setText(todo)
        editText.isEnabled = false
        checkBox.visibility = View.INVISIBLE
        button.setText("Complete")
```

The above portion of code does changes to the user interface (fig. 3.23a).

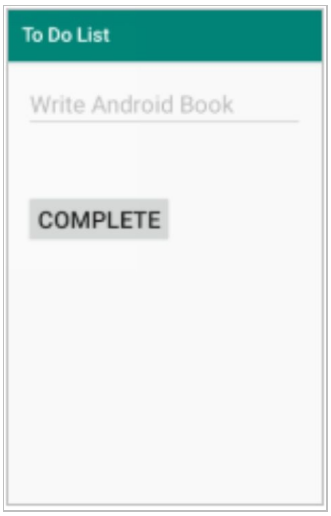


Figure 3.23a

```
val todo = intent.extras.getString("todoDesc")
editText.setText(todo)
```

After getting the existing to-do from *extras*, we set the to-do description to the *text* property of the edit text.

```
editText.isEnabled = false
checkBox.visibility = View.INVISIBLE
```

We disable the *editText* and set the checkbox to be invisible since these values are retrieved from the existing to-do. Suppose if you want to implement editing of the to-do item you will not set these.

```
button.setText("Complete")
```

Lastly, we set button text to ‘Complete’ instead of ‘Add’.

```
button.setOnClickListener{
    var prefs =                 getSharedPreferences("com.example.todolist.sharedprefs",
Context.MODE_PRIVATE)
    var todos = prefs.getStringSet("todos", setOf())
                                        .toMutableSet()
    todos.remove(todo)
    prefs.edit().putStringSet("todos", todos).apply()
    finish()
}
```

The above *button.setOnClickListener* implementation is quite similar to the add button’s. Except that instead of *todos.add(todo)*, we now have *todos.remove(todo)*.

Running your app

Try running your app now. Tap on an existing to-do item and when you tap on complete, you will find that to-do item gone (fig. 3.23b).

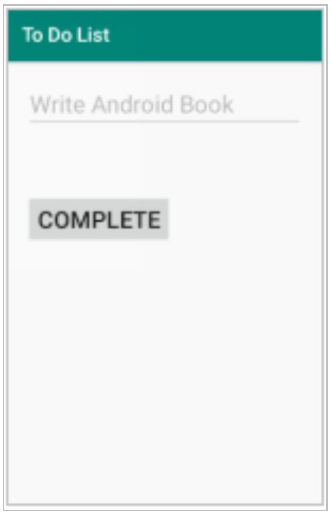


Figure 3.23b

Try it Out

As a challenge, try implementing the editing to-do item on your own. Let me know (support@i-ducate.com) if you encounter problems.

Handling Hard-coded Strings

You might notice that we have a couple of hard-coded strings in our code. For e.g.

```
    override fun onResume() {
        super.onResume()
        var prefs = getSharedPreferences("com.example.todolist.sharedprefs",
Context.MODE_PRIVATE)
        var todos = prefs.getStringSet("todos", setOf()).toMutableSet()
```

And:

```
        override fun onClick(p0: View?) {
            val intent = Intent(view.context,CreateToDo:: class.java)
            intent.putExtra("todoDesc",todoDesc)
            startActivity(view.context,intent,null) // ??? explain
        }
```

You might have even encountered a bug where you are not able to display or store to-do items just because there is a slight typo in the key for e.g. *todoDesc* is not the same as *to**D**oDesc*. When this happens, you might realize that you have unknowingly introduced errors in either displaying or adding to-do items and you are not alerted to them.

In this section, we will see how to make our code more robust and secure. Instead of hard-coding the strings, we will instead store the strings in a common dedicated file where all important strings are stored and accessible in our app. That file is located in *res/values/strings.xml* (fig. 3.24).

Figure 3.24

The file contains by default the strings for *app_name* and *action_settings*:

```
<resources>
    <string name="app_name">To Do List</string>
    <string name="action_settings">Settings</string>
</resources>
```

Let’s add the below:

```
<resources>
    <string name="app_name">To Do List</string>
    <string name="action_settings">Settings</string>
    <string name="shared_pref">com.example.todolist.sharedprefs</string>
    <string name="pref_key">todos</string>
</resources>
```

That is, we define our strings one-time here instead of hardcoding in multiple places in our code. And in *MainActivity.kt*, we replace the hardcoded strings:

```
    override fun onResume() {
        super.onResume()
        var prefs = getSharedPreferences(getString(R.string.shared_pref), Context.MODE_PRIVATE)
        var todos = prefs.getStringSet(getString(R.string.pref_key), setOf()).toMutableSet()
```

Do the same in *CreateToDo.kt*,

```
class CreateToDo : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
          …

      if(intent.hasExtra("todoDesc")) {
          …
         button.setOnClickListener{
            var prefs =
               getSharedPreferences(getString(R.string.shared_pref), Context.MODE_PRIVATE)
            var todos = prefs.getStringSet(getString(R.string.pref_key), setOf()).toMutableSet()
            todos.remove(todo)
            prefs.edit().putStringSet(getString(R.string.pref_key), todos).apply()
            finish()
         }
      }
      else {
         button.setOnClickListener {
          …
            var prefs =
               getSharedPreferences(getString(R.string.shared_pref), Context.MODE_PRIVATE)
            var todos = prefs.getStringSet(getString(R.string.pref_key), setOf()).toMutableSet()
            todos.add(todoDesc)
            prefs.edit().putStringSet(getString(R.string.pref_key), todos).apply()
            finish()
         }
      }
   }
}
```

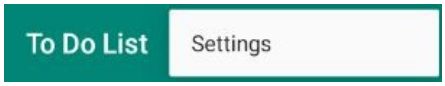
The above code is now less prone to spelling mistakes because if we make a typo error like in fig. 3.25, we get an error. So if you want to have a certain string and ensure that you want to avoid spellings mistakes, place your strings in *strings.xml*. This separates the content of the app from the programming of the app, making it easier to make content changes without impacting programming code.

Figure 3.25

Menu

On the top right of our app, we have a menu button 

.

When you click, it displays the options available: 

We will see how to make use of the menu here and add an option to delete all to-dos.

In *MainActivity.kt*, there is a method *onCreateOptionsMenu*:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
 menuInflater.inflate(R.menu.menu_main, menu)
 return true
}
```

Remember that whenever you see a 'R', it refers to the *res* folder. So if we go to *res/menu/menu_main.xml* and in the 'Text' pane, we can see the menu items defined there.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   tools:context="com.example.todolist.MainActivity">
   <item
      android:id="@+id/action_settings"
      android:orderInCategory="100"
      android:title="@string/action_settings"
      app:showAsAction="never" />
</menu>
```

The *title* string is in turn defined in *res/values/strings.xml* which we introduced earlier,

```
<resources>
   <string name="app_name">To Do List</string>
   <string name="action_settings">Settings</string>
   <string name="shared_pref">com.example.todolist.sharedprefs</string>
   <string name="pref_key">todos</string>
</resources>
```

We will change this to:

```
<resources>
   <string name="app_name">To Do List</string>
```

```
    <string name="action_delete_all">Delete All</string>
    <string name="shared_pref">com.example.todolist.sharedprefs</string>
    <string name="pref_key">todos</string>
</resources>
```

And back in *menu_main.xml*, make the following changes:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.example.todolist.MainActivity">
    <item
        android:id="@+id/action_delete_all"
        android:orderInCategory="100"
        android:title="@string/action_delete_all"
        app:showAsAction="never" />
</menu>
```

Note that other than *title*, we have also changed the id to *action_delete_all* to reflect its functionality better.

To implement the delete all functionality, go back to *MainActivity.kt* and implement *onOptionsItemSelected* as:

```
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        if(item.itemId == R.id.action_delete_all){
            var prefs = getSharedPreferences(getString(R.string.shared_pref),
                    Context.MODE_PRIVATE)
            prefs.edit().putString(getString(R.string.pref_key),null).apply()
            return true
        }
        return super.onOptionsItemSelected(item)
    }
```

Code Explanation

onOptionsItemSelected is called when the user taps on the menu item.

```
            prefs.edit().putString(getString(R.string.pref_key),null).apply()
```

We delete all the to-dos by specifying *null* for the *todos* key-value pair.

Next, we still need to update the recycler view to reflect the deletion. Essentially, we need to call the below code:

```
    var prefs = getSharedPreferences(getString(R.string.shared_pref), Context.MODE_PRIVATE)
    var todos = prefs.getStringSet(getString(R.string.pref_key), setOf()).toMutableSet()

    layoutManager = LinearLayoutManager(this)
    adapter = ToDoAdapter(todos.toList())
    recyclerView.layoutManager = layoutManager
    recyclerView.adapter = adapter
```

But you will notice that this code is already existing in *onResume*. It will thus be better if we put the above code in a method which will then be called by *onResume* and also *onOptionsItemSelected*. So create a new function *updateRecycler* with the above code:

```
fun updateRecycler(){
    var prefs = getSharedPreferences(getString(R.string.shared_pref), Context.MODE_PRIVATE)
    var todos = prefs.getStringSet(getString(R.string.pref_key), setOf()).toMutableSet()

    layoutManager = LinearLayoutManager(this)
    adapter = ToDoAdapter(todos.toList())

    recyclerView.layoutManager = layoutManager
    recyclerView.adapter = adapter
    }
```

And in *onResume*, call *updateRecycler* instead:

```
override fun onResume() {
  super.onResume()
  updateRecycler()
}
```

onOptionsItemSelected will also call *updateRecycler*:

```
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        if(item.itemId == R.id.action_delete_all){
            var prefs = getSharedPreferences(getString(R.string.shared_pref), Context.MODE_PRIVATE)
            prefs.edit().putString(getString(R.string.pref_key),null).apply()
            updateRecycler()
            return true
        }

        return super.onOptionsItemSelected(item)
    }
```

Now run your app, and when you select delete all to-dos from the menu, there shouldn’t be any to-do items left.

Chapter 4: To Do List with Realm

In this chapter, we will improve our To-Do list app by storing the dates of the to-dos when they were created. The date will be displayed below the to-do items (fig. 4.1a).

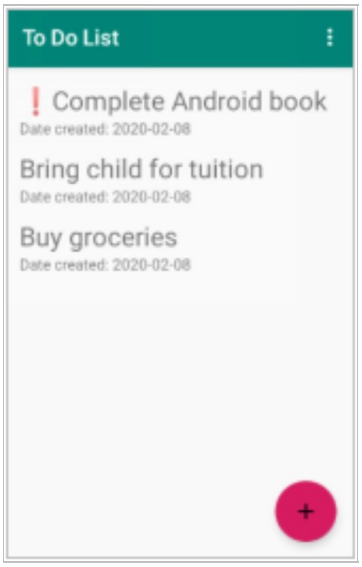


Figure 4.1a

In our previous chapter, we were limited to storing only strings in Shared Preferences. We will explore in this chapter a more comprehensive way of keeping data persistent, store more complex data in the form of objects, and also store more quantity of data.

We will be using a library called Realm to do that. To find out more about Realm, you can go to https://*realm.io* (fig. 4.1b).

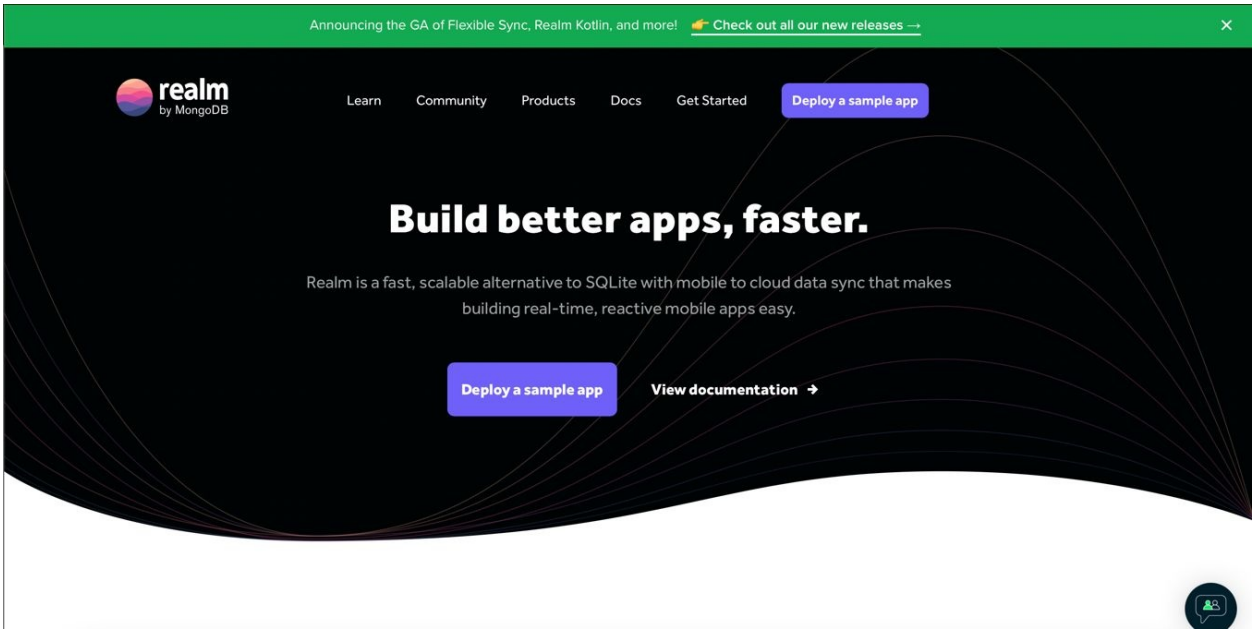


Figure 4.1b

Realm is a mobile database which offers a mobile database solution not only on Android, but also on iOS using the languages: Swift, Java, Javascript and more.

If you search for Android data persistence solutions on the web, you will most likely get results on using SQLite. There is a lot of information on it. Personally, I find that it is much more challenging to work with SQLite where you have to create database tables and use lots of verbose code to connect your Kotlin code to the database. Using Realm however, you can achieve the same effect in a much simpler manner. Realm facilitates an elegant storage solution by allowing us to write classes similar to what we have already done.

Realm also provides us with advantages, like providing notification when data changes, updating data on the fly asynchronously and making them simple to implement. To do the same in SQLite would involve more code, likely introduce more errors and therefore more time spent debugging.

Getting Started

So go to the Realm website and under ‘Get Started’, it will provide you instructions to set up Realm. Realm is installed as a gradle plug-in similar to what we will do with Retrofit. So in the *ToDoList* project from the previous chapter, in *build.gradle (Project: To Do List)*, add in the following:

```
buildscript {
        …
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.3'
        classpath "org.jetbrains.Kotlin:Kotlin-gradle-plugin:$Kotlin_version"
        classpath "io.realm:realm-gradle-plugin:6.0.2"
…
```

Next, in *build.gradle (Module: app)*, add:

```
apply plugin: 'com.android.application'
apply plugin: 'Kotlin-kapt'
apply plugin: 'Kotlin-android'
apply plugin: 'Kotlin-android-extensions'
apply plugin: 'realm-android'
…
```

*Kotlin-kapt is the Kotlin Annotation processing tool.

As you make changes, Android Studio will prompt you to ‘Sync Now’. So go ahead and Android Studio will install the updates. When the installation is done, we are ready to use Realm!

Defining ToDo RealmObject Class

First, we will create a class to present each to-do instance. Create a new Kotlin class file and name it *ToDo.kt*. Fill it with the following code:

```
package com.example.todolist

import io.realm.RealmObject
import io.realm.annotations.PrimaryKey
import java.util.*

open class ToDo : RealmObject(){
    @PrimaryKey
    private var id = UUID.randomUUID().toString()
    var name = ""
    var important = false
    var date = ""

    fun getId(): String{
        return id
```

```
    }
}
```

We have defined a *ToDo* class which is just like any other class, except that it extends the *RealmObject* class. We will later explain the need for a primary key in *id*. But for now, note that we have three fields for a to-do item: *name*, *important* and the date the to-do was created. We already illustrated the *name* and *important* fields in the previous chapter when we first developed our ToDo app. Here, we have a new *date* field which stores the date the to-do was created. This is also to illustrate that Realm allows us to store more complex objects as opposed to just a *String* variable using Shared Preferences.

Saving ToDo RealmObject Class

In this section, we will instantiate a *ToDo* object and save it in Realm. In *CreateToDo*, add the codes below in **bold**:

```
class CreateToDo : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
            …

        if(intent.hasExtra("todoId")) {
            …
        }
        else {
            button.setOnClickListener {
                var todoDesc = ""
                if (checkBox.isChecked) {
                    todoDesc = "      " + editText.text.toString()
                } else {
                    todoDesc = editText.text.toString()
                }

                var todo = ToDo()
                todo.name = todoDesc
                todo.important = checkBox.isChecked
                val current = LocalDateTime.now()
                val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd")
                val formatted = current.format(formatter)
                todo.date = formatted

                val realm = Realm.getDefaultInstance()
                realm.beginTransaction()
                realm.copyToRealm(todo)
                realm.commitTransaction()
```

```
            finish()
        }
    }
```

Code Explanation

```
        button.setOnClickListener {
            var todoDesc = ""
            if (checkBox.isChecked) {
                todoDesc = "      " + editText.text.toString()
            } else {
                todoDesc = editText.text.toString()
            }
        …
```

The existing code on adding an exclamation mark if the *important* checkbox is checked remains unchanged from the previous chapter.

Do make sure to remove the *SharedPreferences* code that we have done earlier, i.e.:

```
var prefs = getSharedPreferences(getString(R.string.shared_pref),
                                 Context.MODE_PRIVATE)
var todos = prefs.getStringSet(getString(R.string.pref_key),
                                 setOf()).toMutableSet()
todos.add(todoDesc)
prefs.edit().putStringSet(getString(R.string.pref_key), todos).apply()
```

Remove the above lines of code.

```
            var todo = ToDo()
```

Instead, we instantiate a *todo* object from the *ToDo* class we defined earlier.

```
            todo.name = todoDesc
            todo.important = checkBox.isChecked
            val current = LocalDateTime.now()
            val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd")
            val formatted = current.format(formatter)
            todo.date = formatted
```

We assign its *name*, *important* and also its *date* attribute with a formatted date. Note that the use of the formatting functions require at least a min API of 26. So

make the change to *minSdkVersion* in the *build.gradle* file:

```
  defaultConfig {
    applicationId "com.example.todolist"
    minSdkVersion 26
    targetSdkVersion 28
        …
```

To save something in Realm, we have to first get an instance of a Realm object. We do it with:

```
        val realm = Realm.getDefaultInstance()
```

Next, we have the three lines of code:

```
        realm.beginTransaction()
        realm.copyToRealm(todo)
        realm.commitTransaction()
```

The storing of our *todo* object has to be done in between *realm.beginTransaction* and *realm.commitTransaction*.

```
        finish()
```

Lastly, we have *finish()* which brings us back to the previous activity.

Setting Up Realm in MainActivity

Before we proceed to run and test what we have written, we have to add two lines of code in *MainActivity* to initialize Realm:

```
…
import io.realm.Realm

class MainActivity : AppCompatActivity() {
        …
  override fun onCreate(savedInstanceState: Bundle?) {
        …
    fab.setOnClickListener { view ->
        …
    }

    Realm.init(this)
…
```

Running and Verifying

Try running your app and create a to-do item. Now how do we verify that we have indeed successfully created our to-do? We can query our database and log the results with the following code in *CreateToDo*:

```
val realm = Realm.getDefaultInstance()
realm.beginTransaction()
realm.copyToRealm(todo)
realm.commitTransaction()


val query = realm.where(ToDo::class.java)
val results = query.findAll()
for(todo in results){
   Log.d("todo",todo.name)
}
finish()
```

Code Explanation

```
val query = realm.where(ToDo::class.java)
val results = query.findAll()
for(todo in results){
   Log.d("todo",todo.name)
}
```

We query all *ToDo* objects with the above code and then use the familiar *for*-loop to print out the results. When you run your app and create a to-do item, you should see the to-do item names logged in the logcat. This also means that we have successfully stored our objects in Realm.

Displaying To-Do items

Next, we have to make some changes to our *ToDoAdapter* and *MainActivity* to display our to-do items. The only change we have to make in *MainActivity* is in *updateRecycler*. Add in the below codes in **bold** and make sure that you remove the commented-out codes.

```
fun updateRecycler(){
   val realm = Realm.getDefaultInstance()
```

```
        val query = realm.where(ToDo::class.java)
        val todos = query.findAll()

        /*
        var prefs = getSharedPreferences(getString(R.string.shared_pref), Context.MODE_PRIVATE)
        var todos = prefs.getStringSet(getString(R.string.pref_key), setOf()).toMutableSet()
         */

        layoutManager = LinearLayoutManager(this)
        adapter = ToDoAdapter(todos.toList())

        recyclerView.layoutManager = layoutManager
        recyclerView.adapter = adapter
    }
```

Code Explanation

Previously, we got our *todos* from Shared Preferences. Now, we get *todos* from querying Realm. As you might note, the code is almost the same as the code we used to query and log our to-do objects. We simply query all *ToDo* objects. That's all for *MainActivity*.

Next, we have to make changes in displaying our *ToDo* items. First, we add a TextView to show the created date of the to-do item. We display this date text view below the to-do description TextView. So in *to_do_row.xml*, drag in another text view as shown (fig. 4.2):

Figure 4.2

Set its id to *dateTextView*. Ensure that its *layout_width* is 'match_parent', *layout_height* is 'wrap_content' and we set its text size to a smaller 14sp. When we have done so, let's go on to *ToDoAdapter.kt*

ToDoAdapter.kt and *ToDoHolder*

class ToDoAdapter(val todos: List<**ToDo**>): …

First, in the class definition, we have to change the object type in our *List* to *ToDo*.

Next in class *ToDoHolder*, make the following changes:

```
class ToDoHolder(v: View) : RecyclerView.ViewHolder(v), View.OnClickListener{
    var view: View = v
    var todoId: String = ""

    fun bindToDo(todo: ToDo){
        this.todoId = todo.getId()
        view.textView.text = todo.name
        view.dateTextView.text = "Date created: ${todo.date}"
    }
        …
}
```

Code Explanation

The above code changes is similar to what we have done earlier. Except now, we declare a *todoId*. We will explain its importance more in detail later. For now, the main change is in *bindToDo*, where we assign the *todo* name to the TextView we already have. In addition, we pass in the to-do date to *dateTextView*.

Can't pass Realm Objects through Intents

Remember that previously we could pass in strings across an intent? i.e.:

```
intent.putExtra("todoDesc",todoDesc)
```

Now, Realm DOES NOT allow us to pass in RealmObjects across intents. For e.g.:

```
    intent.putExtra("todo",todo)
```

To get around this, instead of passing the entire object, we just pass in the to-do id, and in the destination activity, we use this id to retrieve the object. That is the main reason why we had to define a *todoId* in *ToDoHolder* and also a primary key *id* in our *ToDo* class:

```
…
open class ToDo : RealmObject(){
```

```
    @PrimaryKey
    private var id = UUID.randomUUID().toString()
    var name = ""
        var important = false
…
```

So in *onClick* of *ToDoHolder*, make the following changes:

```
        override fun onClick(p0: View?) {
          val intent = Intent(view.context,CreateToDo:: class.java)
          intent.putExtra("todoId",todoId)
          startActivity(view.context,intent,null)
        }
```

We pass in *todoId* with key “todoId”. In the next section, we will show how we can retrieve the ToDo Realm object, display it and allow a user to mark it complete.

Displaying a to-do Item Detail

We will retrieve the *todoId* passed through the intent and retrieve from the Realm database the *ToDo* object whose id equals to the id we supplied. We then set the values for the TextView and check box in the layout.

In *CreateToDo*, replace the existing code with the below code:

```
class CreateToDo : AppCompatActivity() {

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

        if(intent.hasExtra("todoId")) {
            val todoId = intent.extras.getString("todoId")

            val realm = Realm.getDefaultInstance()
            val todo = realm.where(ToDo::class.java).equalTo("id",todoId)
                                        .findFirst()

            editText.setText(todo!!.name)
            editText.isEnabled = false
            checkBox.isChecked = todo!!.important
            checkBox.isEnabled = false
             …
```

Code Explanation

```
if(intent.hasExtra("todoId")) {
   val todoId = intent.extras.getString("todoId")
```

First, we check if there is a *todoId* being passed through the intent. If there is, we retrieve it. This is similar to what we have done before.

```
   val realm = Realm.getDefaultInstance()
   val todo = realm.where(ToDo::class.java).equalTo("id",todoId)
                                   .findFirst()
```

Next, we retrieve from the Realm database *ToDo* objects whose id equals the id we supplied. We use *findFirst()* to return us the first one that it finds. And since *id* is a primary key, it will be the only one.

```
   editText.setText(todo!!.name)
   editText.isEnabled = false
   checkBox.isChecked = todo!!.important
   checkBox.isEnabled = false
```

Having retrieved our *todo* object, we then set the values for the text view and check box in the layout. We use two exclamation marks to ensure that todo definitely will not be null.

Removing ToDo from Realm

Next, we proceed to remove *todo* from Realm when a user hits the ‘Complete’ button. Remove the commented out code where we remove our *todo* from Shared Preferences and instead replace it with the code in **bold** below:

```
if(intent.hasExtra("todoId")) {
            …

   button.setText("Complete")
   button.setOnClickListener{
      realm.beginTransaction()
      todo!!.deleteFromRealm()
      realm.commitTransaction()
      finish()
   }
}
```

Code Explanation

```
realm.beginTransaction()
todo!!.deleteFromRealm()
realm.commitTransaction()
```

Similar to when we store a Realm object, we have to also enclose the deletion of a Realm object in between a Realm *beginTransaction* and *commitTransaction*. In between, we call the *deleteFromRealm* method of the Realm object.

Running your app

When you run your app now, try creating new to-dos and you will see the date displayed below each to-do item (fig. 5.3).

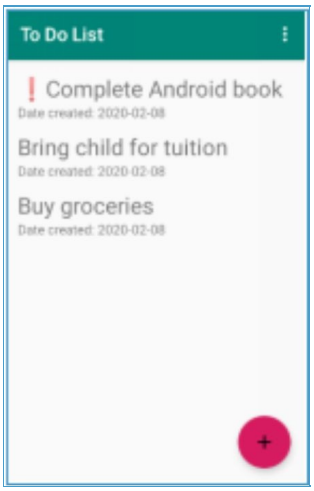


Figure 5.3

Try ‘completing’ a to-do and it will be removed.

Summary

In this chapter, we improved on our To-Do list from the last chapter by letting users include images in their to-do items. In the next chapter, we will learn how to connect our app with an API to retrieve data.

Chapter 5: Connecting to an API: Cryptocurrency Price Tracker App

In this chapter, we will learn how to connect our app with the Internet. We will be connecting to an Application Programming Interface (API) to get the price of different cryptocurrencies to display to the user. Our app will contain a spinner to let the user pick from a range of cryptocurrencies: BTC, ETH, XRP and BCH and the price retrieved will be displayed in various real-world currencies: "USD", "EUR", "JPY" in a nicely formatted fashion (fig. 5.0).

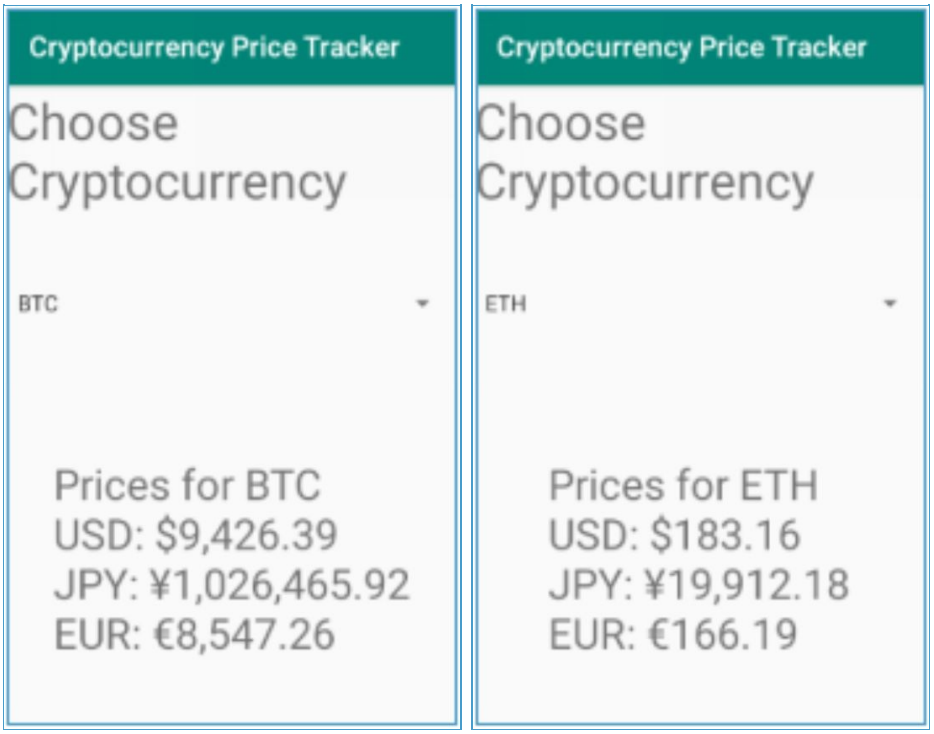


Figure 5.0

Start a new ‘Empty Activity’ Android project and name it ‘Cryptocurrency Price Tracker’. In *minSdkVersion*, set it to be 16.

Proceed to drag and drop a text view on the top (text: “Choose Cryptocurrency”, fig. 5.0a). Set its left, right and top constraints to zero. Change text size to 36sp.

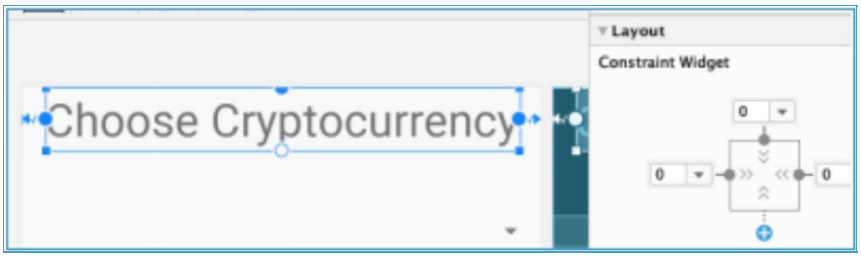


Figure 5.0a

We will use a spinner to populate it with the various crypto-currency options. The *Spinner* is a UI element used to make a selection from multiple choices. So drag a spinner, set left/right constraints to zero and link its top constraint to TextView. Ensure that its *layout_height* is set to *wrap_parent* and *layout_width* to *match_constraints* so that we avoid any unnecessary huge spacing issue.

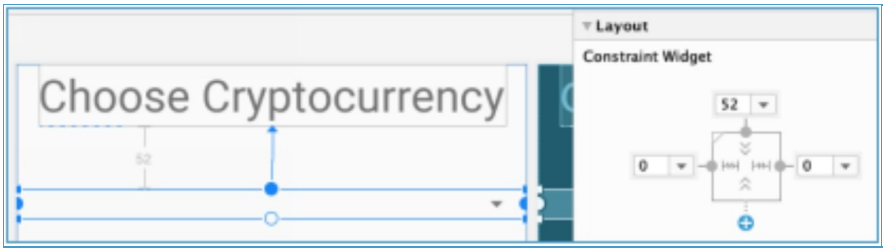


Figure 5.0b

Populating the Spinner

To populate the spinner, we have to specify the spinner item options in *strings.xml*:

```
<resources>
    <string name="app_name">Cryptocurrency Price Tracker</string>
    <string-array name="crypto_currencies">
        <item>BTC</item>
        <item>ETH</item>
        <item>XRP</item>
        <item>BCH</item>
    </string-array>
</resources>
```

We define a string-array *crypto_currencies* with the options BTC, ETH, XRP and BCH.

Then in *MainActivity.kt*, add the following code:

```
…
import android.widget.AdapterView
import android.widget.ArrayAdapter
import Kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

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

        // Create an ArrayAdapter using the string array and a default
        // spinner layout
        ArrayAdapter.createFromResource(
            this,
            R.array.crypto_currencies,
            android.R.layout.simple_spinner_item
              ).also { adapter ->
            // Specify the layout to use when the list of choices appears
                    adapter.setDropDownViewResource(
                            android.R.layout.simple_spinner_dropdown_item)
            // Apply the adapter to the spinner
            spinner.adapter = adapter
        }
    }
}
```

Code Explanation

```
        ArrayAdapter.createFromResource(
            this,
            R.array.crypto_currencies,
            android.R.layout.simple_spinner_item
        )
```

The above creates an *ArrayAdapter* from the *crypto_currencies* string array (2nd argument). The third argument receives a layout resource that defines how the selected choice appears in the spinner (fig. 5.0c).

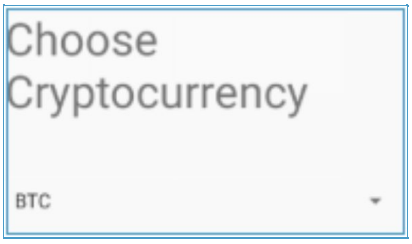


Figure 5.0c

R.layout.simple_spinner_item is the default layout that Android provides.

```
            ).also { adapter ->
        // Specify the layout to use when the list of choices appears
                    adapter.setDropDownViewResource(
                                android.R.layout.simple_spinner_dropdown_item)
        // Apply the adapter to the spinner
        spinner.adapter = adapter
    }
```

We then call *setDropDownViewResource* to specify the layout the adapter uses to display the list of spinner choices (fig 5.0d).

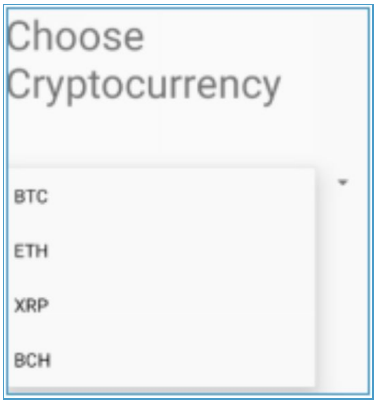


Figure 5.0d

R.layout.simple_spinner_dropdown_item is another standard layout provided by Android.

Selection Using Spinner

If you run your app now, you will see the options populated in your picker. But how do we select what the user has picked?

To detect what the user has selected in our picker, we have to implement the *onItemSelectedListener* of the spinner. When the user selects an item from the spinner, it receives an *onItemSelected* event. Thus, let's define the *onItemSelectedListener* from *MainActivity*:

```
class MainActivity : AppCompatActivity(), AdapterView.OnItemSelectedListener {
```

We then have to implement the *onItemSelected* and *onNothingSelected* methods:

```
    override fun onItemSelected(parent: AdapterView<*>, view: View, pos: Int, id: Long) {
```

```
    // An item was selected. You can retrieve the selected item using
    Log.d("greg",parent.getItemAtPosition(pos).toString())
}

override fun onNothingSelected(parent: AdapterView<*>) {
}
```

We then assign *this* to the *spinner.onItemSelectedListener* because our spinner is in this current view.

```
ArrayAdapter.createFromResource(
    this,
    R.array.crypto_currencies,
    android.R.layout.simple_spinner_item
).also { adapter ->
                // Specify the layout to use when the list of choices appears
                adapter.setDropDownViewResource(
                        android.R.layout.simple_spinner_dropdown_item)
    // Apply the adapter to the spinner
    spinner.adapter = adapter
    spinner.onItemSelectedListener = this
}
```

When you run your app now, you should see your spinner appear. And when you tap on it, the options appear (fig. 5.2).

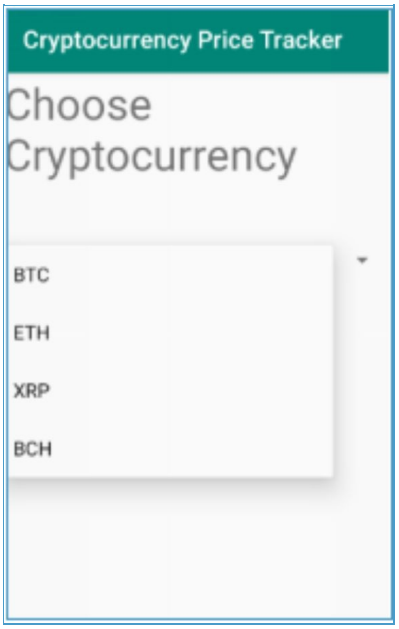


Figure 5.2

And when you select an item, it should be logged in Logcat. Since we have

specified in *onItemSelected*,

```
override fun onItemSelected(parent: AdapterView<*>, view: View, pos: Int, id: Long) {
    // An item was selected. You can retrieve the selected item using
    Log.d("greg",parent.getItemAtPosition(pos).toString())
}
```

You can find out more information regarding the usage of spinners at *https://developer.android.com/guide/topics/ui/controls/spinner*.

Retrieving Data from an API

Now how do we get the cryptocurrency prices? We use *https://min-api.cryptocompare.com/*, a free API for getting cryptocurrency live pricing data (fig. 5.3a).

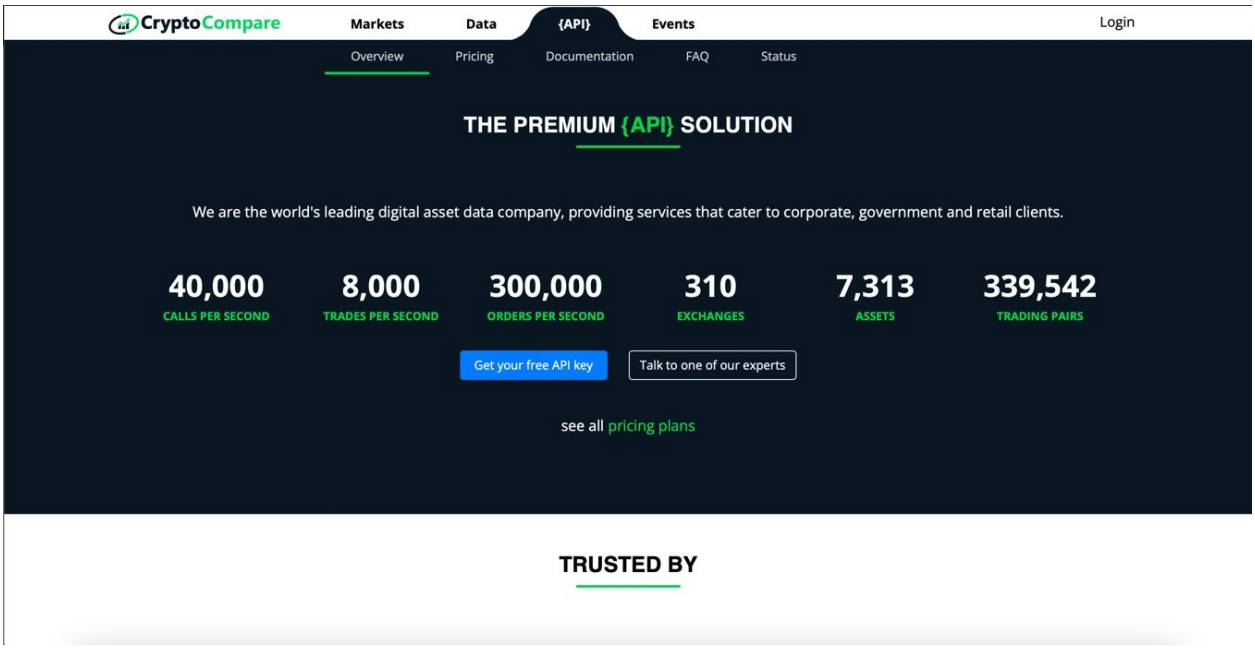


Figure 5.3a

There are APIs for all kinds of data, e.g. weather information, Twitter tweets, Facebook posts. An API could be seen as a way for computers to talk to computers.

The default sample API provided in the site is: *https://min-api.cryptocompare.com/data/price?fsym=BTC&tsyms=USD,JPY,EUR*

which returns the result:

{"USD":9319.45,"JPY":1014307.55,"EUR":8449.23}

That is, get the price of Bitcoin in USD, JPY and EUR. We can retrieve the price of Ethereum by specifying *fsym=ETH* instead of *fsym=BTC*:

https://min-api.cryptocompare.com/data/price?fsym=ETH&tsyms=USD,JPY,EUR

RetroFit

Now how do we process the data retrieved? The API retrieves our data in a format called JSON, which stands for JavaScript Object Notation (pronounced “Jason”). JSON is an easy way to represent data that facilitates data transfer between computers, in contrast to other data formats such as XML (Extensible Markup Language). It’s a standard format which many APIs use.

In Kotlin, we will be using RetroFit to download and parse JSON data. RetroFit is created by Square, the guys who created the payment processing company (fig. 5.3b).

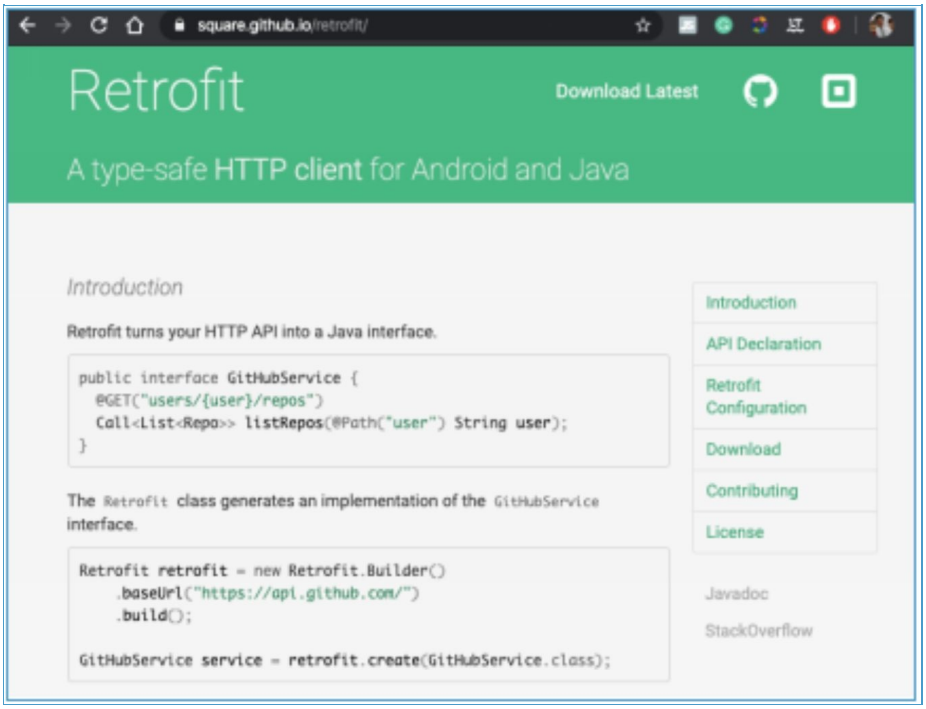
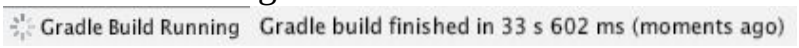


Figure 5.3b

We will next show the steps required to include RetroFit in our project. To do so, we have to touch on *Gradle*. What's Gradle? When you are building or running your project, you might notice near the bottom of Android Studio where it says, "Gradle build running/finished in…"
e.g. 

Gradle is a build automation system. When we run our app on the emulator, Gradle combines all our code/XML files into an app bundle and sends it to the emulator which runs it. Gradle does all these in the background so that we don't have to mess with it.

We can also use Gradle to download code other people have written e.g. RetroFit and include it into our project.

If you look under your left pane, *Gradle Scripts*, *build.gradle (Module:app)*, you can see the dependencies of our app, e.g.:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.Kotlin:Kotlin-stdlib-jdk7:$Kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
}
```

We will now add in the RetroFit dependency. Go to https://square.github.io/retrofit/ (fig. 5.3) or just search 'retrofit' on a search engine of your choice.

Figure 5.3c

Click or scroll down to the ‘Download’ section, where you will see the portion on Gradle (fig. 5.4).

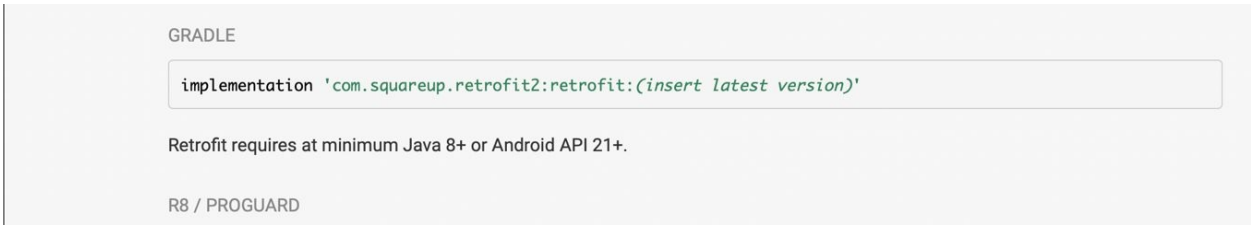


Figure 5.4

Copy the line of code; we will be using 2.3.0 in this book i.e.

implementation 'com.squareup.retrofit2:retrofit:**2.9.0**'

And add it to the *build.gradle* file shown in **bold**:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.Kotlin:Kotlin-stdlib-jdk7:$Kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
}
```

Once again, we will get a warning saying something like “gradle files have changed since last project sync…”. Hit on ‘Sync Now’.

When you do so, Gradle will then behind the scenes go to the RetroFit site, download and integrate Retrofit into our project. We will then be able to start importing RetroFit libraries with the *import* keyword in your code i.e.

import retrofit2…

Using RetroFit

RetroFit has its own documentation website which takes you through how to use it to get data and use it in your app (fig. 5.5).

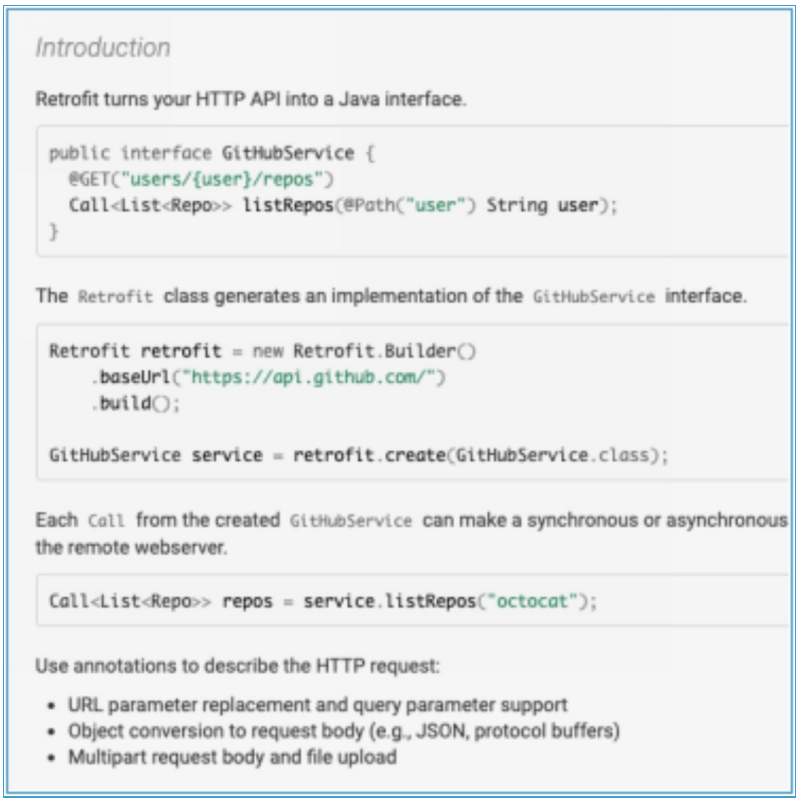


Figure 5.5

But it can be a bit complicated especially for beginners, so I will take you through it. We explore a more straightforward JSON structure with the crypto-currency API in this chapter. The next chapter will explore a slightly more complicated JSON structure. This will put you in good stead later on when you

encounter different and more complicated JSON structures to know how to refer back to this site since you already know the fundamentals.

interface

To integrate RetroFit in our project, create a new Kotlin 'File/Class' to hold our code to work with the currency API and name it *CurrencyAPI*. In it, create an interface *CurrencyAPI* with the following codes:

```
package com.example.cryptocurrencypricetracker
import retrofit2.Call
import retrofit2.http.GET

interface CurrencyAPI{
   @GET("data/price?fsym=BTC&tsyms=USD,JPY,EUR")


   fun getCurrencyPrices(): Call<Price>
}
```

Code Explanation

```
interface CurrencyAPI
```

In the above, we declare an interface called *CurrencyAPI* with the method *getCurrencyPrices*. But we don't actually implement it. That's the purpose of an interface; just declaring the method header but without the implementation.

```
   @GET("data/price?fsym=BTC&tsyms=USD,JPY,EUR")
```

Next, we have the *@GET* call. For this, we need to import *retrofit2.http.GET*. Be reminded that the full URL for the API to retrieve the currency data is:

https://min-api.cryptocompare.com/data/price?fsym=BTC&tsyms=USD,JPY,EUR

If you enter the full URL in a browser, it returns the data something like:

```
{"USD":9362.47,"JPY":1018734.62,"EUR":8484.77}
```

In *@GET*, we provide the ending part of the URL that is, *data/price?fsym=BTC&tsyms=USD,JPY,EUR*. We will specify the base i.e. *https://min-*

api.cryptocompare.com/ in another location.

```
    fun getCurrencyPrices(): Call<Price>
```

We then declare the function *getCurrencyPrices* without its implementation. We specify that *getCurrencyPrices* return the RetroFit *Call* object (import *retrofit2.Call*). Since our data is a ˆobject, we provide the class <*Price*>. You can give this any name you want, e.g. *CryptoPrice*. But it should describe your data that comes from the JSON API.

Price

We next declare the class *Price* as shown in **bold** below:

```
…
interface CurrencyAPI{
    @GET("data/price?fsym=BTC&tsyms=USD,JPY,EUR")

    fun getCurrencyPrices(): Call<Price>
}

class Price(val USD: String, val JPY: String, val EUR: String)
```

It is important to note that the argument names (USD, JPY, EUR) need to have the exact same name and capitalisation as the variables provided in the API i.e., {"USD":9362.47,"JPY":1018734.62,"EUR":8484.77}

Suppose we are requesting from a stock price API which gives us the data {“high” : 99, “low” : 78} , we would declare the class *Price* as:

```
class Price(val high: String, val low: String)
```

PriceRetriever

We next create a *PriceRetriever* class. You can just include it under the class *Price*:

```
package com.example.cryptocurrencypricetracker
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import retrofit2.http.GET
```

```
interface CurrencyAPI{
        …
}

class Price(val USD: String, val JPY: String, val EUR: String)

class PriceRetriever{
   val service: CurrencyAPI

   init{
      val retrofit = Retrofit
         .Builder()
         .baseUrl("https://min-api.cryptocompare.com/")
         .addConverterFactory(GsonConverterFactory.create()).build()
      service = retrofit.create(CurrencyAPI::class.java)
   }

   fun getPrice(callback: Callback<Price>){
      val call = service.getCurrencyPrices()
      call.enqueue(callback)
   }
}
```

Code Explanation

The code in *init()* essentially converts our JSON data into a Kotlin object,

```
val retrofit = Retrofit
   .Builder()
   .baseUrl("https://min-api.cryptocompare.com/")
```

Here is where we provide the *baseUrl* *https://min-api.cryptocompare.com/*. The latter portion of the URL has been previously specified in interface *CurrencyAPI* in @GET("data/price?fsym=BTC&tsyms=USD,JPY,EUR")

```
.addConverterFactory(GsonConverterFactory.create()).build()
```

GsonConverter converts our JSON data into a Kotlin object. Gson is Google's popular JSON converter (thus the name *Gson*) to convert JSON into Java/Kotlin objects and back.

```
service = retrofit.create(CurrencyAPI::class.java)
```

Now, ensure that you have included in your dependencies *retrofit* and *converter-gson*. In this book, we will be using the 2.9.0 versions:

```
dependencies {
        …
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
    testImplementation 'junit:junit:4.12'
            …
}
```

Next, we go through *getPrice()*.

```
    fun getPrice(callback: Callback<Price>){
      val call = service.getCurrencyPrices()
      call.enqueue(callback)
    }
```

getPrice receives an argument of *Callback* type. *callback* is a function we will define in the next section. *callback* will run asynchronously when *call.enqueue* finishes. We need this because we do not want our app to freeze whenever we request currency prices via the API. We want the app to function normally while the request to obtain the currency data runs in the background. This gives the user a better user experience that the app is still reacting rather than froze up. Asynchronous function calls is common especially in web development where we want websites to feel responsive and background API calls continue to run while the user surfs the website.

MainActivity.kt

To test if our connection is right, we call *PriceRetriever* in *MainActivity*'s *onCreate* with:

```
…
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response

class MainActivity : AppCompatActivity() {

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

    override fun onItemSelected(parent: AdapterView<*>, view: View, pos: Int, id: Long) {
            retrievePrices()
    }
```

```
    fun retrievePrices(){
        var retriever = PriceRetriever()

        val callback = object: Callback<Price> {
            override fun onFailure(call: Call<Price>, t: Throwable) {
                Log.d("jason",t.toString())
            }
            override fun onResponse(call: Call<Price>, response: Response<Price>) {
                Log.d("jason",response!!.body()!!.USD)
            }
        }
        retriever.getPrice(callback)
    }
}
```

Code Explanation

```
    fun retrievePrices(){
        var retriever = PriceRetriever()
```

We define a function *retrievePrices()* and in it, create an instance of *PriceRetriever*

```
        val callback = object: Callback<Price> {
            override fun onFailure(call: Call<Price>, t: Throwable) {
                Log.d("greg",t.toString())
            }
            override fun onResponse(call: Call<Price>, response:
                                    Response<Price>) {
                Log.d("greg",response!!.body()!!.USD)
            }
        }
```

We then create a *Callback* object. We have to implement the *onFailure* and *onResponse* methods of the *Callback* object. *onFailure* as its name suggests, will be called when the request fails, and if the request is successful, *onResponse* will be called.

In *onResponse*, the *Price* object which contains the data from the API is returned. We can access it with:

```
            Log.d("greg",response!!.body()!!.USD)
```

We then test our connection by calling *retrievePrices*() from *onItemSelected*(),

that is, each time we select an item from the spinner:

```
override fun onItemSelected(parent: AdapterView<*>, view: View, pos: Int, id: Long) {
    retrievePrices()
}
```

Because our app gets information from the Internet, we have to get permission from the user. We do that by specifying the below in Android Manifest:

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

Running your app

When you run your app, it should log the USD price when we select an item. And that means we have successfully connected to the API!

Displaying Retrieved Prices

Now, instead of logging just the USD price in the console, we should be displaying the different prices in a TextView in our app.

Thus, let's add a TextView to the bottom of the spinner. Set its constraints and give it the id *pricesTextView*.

We then populate *pricesTextView.text* in *onResponse* with the following code:

```
fun retrievePrices(){
    var retriever = PriceRetriever()

    val callback = object: Callback<Price> {
        override fun onFailure(call: Call<Price>, t: Throwable) {
            Log.d("jason",t.toString())
        }

        override fun onResponse(call: Call<Price>, response:
                                Response<Price>) {
            pricesTextView.text = "USD: ${response!!.body()!!.USD}" +
                    "\nJPY: ${response!!.body()!!.JPY}" +
                    "\nEUR: ${response!!.body()!!.EUR}"
        }
    }
    retriever.getPrice(callback)
}
```

If you run your app now, our prices finally get displayed (fig. 5.6)!

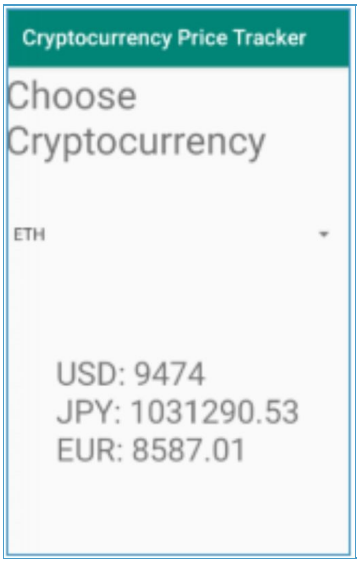


Figure 5.6

However, it would be better if we format our prices with the correct currency symbol, e.g. USD with $, JPY with ¥ (pronounced “yen”). To do so, we use the *NumberFormat* class.

Formatting Currencies

We use the *NumberFormat* to format our currencies with the following code:

```
import java.text.NumberFormat
import java.util.*

…

override fun onResponse(call: Call<Price>, response: Response<Price>) {

        val formatUSD = NumberFormat.getCurrencyInstance()
        formatUSD.setMaximumFractionDigits(2)
        formatUSD.setCurrency(Currency.getInstance("USD"))

        val formatJPY = NumberFormat.getCurrencyInstance()
        formatJPY.setMaximumFractionDigits(2)
        formatJPY.setCurrency(Currency.getInstance("JPY"))

        val formatEUR = NumberFormat.getCurrencyInstance()
        formatEUR.setMaximumFractionDigits(2)
        formatEUR.setCurrency(Currency.getInstance("EUR"))

         pricesTextView.text =
```

```
            "USD: ${formatUSD.format(response!!.body()!!.USD.toDouble())}" +
            "\nJPY: ${formatJPY.format(response!!.body()!!.JPY.toDouble())}" +
            "\nEUR: ${formatEUR.format(response!!.body()!!.EUR.toDouble())}"
}
```

Using the *NumberFormat* object, we specify the currency code (e.g. USD, JPY, EUR) with *setCurrency*, specify fraction digits to 2 and finally use the *format* function to get our formatted price, i.e. €185.47 (fig. 5.7).

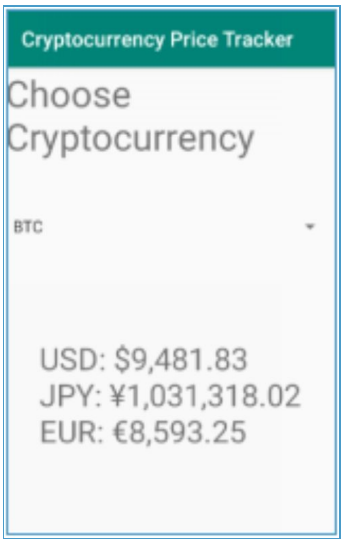


Figure 5.7

Adding Query Arguments in RetroFit

So far, our URL has been hard-coded to retrieve only ‘BTC’ or Bitcoin prices. i.e.:

```
@GET("data/price?fsym=BTC&tsyms=USD,JPY,EUR")
```

We hardcode the query argument *fsym=BTC*. Let’s make it to retrieve the price of the cryptocurrency we have selected in the spinner.

To do so, in *CurrencyAPI.kt*, we remove the hardcoding of *fsym* and replace it with a query parameter:

```
…
import retrofit2.http.GET
import retrofit2.http.Query
```

```
interface CurrencyAPI{
   //@GET("data/price?fsym=BTC&tsyms=USD,JPY,EUR")
   @GET("data/price?tsyms=USD,JPY,EUR")

   fun getCurrencyPrices(@Query("fsym") currency: String): Call<Price>
}
```

Code Explanation

```
   //@GET("data/price?fsym=BTC&tsyms=USD,JPY,EUR")
   @GET("data/price?tsyms=USD,JPY,EUR")
```

Previously, we hardcoded the query *fsym* to be BTC. We will remove this hardcode and instead specify in *getCurrencyPrices* :

```
   fun getCurrencyPrices(@Query("fsym") currency: String): Call<Price>
```

Query parameters can be added in the URL with *@Query* and then specifying the query argument, i.e. *fsym*. To specify this additional currency symbol, we have to add a string argument in our class *PriceRetriever* as shown:

```
class PriceRetriever{
          …

   fun getPrice(callback: Callback<Price>, currency: String){
      val call = service.getCurrencyPrices(currency)
      call.enqueue(callback)
   }
}
```

In turn, we have to add currency also in *MainActivity.kt*:

```
fun retrievePrices(currency: String){
   var retriever = PriceRetriever()

   val callback = object: Callback<Price> {
     override fun onFailure(call: Call<Price>, t: Throwable) {
   }

   override fun onResponse(call: Call<Price>, response: Response<Price>) {
          …
      pricesTextView.text = "Prices for ${currency}" +
      "\nUSD: ${formatUSD.format(response!!.body()!!.USD.toDouble())}" +
      "\nJPY: ${formatJPY.format(response!!.body()!!.JPY.toDouble())}" +
      "\nEUR: ${formatEUR.format(response!!.body()!!.EUR.toDouble())}"
    }
```

```
    }
    retriever.getPrice(callback,currency)
}
```

In *pricesTextView.text = "**Prices for ${currency}**"* , we also output to the text the current cryptocurrency we are showing.

```
    override fun onItemSelected(parent: AdapterView<*>, view: View, pos: Int, id: Long) {
        retrievePrices(parent.getItemAtPosition(pos).toString())
    }
```

Lastly, in *onItemSelected*, we pass in the selected currency.

When you run your app now, you can select different cryptocurrencies and see its prices displayed (fig. 5.8).

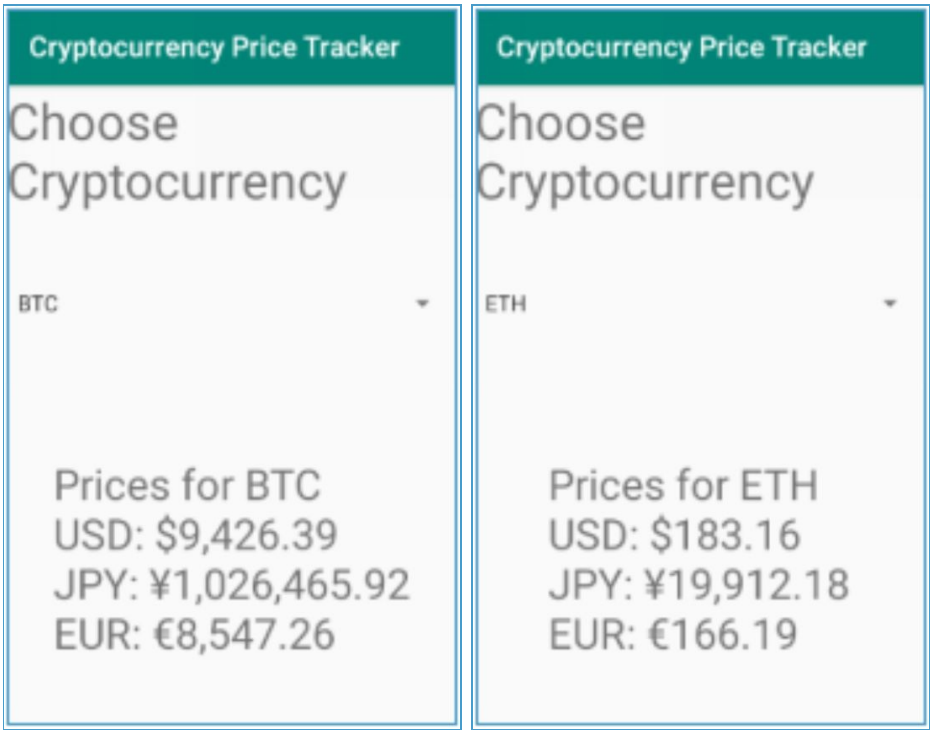


Figure 5.8

Chapter 6: Reading From GitHub API

In the previous chapter, we were introduced to using RetroFit to get data through RESTful APIs. The API we explored in the last chapter was rather straightforward. In this chapter, we will illustrate using RetroFit with an API that has a more complicated JSON structure. We also show how to download and display images that come in the API.

GitHub RESTful API

We will illustrate this by connecting to the GitHub RESTful API to retrieve and manage GitHub content. You can know more about the GitHub API at

https://developer.github.com/v4/

But as a quick introduction, we can get GitHub users data with the following URL,

https://api.github.com/search/users?q=<search term>

We specify our search term in the URL to get GitHub data for user with name matching our search term. An example is shown below with search term *greg* .

https://api.github.com/search/users?q=greg

When we make a call to this URL, we will get the following JSON objects as a result (fig. 6.1).

fig. 6.1

Getting Data

We will again use RetroFit to get data through this API. Create a new ‘Empty Activity’ Android Studio project, name it ‘Find GitHub Users’.

Let’s first design the *MainActivity* layout. In *activity_main.xml*, delete the existing TextView. Then drag in an EditText, center it horizontally in parent (fig. 6.2) and also set its top constraint.

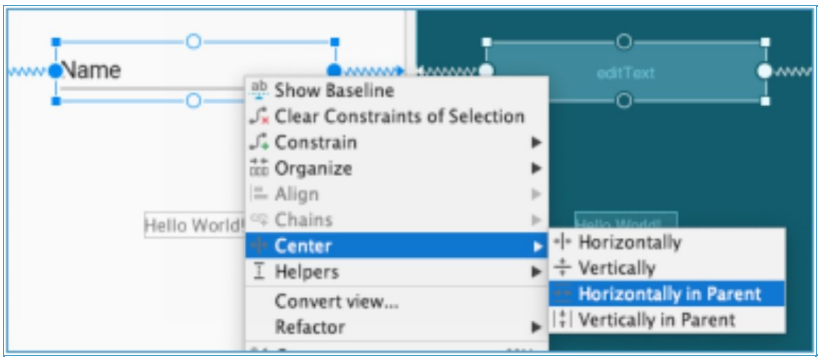


Figure 6.2

Set its hint as ‘Enter GitHub User’ and id ‘searchEditText’.

Next, drag in a button below the EditText. Center it and set its top constraint to the bottom of the EditText. Change its text to ‘Search’ and set its id to *searchButton* (fig. 6.3).

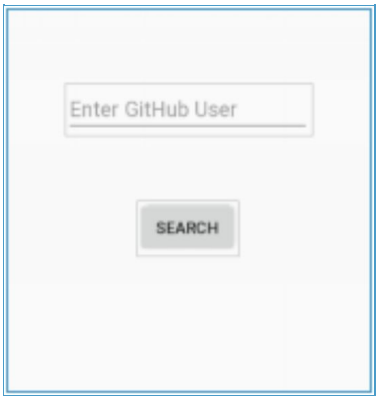


Figure 6.3

This activity will be where a user keys in her search term to find a GitHub User.

Next in *MainActivity*, we implement the *setOnClickListener* to link to the activity which displays the search results. Add in the below:

```
…
import android.content.Intent
import Kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
            …
        searchButton.setOnClickListener{
```

```
        val intent = Intent(this,SearchResultActivity::class.java)
        intent.putExtra("searchTerm",searchEditText.text.toString())
        startActivity(intent)
      }
    }
}
```

Next, let's create the activity which displays the search results.

SearchActivity

Create a new empty activity and call it *SearchResultActivity*. In it, add the below to retrieve the search term:

```
class SearchResultActivity : AppCompatActivity() {

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

        val searchTerm = intent.extras.getString("searchTerm")
    }
}
```

RetroFit Gradle

We now need to add RetroFit and Gson to our project through Gradle as what we have done before in the previous chapter. So add into our *build.gradle* file:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.Kotlin:Kotlin-stdlib-jdk7:$Kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}
```

GitHubAPI

Next, create a new Kotlin file, *GitHubAPI.kt* and declare the following interface

in it:

```
import retrofit2.Call
import retrofit2.http.GET
import retrofit2.http.Query

interface GitHubService{
   @GET("search/users")

   fun searchUsers(@Query("q") searchTerm: String) : Call<GitHubSearchResult>
}
```

Because our full URL is: https://api.github.com/search/users?q=greg, we pass into @GET "search/users".

We also declare a function *searchUsers* where we can specify the argument for query parameter '*q*'.

API Structure

Now let's look at our API structure first to understand it and to query it accurately. Below is a sample extract where I have filtered out some non-essentials:

```
{
  "total_count": 25967,
  "incomplete_results": false,
  "items": [
    {
      "login": "greg",
                …
    },
    {
      "login": "gregkh",
                …
    },
    {
      "login": "greggman",
                …
    },
    {
      "login": "wincent",
                …
    },
    …
```

Our list of users is under the ‘items’ array i.e. greg, gregkh, greggman, wincent. So to query items, we have to define:

```
class GitHubSearchResult(val items: List<User>)
```

Remember that we specify the argument *items* because it has to match exactly the attribute name in the API. And in *items*, we are expecting a *List* of Users. This is where it differs from the previous chapter where we just requested:

```
fun getCurrencyPrices(): Call<Price>
```

which is a single object. But we now use:

```
class GitHubSearchResult(val items: List<User>)
```

because we get an array of objects.

Inside items array

And if we look at what’s inside *items* in the API:


```
{
  "total_count": 25967,
  "incomplete_results": false,
  "items": [
    {
      "login": "greg",
      "id": 1658846,
      "node_id": "MDQ6VXNlcjE2NTg4NDY=",
      "avatar_url": "https://avatars3.githubusercontent.com/u/1658846?v=4",
      "gravatar_id": "",
      "url": "https://api.github.com/users/greg",
      "html_url": "https://github.com/greg",
      "followers_url": "https://api.github.com/users/greg/followers",
      "following_url": "https://api.github.com/users/greg/following{/other_user}",
      "gists_url": "https://api.github.com/users/greg/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/greg/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/greg/subscriptions",
      "organizations_url": "https://api.github.com/users/greg/orgs",
      "repos_url": "https://api.github.com/users/greg/repos",
      "events_url": "https://api.github.com/users/greg/events{/privacy}",
      "received_events_url": "https://api.github.com/users/greg/received_events",
      "type": "User",
```

```
    "site_admin": false,
    "score": 1.0
        },
        …
```

You can get things like *login*, *score*, *avatar_url* (user image), *html_url* (user site url). We thus define class *User* as:

```
…
class GitHubSearchResult(val items: List<User>)
class User(val login: String, val score: Double, val avatar_url: String, val html_url: String)
```

The following steps of creating *GitHubRetriever* and *SearchResultActivity* are similar to what we have done in the previous chapter. So, let's create *GitHubRetriever* with the following code:

```
…
import retrofit2.Callback
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
…
class GitHubRetriever{
   val service : GitHubService

   init{
      val retrofit = Retrofit
         .Builder()
         .baseUrl("https://api.github.com/")
         .addConverterFactory(GsonConverterFactory.create()).build()
      service = retrofit.create(GitHubService::class.java)
   }

   fun searchUsers(callback: Callback<GitHubSearchResult>, searchTerm: String){
      val call = service.searchUsers(searchTerm)
      call.enqueue(callback)
   }
}
```

The above is similar to *PriceRetriever* in the previous chapter.

Next in *SearchResultActivity.kt*, add the following:

```
class SearchResultActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
         …
      val searchTerm = intent.extras.getString("searchTerm")
```

```
        val retriever = GitHubRetriever()
        val callback = object: Callback<GitHubSearchResult> {
            override fun onFailure(call: Call<GitHubSearchResult>, t: Throwable) {
            }
            override fun onResponse(call: Call<GitHubSearchResult>, response:
Response<GitHubSearchResult>) {
                val searchResult = response?.body()
                if(searchResult != null) {
                    for (user in searchResult!!.items) {
                        Log.d("user", user.login)
                    }
                }
            }
        }

        retriever.searchUsers(callback,searchTerm)
    }
}
```

The above is similar to what we have implemented in ***retrievePrices*** in the previous chapter except that this time round in *onResponse*, because we are returned an array items, we use a *for*-loop to traverse it.

Before we test run our app, remember to add Internet permissions in *AndroidManifest.xml*:

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

Running your app

When you run your app now, it should log the users in Logcat. In the next section, we will display this user's information in a RecyclerView.

Custom RecyclerView Rows

We will now implement a page that displays our GitHub user data nicely like in figure 6.4a.

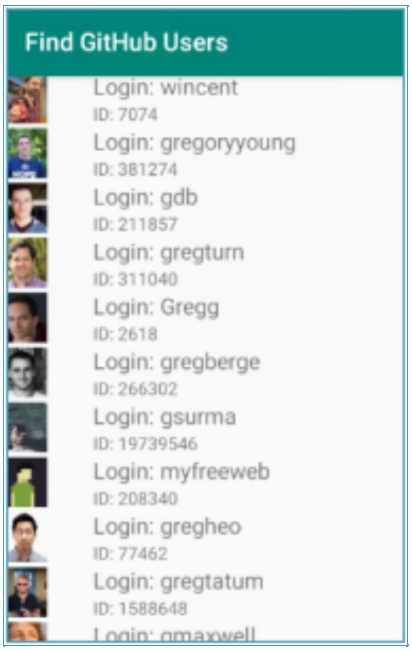


Figure 6.4a

To do so, we want to have our users' data displayed in a RecyclerView. Let's first design the row layout. So, in */res/layout*, create a new 'layout resource file'. Leave the root element as the default 'LinearLayout' and name the file *recycler_user_row*. We want our row to look something like (fig. 6.4b):

Figure 6.4

First, ensure that the *LinearLayout* has *layout_height* set to *wrap content*. As its name suggests, all the UI items within a *LinearLayout* are laid out linearly. Importantly, the *Linearlayout* should be set to horizontal, since we want the image and TextView to be horizontally side by side.

Next, drag in the ImageView and set its *layout_height* to match_parent. We then want to have two TextViews (vertical of each other) set beside the ImageView. To do so, we need to drag in another *LinearLayout* and set it to the vertical option. With that, you are then able to drag in two TextViews into the vertical linear layout. Set the id for the TextView on the top to be *loginTextView* and the TextView below to *idTextView*. You can also increase the text size for

loginTextView to be bigger than *idTextView*.

Your component tree should now look like the following (fig. 6.5):

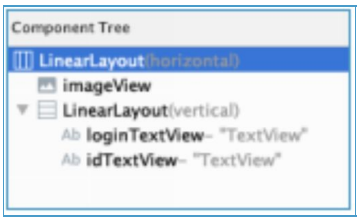


Figure 6.5

That is, we have a LinearLayout containing another LinearLayout. This is done quite often in UI design. Ensure that the inner vertical LinearLayout's layout width and height are set to match_parent (i.e. its parent is the outer LinearLayout). Your row should look something like (fig. 6.6):

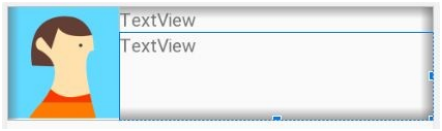


Figure 6.6

Activity_search_result.xml

Next, in *Activity_search_result.xml*, drag in a RecyclerView and set its constraints on all four sides. Give it an id *recyclerView*. We next implement the adapter for our RecyclerView.

UserAdapter.kt

Create a new Kotlin file called *UserAdapter.kt*. This will be the adapter for our RecyclerView. We have previously developed RecyclerView adapters so the following steps should be familiar to you. Fill in *UserAdapter.kt* with the following code:

```
package com.example.findgithubusers

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
```

```
class UserAdapter(val users: List<User>): RecyclerView.Adapter<UserAdapter.UserViewHolder>() {

   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): UserViewHolder {
    return   UserViewHolder(LayoutInflater.from(parent.context)
                                    .inflate(R.layout.recycler_user_row,parent,false))
  }

  override fun getItemCount(): Int {
    return users.count()
  }

  override fun onBindViewHolder(holder: UserViewHolder, position: Int) {
    var user = users[position]
    holder.bindUser(user)
  }
}
```

For now, you will get the error that we have not defined *UserViewHolder*. We will do that later. Let's explain this portion of code first.

Code Explanation

```
class UserAdapter(val users: List<User>): RecyclerView.Adapter<UserAdapter.UserViewHolder>() {
```

UserAdapter will take in a *List* of *User* objects from *SearchResultActivity*.

The rest of the code should be familiar to you. i.e. *getItemCount*, *onBindViewHolder*. Now, let's proceed to implement the inner class *UserViewHolder*.

UserViewHolder

Inside the class *UserAdapter*, add the following:

```
class UserAdapter(val users: List<User>): RecyclerView.Adapter<UserAdapter.UserViewHolder>() {
…

  class UserViewHolder(v: View):RecyclerView.ViewHolder(v), View.OnClickListener{
    var view:View = v
    var htmlUrl: String = ""

    init {
```

```
    v.setOnClickListener(this)
  }

  override fun onClick(p0: View?) {
    val openURL = Intent(Intent.ACTION_VIEW)
    openURL.data = Uri.parse(this.htmlUrl)
    startActivity(view.context,openURL,null)
  }

  fun bindUser(user: User){
    view.loginTextView.text = "Login: ${user.login}"
    view.idTextView.text = "ID: ${user.id}"
    this.htmlUrl = user.html_url

    Picasso.get().load(user.avatar_url).resize(60, 60)
      .centerCrop()
     .into(view.imageView);
  }
 }
}
```

You will be asked to import libraries for the added code. In the end, your *import* statements should be something like:

```
import android.content.Intent
import android.net.Uri
import android.view.View
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import androidx.core.content.ContextCompat.startActivity
import com.squareup.picasso.Picasso
import Kotlinx.android.synthetic.main.recycler_user_row.view.*
```

Now, let’s go back to explaining the code.

Code Explanation - bindUser

```
  fun bindUser(user: User){
    view.loginTextView.text = "Login: ${user.login}"
    view.idTextView.text = "ID: ${user.id}"
    this.htmlUrl = user.html_url

    Picasso.get().load(user.avatar_url).resize(60, 60)
      .centerCrop()
     .into(view.imageView);
  }
```

bindUser binds *login*, *id*, *html_url* and *avatar_url* retrieved from the API to the two text views as shown below:

```
view.loginTextView.text = "Login: ${user.login}"
view.idTextView.text = "ID: ${user.id}"
this.htmlUrl = user.html_url
```

We store the retrieved *html_url* in *htmlUrl* where we will later open a browser linking to that URL.

```
Picasso.get().load(user.avatar_url).resize(60, 60)
  .centerCrop()
 .into(view.imageView);
```

In the above, we use a library called *Picasso* to help us download the image located at *avatar_url*.

To find out more about *Picasso*, you can go to *https://square.github.io/picasso/*. Like RetroFit, Picasso is also provided by Square. It is a powerful image downloading and caching library for Android. It not only provides us with helper functions to resize, crop and assign images to our ImageView easily; it also helps us cache them such that when we request for the same image, there is faster loading time because that image is temporarily stored in the system for quicker access.

To use Picasso, we have to include in *build.gradle*:

```
dependencies {
        …
   implementation 'com.squareup.retrofit2:retrofit:2.3.0'
   implementation 'com.squareup.retrofit2:converter-gson:2.3.0'
     implementation 'com.squareup.picasso:picasso:2.71828'
            …
```

Let’s continue on our code explanation for *onClick*:

Code Explanation – onClick

```
class UserViewHolder(v: View):RecyclerView.ViewHolder(v), View.OnClickListener{
  var view:View = v
  var htmlUrl: String = ""
```

```
init {
  v.setOnClickListener(this)
}

override fun onClick(p0: View?) {
  val openURL = Intent(Intent.ACTION_VIEW)
  openURL.data = Uri.parse(this.htmlUrl)
  startActivity(view.context,openURL,null)
   }
```

In our class header, *class UserViewHolder(v: View):RecyclerView.ViewHolder(v),* ***View.OnClickListener*** **,** we implement *View.OnClickListener* to enable code to be run when a user clicks on a specific row. In our case, we want to navigate to the URL of the GitHub user's page in the browser.

```
override fun onClick(p0: View?) {
  val openURL = Intent(Intent.ACTION_VIEW)
  openURL.data = Uri.parse(this.htmlUrl)
  startActivity(view.context,openURL,null)
   }
```

So, other than extending the *RecyclerView.ViewHolder*, we also have to implement *OnClickListerer* and its *onClick* method. In *onClick*, we then open the URL in a browser with the above code.

Finally, we go back to *SearchResultActivity.kt* and initialize our layout and adapter like what we had previously done when we implemented RecyclerViews. Add the following in bold:

```
class SearchResultActivity : AppCompatActivity() {

  lateinit var layoutManager: LinearLayoutManager
  lateinit var adapter: UserAdapter

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

    layoutManager = LinearLayoutManager(this)
    recyclerView.layoutManager = layoutManager

    val searchTerm = intent.extras.getString("searchTerm")
    val retriever = GitHubRetriever()

    val callback = object: Callback<GitHubSearchResult> {
      override fun onFailure(call: Call<GitHubSearchResult>, t: Throwable) {
      }
```

```
        override fun onResponse(call: Call<GitHubSearchResult>, response:
Response<GitHubSearchResult>) {
            val searchResult = response?.body()
            if(searchResult != null) {
               adapter = UserAdapter(searchResult.items)
               recyclerView.adapter = adapter
            }
         }
      }
      retriever.searchUsers(callback,searchTerm)
   }
}
```

The above code should be self-explanatory since we have gone through in the previous chapters about RecyclerViews. If there is any part that you do not understand, please go back to that chapter. The main thing to note is that in instantiating *UserAdapter*, we pass in the items array i.e.:

```
adapter = UserAdapter(searchResult.items)
```

And when you run your app now, fill in a user search term e.g. ‘greg’, you should get the results displayed nicely like in fig. 6.7

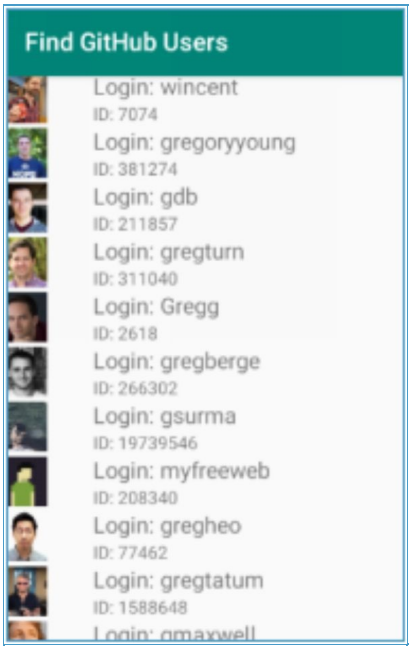


Figure 6.7

Firstly, note that the images that we get from GitHub are of different height and width. But because Picasso allows us to specify that we want images of size e.g. (60,60)

```
Picasso.get().load(user.avatar_url).resize(60, 60)
    .centerCrop()
    .into(view.imageView);
```

We get the specified sizes. See how useful Picasso is?

Summary

In the chapter, we learned how to implement a GitHub User Search application with the help of facilities like RetroFit and Picasso, as well as deepen our understanding of APIs.

Chapter 7: Face Detection, Text Recognition with ML Kit

In this chapter, we will explore using Firebase and its ML Kit (machine learning kit). We will make an app that uses Google ’ s machine learning expertise to detect faces, features on its faces e.g. is its smiling, its left/right eye is open or close etc, and also show probabilities of them. For example in figure 7.0a, we have a picture of a young girl who is not smiling and has both eyes closed.

Figure 7.0a

Our app will then detect the probability of her smiling as only 1.46%, probability of her left eye being opened at 16.25% and for the right eye 2.36% (fig. 7.0b)

Figure 7.0b

The face detection app just serves as an introduction on how to harness the

power of ML Kit in our Android apps. ML Kit offers much more than face recognition abilities. They provide other functionalities like image classification, recognization of landmarks, scanning of bar codes etc. But once you know how to use one of them, it is much easier to learn how to use the rest on your own. Once we are finished with our face recognition app, we will illustrate how simple it is to transit to text recognition and these will give you the building blocks of how to use the rest of the services ML Kit offers.

Now to use ML Kit, we will have to deploy our app to a real device. If you have not deployed any apps to an actual device, revisit the instructions on how to do so back in chapter one.

Connecting with ML Kit in Firebase

First, let ' s create a new empty activity Android project and name it ' Face Detector ' .

On the top right corner of Android studio, click on  and you will be prompted to sign into the firebase console with a Google account (fig. 7.1).

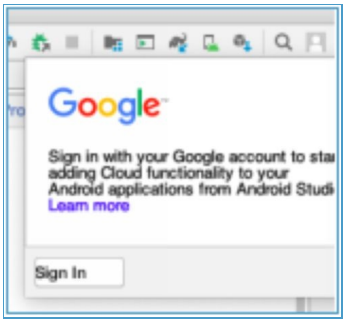


Figure 7.1

Proceed to sign in with your Google account. It will next prompt you that Android Studio wants to access your Google account (fig. 7.2). Make sure you enable access.

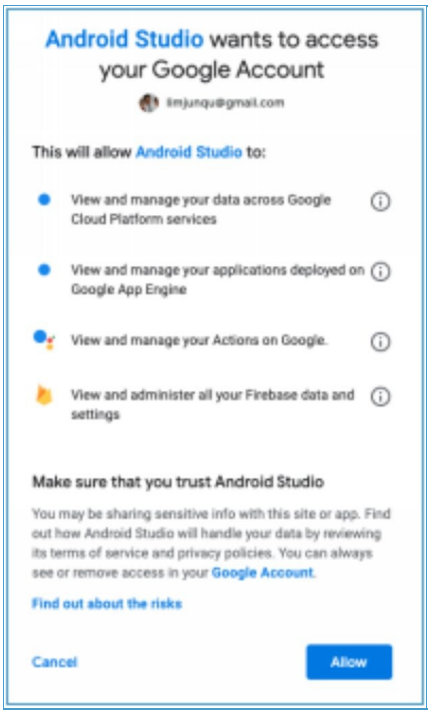


Figure 7.2

Click on ‘ Firebase ’ (fig. 7.3).

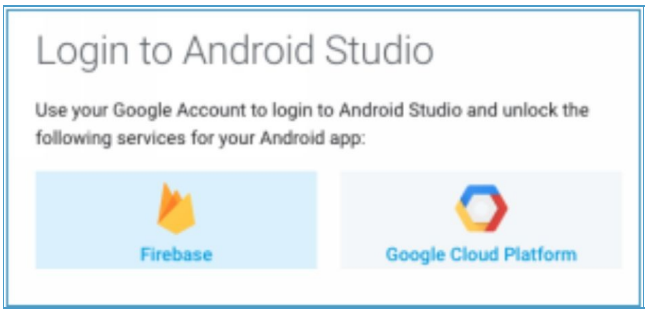


Figure 7.3

Click on ‘ Get Started ’ , and then select ‘ Add Project ’ (fig. 7.4).

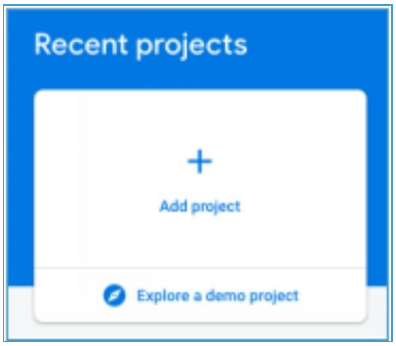


Figure 7.4

Go through the steps as prompted. Give your project a name. I have named mine ‘ Facedetector ’ .

We will enable Google Analytics for this project as recommended (fig. 7.5).

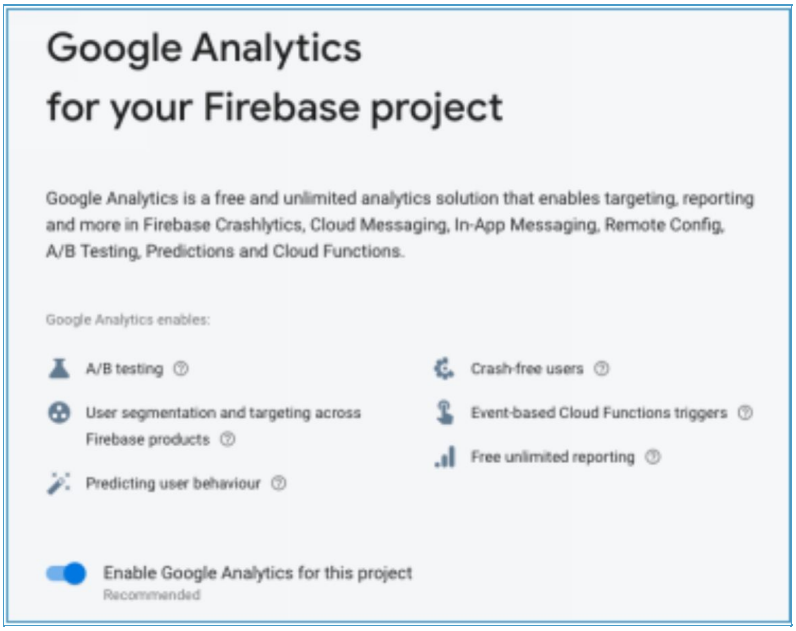


Figure 7.5

Choose and configure an account for analytics. When you are finished, you will have your project created.

Now, go back to Android Studio, and under ‘ Tools ’ , select ‘ Firebase ’ (fig. 7.6).

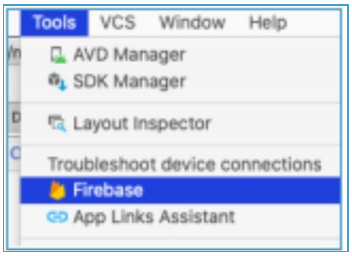


Figure 7.6

It will show you the tools offered by Firebase. For now, we want to connect to Firebase and we can do so by selecting any of the tools. Let ’ s just choose

‘ Analytics ’ , ‘ Log an Analytics event. ’ (fig. 7.7)

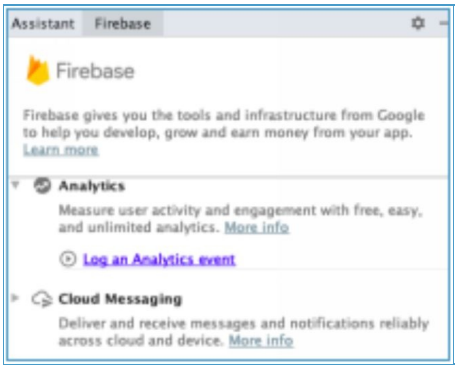


Figure 7.7

Under ‘ Connect your app to Firebase ’ (fig. 7.8), hit ‘ Connect to firebase ’ and select the project we have just created. After a while, our app should be connected to firebase services.

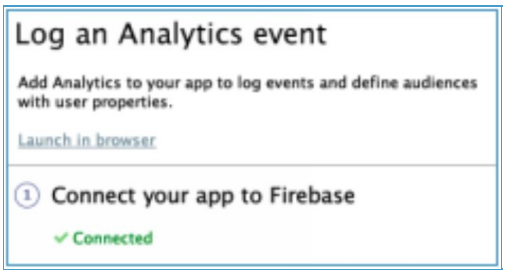


Figure 7.8

The connection will have added a few lines of dependencies into our gradle files which will go through in the next section.

Gradle Files

Now in *build.gradle(Module: app)*, we should have the following added (or add it if it is not there):

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.Kotlin:Kotlin-stdlib-jdk7:$Kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
      implementation 'com.google.firebase:firebase-ml-vision:24.0.1'
     ...
```

As per time of this book ’ s writing, the version is 24.0.1.

Also add the plugin:

```
apply plugin: 'com.android.application'
apply plugin: 'Kotlin-android'
apply plugin: 'Kotlin-android-extensions'
apply plugin: 'com.google.gms.google-services'
…
```

Next, in *build.gradle (Project …)*, add the Google services dependency:

```
…
   dependencies {
      classpath 'com.android.tools.build:gradle:3.5.3'
      classpath "org.jetbrains.Kotlin:Kotlin-gradle-plugin:$Kotlin_version"
      classpath 'com.google.gms:google-services:4.3.3'
    }
…
```

As per time of writing, the version is 4.3.3.

Layout Design

Next in *activity_main.xml*, we create a simple UI for our app. Drag in a button and place it near the bottom. Center it and change its text to ‘ Camera ’ with an id of *cameraButton*.

Next, drag in a text view and center it in the app (fig. 7.9). This will be the text view where we display the recognition results.

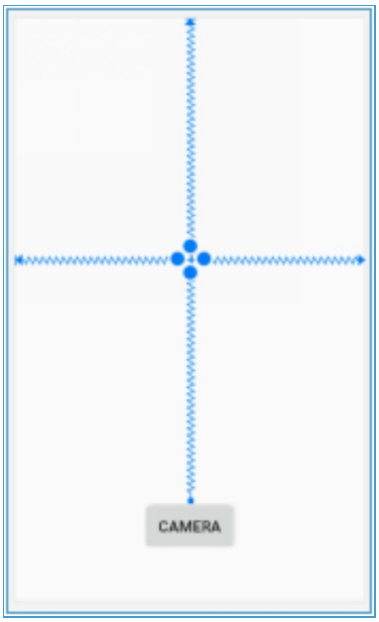


Figure 7.9

Initializing Firebase

To initialize firebase, in *MainActivity.kt* add the following in *onCreate*:

```
class MainActivity : AppCompatActivity() {


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

        FirebaseApp.initializeApp(this)
        }
    …
```

Opening Camera on a Real Device

Next, we want to open the camera to take a picture when user hits the ‘ Camera ’ button. Add in the following codes:

```
class MainActivity : AppCompatActivity() {

    private val REQUEST_IMAGE_CAPTURE = 1

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

```
    FirebaseApp.initializeApp(this)

    cameraButton.setOnClickListener{
        val takePhotoIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

        if(takePhotoIntent.resolveActivity(getPackageManager()) != null){
            startActivityForResult(takePhotoIntent,REQUEST_IMAGE_CAPTURE)
        }
    }
}
```

Code Explanation

```
private val REQUEST_IMAGE_CAPTURE = 1
```

First, we declare a variable *REQUEST_IMAGE_CAPTURE* and assign it a value of 1. We could assign it any random value for e.g. 123, 456, but the essential thing here is to assign a specific value so that we can identify later that this is the specific request we are making. This will be clearer later when we go through the implementation for *onActivityResult*.

```
    cameraButton.setOnClickListener{
        val takePhotoIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

        if(takePhotoIntent.resolveActivity(getPackageManager()) != null){
            startActivityForResult(takePhotoIntent,REQUEST_IMAGE_CAPTURE)
        }
    }
```

We then implement the *setOnClickListener* of *cameraButton*.

```
        val takePhotoIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
```

The *MediaStore.ACTION_IMAGE_CAPTURE* intent opens up the camera.

```
        if(takePhotoIntent.resolveActivity(getPackageManager()) != null){
            startActivityForResult(takePhotoIntent,REQUEST_IMAGE_CAPTURE)
        }
```

The *resolveActivity(getPackageManager()) != null* check ensures that the returned activity can handle the intent. Else, if there is no activity to handle the intent, the app will crash.

We then use *startActivityForResult* so that we can detect when a result is returned from the Intent and thus call *onActivityResult*. In contrast, *startActivity* is often used just for moving between Activities.

So, let ’ s go on to implement *onActivityResult* as the below code shows:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
   super.onActivityResult(requestCode, resultCode, data)

   if(requestCode == REQUEST_IMAGE_CAPTURE &&
                  resultCode == Activity.RESULT_OK){
      val imageBitmap = data?.extras?.get("data") as Bitmap
      detectFace(imageBitmap)
   }
  }
```

Code Explanation

```
if(requestCode == REQUEST_IMAGE_CAPTURE &&
                  resultCode == Activity.RESULT_OK){
```

We first check that the *requestCode* returned to *onActivityResult* is *REQUEST_IMAGE_CAPTURE* to ensure that we get the specific request that we want. This because there can be multiple requests and we want to be sure that the request is coming from our own app. We also check that *resultCode* is *RESULT_OK*. This is to ensure that the user has selected a picture rather than if he selects ‘ Cancel ’ for example.

```
val imageBitmap = data?.extras?.get("data") as Bitmap
detectFace(imageBitmap)
```

The Android camera application encodes the photo in the returned Intent delivered to onActivityResult as a small Bitmap in *extras* under the key “ data ” . We thus retrieve the image *bitmap* and then pass it to *detectFace* which does the actual face detection.%%%

detectFace

Let ’ s proceed to implement the actual face detection code. Implement *detectFace* as follows:

```
fun detectFace(bitmap : Bitmap){
    val image : FirebaseVisionImage
    val detector : FirebaseVisionFaceDetector

    val options = FirebaseVisionFaceDetectorOptions.Builder()
        .setPerformanceMode(FirebaseVisionFaceDetectorOptions.ACCURATE)
        .setLandmarkMode(FirebaseVisionFaceDetectorOptions.ALL_LANDMARKS)
        .setClassificationMode(FirebaseVisionFaceDetectorOptions.ALL_CLASSIFICATIONS)
        .setMinFaceSize(0.15f)
        .enableTracking()
        .build()

    try {
        image = FirebaseVisionImage.fromBitmap(bitmap)

        detector = FirebaseVision.getInstance()
                                        .getVisionFaceDetector(options)


        detector.detectInImage(image).addOnSuccessListener { faces ->
            var resultText: String = ""

            var i = 1
            for (face in faces) {
                resultText = resultText + "\n${i}." +
                        "\nSmile: ${face.smilingProbability * 100} %" +
                        "\nLeft eye: ${face.leftEyeOpenProbability * 100} %" +
                        "\nRight eye: ${face.rightEyeOpenProbability * 100} %"
                i++
            }
            if(faces.count() == 0){
                resultText = "No faces"
            }
            textView.text = resultText
        }.addOnFailureListener { e ->
            Log.d("jason",e.toString())
        }
    } catch (e: IOException) {
        e.printStackTrace()
    }
}
```

Code Explanation

```
fun detectFace(bitmap : Bitmap){
    val image : FirebaseVisionImage
    val detector : FirebaseVisionFaceDetector
```

detectFace takes in a *Bitmap* argument as passed in from *onActivityResult*.

We declare a *FirebaseVisionImage* variable which represents an image object used for detection. We also declare a *FirebaseVisionFaceDetector* object which detects *FirebaseVisionFace*(s) in a supplied image.

```
val options = FirebaseVisionFaceDetectorOptions.Builder()
    .setPerformanceMode(FirebaseVisionFaceDetectorOptions.ACCURATE)
    .setLandmarkMode(FirebaseVisionFaceDetectorOptions.ALL_LANDMARKS)
    .setClassificationMode(FirebaseVisionFaceDetectorOptions
                        .ALL_CLASSIFICATIONS)
    .setMinFaceSize(0.15f)
    .enableTracking()
    .build()
```

We have to specify some options in our face detection process. For example, in *setPerformanceMode* we can specify whether the recognition process should favor either speed (FAST_MODE) or accuracy (ACCURATE_MODE). setLandmarkMode is used to declare whether the recognition process should recognize facial landmarks like nose, eyes, mouth, etc. For more comprehensive explanations about the different modes and options, go to the ML Kit Face Detection documentation at
https://firebase.google.com/docs/ml-kit/android/detect-faces

```
try {
    image = FirebaseVisionImage.fromBitmap(bitmap)
    detector = FirebaseVision.getInstance()
                                .getVisionFaceDetector(options)

    detector.detectInImage(image).addOnSuccessListener { faces ->
                        …
    }.addOnFailureListener { e ->
                …
    }
} catch (e: IOException) {
    e.printStackTrace()
}
```

When that is done, we then have to enclose our face detection code in a *try-catch* block. We use *FirebaseVisionImage.fromBitmap(bitmap)* to return us our image in *FirebaseVisionImage* format.
We get an instance of the face detector with *FirebaseVision.getInstance().getVisionFaceDetector(options).*

We then call *detectInImage* and pass in *image*. Similar to in Retrofit when we

have *onSuccess* and *onFailure* handlers, *detectInImage* provides us with the *addOnSuccessListener* and *addOnFailureListener*.

```
detector.detectInImage(image).addOnSuccessListener { faces ->
    var resultText: String = ""

    var i = 1
    for (face in faces) {
        resultText = resultText + "\n${i}." +
                "\nSmile: ${face.smilingProbability * 100} %" +
                "\nLeft eye: ${face.leftEyeOpenProbability * 100} %" +
                "\nRight eye: ${face.rightEyeOpenProbability * 100} %"
        i++
    }
    if(faces.count() == 0){
        resultText = "No faces"
    }

    textView.text = resultText
}
```

In *addOnSuccessListener*, we are returned a list of *FirebaseVisionFace* objects upon successful detection. The reason we have the *faces* list is because there can be multiple faces detected in an image. Each *FirebaseVisionFace* object represents a face detected in the image. We loop through *face* and for each, we access its *smilingProbability*, *leftEyeOpenProbability* etc. and concatenate them to the string *resultText*. We then assign *resultText* to our text view.

And in case you are missing any *import* statements, here they are:

```
import android.app.Activity
import android.content.Intent
import android.graphics.Bitmap
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.provider.MediaStore
import android.util.Log
import com.google.firebase.FirebaseApp
import com.google.firebase.ml.vision.FirebaseVision
import com.google.firebase.ml.vision.common.FirebaseVisionImage
import com.google.firebase.ml.vision.face.FirebaseVisionFaceDetector
import com.google.firebase.ml.vision.face.FirebaseVisionFaceDetectorOptions
import Kotlinx.android.synthetic.main.activity_main.*
import java.io.IOException
```

Running our app

Now, we are ready to run our app. When I use my app to take the following picture (7.10),

Figure 7.10a

I get the following probabilities:

Figure 7.10b

And yes, it is pretty accurate right? Indeed the girl is not smiling and its low probability that both eyes are opened.

And when I test my app on the following picture (fig. 7.11a),

Figure 7.11a

I get:

Figure 7.11b

And it detects three faces with all three faces having high probability of smiling. So have some fun and test your app on more pictures! Practical usages of face detection include video chat or games that respond to user ’ s expressions or generating avatars from a user ’ s photo.

Text Recognizer

Now, we will see how easy it is to change this app to one that recognizes text.

Much of the code will remain the same as our Face Detection app. For example, we will use the same UI layout (we will still use the camera button), initializing

of firebase will be the same. But in *onActivityResult*, we call *detectText* instead of *detectFace*:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if(requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK){
        val imageBitmap = data?.extras?.get("data") as Bitmap
        detectText(imageBitmap)
    }
}
```

Implement *detectText* as shown:

```
    fun detectText(bitmap : Bitmap){
        val image : FirebaseVisionImage
        val detector : FirebaseVisionTextRecognizer

        try {
            image = FirebaseVisionImage.fromBitmap(bitmap)

            detector = FirebaseVision.getInstance().cloudTextRecognizer
            detector.processImage(image).addOnSuccessListener
                    { firebaseVisionText ->
                val resultText = firebaseVisionText.text
                if(resultText.isEmpty()){
                    textView.text = "Nothing detected"
                }
                else{
                    textView.text = resultText
                }
                Log.d("jason",resultText)
            }.addOnFailureListener { e ->
            }
        } catch (e: IOException) {
            e.printStackTrace()
        }
    }
```

Code Explanation

```
    fun detectText(bitmap : Bitmap){
        val image : FirebaseVisionImage
        val detector : FirebaseVisionTextRecognizer
```

Similar to *detectFace*, we have our *FirebaseVisionImage* . But we also have a *FirebaseVisionTextRecognizer* object that performs optical character recognition (OCR)

on an input image.

```
detector = FirebaseVision.getInstance().cloudTextRecognizer
```

We get an instance of our text recognizer with the above. There are two versions of text recognizer, one that is *onDevice* and another which is cloud-based. As its name suggests, the *onDevice* text recognizer allows you to recognize text with the machine learning model on your device. That is, you do not need an Internet connection to connect to the cloud machine learning model. But the cloud model is obviously more accurate. In this Text recognizer example, we will be using the cloud model. To use the cloud model however, you need to upgrade your firebase plan to the Blaze plan (https://firebase.google.com/pricing). Don ' t worry about paying any money in following along this book however. On the Blaze plan, your account's first 1000 Cloud Vision API calls/month are free.

```
detector.processImage(image).addOnSuccessListener
        { firebaseVisionText ->
    val resultText = firebaseVisionText.text
    if(resultText.isEmpty()){
        textView.text = "Nothing detected"
    }
    else{
        textView.text = resultText
    }
    Log.d("jason",resultText)
}.addOnFailureListener { e ->
}
```

Similar to before, *processImage* provides the *addOnSuccessListener* and *addOnFailureListener* listeners.

In *addOnSuccessListener* , this time its more straight forward as compared to when we had to loop through the *faces* list. This time, we just retrieve the detected text from *firebaseVisionText.text*. And if it is not empty, we display it in our text view.

Now having made these changes, let ' s run our app.

Running our App

When I try to detect the text in figure 7.12a.

Figure 7.12a

Mine was able to successfully detect the text (fig. 7.12b).

Figure 7.12b

So congratulations if you made it all the way here! Note that we should probably change that ‘ Face Detector ’ heading to ‘ Text Detector ’ though.

So these are just some examples of apps we could do with ML Kit. Try exploring more!

Chapter 8: Publishing Our App on Google Play Store

To begin publishing our app on the Google Play Store, we have to enroll for a Google Play developer account (*https://play.google.com/apps/publish/signup/*) which costs a one time fee of $25. You will be granted a Google Play developer account and given access to the Google Play Console (*https://play.google.com/apps/publish/*).

In Google Play Console, you can upload new apps, manage/update existing ones, check app analytics (how many people are using your app, the countries they are from), check sales/trends (how much app downloads, how much money you are making per day/week) and more.

The first step to uploading an app into the console is to create an app icon for it.

App Icon

So far, our apps do not have app icons. They just have the default android looking icon (fig. 8.1). We want an actual app icon to show our users.


Figure 8.1

Open any existing Android project in Android Studio. I will be using my Cryptocurrnecy Price Tracker app. Right click on your *res* folder, select ‘ New ’ , ‘ Image Asset ’ (fig. 8.2).

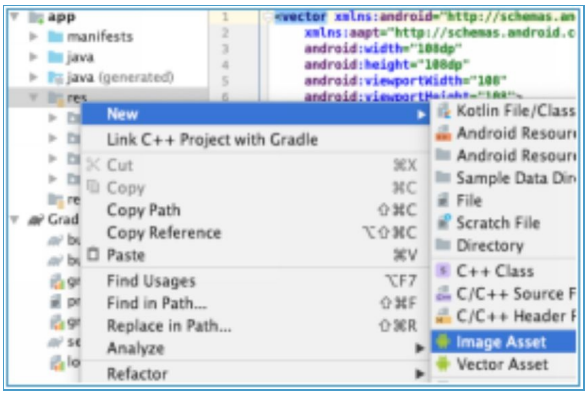


Figure 8.2

The Image Studio will appear (fig. 8.3). As quoted from *https://developer.android.com/studio/write/image-asset-studio*, “ Image Asset Studio helps you generate your own app icons from material icons, custom images, and text strings. It generates a set of icons at the appropriate resolution for each pixel density that your app supports. Image Asset Studio places the newly generated icons in density-specific folders under the *res*/ directory in your project. At runtime, Android uses the appropriate resource based on the screen density of the device your app is running on. ”

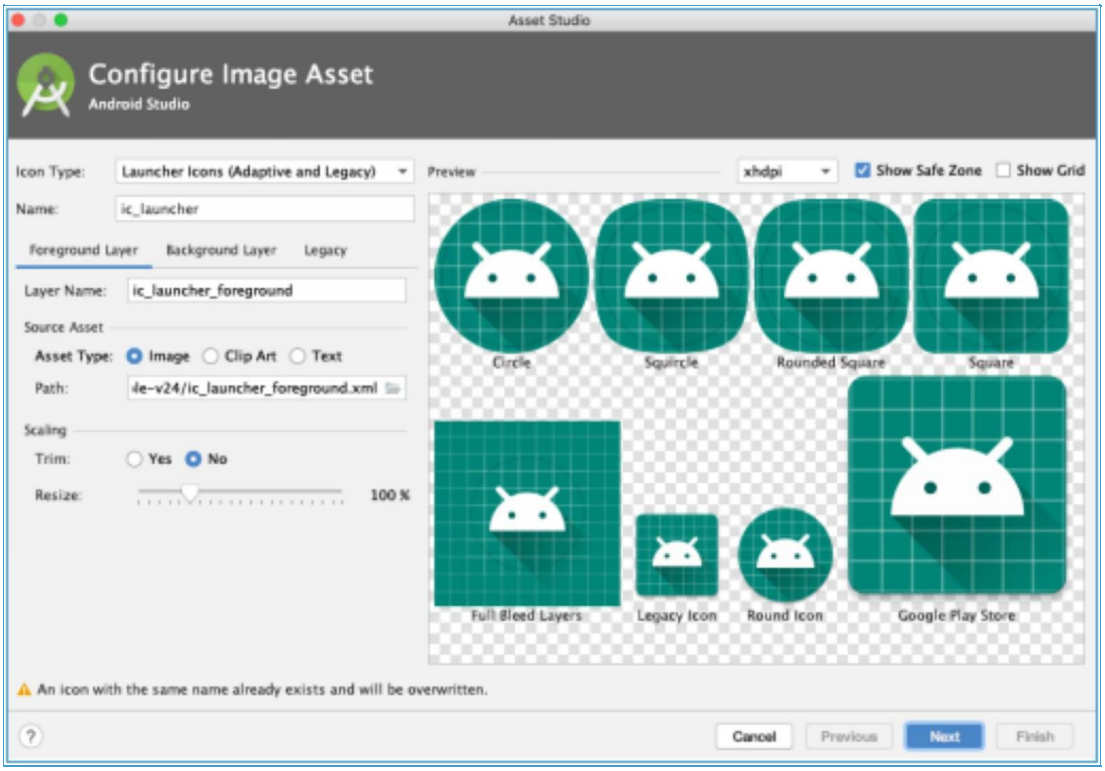


Figure 8.3

Under *Icon Type*, leave the selection as ‘ Launcher Icons (Adaptive and Legacy) ’ . Adaptive launcher icons can display as a variety of shapes across different device models (available in API level 26 and higher).

We next have to apply a foreground and a background. Under ‘ Foreground Layer ’ , ‘ Asset Type ’ check ‘ Image ’ . Upload a 512 by 512 pixel *PNG* picture (e.g. fig. 8.4). 512 by 512 is the recommend size by Google to upload to the App store.

Figure 8.4

Upload your picture by specifying the image file path in ‘ Path ’ .

You can also specify a color or image to use as a background layer (fig. 8.5).

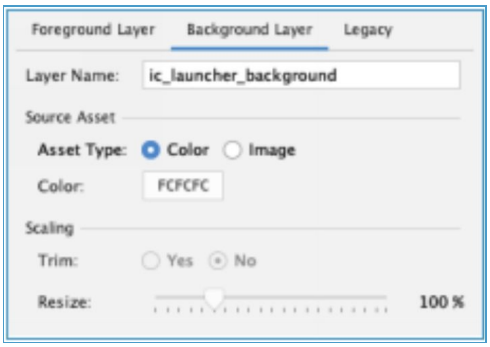


Figure 8.5

In both the Foreground Layer and Background Layer, you are given the option under ‘ Scaling ’ to trim and resize your picture to fit the various icon shapes (fig. 8.6).

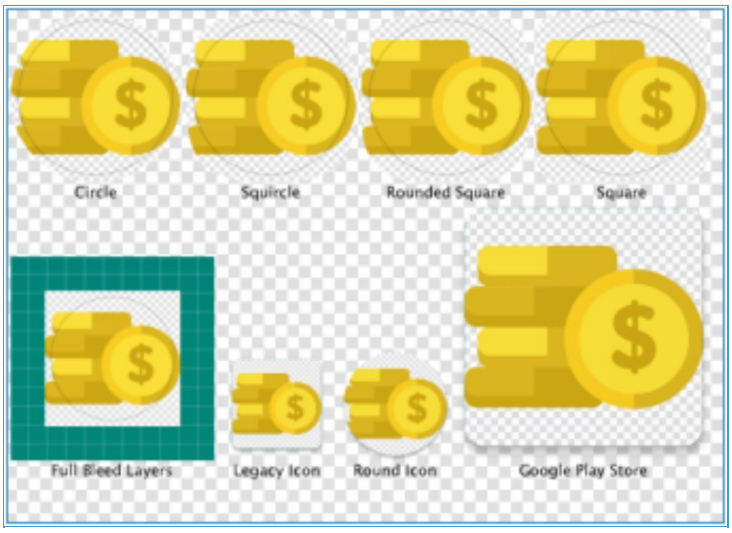


Figure 8.6

You can also optionally change the name for each of the Foreground Layer and Background Layer.

When you have finished resizing, click *Next*, and *Finish*. When that is done, Image Asset Studio adds the images to the *mipmap* folders for the different densities.

And when you run your app on your virtual or actual device, you should be able to see your app icon displayed (fig. 8.7). See how easy it is?

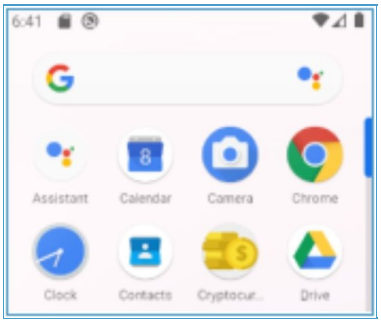


Figure 8.7

Uploading our App on to the Google Play Console

Now that we have our app icon, we are ready to upload our app in the Google Play Console. The steps to publishing on Google ’ s Play Store are not complicated. They are, however, quite in-depth and a little laborious. Most of the

steps involve entering personal information and images about your app. The entire process however should be intuitive.

You should be fine going through the steps on your own as the console provides much detailed instructions. I will however still provide a walk-through in the following sections.

Back in the Google Play Console (*https://play.google.com/apps/publish*), click on ‘ CREATE APPLICATION ’ 

.

Store Listing

Next are the steps to fill in your app ’ s ‘ Store listing ’ (fig. 8.8) information like Default Language, App Title, Short description, Full description, high-res icon image, at least two screenshots of your app, feature graphic, an optional promo video and so on.

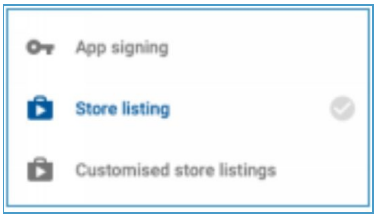


Figure 8.8

Filling up the form fields should be relatively straight-forward to do so on your own.
Tip: to take screenshots of your app on the emulator, you can click on the ‘ Camera ’ icon  (fig. 8.9).

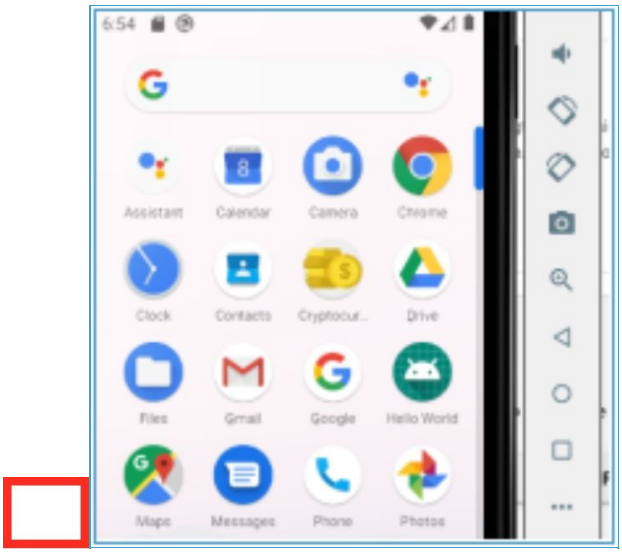


Figure 8.9

By default, the screenshots will be saved on your Desktop. You can change the location by selecting ‘ More ’ in the emulator, and under ‘ Settings ’ , specify the location for your screenshot.

It was once possible to optionally opt out of having a privacy policy by selecting an option called ‘ Not submitting a privacy policy URL at current time. ’ However, it is now a basic requirement that you must have a clear privacy policy when you upload your app to the Google Play Store, depending upon which jurisdiction you are in. If you are based in the European Union (EU) or interact with EU users, it is also important to comply with the General Data Protection Regulation (GDPR).

When all the fields are done, click on ‘ Save Draft ’ and the ‘ Store Listing ’ section should be completed.

Pricing and Distribution

Next, under ‘ Pricing & distribution ’ , you have to specify if want your app to be free or paid (fig. 8.10). Under current policies in the Google Play Store, if you make your app free, you cannot later change it to paid and you will have to upload a new version of your app to do this. There is also the option of having subscriptions and in-app purchases, but these are beyond the scope of this book.

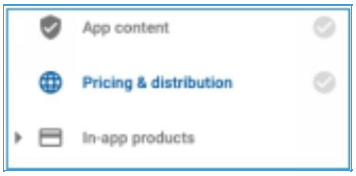


Figure 8.10

You have to specify the countries that your app will be made available in. Check on the ‘ Available ’ radio button at the top of the column to check for all countries (fig. 8.11).

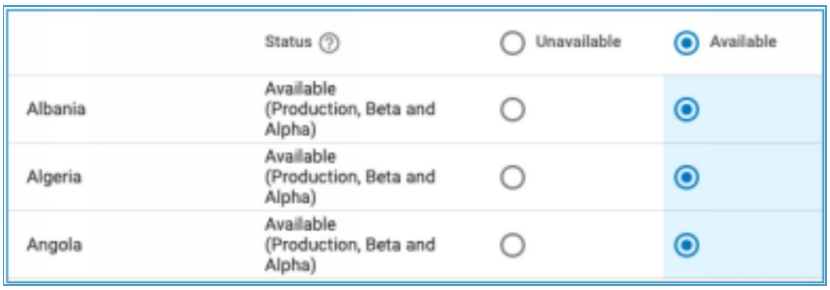


Figure 8.11

Select if it has or has no ads (fig. 8.12).

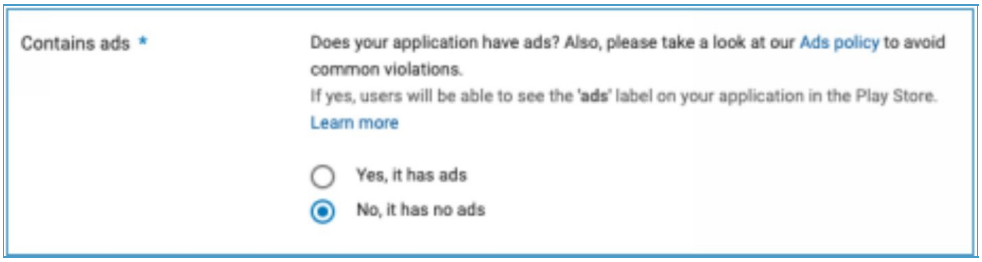


Figure 8.12

You have to declare that your app meets the Content guidelines and US export laws (fig. 8.13).

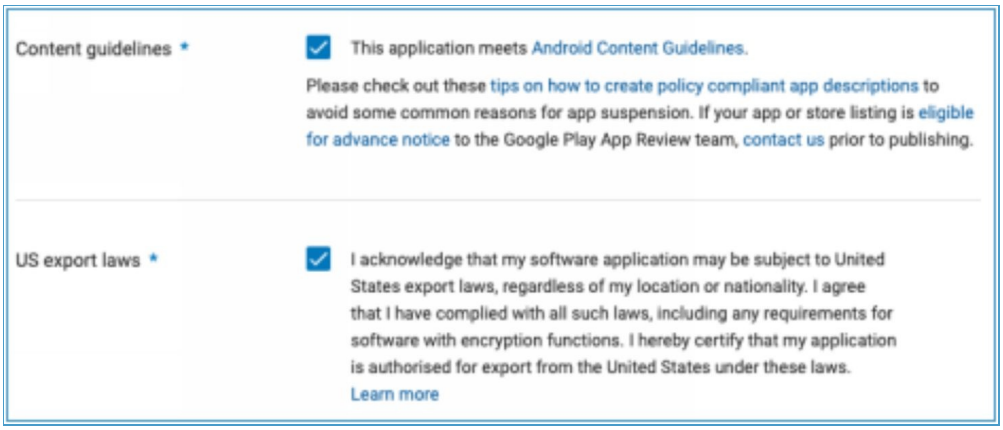


Figure 8.13

Click on ‘ Save Draft ’ and the ‘ Pricing & Distribution ’ section should be completed.

Next, let ’ s see how to generate our APK.

Uploading and Generating our APK

To upload our APK, in the console, go to ‘ App releases ’ (fig. 8.14).

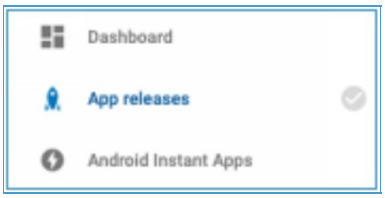


Figure 8.14

Under ‘ Production ’ , click on ‘ Manage ’ (fig. 8.15) to add our Android App Bundle.

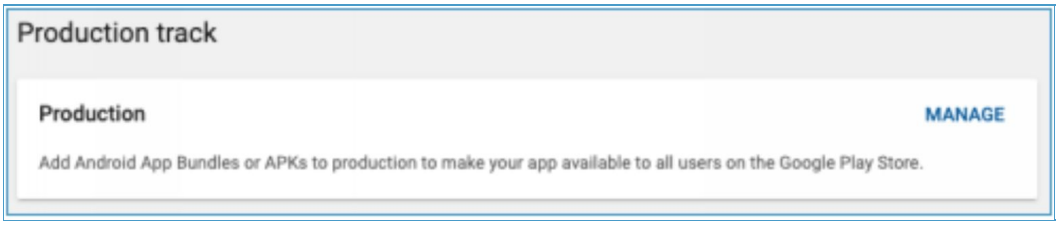


Figure 8.15

Click on ‘ Create Release ’ to publish a version of our app (fig. 8.16).

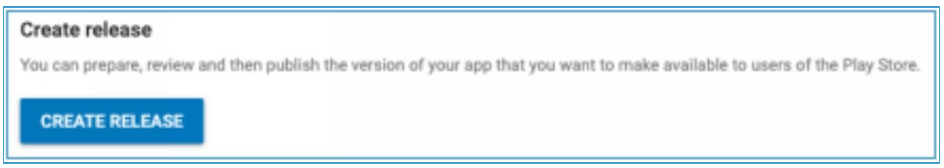


Figure 8.16

Next, select ‘ Continue ’ to let Google manage and protect our app signing key (fig. 8.17).

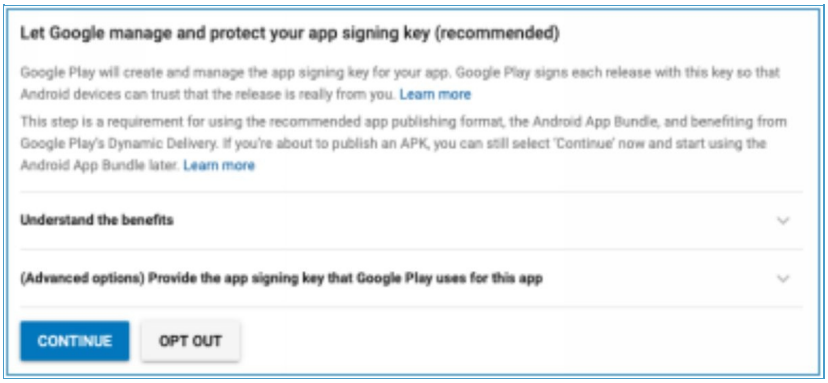


Figure 8.17

At this point, we have to upload our app ’ s APK file. To generate our APK, in Android Studio, under ‘ Build ’ , select ‘ Generate Signed Bundle/APK …’ (fig. 8.18)

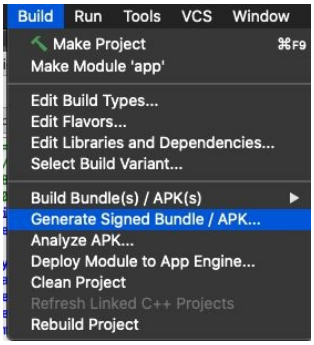


Figure 8.18

Select ‘ Android App Bundle ’ and click ‘ Next ’ (fig. 8.19). We will later explain why we select this option rather than the ‘ APK ’ option.

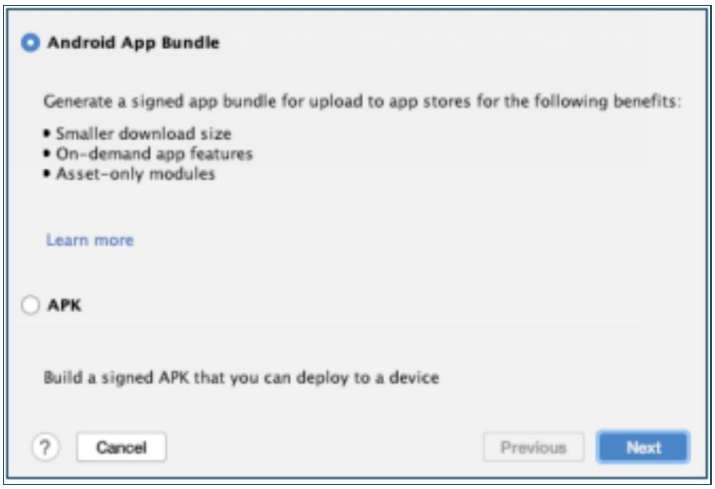


Figure 8.19

In the next form, under ‘ Key store path ’ , select ‘ Create new …’ . Enter in a Key store path, provide a password and fill in the other ‘ Certificate ’ information (fig. 8.20).

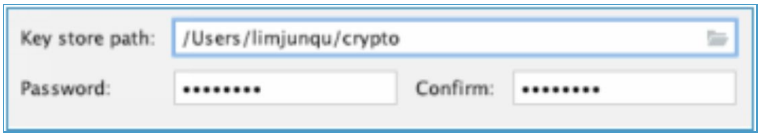


Figure 8.20

Android Studio will use the key store and certificate information provided to cryptographically sign your app. It is a way of proving that the app belongs to you so that only you can make changes or future updates to your app.

Note where you have saved your key store and make sure that you keep it somewhere safe. If someone gets your key store, they can upload a new version of your app. When you are done, click on ‘ Next ’ .

You will then be prompted to specify your app bundle location and also if you want to build the ‘ debug ’ or/and ‘ release ’ app bundles. To build both, highlight both and then click ‘ Finish ’ (fig. 8.21).

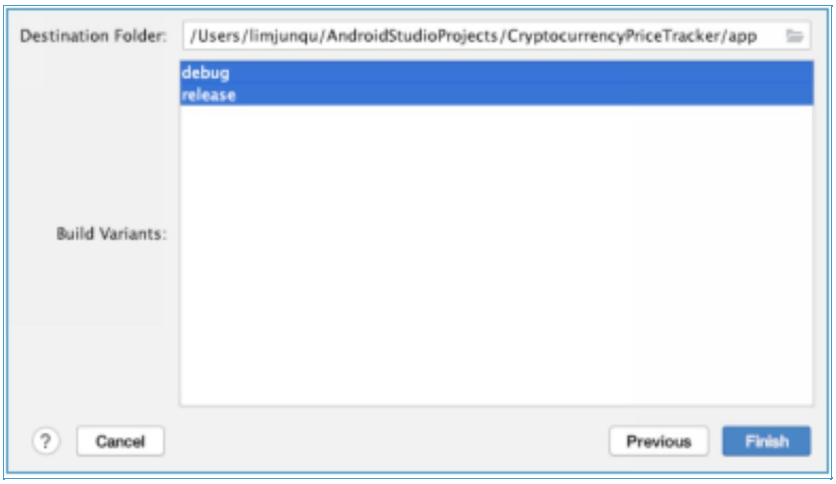


Figure 8.21

Android Studio will take some time to generate the app bundle and also sign it. If you look in your specified folder, you will see the generated ‘ .aab ’ file.

Now, where is the ‘ .apk ’ file? Why are we having an ‘ .aab ’ file instead? This is because we have chosen the ‘ Android App Bundle ’ option earlier. ‘ .aab ’ is actually a new upload format called Android App Bundle. The ‘ .aab ’ file actually includes all our app ’ s compiled code and resources. We will upload the ‘ .aab ’ and as quoted from *https://developer.android.com/guide/app-bundle*, “ Google Play ’ s new app serving model, called Dynamic Delivery then uses this app bundle to generate and serve optimized APKs for each user ’ s device configuration, so they download only the code and resources needed to run the app. You no longer have to build, sign, and manage multiple APKs to support different devices, and users get smaller, more optimized downloads. ”

With the ‘ .aab ’ , upload it as shown in figure 8.22:

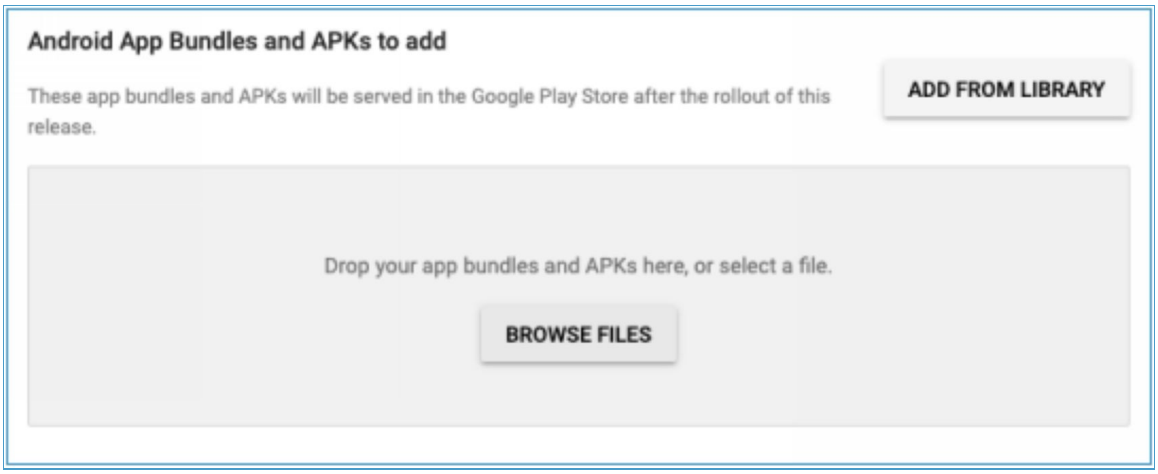


Figure 8.22

At this point, if you have not changed your package name from ‘ com.example. ’ , the console will prompt you to rename your package. To rename your package, go to your *build.gradle (Module: app)* file and specify your package name, for example:

```
…
android {
defaultConfig{
   compileSdk 28
   buildToolsVersion "29.0.2"
   defaultConfig {
      applicationId "com.iducate.cryptocurrencypricetracker"
      minSdk 16
…
```

Note, if you rename your package, you will have to ‘ Build and Generate Signed Bundle/APK …’ again.

When that ’ s done, back in the console, specify the ‘ Release name ’ and “ What ’ s new in this release ” (fig. 8.23).

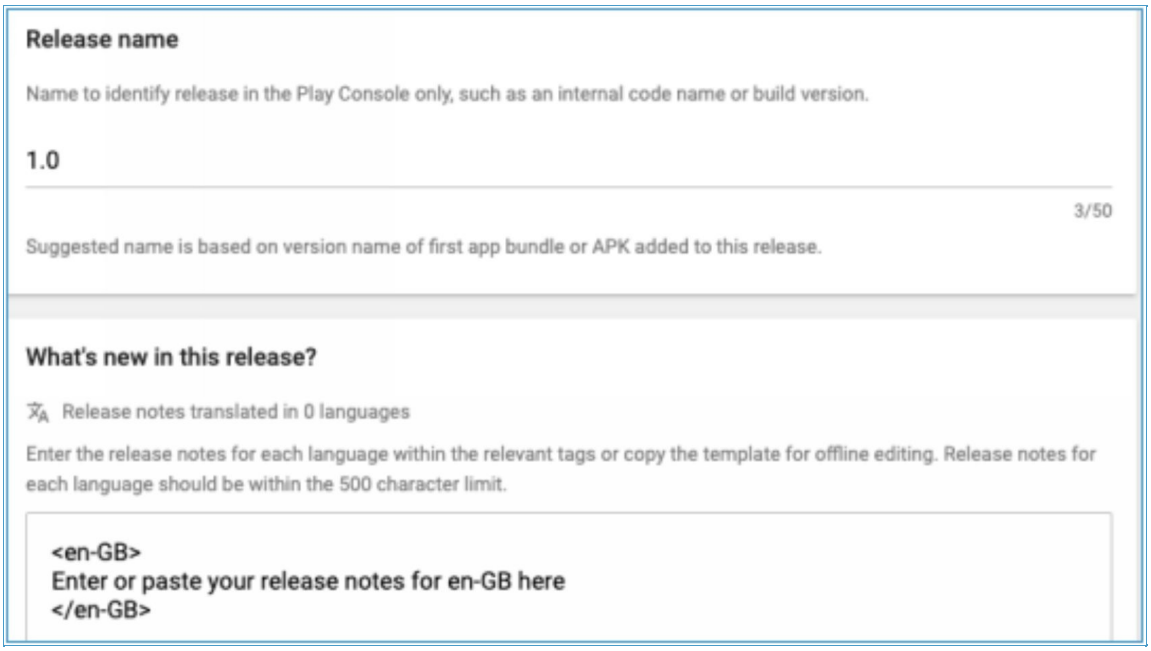
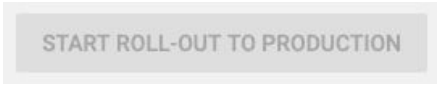


Figure 8.23

Click ‘ Save ’ after which you should be able to click ‘ Review ’ . At this point, the ‘ Start Roll-out to Production ’ button will still be greyed out



since we have not completed the ‘ Content Rating ’ . So let ’ s go ahead and do that.

Content Rating

If you go to ‘ Content Rating ’ , you will be prompted to fill in an email address and select a category (fig 8.24).

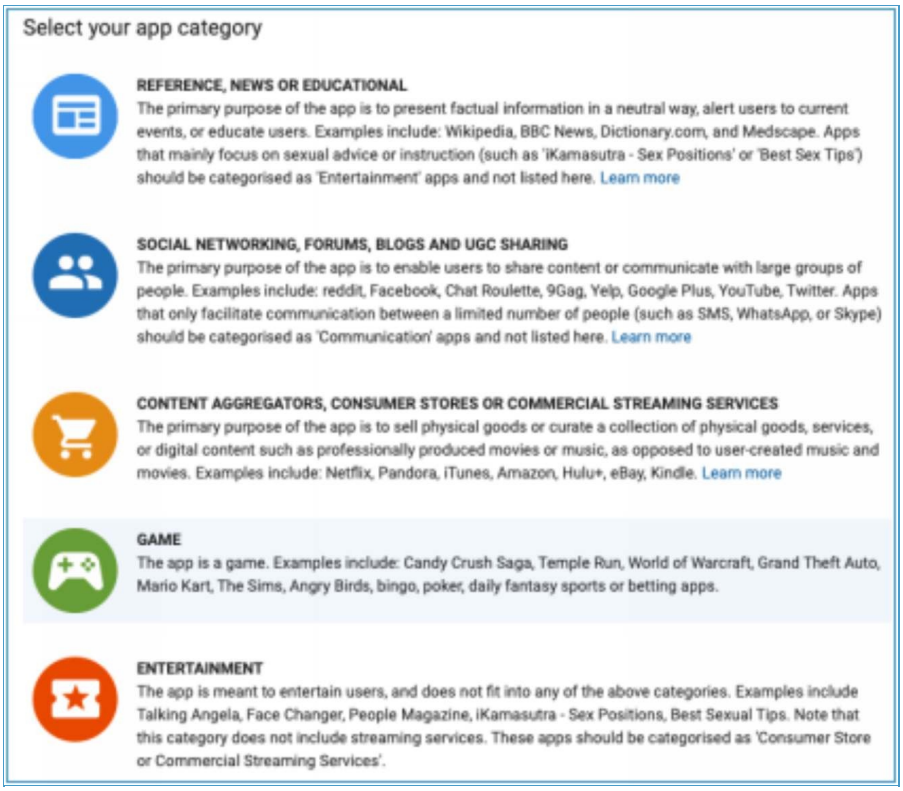


Figure 8.24

You will be then be asked a series of questions which will determine your app ’ s content rating (fig. 8.25).

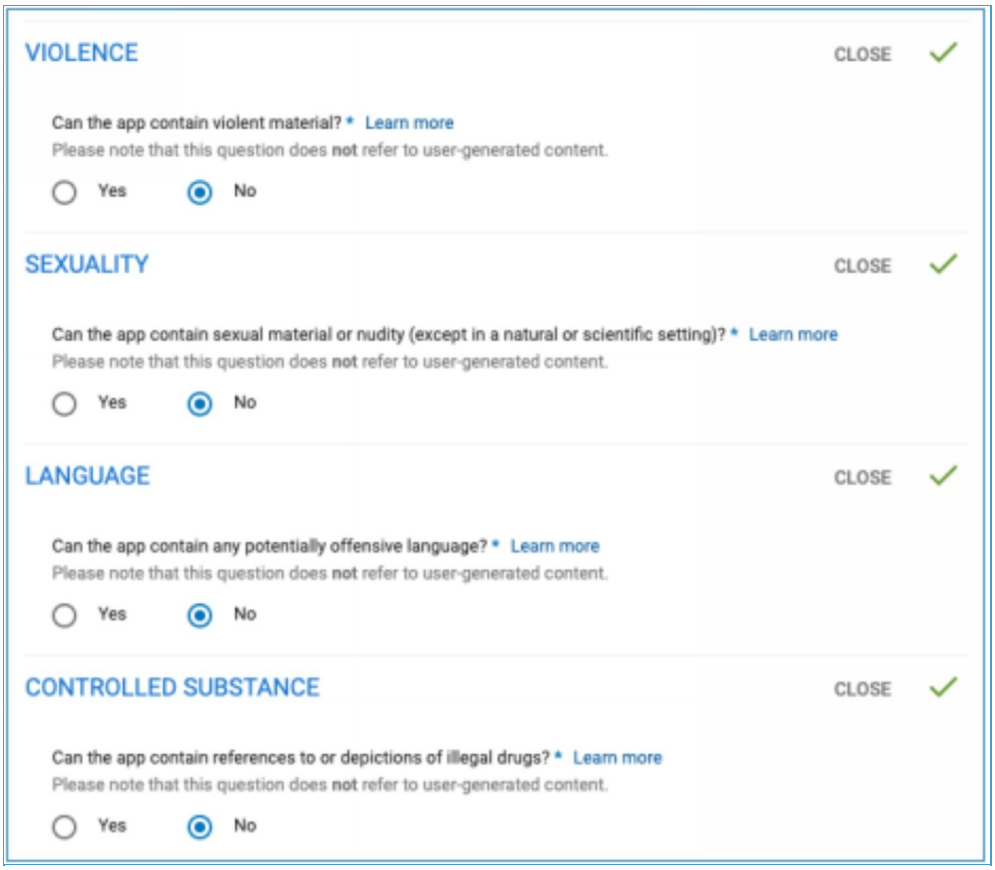


Figure 8.25

Select your answers to the questions and then click ‘ Calculate Rating ’ which will then generate your app ’ s rating. Click ‘ Apply Rating ’ and ‘ Content Rating ’ should now have a green tick marking its completion (fig. 8.26).

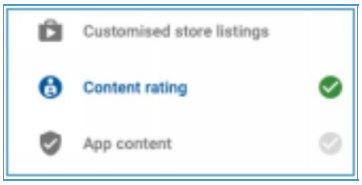


Figure 8.26

App Content

Now, let ’ s go ahead to fill out the form fields under ‘ App content ’ to complete it (fig. 8.27).

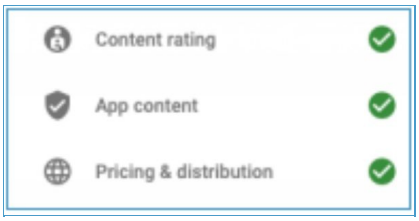


Figure 8.27

App Releases

At this point, go back to ‘ App releases ’ , ‘ Edit Release ’ to complete our release. Click on ‘ Review ’ and we should now be able to click on ‘ Start Roll-out to Production ’ 

. When you do so, there will be a pop up (fig. 8.28) asking you to confirm that your app will be uploaded on the Google Play store.

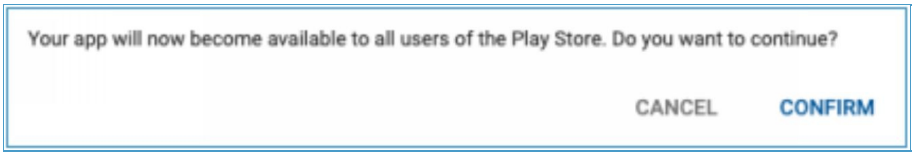


Figure 8.28

Click on ‘ Confirm ’ and your app will begin to propagate in the Play Store!

In some cases, Google might notify you that your app has some issues to fix before it is accepted into the Store. If that happens, just make the corrections as specified by the Play Store and you should be fine.

Summary

We have gone through quite a lot of content to equip you with the skills to create an Android app built with Kotlin and to submit it to the Google Play Store.

Hopefully, you have enjoyed this book and would like to learn more from me. I would love to get your feedback, learning what you liked and didn't for us to improve.

Please feel free to email me at support@i-ducate.com if you encounter any errors with your code or to get updated versions of this book.

If you didn't like the book, or if you feel that I should have covered certain additional topics, please email us to let us know. This book can only get better thanks to readers like you.

If you like the book, I would appreciate if you could leave us a review too. Thank you and all the best for your learning journey in Android app development!

About the Author

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing.

Contact Greg at support@i-ducate.com.